Table of Contents

Programming

The Story So Far...

After defeating the giant evil space bear Grizzlebutt, the Space Buddies were headed home to celebrate their victory. However, a wheel of fancy space cheese the size of a small asteroid hurled them off course, forcing them to crash land on the nearest planet: Earth.

The force of the mighty cheese wheel had knocked the ten Space Buddies so far apart that each one ended up in a completely different location on Earth in their musical spaceships of glory.

Unfortunately for the Space Buddies, the only way to get back to their home planet, Symphonia, was to open a hidden space gateway known as Vortex in G Minor. This special vortex could only be opened when all ten of their spaceships played their unique tune together.

Now they need your help to bring them together and re-establish communication. By collecting all ten tunes you can send them back on their journey home!

Mathematics

A lot of people that enquire about programming and electronics are often made to believe that you need to have amazing strong abilities with mathematics. While it is beneficial to have skills in mathematics, you don’t need to be a mathematician to learn programming or electronics. Some of the most useful skills that you would ever need to know will be explained in this book.

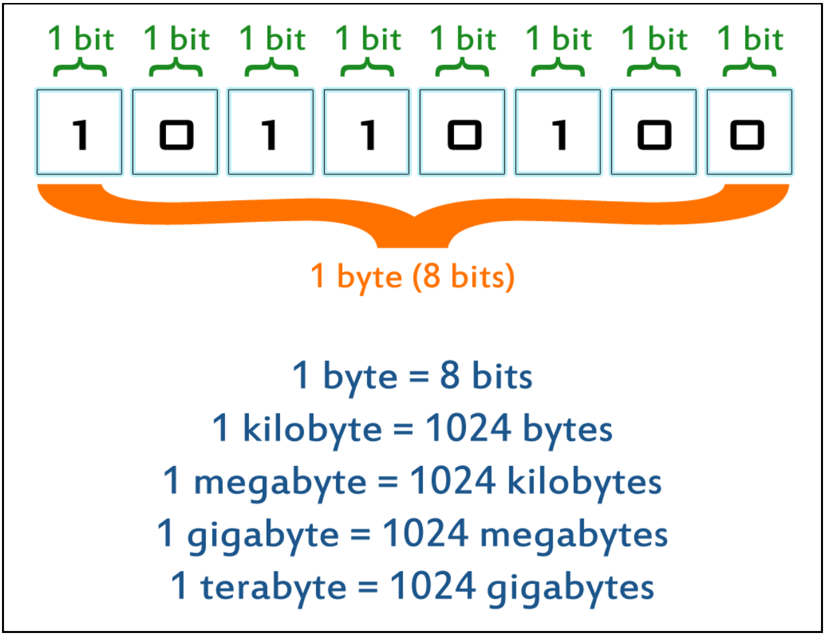
Binary

Bits and Bytes

Many people already know the basics about binary, namely that it is a bunch of ones and zeros, but never anything beyond that. A single one or zero is referred to as a bit or sometimes as a boolean, (depending on the context of what you are talking about), but they are one in the same.

A boolean is an a mathematical notation to represent that a given value is always one of two possibilities, which are most commonly used a way of identifying True and False. Since a bit can only be either a one or a zero, it is the embodiment of what boolean represents, where a one is True and a zero is False.

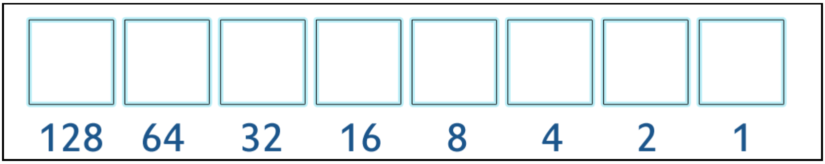
When you begin thinking about data stored on a computer, you often think about the amount of space that is used up, such as megabytes, gigabytes, and so on. Well this is all calculated with binary information. A group of 8 bits is known as 1 byte, and 1024 bytes is known as a kilobyte.



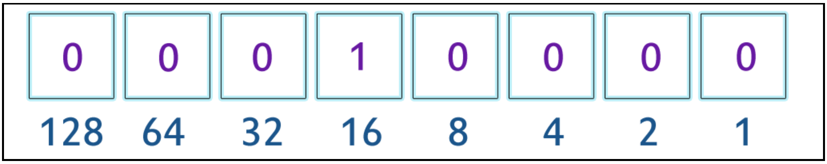
1 byte = 8 bits
1 kilobyte = 1024 bytes
1 megabyte = 1024 kilobytes
1 gigabyte = 1024 megabytes
1 terabyte = 1024 gigabytes

Counting

Since we already know that bits are grouped together in lots of eight, let us look into how numbers are stored in binary. Examine the diagram below



Now it may seem strange, but the above grid representation of eight bits (or one byte) can in fact hold any number between 0 and 255. The way it works is actually quite simple and you can think of it as a game. To begin, we will see how to store a number inside this byte with a few examples. To store the number 16, we would do the following



0	0	0	1	0	0	0	0
128	64	32	16	8	4	2	1

Now each bit as we discovered earlier can only store either a one or a zero, so we use that for counting on this grid. So how did this happen? Well let us break it down a bit further, we want to store the number 16.

Is 16 greater than or equal to 128? No, so this is 0	128	0
Is 16 greater than or equal to 64? No, so this is a 0	64	0
Is 16 greater than or equal to 32? No, so this is a 0	32	0
Is 16 greater than or equal to 16? Yes, so this is a 1	16	1
16 has already been used, so this is 0	8	0
16 has already been used, so this is 0	4	0
16 has already been used, so this is 0	2	0
16 has already been used, so this is 0	1	0

That was a simple example, so let’s go with one that’s a bit more complicated, like the number 30.

Is 30 greater than or equal to 128? No, so this is 0	128	0
Is 30 greater than or equal to 64? No, so this is 0	64	0
Is 30 greater than or equal to 32? No, so this is 0	32	0
Is 30 greater than or equal to 16? Yes, so this is 1	16	1
Since the previous answer was yes, we need to use the remainder. So 30 subtract 16 leaves us with 14. Is 14 greater than or equal to 8? Yes, so this is a 1	8	1
Again since the previous was a yes, we use the remainder. 14 subtract 8 leaves us with 6. Is 6 greater than or equal to 4? Yes, so this is a 1	4	1
One more time. 6 subtract 4 leaves us with 2. Is 2 greater than or equal to 2? Yes, so this is a 1	2	1
We have nothing else left, so this is a 0	1	0

Now that we know how to store numbers inside of this byte, let us look into how we pull them back out. This is actually much easier, as all we have to do is add up all the numbers which have a 1 next to them.

0	0	0	1	1	1	1	0
128	64	32	16	8	4	2	1

So all the numbers with a 1 are : 16, 8, 4, 2 and we want to add these together.
16 + 8 + 4 + 2 = 30

It’s really that simple, let’s look at the biggest number 255

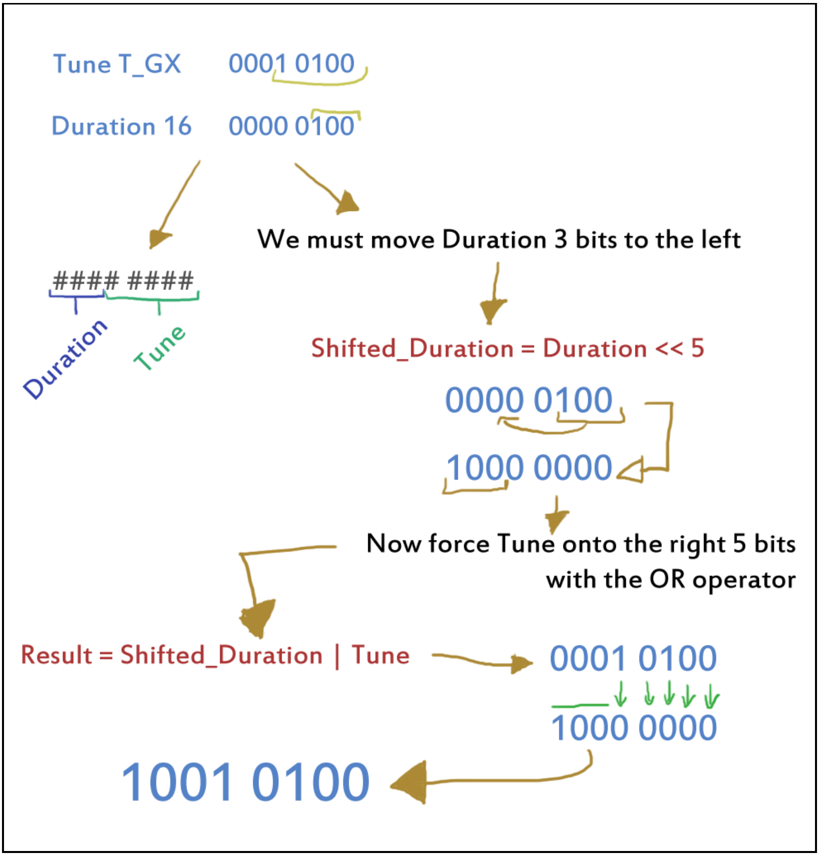
1	1	1	1	1	1	1	1
128	64	32	16	8	4	2	1

So again we add all the numbers with a 1 together.
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

Bit Shifting

What is bitshifting? It is simply the name given to the process of moving bits inside a byte left or right.

Why do we need this? When you're working with microcontrollers, space is very limited, so we do everything we can to save on space. We make use of bit shifting to create tunes in Space Buddies. The tune pitch is 5 bits, and the duration is another 3 bits. Everything comes in from the right and we need to move it to the left so can we fit it all within the same byte. See the follow diagram for a better understanding.


Tune T_GX
0001 0100
Duration 16
0000 0100
We must move Duration 3 bits to the left
####
Duration
Tune
Shifted_Duration = Duration << 5
0000 0100
1000 0000
Now force Tune onto the right 5 bits with the OR operator
Result = Shifted_Duration | Tune
0001 0100
1000 0000
1001 0100

Now on a much simpler level, let’s look at the binary value for the number 2 as illustrated below.

