Australian **Computing** Academy

DT Challenge Arduino

Sound

1. Hello, Arduino!
2. Using Sensors
3. Making decisions
4. Using numbers
5. Looping
6. Arrays
7. Project: Making Music

(https://creativecommons.org/licenses/by/4.0/)

The Australian Digital Technologies Challenges is an initiative of, and funded by the Australian Government Department of Education and Training (https://www.education.gov.au/).

© Australian Government Department of Education and Training.

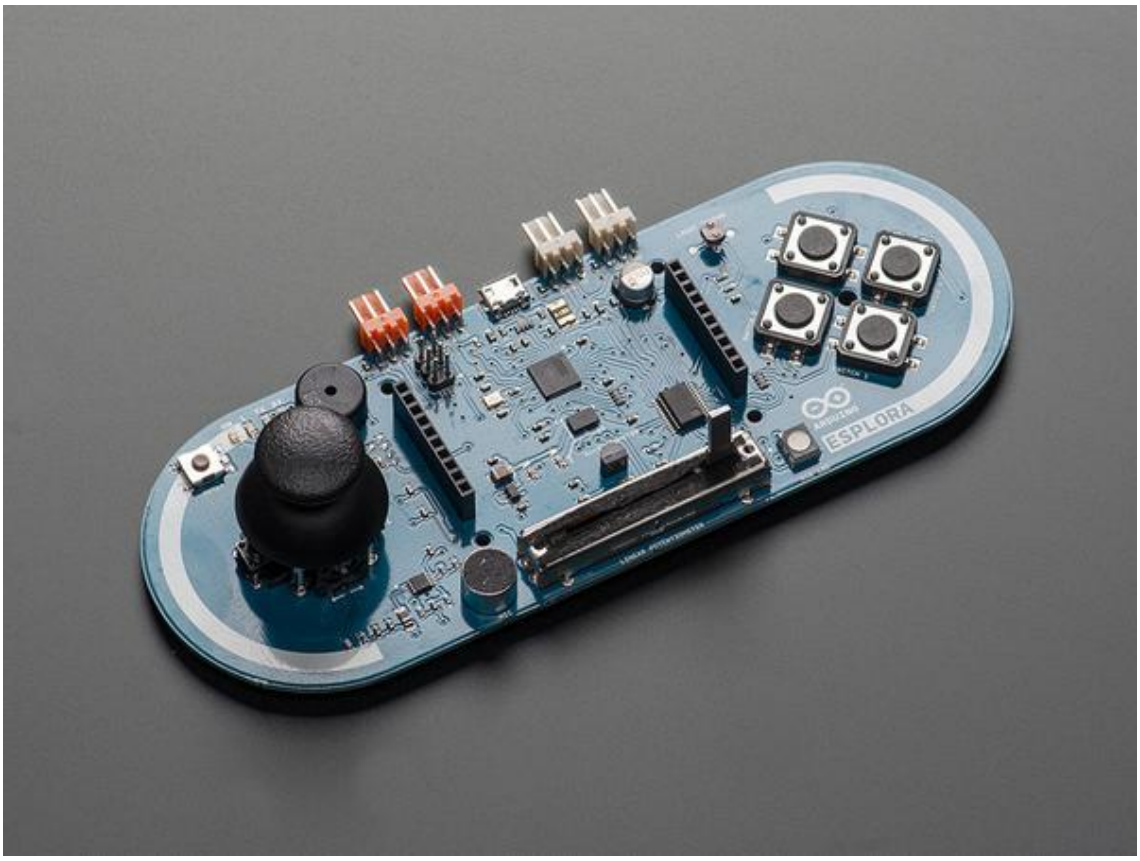
1

HELLO, ARDUINO!

1.1. Getting started

1.1.1. Arduino Esplora

In this course, we'll be the learning how to program the Arduino. The little computers inside the Arduino are used to run almost every device that you use today.



Arduino Esplora - image from Adafruit (https://www.adafruit.com/)

The type of Arduino we'll be using is called the Arduino Esplora, it has:

- **an RGB LED** (red, green and blue light emitting diode)
- **a joystick, with a joystick switch**
- **4 buttons**
- **a slider**
- **a temperature sensor**
- **a speaker**
- **a microphone**

