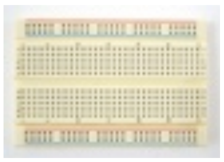
Project 1

LED Flasher

Project 1 - LED Flasher

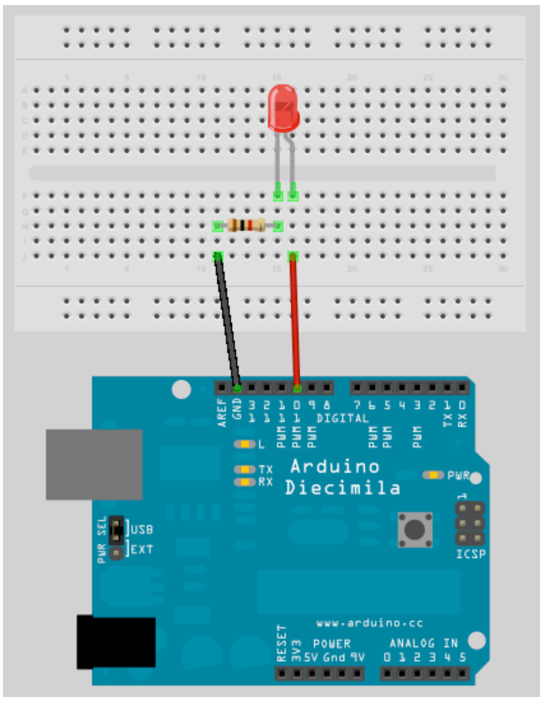
In this project we are going to repeat what we did in setting up and testing the Arduino, that is to blink an LED. However, this time we are going to use one of the LED's in the kit and you will also learn about some electronics and coding in C along the way.

What you will need

Breadboard	
Red LED	
150Ω Resistor	
Jumper Wires	

Connect it up

Now, first make sure that your Arduino is powered off. You can do this either by unplugging the USB cable or by taking out the Power Selector Jumper on the Arduino board. Then connect everything up like this :-



It doesn't matter if you use different coloured wires or use different holes on the breadboard as long as the components and wires are connected in the same order as the picture. Be careful when insterting components into the Breadboard. The Breadboard is brand new and the grips in the holes will be stiff to begin with. Failure to insert components carefully could result in damage.

Make sure that your LED is connected the right way with the longer leg connected to Digital Pin 10. The long led is the Anode of the LED and always must go to the +5v supply (in this case coming out of Digital Pin 10) and the short leg is the Cathode and must go to Gnd (Ground).

When you are happy that everything is connected up correctly, power up your Arduino and connect the USB cable.

Enter the code

Now, open up the Arduino IDE and type in the following code :-

```
// Project 1 - LED Flasher

int ledPin = 10;

void setup() {
        pinMode(ledPin, OUTPUT);
}

void loop() {
        digitalWrite(ledPin, HIGH);
        delay(1000);
        digitalWrite(ledPin, LOW);
        delay(1000);
}
```

Now press the Verify/Compile button at the top of the IDE to make sure there are no errors in your code. If this is successful you can now click the Upload button to upload the code to your Arduino.

If you have done everything right you should now see the Red LED on the breadboard flashing on and off every second.

Now let's take a look at the code and the hardware and find out how they both work.

Project 1 - Code Overview

```
// Project 1 - LED Flasher

int ledPin = 10;

void setup() {
        pinMode(ledPin, OUTPUT);
}

void loop() {
        digitalWrite(ledPin, HIGH);
        delay(1000);
        digitalWrite(ledPin, LOW);
        delay(1000);
}
```

So let's take a look at the code for this project. Our first line is

```
// Project 1 - LED Flasher
```

This is simply a comment in your code and is ignored by the compiler (the part of the IDE that turns your code into instructions the Arduino can understand before uploading it). Any text entered behind a // command will be ignored by the compiler and is simply there for you, or anyone else that reads your code. Comments are essential in your code to help you understand what is going on and how your code works. Comments can also be put after commands as in the next line of the program.