0	0	0	0	0	0	1	0
128	64	32	16	8	4	2	1

What if we wanted to see this as eight different boolean values, each representing cat houses and we only have one cat. Our cat is currently sitting in the house numbered 2, but now he wants to move into the house numbered as 16 to take a nap. We want to move the number 1 (our cat) over to the number 16, so you can tell the byte to shift everything to the left by 3. This is represented as

2 << 3

0	0	0	1	0	0	0	0
128	64	32	16	8	4	2	1

Picture the pair of less than symbols as an arrow telling the byte how many bits to move and in which direction. So the number the byte represents is on the left, followed by your operator (<< or >>) and lastly the number of bits you want to shift. Here's another example.

0	0	0	1	0	1	0	0
128	64	32	16	8	4	2	1

Let's say we want to move everything over to the right by 1 bit. We need to take the number representation of our byte, (which is 16+4=20 in this instance), the operator for moving towards the right (>>) and then the number of bits we want to move across which is 1, leaving us with:

20 >> 1

0	0	0	0	1	0	1	0
128	64	32	16	8	4	2	1

Bit shifting values is extremely easy to understand, as long as you remember that you are shifting EVERY single bit within the byte. Another thing to remember is that when you are shifting a value off of the left or right, it disappears and it's never seen again.

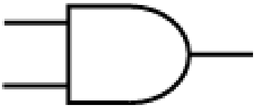
Logic Gates

Computers do not understand the same type of logic that we do and are in fact limited to a number of options in how they are able to do calculations. When we are working with electronics, to get the very best performance, we try to make use of these logic gates whenever possible.

What is a Logic gate? It is one of the building blocks of all digital electronics, and there are a load of different types, but we will only cover the most commonly used types. These all take in 2 binary values, and return a single binary value as a result.

When working with whole numbers, you need to understand that it works on a bit by bit level, rather than looking at the entire number as a whole.

AND



The *AND* logic is extremely simple: both A and B must be 1 for C to be 1, otherwise it is 0.

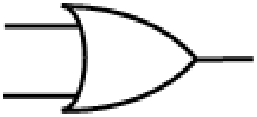
A	0	0	1	0	1	0	0	1
B	1	0	0	1	1	1	0	1
C	0	0	0	0	1	0	0	1

Within C/C++ it would look like this :

```
uint8_t a = 41;
uint8_t b = 157;
uint8_t c = a & b;
```

The result is 9. Since everything is working on a bit by bit level, you might not be able to see why you would ever need to use this, but when working with electronics, using this type of logic is often necessary. Later on you will see some practical examples.

OR



The *OR* logic requires that either A or B is 1 for C to be 1. If they are both 0 though, then C will be 0.

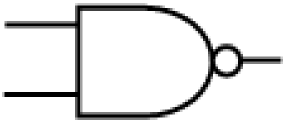
A	0	0	1	0	1	0	0	1
B	1	0	0	1	1	1	0	1
C	1	0	1	1	1	0	0	1

Within C/C++ it would look like this :

```
uint8_t a = 41;
uint8_t b = 157;
uint8_t c = a | b;
```

This results in 185.

NAND



The *NAND* logic works exactly the same as the *AND* logic, however the end results are inverted.

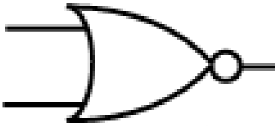
A	0	0	1	0	1	0	0	1
B	1	0	0	1	1	1	0	1
C	1	1	1	1	0	1	1	0

Within C/C++ it would look like this :

```c
uint8_t a = 41;
uint8_t b = 157;
uint8_t c = ~(a & b);
```

The result is 246.

NOR



The *NOR* logic works exactly the same as the *OR* logic, however the end results are inverted.

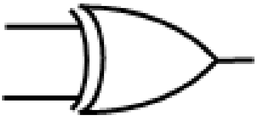
A	0	0	1	0	1	0	0	1
B	1	0	0	1	1	1	0	1
C	0	1	0	0	0	1	1	0

Within C/C++ it would look like this :

```
uint8_t a = 41;
uint8_t b = 157;
uint8_t c = ~(a | b);
```

The result is 70.

XOR



The *XOR* is logic is a strange one. Either A or B needs to be 1 for C to be 1, however not both.

A	0	0	1	0	1	0	0	1
B	1	0	0	1	1	1	0	1
C	1	0	1	1	0	1	0	0

Within C/C++ it would look like this :

```
uint8_t a = 41;
uint8_t b = 157;
uint8_t c = a ^ b;
```

The result is 180.

Electronics

Introduction

How circuits work

Electronics are all about electricity, and how we manipulate it to do all sorts of things for us. Using circuits that we design, we are able to control the flow of electricity which in turn can drive motors, generate light, cause vibrations which form sound, and much more.

Batteries

Batteries are a source of portable electricity, however they are always limited on how much power they hold. There are a great number of different types of batteries. Depending on what is used to make the battery changes whether they are rechargeable or not, along with how much power they are capable of storing. All batteries however, share the same general structure: they have a positive electrode, a negative electrode and an electrolyte separating the two nodes. The electrolyte could be a liquid or a solid, as each battery's makeup is different.

On either side of the outside of the battery you will find a positive terminal and a negative terminal. When these two connect via a circuit, a chemical reaction occurs inside the battery which releases electrical energy which becomes our power source.

Voltage and Current

Electricity is a lot like water flowing through a pipe. Current (measured in amps) is how much electricity will flow through the circuit (or pipe), and when you add resistance you are essentially making the pipe smaller. Voltage can be thought as the pressure of the water, which is how fast/low it flows through the pipe.

AC and DC

If you want to understand electricity then perhaps the most important thing you should know is the difference between AC and DC.

Starting with DC as it is much easier to explain, is Direct Current which is simply a power source which provides power constantly. A very common place that you would find DC power would be batteries.

Now AC, also known as Alternating Current is quite a bit different to DC in that the polarity is constantly switching, meaning that the power and the ground lines are constantly swapping. Now why would this be of use to anybody, you may be asking? Well the short and simple answer is that it is a lot more power efficient as power is only going to the power line 50% of the time. This also is how some devices such as speakers are powered.

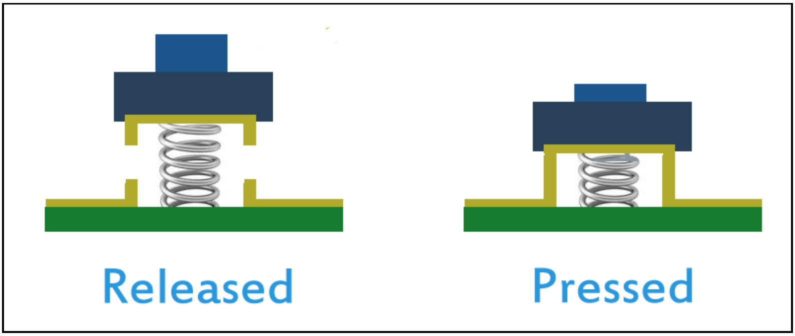
Now why would you want to use DC power when AC is so efficient? Unfortunately, not everything is capable of handling AC as a power source, such as a microcontroller which requires constant power at all times.

Components

There are a lot of electronic components out there, but a handful of them are the most commonly found and used. In fact, many other components are simply composed of a number of the more common components mixed together to achieve a given effect. Electronic components are a lot like a painting. You can put paint wherever you want and you will always end up with something at the end. However if you mix a certain combination of colours in specific locations, you end up with a masterpiece. Electronics are the same. You can put them anywhere you like, but once you put them in the correct places you end up with something magical.

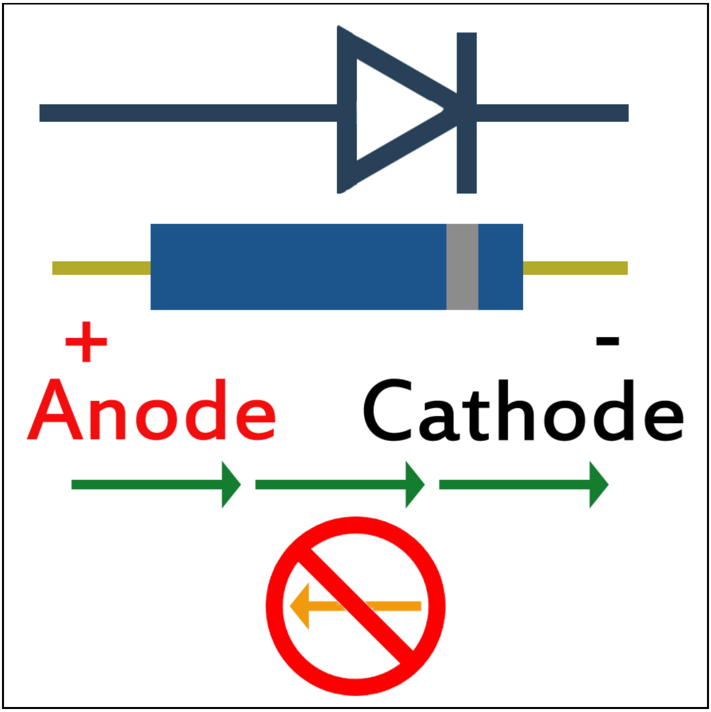
Buttons and Switches

Buttons and Switches are very simple when you understand them. Essentially you have a break in your circuit which is bridged by a button or a switch. The button only completes the circuit when you press the button, and upon releasing the button the circuit is broken once more. The same behaviour is found with switches, however the difference is that once you move them into place they remain there until you physically move them again.