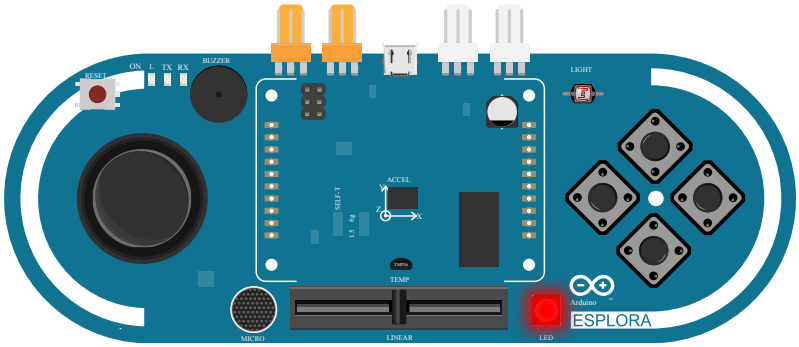
- **an accelerometer** (to tell which way the Esplora is facing)
- **a light sensor**
- **header pins** (to add a screen)
- **inputs and outputs** (at the top of the board)

> **If you don't have a real Esplora...**
>
> You can still do this course. It includes a full simulator, so you'll be able to do everything you'd do on a real Arduino Esplora!

1.1.2. First program

Let's run a simple program that turns on the LED! Click the ▶ button in the grey box below to run the code!

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 0);
}
```



Click on the ↰ button to swap the code back to the original. Click it again to swap back to your version.

There are a lot of separate parts to the above program, let's take a look at each one:

`#include` includes the Esplora library so we can access the things on the board.

A **setup**, a block of code that only runs *once*.

A **loop**, this block of code runs forever. Most of the code that we use will go in the loop.

Inside the loop is:

```
Esplora.writeRGB(255, 0, 0);
```

this turns on the red LED.

> **Blocks of code**
>
> Indenting the blocks of code that are surrounded by the `{` and `}` makes it easier to read them.

1.1.3. Control flow example

Let's take a look at how the Arduino runs!

The `setup` function runs once, then the `loop` function runs forever.

```
#include <Esplora.h>
void setup() {
  Serial.println("Welcome to Arduino :)");
  delay(2000);
}
void loop() {
  Serial.println("We're looping!");
  delay(2000);
}
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
Welcome to Arduino :)
We're looping!
```

No output is visible on the board. The `Serial.println` function prints all output to the Serial monitor.

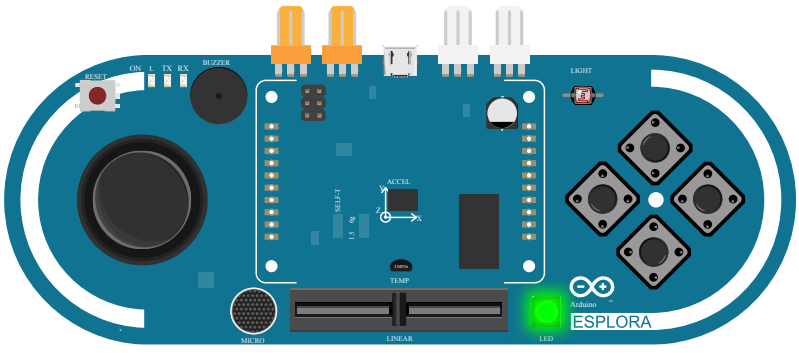
1.1.4. Setting the LED

Let's make some light!

Our little Esplora RGB LED consists of three tiny LEDs that we can control individually. To turn the RGB LED on, we use the `Esplora.writeRGB` function. It takes 3 numbers that go from 0 to 255. These numbers control the brightness of the red, green and blue LEDs. RGB stands for red, green, blue. We can make any colour by mixing these three colours.

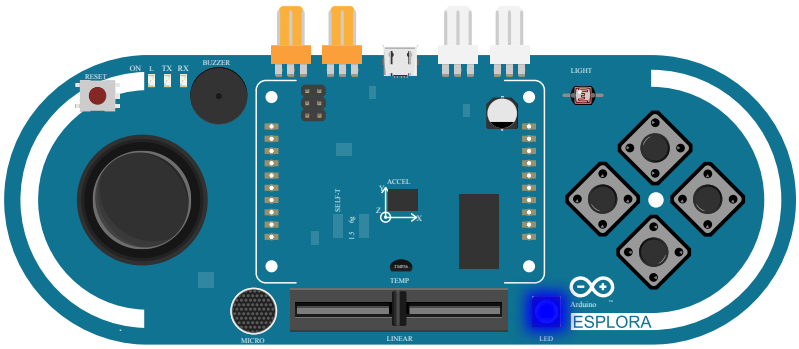
Turning on the green LED to full brightness would be done in the following way.

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(0, 255, 0);
}
```



Because we're only setting the colour *once*, the line can go inside the `setup` instead of the `loop`, because the setup only runs once.

```
#include <Esplora.h>
void setup() {
  Esplora.writeRGB(0, 0, 255);
}
void loop() { }
```



writeRGB

The `writeRGB(red, green, blue)` function above takes 3 numbers. Where the value of each number controls the amount of that particular colour.