Later on as your projects get more complex and your code expands into hundreds or maybe thousands of lines, comments will be vital in making it easy for you to see how it works. You may come up with an amazing piece of code, but if you go back and look at that code days, weeks or months alter, you may forget how it all works. Comments will help you understand it easily. Also, if your code is meant to be seen by other people (and as the whole ethos of the Arduino, and indeed the whole Open Source community is to share code and schematics. We hope when you start making your own cool stuff with the Arduino you will be willing to share it with the world) then comments will enable that person to understand what is going on in your code.

You can also put comments into a block statement by using the /* and */ commands. E.g.

```
/* All of the text within
the slash and the asterisks
is a comment and will be
ignored by the compiler */
```

The IDE will automatically turn the colour of any commented text to grey.

The next line of the program is

```
int ledPin = 10;
```

This is what is know as a variable. A variable is a place to store data. In this case you are setting up a variable of type int or integer. An integer is a number within the range of -32,768 to 32,767. Next you have assigned that integer the name of ledPin and have given it a value of 10. We didn't have to call it ledPin, we could have called it anything we wanted to. But, as we want our variable name to be descriptive we call it ledPin to show that the use of this variable is to set which pin on the Arduino we are going to use to connect our LED. In this case we are using Digital Pin 10. At the end of this statement is a semi-colon. This is a symbol to tell the compiler that this statement is now complete.

Although we can call our variables anything we want, every variable name in C must start with a letter, the rest of the name can consist of letters, numbers and underscore characters. C recognises upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names. Keywords are constants, variables and function names that are defined as part of the Arduino language. Don't use a variable name that is the same as a keyword. All keywords within the sketch will appear in red.

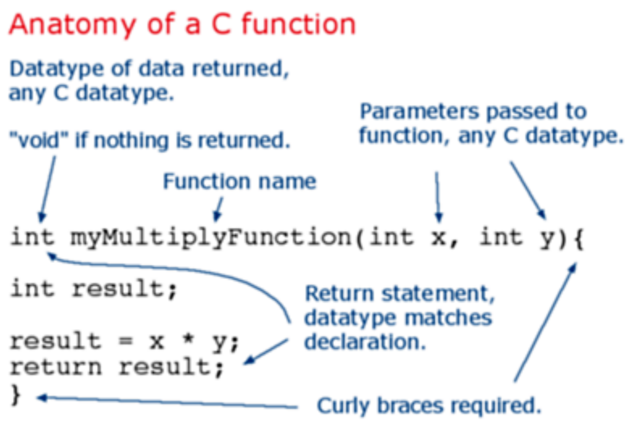
So, you have set up an area in memory to store a number of type integer and have stored in that area the number 10. Imagine a variable as a small box where you can keep things. A variable is called a variable because you can change it. Later on we will carryout mathematical calculations on variables to make our program do more advanced stuff.

Next we have our setup() function

```
void setup() {
        pinMode(ledPin, OUTPUT);
}
```

An Arduino sketch must have a setup() and loop() function otherwise it will not work. The setup() function is run once and once only at the start of the program and is where you will issue general instructions to prepare the program before the main loop runs, such as setting up pin modes, setting serial baud rates, etc.

Basically a function is a block of code assembled into one convenient block. For example, if we created our own function to carry out a whole series of complicated mathematics that had many lines of code, we could run that code as many times as we liked simply by calling the function name instead of writing


Anatomy of a C function
Datatype of data returned, any C datatype.
"void" if nothing is returned.
Function name
Parameters passed to function, any C datatype.
int myMultiplyFunction(int x, int y){
int result;
result = x * y;
return result;
}
Return statement, datatype matches declaration.
Curly braces required.

out the code again each time. Later on we will go into functions in more detail when we start to create our own.
In the case of our program the setup() function only has one statement to carry out. The function starts with

```
void setup()
```

and here we are telling the compiler that our function is called setup, that it returns no data (void) and that we pass no parameters to it (empty parenthesis). If our function returned an integer value and we also had integer values to pass to it (e.g. for the function to process) then it would look something like this

```
int myFunc(int x, int y)
```

In this case we have created a function (or a block of code) called myFunc. This function has been passed two integers called X and Y. Once the function has finished it will then return an integer value to the point after where our function was called in the program (hence `int` before the function name).