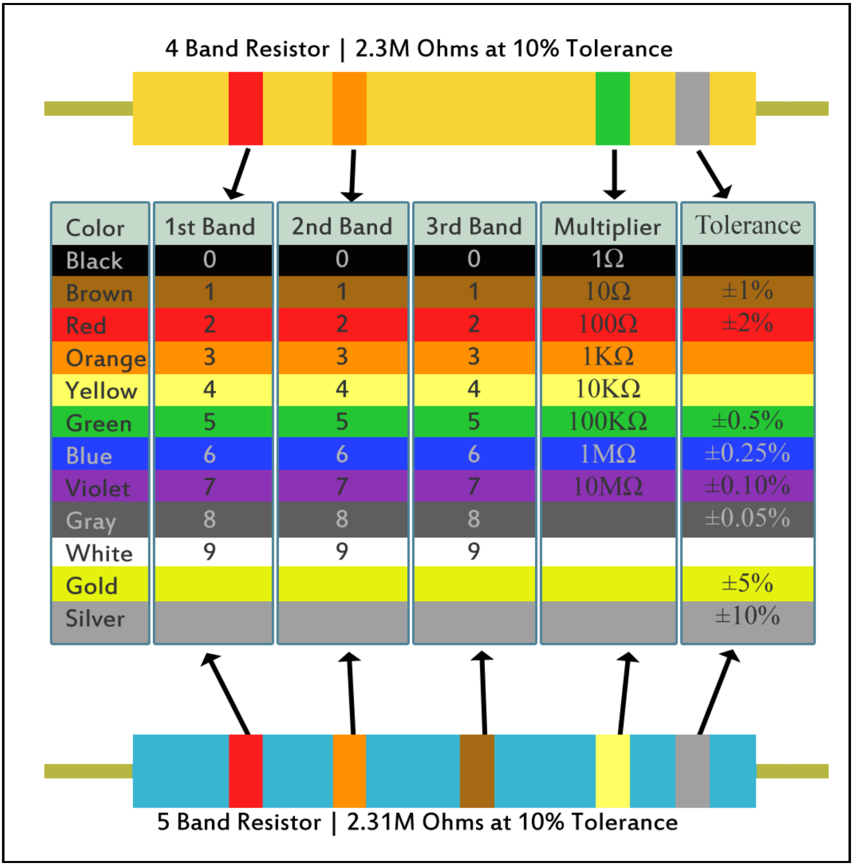
Diode

Diodes can be seen a bit like little traffic cops which restrict current to flow in one direction and only that direction. AC Electricity is capable of flowing in two directions, and this essentially prevents it from going back after it passes through this component.



Resistor

Resistors are very simple components which simply restrict the amount of current that flows through them. The coloured bands are used to indicate how much resistance is imposed by the resistor.



4 Band Resistor | 2.3M Ohms at 10% Tolerance

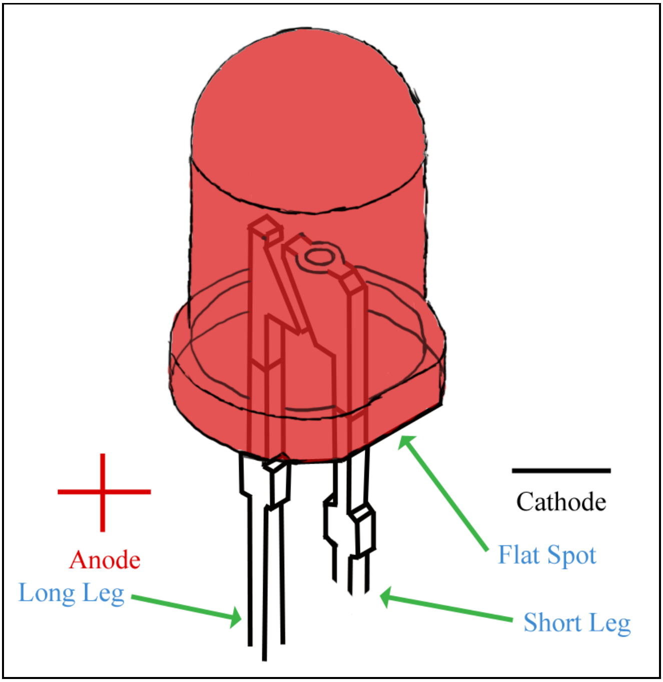
Color	1st Band	2nd Band	3rd Band	Multiplier	Tolerance
Black	0	0	0	1Ω	
Brown	1	1	1	10Ω	±1%
Red	2	2	2	100Ω	±2%
Orange	3	3	3	1KΩ	
Yellow	4	4	4	10KΩ	
Green	5	5	5	100KΩ	±0.5%
Blue	6	6	6	1MΩ	±0.25%
Violet	7	7	7	10MΩ	±0.10%
Gray	8	8	8		±0.05%
White	9	9	9		
Gold					±5%
Silver					±10%

5 Band Resistor | 2.31M Ohms at 10% Tolerance

Light Emitting Diode (LED)

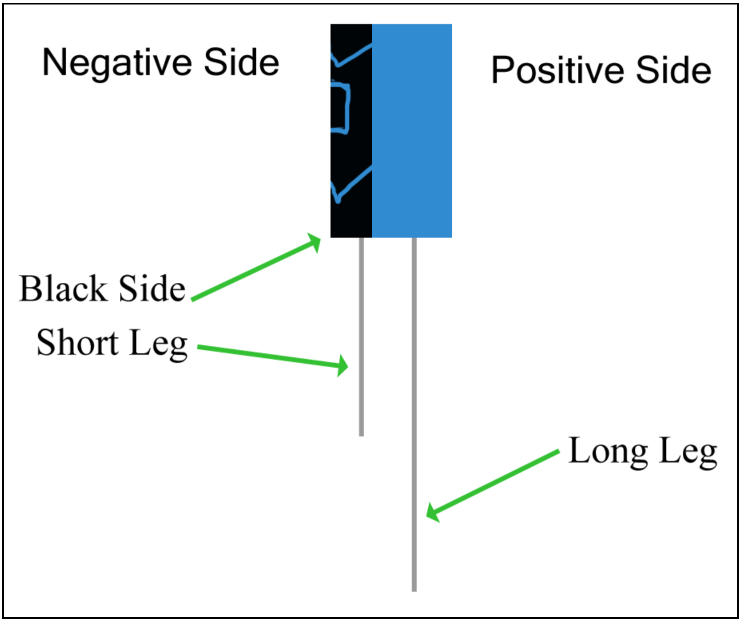
A Light Emitting Diode, or LED, is exactly the same as a regular Diode with the one difference that when current is passed through it, it will light up. LEDs come in a number of different shapes, sizes, colours and luminosities.

You can also find RGB LEDs (used within the Space Buddies kits). These types of LEDs are in fact just 3 really small LEDs sat inside the same little package, allowing you to create a wide range of colours.



Capacitor

Capacitors store energy very much like a battery, and the charge/discharge rate depends on the the circuit. You get a number of different types of capacitors, and they can be used for a number of different things.



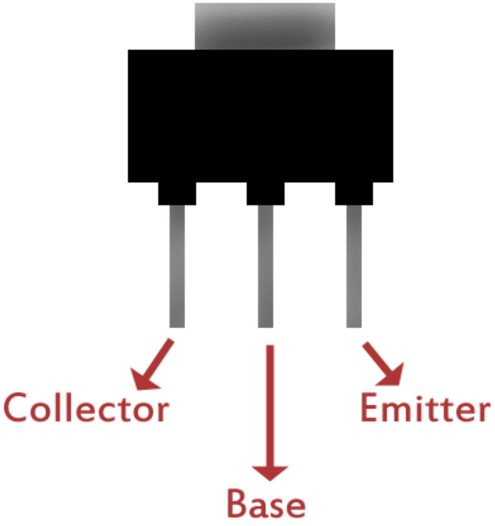
Transistors

Transistors play a very important role in electronics. When you're just starting out you may not use them for very simple circuits, however as you begin to work more with electronics, you may begin to find a good use for these little gems.

The purpose of a transistor is to switch or boost current and signals. The transistors which switch signals are heavily used inside of microcontrollers, but can also be used on their own to control when a current or a signal is allowed to reach its destination - much like a gate.

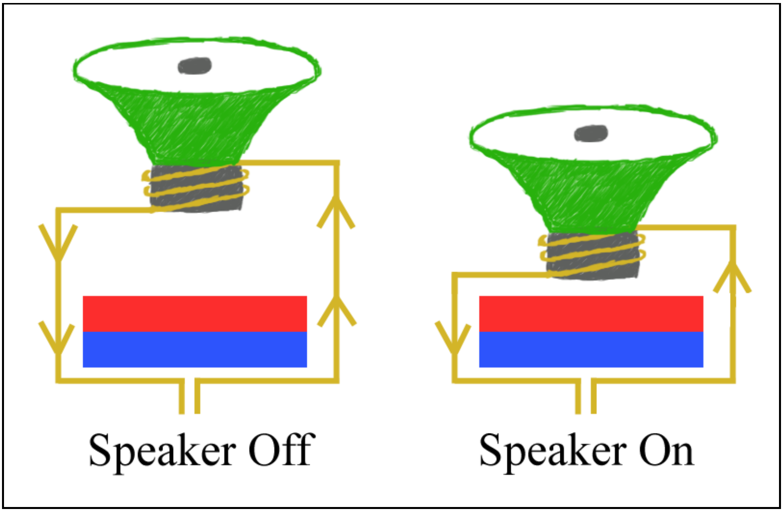
The other type which boost current and signals are often used with speakers to boost the volume, in radios to boost the signal, and a number of other places in common household electronics.

For more information about the different types of transistors and how to use them, visit the Space Buddies website.



Speaker

Everybody knows what a speaker can do, but not everybody knows what it actually is. A speaker is basically a magnet, a coil and a cone stacked on top of each other. When you send an electric current through the coil, it itself becomes a magnet and is pushed away from the magnet already there, which in turn pushes against the cone which makes a sound. Thanks to the quick speed of electricity, we are able to move the cone with the coil and magnet extremely fast, which allows us to make sound.



Potentiometer

These little components are often referred to as variable resistors, and come in all shapes and sizes. The two most common shapes are dials and slides.

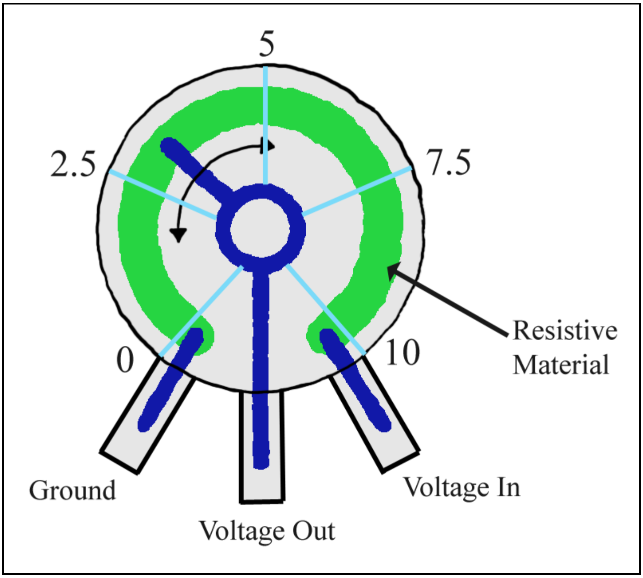
A dial you often see used to control the value volume where you twist it one way to increase the volume and the other to decrease.

A slide you will most commonly see on a mixing deck.

They both do the exact same thing, and it’s all cosmetic on which one you decide to use. A dial takes less physical space, but you might find that a slide looks better on your specific project.