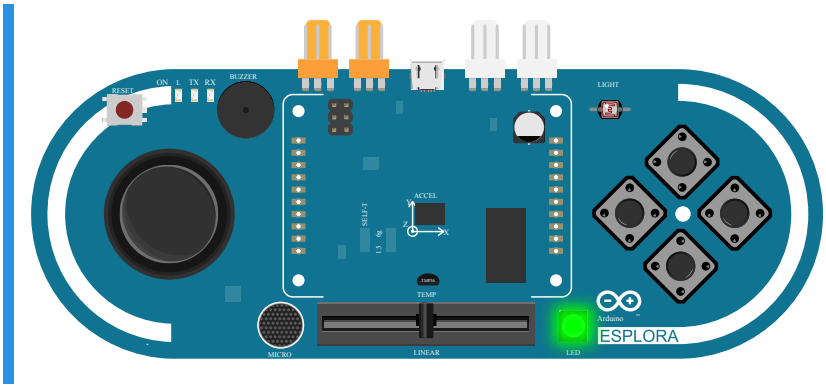
1.1.5. Problem: Setting the LED

Your task is to write a program that turns on the green LED to the maximum value of 255.

To do this we use the `Esplora.writeRGB` function.

The writeRGB function sets the brightness of the red, green and blue LEDs on the Arduino Esplora. It accepts numbers for 0 (off) to 255 (maximum brightness), for each colour.

Your program should behave like the following output:



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRGB(0, 0, 0);
}
```

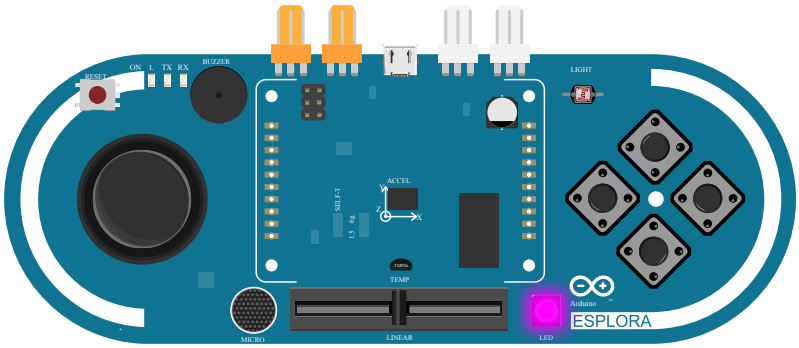
Testing

- [ ] Testing that the LED turns on.
- [ ] Testing that the LED is green.
- [ ] Testing that the LED is green at full brightness.
- [ ] Testing that red is off.
- [ ] Testing that blue is off.
- [ ] Testing that the LED stayed green forever.

1.1.6. Setting multiple colours!

The **`writeRGB`** function sets *all* the colours at once. So we can set multiple colours at the same time, let's make the LED magenta (red and blue)!

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 255);
}
```



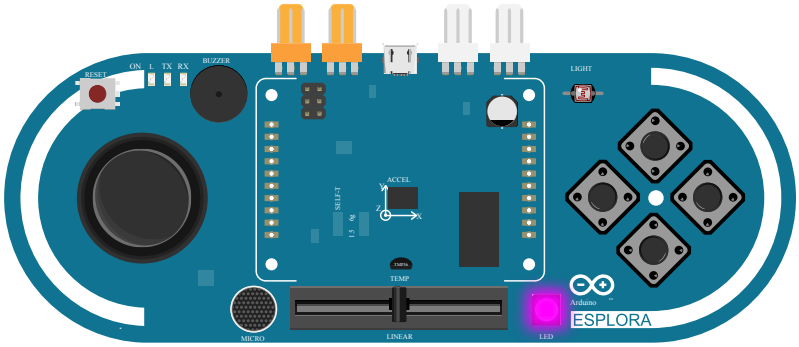
1.1.7. Problem: Setting two colours!

Use the RGB LED to make the colour magenta at full brightness, by setting the colours to be:

- RED: 255
- GREEN: 0
- BLUE: 255

When you set two different colours, those colours add together to make a new colour. This is exactly how an LED monitor (https://en.wikipedia.org/wiki/LED_display) makes colours.