All of the code within the function is contained within the curly braces. A { symbol starts the block of code and a } symbol ends the block. Anything in between those two symbols is code that belongs to the function.

We will go into greater detail about functions later on so don't worry about them for now. All you need to know is that in this program, we have two functions, the first function is called setup and it's purpose is to setup anything necessary for our program to work before the main program loop runs.

```
void setup() {
      pinMode(ledPin, OUTPUT);
}
```

Our setup function only has one statement and that is pinMode. Here we are telling the Arduino that we want to set the mode of one of our digital pins to be Output mode, rather than Input. Within the parenthesis we put the pin number and the mode (OUTPUT or INPUT). Our pin number is ledPin, which has been previously set to the value 10 in our program. Therefore, this statement is simply telling the Arduino that the Digital Pin 10 is to be set to OUTPUT mode.

As the setup() function runs only once, we now move onto the main function loop.

```
void loop() {
      digitalWrite(ledPin, HIGH);
      delay(1000);
      digitalWrite(ledPin, LOW);
      delay(1000);
}
```

The loop() function is the main program function and runs continuously as long as our Arduino is turned on. Every statement within the loop() function (within the curly braces) is carried out, one by one, step by step, until the bottom of the function is reached, then the loop starts again at the top of the function, and so on forever or until you turn the Arduino off or press the Reset switch.

In this project we want the LED to turn on, stay on for one second, turn off and remain off for one second, and then repeat. Therefore, the commands to tell the Arduino to do that are contained within the loop() function as we wish them to repeat over and over.

The first statement is

```
digitalWrite(ledPin, HIGH);
```

and this writes a HIGH or a LOW value to the digital pin within the statement (in this case ledPin, which is Digital Pin 10). When you set a digital pin to HIGH you are sending out 5 volts to that pin. When you set it to LOW the pin becomes 0 volts, or Ground.

This statement therefore sends out 5v to digital pin 10 and turns the LED on.

After that is

```
delay(1000);
```

and this statement simply tells the Arduino to wait for 1000 milliseconds (to 1 second as there are 1000 milliseconds in a second) before carrying out the next statement which is

```
digitalWrite(ledPin, LOW);
```

which will turn off the power going to digital pin 10 and therefore turn the LED off. There is then another delay statement for another 1000 milliseconds and then the function ends. However, as this is our main loop() function, the function will now start again at the beginning. By following the program structure step by step again we can see that it is very simple.

```
// Project 1 - LED Flasher

int ledPin = 10;

void setup() {
      pinMode(ledPin, OUTPUT);
}

void loop() {
      digitalWrite(ledPin, HIGH);
      delay(1000);
      digitalWrite(ledPin, LOW);
      delay(1000);
}
```

We start off by assigning a variable called ledPin, giving that variable a value of 10.

Then we move onto the setup() function where we simply set the mode for digital pin 10 as an output.

In the main program loop we set Digital Pin 10 to high, sending out 5v. Then we wait for a second and then turn off the 5v to Pin 10, before waiting another second. The loop then starts again at the beginning and the LED will therefore turn on and off continuously for as long as the Arduino has power.

Now that you know this you can modify the code to turn the LED on for a different period of time and also turn it off for a different time period.

For example, if we wanted the LED to stay on for 2 seconds, then go off for half a second we could do this:-

```
void loop() {
      digitalWrite(ledPin, HIGH);
      delay(2000);
      digitalWrite(ledPin, LOW);
      delay(500);
}
```

or maybe you would like the LED to stay off for 5 seconds and then flash briefly (250ms), like the LED indicator on a car alarm then you could do this:-

```
void loop() {
      digitalWrite(ledPin, HIGH);
      delay(250);
      digitalWrite(ledPin, LOW);
      delay(5000);
}
```

or make the LED flash on and off very fast

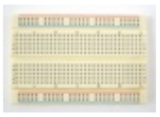
```
void loop() {
      digitalWrite(ledPin, HIGH);
      delay(50);
      digitalWrite(ledPin, LOW);
      delay(50);
}
```

By varying the on and off times of the LED you create any effect you want. Well, within the bounds of a single LED going on and off that is.