All that they both do is increase and decrease resistance or voltage on any power that flows through it.

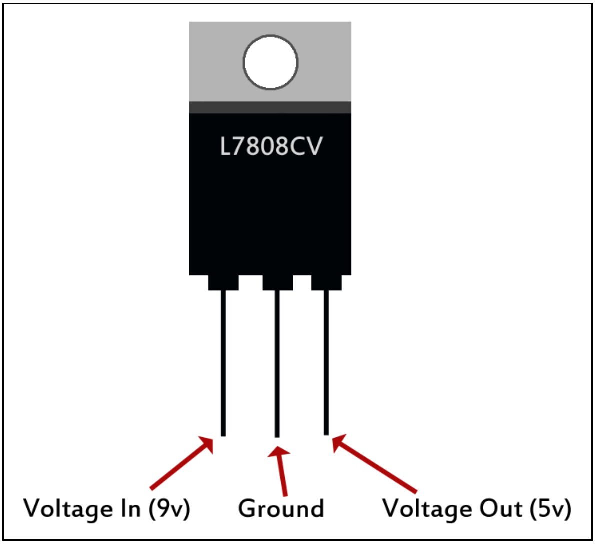
The other variation is that you can get digital and analog variations of potentiometers, to find out more visit the Space Buddies website.



Voltage Regulator

A voltage regulator is really simple: you provide it with a high voltage power source and it will drop the power source to a known amount and ensures that it never exceeds that.

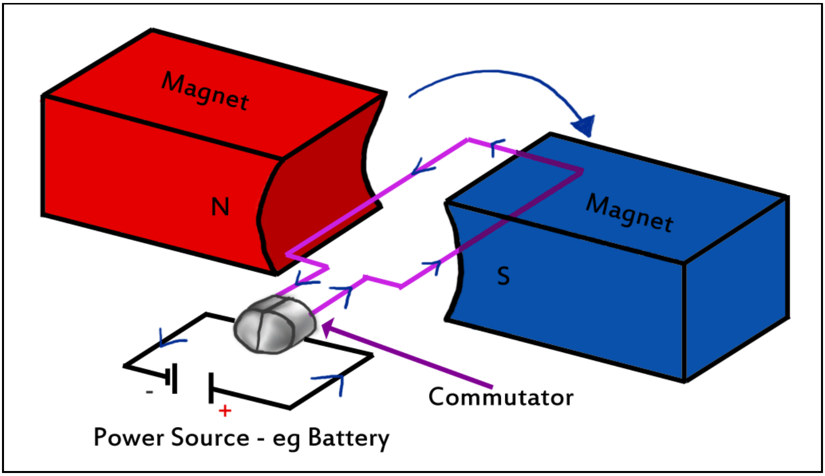
What does this all mean though? Well, if your electronic component needs 3.3v to operate, and you give it more than 3.3v it will just break and you could actually easily cause it to catch fire depending on the type of component. Voltage regulators restrict voltage much like resistors restrict current. So if you use a 3.3v regulator and you give it a 9v or a 12v power source, it will always give out 3.3v so your components do not break and catch fire.



Electric Motor

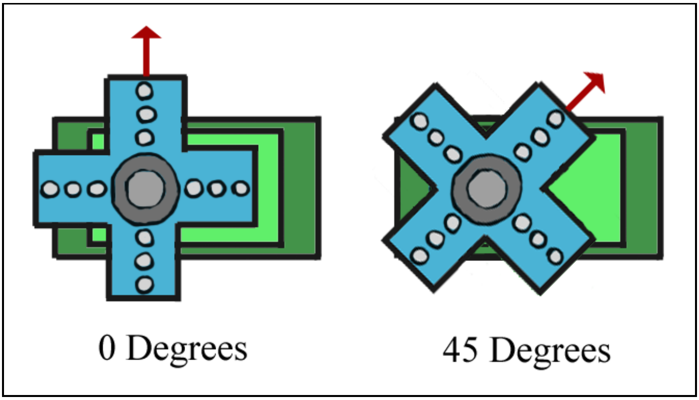
A motor takes in electrical energy and converts that into mechanical energy, or more specifically it allows you to make things move. Its speed is controlled by a mixture of the motor's capabilities and the circuit providing it with power.

Breaking it down to its simplest form, the average electric motor will consist of two magnets charged with the two opposite polarities. A metal rod attached to a commutator which in turn is connected to a power source. As the power goes into the rod, it flows in one direction, which causes one magnet to force it away which rotates it. After the rod is pushed a certain distance away the commutator will reverse the direction of the power flow, forcing the process to start again but this time from the other side of the rod.



Servo

A servo is very similar to a motor, however instead of turning constantly it will be capable of moving a given number of degrees of movement, normally 180 or 360. You provide it with power as you would a motor, however you then provide it with the angle you want it to rotate towards, and it will go there as fast as it can and remain locked there until you tell it to move again.



Micro Controller Units (AVR and PIC)

Micro Controller Units are basically the name for the microchips which you upload code to on your circuit boards. While there are a number of different manufacturers, the two main contenders as far as hobbyists are concerned would be AVR chips developed by Atmel and PIC chips developers by Microchip. You can argue that one is better than the other, but in the end both are very good and each has its own pros and cons.

When you open up most commercial products, you will often find they are making use of a PIC and very rarely would you find an AVR. The reason for this is that for a very long time PIC have always made very good chips full of many useful features, sold at reasonable prices, and their official development kits have always been cheap. Within the last few years they have also begun to give out monthly free samples of almost all of their chips, which means that most hobbyists could easily get away with never having to pay for an MCU if they used PIC exclusively.

Atmel however are extremely popular thanks to the birth of the Arduino platform. The Arduino opened up a whole new way for beginners to get into electronics with minimal knowledge and very low costs. They developed their platform in such a way that it makes use of other circuit boards which stack on top of theirs. They call these shields, and they basically add additional functionality to the original board. The open design attracted so many people that it has become a very commonly used platform for students, artists and more. As a result, more people started developing systems targeted for the Arduino, including a number of different hardware programmers.

Ultimately though, if you learn how to write regular code for electronics avoiding platforms such as the Arduino and looking directly at PIC and AVR, you will find that your code will easily work on both with minimal effort, and your only decision on which to use in the future would simply be based on which chip was better for the job you needed it to perform.

Hardware Programmers

Hardware programmers are physical devices which are used to allow your computer to interface with your microcontroller, and upload your code from your computer to the microchip. There are dozens, if not hundreds of different programmers out there, however the one we are using with Space Buddies is the USBTinyISP. For information on how to use it, please visit the Space Buddies website.

AVRDude

AVRDude is a software application which interfaces with a hardware programmer, allowing you to download, upload and generally manipulate different aspects of AVR microcontrollers. There is also another utility you can download called AVRDudess which simply adds a graphical interface to AVRDude, making it a lot easier to work with.

Fuse Settings

Fuses are a very important part of every microcontroller, and can be seen as their configuration settings. They allow you to tell the chip how fast to run, among other things. You need only set these once, and they are remembered by the chip forever, unless of course you change them, at which point the settings will be updated to whatever you choose.

Program Memory

Program memory is the space on a micro controller where your application code lives after you have uploaded it. This area does not change during the lifetime of the application running and can only be changed when you actually upload replacement code.

Data Memory

Data memory is a bit like a notebook that the program carries around, with a limited number of pages, and behaves as the memory for the program. As soon as you stop providing power to the microcontroller, everything in the notebook is erased and you start with a clean slate when power returns.

EEPROM

EEPROM is very much like Data Memory, however it is slower to access than Data Memory. It does have one major benefit however: when you stop proving power to the micro controller, it does not lose what it has stored.

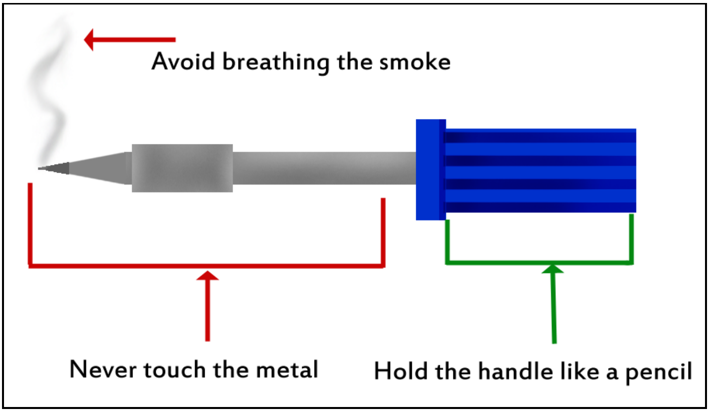
Soldering

An overview

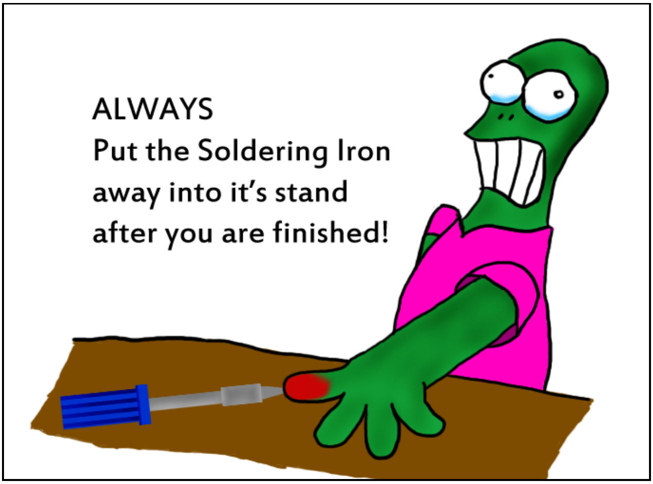
Soldering - in the simplest terms - is basically glueing two bits of metal together with another bit of metal known as solder. Solder comes in a number of forms, but generally it either comes on a reel similar to wire, or in a tub resembling a grey paste.
Now most commonly, anybody that wants to do soldering at home will be doing that with a soldering iron. Once people are a bit more serious about soldering, they invest in more expensive equipment, but in this book we will only explain things relevant to a hobbyist, such as the standard soldering iron.

Safety

When using a soldering iron you need to be extremely careful because it gets really, REALLY hot. Always hold it by the handle, and never touch the hot end. Kids should always be supervised by an adult for this very reason. When you know how to operate them correctly, they are perfectly safe to use, but just like a toaster, put your hand in the wrong place and you get burned.

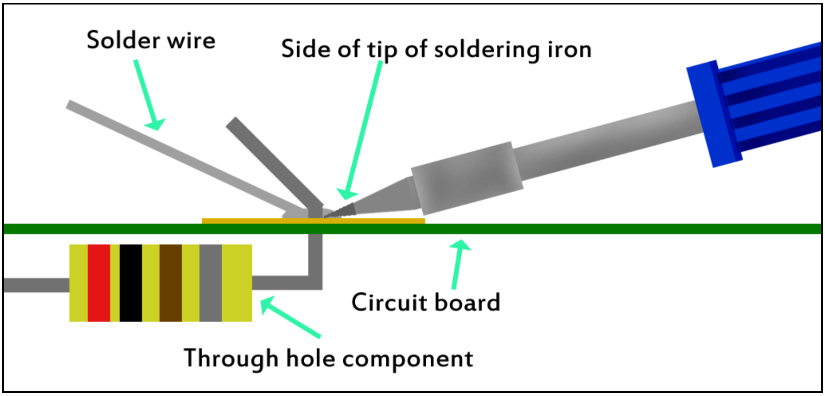


As soon as you are finished, put the soldering iron back in its holder. Never leave it laying on the table because it could very easily roll off of the table and burn you.



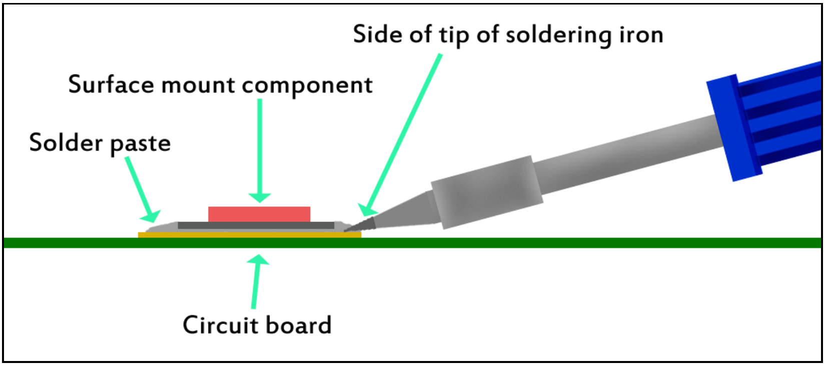
Through hole soldering

Through hole soldering is the term used when you are soldering through hole components, often referred to as TH components. These types of components actually closely resemble their name, as they are components which which physically go through a hole and are soldered through that hole.



Surface mount

Surface mount soldering also refers to the type of components that you are soldering. Surface mount components do not go through holes, they sit on top of the circuit board and are soldering directly onto the circuit board. This is often done with solder paste, however you can use regular solder wire with certain components.



Programming

The basics

A lot of people get scared when they hear about programming, and instantly think about a lot of complicated maths! The truth is that everybody is doing programming every single day of their lives, and you just never knew it. Have you ever given anybody directions, or told them how to perform a task, or told them which book you wanted them to grab off of the shelf for you? In all these cases, you were doing programming in a way.
Programming, when you break it down to its simplest form, is what we call giving a set of set by step instructions to a computer. The only real difference is that for you to be able to communicate with a computer, you need to do so with code that the computer can understand. Much like we speak languages to communicate with other humans, you need to speak code when communicating with computers. Unlike humans, a computer has no common sense and will follow every single instruction you give it - down to the exact letter. So if you give it some bad instructions, it might not do exactly what you think it should. It won't correct itself when it thinks something is wrong, instead it will follow every command you have asked of it, or at least attempt to.

Code

Code is language of computers. This language is really a set of instructions you write for the computer to follow. These instructions can be written in a number of different computer languages, designed to make it easier for us as humans to communicate with machines. Programmers often have their own favourite language, and will have their own opinions of why one may be better than another. The truth is that all of this is really personal opinion when you're starting out, and you should just pick one that you understand enough to do what you want to do. In time as your skills increase, you will learn more languages and will eventually develop opinions. In this book we will be explaining the basics of the C/C++ languages.

Compilers

A compiler is a computer program which takes the instructions you wrote and translates that into the actual language that the computer speaks. Think of it like this: you speak English but your friend only speaks German, so you need a translator to turn your English sentence into a German one. The code you write is designed for humans to easily interpret, but is like a foreign language to a computer. For the computer to understand your code, you use a compiler which in essence ‘translates’ your code to a machine interpretable language.

Integrated Development Environment (IDE)

An IDE is software which helps you write code. There are many options based on the type of computer and type of code you are doing. For up to date information about getting one setup for Space Buddies, please visit the Space Buddies website.

C/C++ syntax

Variables

You are always going to need to store some information when you are writing code. However when working with a lot of information, you need a way to group that information by name, so that you know what each bit of information is and this is exactly what a variable will do for you. There are of course a number of different types of variables, and the key to using them well is to understand them all.

Integers

An integer is just a fancy name for a whole number, and no fractions. Now there are two types of integers with computer programming, signed and unsigned. Signed integers have both positive and negative values, whilst unsigned only have positive numbers as well as the number zero. The easy way to identify an unsigned number in C/C++ is by the letter U in front of the variable declaration. You may be wondering what's the point of having signed and unsigned variables. The point is that an unsigned integer can count twice as high that a signed integer can, so unless you need to count into negative numbers it is best to make use of an unsigned integer as you can count higher.

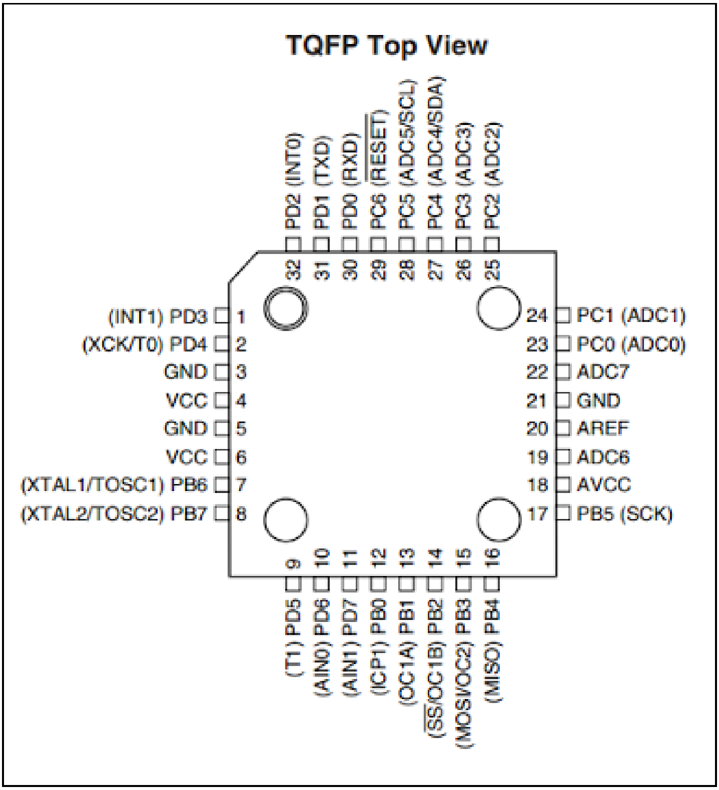
Declaration	Description	Size	Range
uint8_t	Unsigned 8-bit Integer	1 Byte	0 to 255
uint16_t	Unsigned 16-bit Integer	2 Bytes	0 to 65535
int	Signed 32-bit Integer	4 Bytes	-2,147,483,648 to 2,147,483,647
long	Signed 64-bit Integer	8 Bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Code Examples

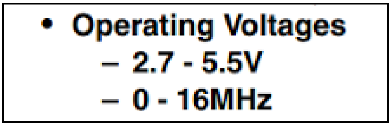
There is a lot of technical details you could get into for writing code, but in all honesty, it’s best to not let any of that keep you from making something awesome!

Understanding Pin Mappings on a Datasheet

With the Space Buddies project we are using the ATMega8 microcontroller. Every microcontroller comes in multiple shapes and sizes, and for Space Buddies we decided on the TQFP size simply because it went well on the spaceship.



The first word just next to each pin on the microcontroller is what is important to us.

GND	Pins marked with GND are pins that connect to ground, which is also the negative side on your power source.
VCC	Pins marked with VCC are pins that connect to the positive side of your power source. Please pay attention to what voltage the datasheet says though! The ATMega8’s details look like this  • Operating Voltages – 2.7 - 5.5V – 0 - 16MHz This tells us that a voltage between 2.7v and 5.5v is perfectly acceptable, which means we can use coincell batteries (3.3v) or even direct power from a USB cable (5v) and it will still work just fine.
PB# **PC#** **PD#**	Pins marked with PB0-7, PC0-7, PD0-7 are groups of data pins. This are the pins you would use in your code to do just about everything. They are are in groups of no more than eight, simply because when you speak to them via code, you speak to the group rather than the individual pins, and this is where the bitwise math comes into play! It’s not all that scary once you see the upcoming examples.
Other pins	The other pins all play an important role, but to do the basics we only really care about the above mentioned ones. At the end of this book you will find a link to a website which can tell you more about all the other pins.

Pin State

With electronics each pin on a microcontroller has a state, which says if it is input or output.
When you set the pin to be an input, that means you will be reading information from something connected to that pin. One example would be reading when a button is pushed.
When you set the pin to be an output, that means that you are talking to give instructions to some other device. A fine example is turning an LED on or off.

DDR# controls the state of a pin group. So if you want to change the pin state for a pin in group B you would use DDRB, if you want to change the state of a pin in group C, you would use DDRC, and so on.

Sending Data

When a pin is set to output you can send data from it, and do things like turning an LED on and off, play a sound, and much more.
Much like when changing pin state, you control everything on the group level and with sending data it is with PORT#. So for a pin in group B, you would use PORTB, and a pin in group C, you would use PORTC, and so on.

Reading Data

When a pin is set to input you can read data from it, and do things like reading data from a sensor, a button being pushed, a switch being turned on/off, and more.
Much the same as with sending data and changing pin state, you control everything on the group level however for reading data you use PIN#. So for a pin in group B, you would use PINB and a pin in group C you would use PINC.