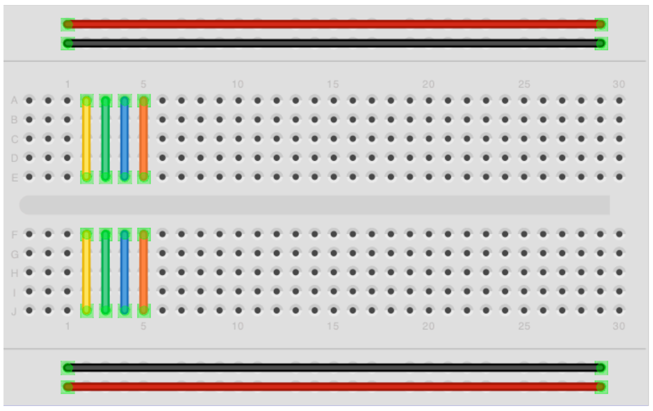
Before we move onto something a little more exciting let’s take a look at the hardware and see how it works.

Project 1 - Hardware Overview

The hardware used for this project was :-

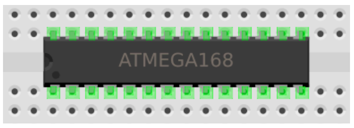
Breadboard	
Red LED	
150Ω Resistor	
Jumper Wires	

The breadboard is a reusable solderless device used generally to prototype an electronic circuit or for experimenting with circuit designs. The board consists of a series of holes in a grid and underneath the board these holes are connected by a strip of conductive metal. The way those strips are laid out is typically something like this:-



The strips along the top and bottom run parallel to the board and are design to carry your power rail and your ground rail. The components in the middle of the board can then conveniently connect to either 5v (or whatever voltage you are using) and Ground. Some breadboards have a red and a black line running parallel to these holes to show which is power (Red) and which is Ground (Black). On larger breadboards the power rail sometimes has a split, indicated by a break in the red line. This is in case you want different voltages to go to different parts of your board. If you are using just one voltage a short piece of jumper wire can be placed across this gap to make sure that the same voltage is applied along the whole length of the rail

The strips in the centre run at 90 degrees to the power and ground rails in short lengths and there is a gap in the middle to allow you to put Integrated Circuits across the gap and have each pin of the chip go to a different set of holes and therefore a different rail.



The next component we have is a Resistor. A resistor is a device designed to cause 'resistance' to an electric current and therefore cause a drop in voltage across it's terminals. If you imagine a resistor to be like a water pipe that is a lot thinner than the pipe connected to it. As the water (the electric current) comes into the resistor, the pipe gets thinner and the current coming out of the other end is therefore reduced. We use resistors to decrease voltage or current to other devices. The value of resistance is known as an Ohm and it's symbol is a greek Omega symbol Ω.

In this case Digital Pin 10 is outputting 5 volts DC at (according to the Atmega datasheet) 40mA (milliamps) and our LED's require (according to their datasheet) a voltage of 2v and a current of 20mA. We therefore need to put in a resistor that will reduce the 5v to 2v and the current from 40mA to 20mA if we want to display the LED at it's maximum brightness. If we want the LED to be dimmer we could use a higher value of resistance.

To work out what resistor we need to do this we use what is called Ohm's law which is I = V/R where I is current, V is voltage and R is resistance. So to work out the resistance we arrange the formula to be R = V/I which is R = 3/0.02 which is 150 Ohms. V is 3 because we need the Voltage Drop, which is the supply voltage (5v) minus the Forward Voltage (2v) of the LED (found in the LED datasheet) which is 3v. We therefore need to find a 150Ω resistor. So how do we do that?

A resistor is too small to put writing onto that could be readable by most people so instead resistors use a colour code. Around the resistor you will typically find 4 coloured bands and by using the colour code in the chart on the next page you can find out the value of a resistor or what colour codes a particular resistance will be.