Blinking an LED

That’s enough theory for today, let’s just make something work! Connect the long leg of an LED to PB1 and the short leg to GND, and use the following code. If you are using the Space Buddies board, this is already hooked up and you should see the green LED on the right begin to blink.

```
// Blink an LED
void main() {
 // Set pin 1 on PORT B to OUTPUT
 DDRB |= (1 << PB1);
 // Begin an infinite loop
 while(1) {
   // Invert pin 1 on PORT B
   PORTB ^= (1 << PB1);
   // Do nothing for half a second, this causes the blinking.
   _delay_ms(500);
 }
}
```

Change the pin, use more than 1 LED, change the delay, play around and see what happens!

Reading when a button is pushed

Much the same with the blinking LED example, keep the LED attached to PB1. Now attach the positive side of an IR LED to PC5, and the other side to GND. Run this code, point a television remote control at the IR LED and press any button. You will notice that each time you press the button, the LED will change its state. If it was on, it will turn off and vice versa.

```
void main()
{
 // Set pin 1 on PORT B to OUTPUT
 DDRD |= (1 << PB1);
 // SET pin 2 on PORTB to INPUT
 DDRC &= ~(1 << PC5);
 // Set pin 2 on PORTB to HIGH it goes LOW when receiving
signal
 PORTC |= (1 << PC5);
 // Begin an infinite Loop
 while(1) {
   // Check if pin 5 on PORT C is receiving a signal. 1
(true) if it is, and 0 (false) if it is not.
   if (!(PINC & (1 << PC5))) {
     // Invert pin 1 on PORT B
     PORTB ^= (1 << PB1);
     // Do nothing for half a second
     //   this is for "debouncing" the input.
     _delay_ms(500);
   }
 }
}
```

Reading data from the EEPROM

When you have large amounts of data like the tunes used with the Space Buddies, you need to store them in the EEPROM. This has a lot more space for storing tunes, however you do not read the data back out the same as any regular variable. Below is an example of how you can initialise data inside of the EEPROM and how to read it back out.

```
// Create an array that is 10 long
//   Initialise it with 10 numbers
//   EEMEM changes this to a pointer in memory in the EEPROM
//   All initialised data will be saved in EEPROM
//   You cannot read this like a regular variable
uint8_t data[10] EEMEM = {
 1, 2, 3, 4, 5, 6, 7, 8, 9, 0
};

void main()
{
 // Create a constant pointer to the EEPROM
 const uint8_t *ptr = data;
 // Create a loop the same size as the array
 for (uint8_t i = 0; i < 10; i++)
 {
   // Read the current byte
   uint8_t value = eeprom_read_byte(ptr++);
 }
}
```

Writing data to the EEPROM

When you have a lot of data which you cannot keep in memory because you need the space for something else, but you do not want to throw it away, you can store it inside the EEPROM. Below is some code showing you how to change a value stored inside the EEPROM from 5 to 100. You can use this same method to store many different types of data you collect.

```
// Create an array that is 10 long
//   Initialise it with 10 numbers
//   EEMEM changes this to a pointer in memory in the EEPROM
//   All initialised data will be saved in EEPROM
//   You cannot read this like a regular variable
uint8_t data EEMEM = 5;

// This code will set every value in the array to 100
void main()
{
 // Create a constant pointer to the EEPROM
 const uint8_t *ptr = data;
 // Set the current byte to 100
 eeprom_update_byte(ptr, 100);
}
```

Space Buddies

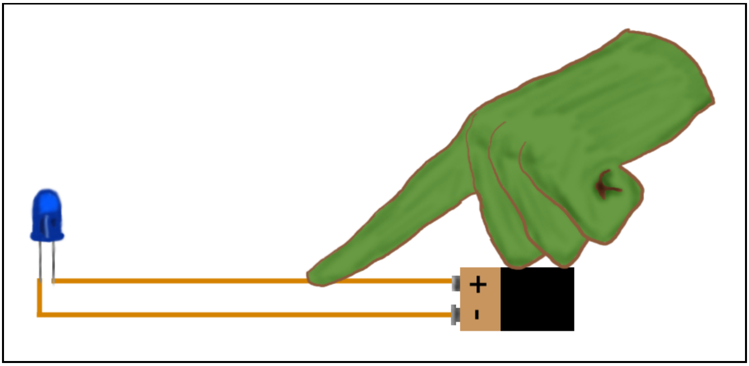
State machine

The Space Buddies code makes use of a programming concept called a state machine. It's an extremely simple concept where you have a variable, and depending on what your application is meant to be doing, you set the variable accordingly. This is a very simple way for you to track what is happening at any point in time. For example, the Space Buddies code has five states which are as follows.

0	The program is currently waiting for buttons to be pressed
1	The program is attempting to read infra-red data
2	The program is sending infrared data
3	The program is saving a received tune into the EEPROM
4	The program is playing a selected tune

Capacitive buttons

These are very special buttons on the Space Buddies which instead of using extra components to behave as buttons, they are actually making use of human body capacitance as the input. What is capacitance though? This is a name given to any object which has the ability to store an electric charge, or more simply, is capable of having an electric charge passed through it. So these buttons actually have two disconnected circuit traces. One is constantly transmitting a small electrical charge and the other is constantly waiting for a charge to be received. These traces are small enough that you cannot help but to touch both of them at the same time. When you touch them, the small electrical charge flows into your finger and back out again onto the circuit waiting for a charge to be received. Now due to the number of different things which makes up the human body, there is a small amount of resistance so we cater for that with our code by checking for the signal seventeen times within a very short period of time, and as long as we see a signal at least twice we know somebody has pressed the button. The number two is just a number obtained by trial and error and holds no scientific reasoning behind it.



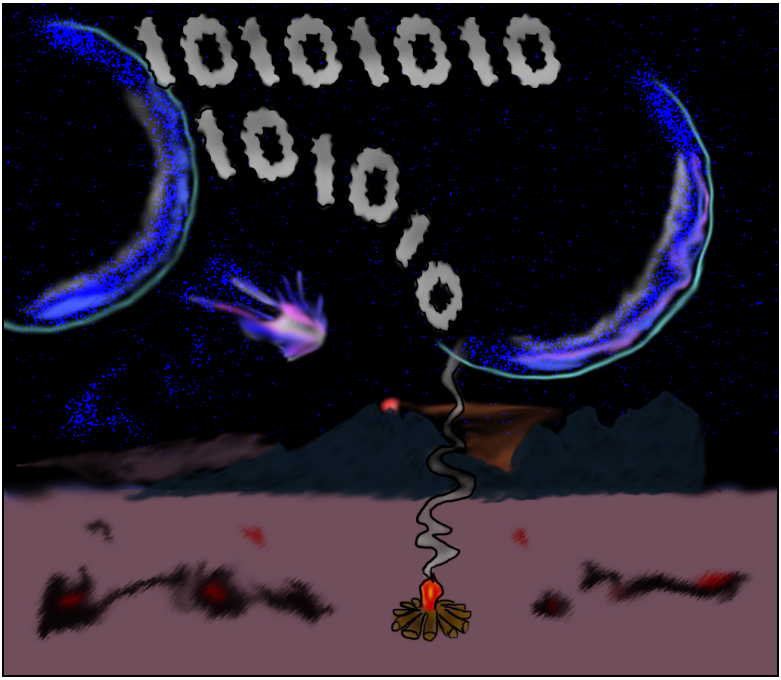
Timers and Interrupts

On every microcontroller you can make use of timers, however the number of times and the types of timers is of course different on each microcontroller. However the most common one across them all is the 8-bit timer, and this is basically a timer that is constantly counting from 0 to 255, and each time it starts again it will call a function. You can also create an interrupt to call another function every time the counter hits a number of your choosing, such as 50. You can also create input based interrupts which basically call yet another function when the value of a pin changes - this is most commonly used to detect when a button is pressed. We make use of timers to basically keep track of time, and ensure parts of our code run at the correct times if we need them to. If you are interested in learning more about timers, visit the Space Buddies website.

Software PWM

PWM stands for Pulse Wave Modulation which basically means the ability to turn a signal on and off very quickly. So with electronics, if you ever see an LED that can go dimmer and/or brighter, this is most probably PWM. LEDs can only be on, or off and there are only two ways to dim them. The first would be to add resistance, however all LEDs have a minimum power requirement so if your resistance exceeds this, the LED will simply not turn on. The other way to dim one is to make use of PWM. Imagine being able to turn an LED on and off repeatedly 255 times a second. Now imagine only turning it on for the first 100 times, and leaving it off the remaining 155 times, but repeating that for every single second that may pass. Your eyes will see it as a dimmed out LED, when in fact it is just turning on and off very quickly. Now if you increase how many times it is on, the LED appears brighter, and the inverse is also true for when you decrease the number of times it is on, the LED appears dimmer. Since you are always providing it with all the power it asks for, you will never hit the problem that you would if you were to add resistance instead.

There is one drawback to PWM though, and that is it requires a lot of processing power to handle doing this at such a high speed. Fortunately you often get a number of pins on every microcontroller that have the capability to handle PWM for you automatically and you need only tell it a value between 0 and 255, and it does all the work for you. Unfortunately each microcontroller is different and you will often not have enough of these special pins to control every LED you want, depending on how many you need to control. This is where Software PWM comes in, and we handle all of the flashing on and off ourselves with the clever usage of timers. This will slow down the rest of our code a little bit when we are running this timer, but the difference is almost unnoticeable and the effect is worth it.



Data structure of a tune

With the Space Buddies project we have a very limited amount of space to store tunes. Additionally, we want to wirelessly transmit them over infrared so the smaller they are in memory size, the better it is for us. To shrink everything, we have taken musical notes from both minor and major along with a few common durations.

We have the following notes

Minors
C, C#, D, E flat, E, F, F#, G, A flat, A , B flat, B
Majors
C, C#, D, E flat, E, F, F#, G, A flat, A , B flat, B

Now on top of these notes, we also need a representation of silence, so we store all of these inside of an enum which essentially saves them all as integers. The stored notes are translated as follows.