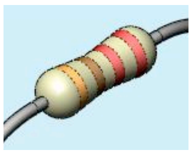
WARNING:
Always put a resistor (commonly known as a current limiting resistor) in series with an LED. If you fail to do this you will supply too much current to the LED and it could blow or damage your circuit.

Colour	1st Band	2nd Band	3rd Band (multiplier)	4th Band (tolerance)
Black	0	0	x10^0	
Brown	1	1	x10^1	±1%
Red	2	2	x10^2	±2%
Orange	3	3	x10^3	
Yellow	4	4	x10^4	
Green	5	5	x10^5	±0.5%
Blue	6	6	x10^6	±0.25%
Violet	7	7	x10^7	±0.1%
Grey	8	8	x10^8	±0.05%
White	9	9	x10^9	
Gold			x10^{-1}	±5%
Silver			x10^{-2}	±10%
None				±20%

We need a 150Ω resistor, so if we look at the colour table we see that we need 1 in the first band, which is Brown, followed by a 5 in the next band which is Green and we then need to multiply this by 10^1 (in other words add 1 zero) which is Brown in the 3rd band. The final band is irrelevant for our purposes as this is the tolerance. Our resistor has a gold band and therefore has a tolerance of ±5% which means the actual value of the resistor can vary between 142.5Ω and 157.5Ω. We therefore need a resistor with a Brown, Green, Brown, Gold colour band combination which looks like this:-

If we needed a 1K (or 1 kilo-ohm) resistor we would need a Brown, Black, Red combination (1, 0, +2 zeros). If we needed a 570K resistor the colours would be Green, Violet and Yellow.

In the same way, if you found a resistor and wanted to know what value it is you would do the same in reverse. So if you found this resistor and wanted to find out what value it was so you could store it away in your nicely labelled resistor storage box, we could look at the table to see it has a value of 220Ω.



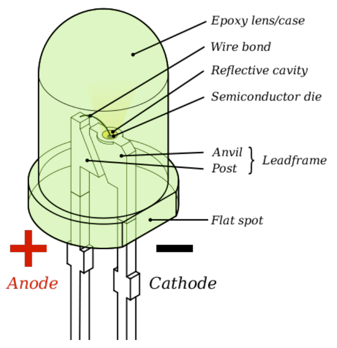
Our final component is an LED (I'm sure you can figure out what the jumper wires do for yourself), which stands for Light Emitting Diode. A Diode is a device that permits current to flow in only one direction. So, it is just like a valve in a water system, but in this case it is letting electrical current to go in one direction, but if the current tried to reverse and go back in the opposite direction the diode would stop it from doing so. Diodes can be useful to prevent someone from accidently connecting the Power and Ground to the wrong terminals in a circuit and damaging the components.

An LED is the same thing, but it also emits light. LED's come in all kinds of different colours and brightnesses and can also emit light in the ultraviolet and infrared part of the spectrum (like in the LED's in your TV remote control).

If you look carefully at the LED you will notice two things. One is that the legs are of different lengths and also that on one side of the LED, instead of it being cylindrical, it is flattened. These are indicators to show you which leg is the Anode (Positive) and which is the Cathode (Negative). The longer leg gets connected to the Positive Supply (3.3v) and the leg with the flattened side goes to Ground.

If you connect the LED the wrong way, it will not damage it (unless you put very high currents through it) and indeed you can make use of that 'feature' as we will see later on.

It is essential that you always put a resistor in series with the LED to ensure that the correct current gets to the LED. You can permanently damage the LED if you fail to do this.



As well as single colour resistors you can also obtain bi-colour and tri-colour LED's. These will have several legs coming out of them with one of them being common (i.e. Common anode or common cathode).

Supplied with your kit is an RGB LED, which is 3 LED's in a single package. An RGB LED has a Red, Green and a Blue (hence RGB) LED in one package. The LED has 4 legs, one will be a common anode or cathode, common to all 3 LED's and the other 3 will then go to the anode or cathode of the individual Red, Green and Blue LED's. By adjusting the brightness values of the R, G and B channels of the RGB LED you can get any colour you want. The same effect can be obtained if you used 3 separate red, green and blue LED's.

Now that you know how the components work and how the code in this project works, let's try something a bit more interesting.