Code	Note	Major/Mino r	Integer Value	Binary Value
T_REST	Silence	N/A	0	0000 0000
T_C	C	Minor	1	0000 0001
T_CS	C#	Minor	2	0000 0010
T_D	D	Minor	3	0000 0011
T_EB	E Flat	Minor	4	0000 0100
T_E	E	Minor	5	0000 0101
T_F	F	Minor	6	0000 0110
T_FS	F#	Minor	7	0000 0111
T_G	G	Minor	8	0000 1000
T_AB	A Flat	Minor	9	0000 1001
T_A	A	Minor	10	0000 1010

T_BB	B Flat	Minor	11	0000 1011
T_B	B	Minor	12	0000 1100
T_CX	C	Major	13	0000 1101
T_CSX	C#	Major	14	0000 1110
T_DX	D	Major	15	0000 1111
T_EBX	E Flat	Major	16	0001 0000
T_EX	E	Major	17	0001 0001
T_FX	F	Major	18	0001 0010
T_FSX	F#	Major	19	0001 0011
T_GX	G	Major	20	0001 0100
T_ABX	A Flat	Major	21	0001 0101
T_AX	A	Major	22	0001 0110
T_BBX	B Flat	Major	23	0001 0111
T_BX	B	Major	24	0001 1000

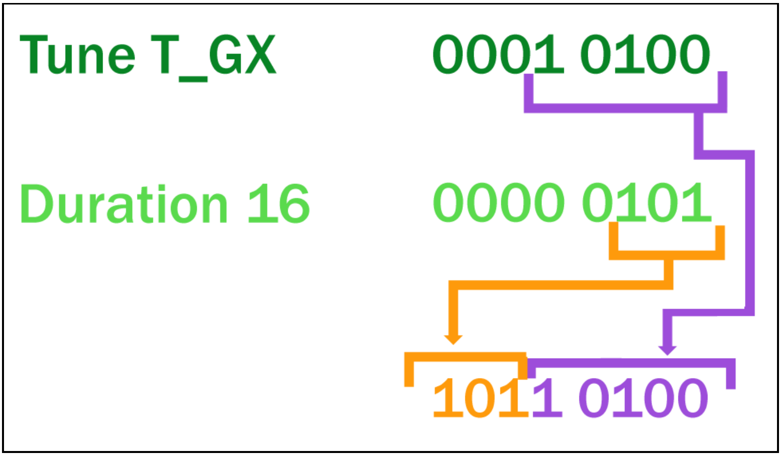
If you take a look at all the binary values, the first three digits are always a zero, meaning that they are never used for holding musical note information, which leaves us three bits for every byte to hold duration information. Three bits limits us to eight potential durations, so we have gone with the following.

2, 4, 6, 8, 12, 16, 18, 24

Now instead of looking at them as those actual numbers, let's look at them as if they were an array counting from 0, and look at the array index values as binary.

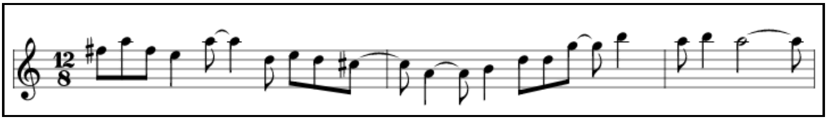
Duration	Index	Index as Binary
2	0	0000 0000
4	1	0000 0001
6	2	0000 0010
8	3	0000 0011
12	4	0000 0100
16	5	0000 0101
18	6	0000 0110
24	7	0000 0111

Again look at the binary values: we only make use of the far right three bits for each byte, so let us take a look at how we can stick these together with the tunes within the same byte to save up on the amount of space used.



To make everything easier, there is a macro called NOTE() which takes in a note and a duration index. This stores the data into the correct areas of the byte as seen in the above diagrams.
This is how we store the data for each tune. We still have allowance to add more notes, however with the way music works, we have all the notes we could ever need and the only thing limiting us is our range of durations to hold each note. Therefore we need to get a little creative when translating a musical score into tune data. There is one thing we can be absolutely certain of though, and that is with our current data definitions, we will never hold a byte that is all ones or more specifically in value 255. This gives us the benefit of using this as a marker to state where a tune may end when transmitting it wirelessly. We will get into this with more detail in an upcoming chapter.

Let us look into an example of how we converted a musical score into data. We begin with a music score as seen below.



Now if we change the above score into written out notation, we get the following:

F# with duration of 5	A with duration of 5	F with duration of 5	E with duration of 5
A with duration of 5	A with duration of 5	D with duration of 5	E with duration of 5
D with duration of 5	C# with duration of 5	C with duration of 5	A with duration of 4
A with duration of 4	B with duration of 4	D with duration of 5	D with duration of 5
G with duration of 5	G with duration of 5	B with duration of 5	A with duration of 5
B with duration of 5	A with duration of 5	A with duration of 5	

After we convert this by hand into notes and durations that match what we have available we have this:

T_FSX with 8	T_AX with 8	T_FSX with 8	T_EX with 16
T_AX with 16	T_DX with 8	T_EX with 8	T_CSX with 16
T_A with 16	T_B with 16	T_DX with 8	T_DX with 8
T_GX with 16	T_BX with 16	T_AX with 8	T_BX with 16
T_AX with 7			

Once more time, let's convert the above into how we plan on storing it inside of our application.

```
NOTE(T_FSX, 3), NOTE(T_AX, 3), NOTE(T_FSX, 3), NOTE(T_EX, 5),
NOTE(T_AX, 5),
NOTE(T_DX, 3), NOTE(T_EX, 3), NOTE(T_EX, 3), NOTE(T_CSX, 5),
NOTE(T_A, 5),
NOTE(T_B, 5), NOTE(T_DX, 3), NOTE(T_DX, 3), NOTE(T_GX, 5),
NOTE(T_BX, 5),
NOTE(T_AX, 3), NOTE(T_BX, 5), NOTE(T_AX, 7), END_MARKER
```

We now have a complete tune stored within 18 bytes of data, plus an extra one for the end marker.

Playing a tune

Playing a tune is much simpler than creating one, so by reaching this chapter you can rest assured that the toughest part is over. When playing a sound it may not be obvious but the sound is very similar to PWM. You do not send data to a speaker, but in fact turn a speaker on and off at a high speed. The longer you keep the speaker on, the higher the pitch that comes out, so we need to convert our musical notes into amounts of time that we should keep the speaker on to create each note. Below is a table that helps with that.

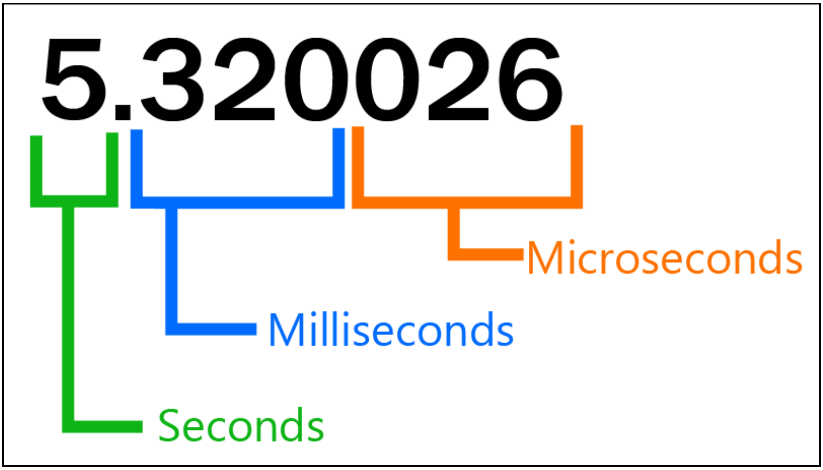
Note	Enum	Value
Minor C	T_C	1911
Minor C#	T_CS	1804
Minor D	T_D	1703
Minor E Flat	T_EB	1607
Minor E	T_E	1517
Minor F	T_F	1432
Minor F#	T_FS	1352
Minor G	T_G	1276
Minor A Flat	T_AB	1204
Minor A	T_A	1136
Minor B Flat	T_BB	1073
Minor B	T_B	1012

Note	Enum	Value
Major C	T_CX	955
Major C#	T_CSX	902
Major D	T_DX	851
Major E Flat	T_EBX	803
Major E	T_EX	758
Major F	T_FX	716
Major F#	T_FSX	676
Major G	T_GX	638
Major A Flat	T_ABX	602
Major A	T_AX	568
Major B Flat	T_BBX	536
Major B	T_BX	506

With all music though you need to have a speed at which it plays, and with Space Buddies we configure that with the variable named vel, which is set to 10,000. By changing this value you can change the speed at which the tune plays.

Transmitting a tune over infrared

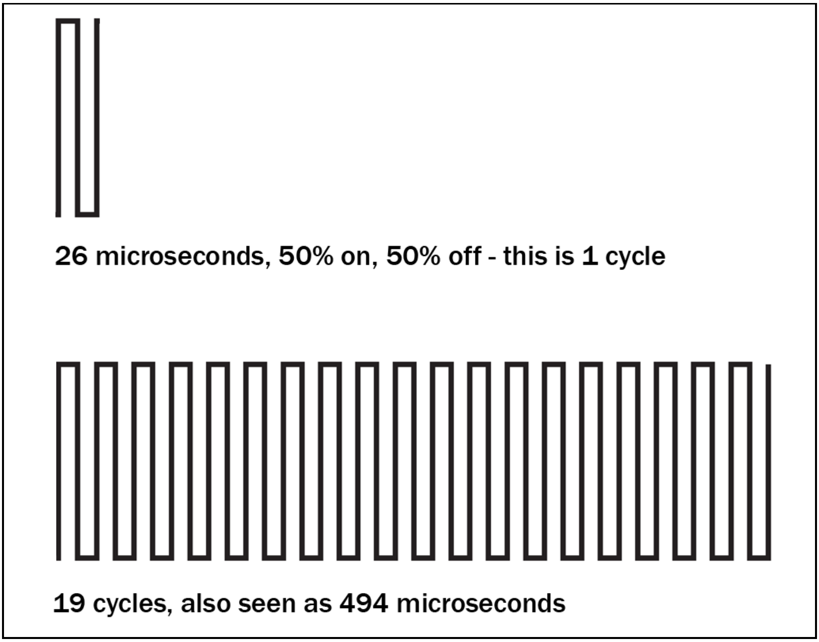
The infrared receiver used with the Space Buddies kit operates at 38KHz, but people just starting out don't really know what that means. 1Hz means one operation per second and as you increase the Hz, you increase how many operations occur per second. Now with that said, 1KHz is the same as saying 1,000Hz which means 1,000 operations per second. Now that we know the infrared receiver works at 38KHz, we know that unless we transmit data at the same speed, it will not work. A little more math is needed: we need to do 1 second divided by 38,000 which gives us 0.00002631578



Now we know that we need to operate at 26 microseconds, which is extremely fast and this is now one of the few times we actually care about how fast it takes to turn an LED on or off. So to operate at 26 microseconds, we need to turn our infrared LED on for 13 microseconds, and off for 13 microseconds.

We need to send binary data wirelessly and rather than trying to come up with a complex system we can use one that has been around for many years. The first that comes to mind is morse code, where you have a long pulse, a short pulse, and no pulse at all. We can use this. Our data is binary, so let us send a long pulse for every 1, a short pulse for every 0, and a break between each of those.

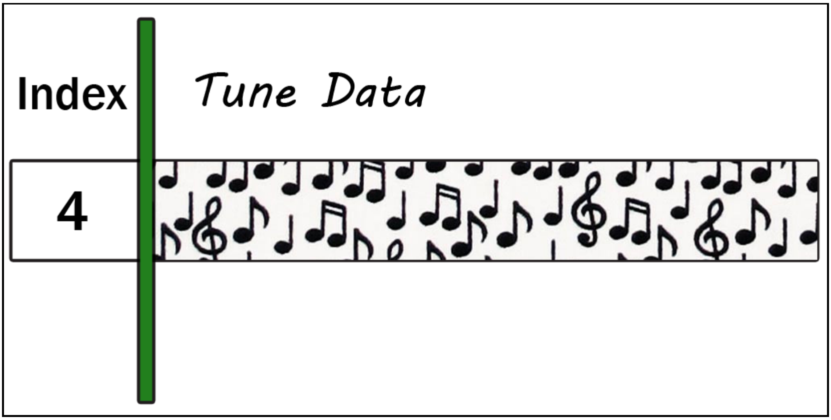
Sending data this way can be a little troublesome because there's a chance that the receiver will not notice the signal straight away, so it's safest to use nice big numbers. With Space Buddies we are sending 220 cycles for binary value of one, and 100 cycles for binary value of zero, with a gap of 250 microseconds between each signal.



26 microseconds, 50% on, 50% off - this is 1 cycle

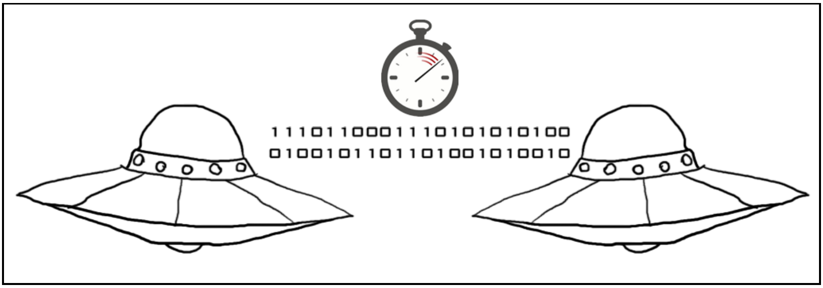
19 cycles, also seen as 494 microseconds

The only thing we need to add to our transmissions is information about which tune we are sending. There are 10 tunes in total, so for the first byte we send a number which represents the tune number we are sending.

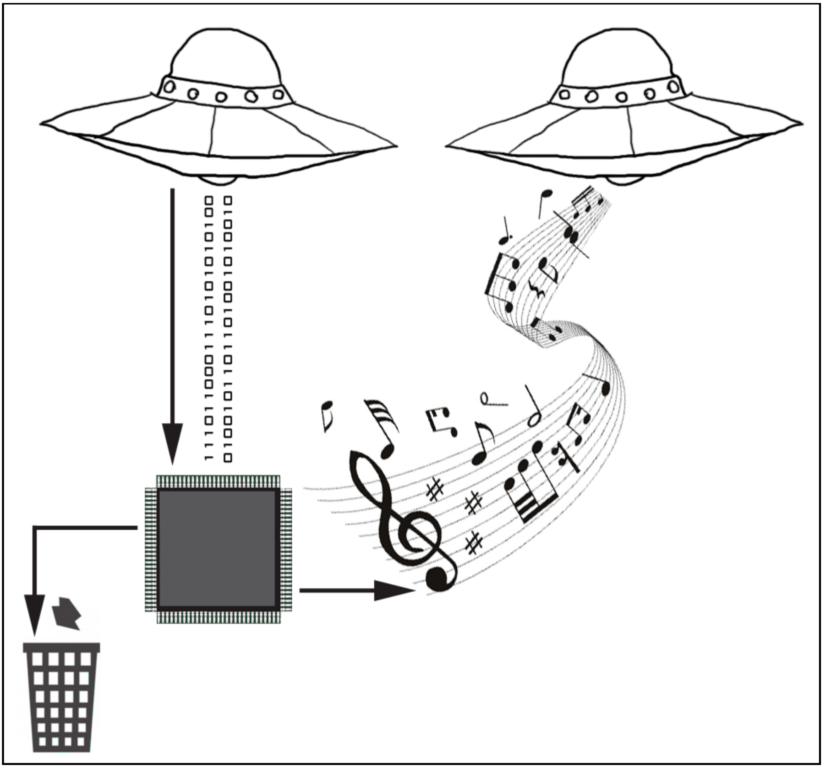


Receiving a tune over infra-red

Receiving a tune over infrared is a little more complicated than sending one as far as the actual code is concerned, but the logic behind it is pretty simple. We start by counting how many cycles the signal is high and the same for how many cycles it is low. Now if either of those exceed a number that we decide upon as the maximum allowable, we can stop recording data and take a look at what we have received.



When we begin to examine the data, the very first thing we check is to see that we have received full bytes of data. If we are even a single bit short of a byte, the data is invalid and we do nothing with it, otherwise we just store it.



Reading and writing EEPROM

Reading and writing data into the EEPROM can be a little tricky compared to regular memory at first, but once you understand how to work with it, it's not all that bad. To begin, you need to declare some variables of the size you need to take up in the EEPROM but make sure to decorate them with EEMEM. This will tell the compiler that your variable is in fact a pointer to a location within the EEPROM, and if you initialise it with any values, when you compile it you will get a special file to burn to the microcontroller that inserts those values directly into the EEPROM.

```
uint8_t tune1[50] EEMEM = {
  NOTE(T_FSX, 3), NOTE(T_AX, 3), NOTE(T_FSX, 3), NOTE(T_EX,
5), NOTE(T_AX, 5),
  NOTE(T_DX, 3), NOTE(T_EX, 3), NOTE(T_EX, 3), NOTE(T_CSX,
5), NOTE(T_A, 5),
  NOTE(T_B, 5), NOTE(T_DX, 3), NOTE(T_DX, 3), NOTE(T_GX, 5),
NOTE(T_BX, 5),
  NOTE(T_AX, 3), NOTE(T_BX, 5), NOTE(T_AX, 7), END_MARKER
};
```

When you are reading from these EEPROM pointers, you need to take into account that it is slower than regular variables, and that you are only able to read a single byte at a time. This is done with the eeprom_read_byte() function, and normally what you would do is populate a local variable one byte at a time with everything you need.

```
for(uint8_t data = eeprom_read_byte(ptr++); data !=
END_MARKER; data = eeprom_read_byte(ptr++))
{
  buffer[i++] = data;
}
```

When you are writing to these EEPROM pointers, the same rules apply that you have when you are reading, including the limit of a single byte at a time. This is done with the eeprom_update_byte() function.

```
for (uint8_t i = 1; i < 50; i++)
{
  eeprom_update_byte(ptr++, buffer[i]);
}
```

Credits

I would like to thank the following people for helping me put this book together, without them I may never have finished as quickly as I had, nor would it be half as readable as it is right now.

Alessandra Williams-Belotti
http://thetechnofringe.blogspot.co.uk/
Alessandra proof read this booklet a good number of times, ensuring that my ramblings actually made sense, and that I provided diagrams wherever they were needed. Without her help, this booklet would probably be completely unreadable by anybody other than myself, with a number of useful diagrams missing!

Jason Hotchkiss
http://hotchk155.blogspot.co.uk/
Jason taught me how to do surface mount soldering efficiently, without paste, and introduced me into the world of macros in AVR C. His projects have always inspired me, and I feel that without his help and inspiration, I may never have even started this project.

Chris Holden
http://nerdclub-uk.blogspot.co.uk/
Chris was my original mentor when it came to electronics, and whenever I had a question he was always more than happy to help explain things to me as long as there was coffee and chocolate hobnobs.

Steve Carpenter
https://plus.google.com/+SteveCarpenter1/
Steve is a successful toy maker, having invented a number of great products that are on sale in stores around the globe. He always seems to find time out of his busy schedule to help others with all aspects of their projects, regardless of what they may be. With Space Buddies he helped me turn my vague idea of a PCB shape and layout into something that just looks great and works.

Adam Brunette
http://www.adambrunette.com
Adam has helped me with every electronic project I have started, including all the projects I never finished. His endless patience and experience is a great inspiration to me.

Andrew Male
https://twitter.com/AndyM84
Andy always said that I should do a project like this, and offered me some great suggestions on structuring this book.

Jari Tulilahti
http://www.facebook.com/jartza
Jari helped me gain a much better understanding of timers, and to debug some issues I had run into with the infra-red code not always working which turned out to be a limitation of the hardware I was using.

Nick Johnson
http://www.arachnidlabs.com/
Nick showed me a great way to structure my data storage and offered ideas to improve my code. He is an absolute wealth of knowledge and always happy to provide help when asked.

Alexander Hosford
http://www.alexanderhosford.co.uk/
Alexander is my musically talented friend who I always go to when I have questions about anything to do with music and audio. I provided him with the details of what notes I had, and he helped me with the music to data conversion chapter, along with making up the tunes used in Space Buddies.

Glossary

AC	Alternating Current - a power source that constantly switches polarity, so it appears that it is 50% of the time on, and 50% of the time off
DC	Direct Current - a power source, such as a battery which is always on, unlike AC
MCU	Microcontroller unit - a small computer on a single integrated circuit.
Duty Cycle	A duty cycle is the percentage of time that a signal is on. AC for example is 50% duty cycle, whilst DC is 100% duty cycle.
PWM	Pulse Wave Modulation - this is the name given for obtaining analog results with digital means, or more specifically to be able to turn a signal on and off rapidly. The Space Buddies use this to play music with the DC power source.
PIC	A brand of microcontrollers by a company called Microchip.
AVR	A brand of microcontrollers by a company called Atmel.
EEPROM	Electrically Erasable Programmable Read-Only Memory - used by microcontrollers to store small amounts of data that must be saved when power is turned off.
Debouncing	The name given to a term for preventing calling a function more than once within a short period of time.
uS	The shorthand for microseconds
ms	The shorthand for milliseconds