Because red, green and blue are primary colours, we can make any colour you can imagine.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRGB(255, 255, 0); // red and green is yellow
}
```

Testing

- [ ] Testing that the LED turns on.
- [ ] Testing that the red is turned on.
- [ ] Testing that the red is at full brightness.
- [ ] Testing that the blue is turned on.
- [ ] Testing that the blue at is full brightness.
- [ ] Testing that the green is off.
- [ ] Testing that the led stays magenta for a long time.

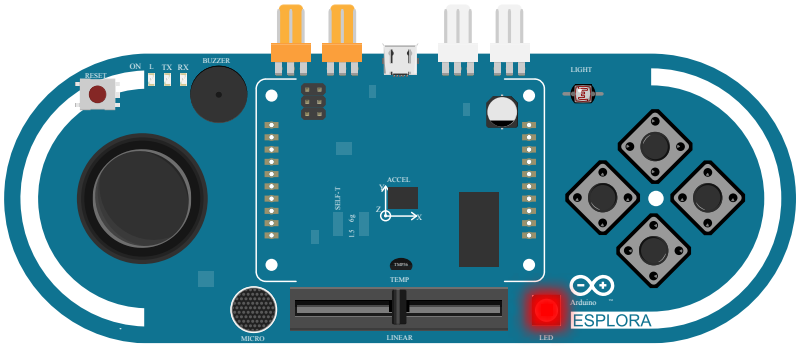
1.2. Arduino Syntax

1.2.1. Blocks of Code

Any language, including English, is governed by syntax, which is the set of rules, principles, and processes that govern the structure of sentences. Without agreed syntax, we wouldn't be able to understand each other.

In computers, this is also true. We program in different *programming languages*, and each language has a specific syntax that must be followed in order for the Arduino to understand what we want it to do.

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 0);
}
```



The `setup` function has the curly brackets { and } to denote when that *block of code* starts and finishes. All blocks of code need to start and finish with opening and closing curly brackets.

1.2.2. The Esplora Library

In order to be able to access all the sensors of the Arduino Esplora, we `#include` the *Esplora library*.

This is done with the line `#include <Esplora.h>`. You see this line at the top of almost every Esplora program.

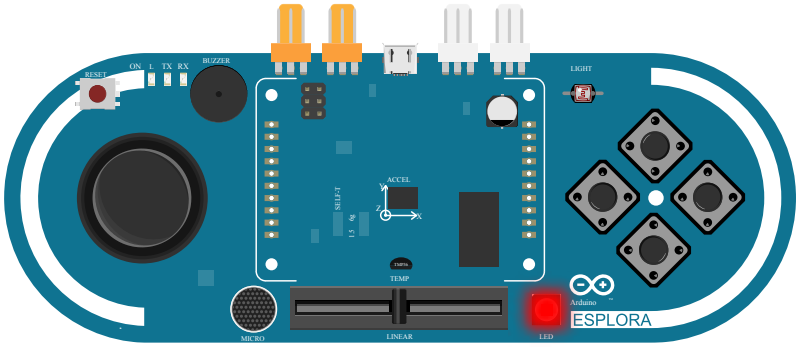
Try running the program below and see what happens...

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 0);
}
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
```

The error message is `Esplora was not declared in this scope`. To fix it we need to include the library, shown below.

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 0);
}
```



Error Messages

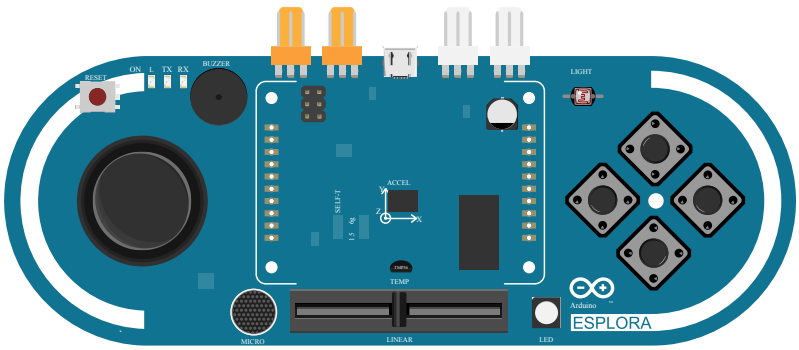
There will often be errors in your code. Even experienced programmers get them all the time. With practice, you will be able to avoid them and interpret the error messages should they occur.

1.2.3. Comments

Sometimes, you will want to leave yourself a little note, which tells yourself or someone reading your code what that code does. This is possible by adding *comments*.

Comments can be made by adding `//` symbols on the same line.

```
#include <Esplora.h>
// This is a comment. ¯\_(ツ)_/¯
void setup() { }
void loop() {
  // this code does nothing! :)
  // because the line below is commented-out
  // Esplora.writeRGB(255, 0, 0);
}
```



1.2.4. Semi-colons

Sketches are made up of *sets of instructions*. We have *functions* which run through a given number of *instructions*.

At the end of each instruction, Arduino expects there to be a semi-colon `;`.

For example the line `Esplora.writeRGB(255, 0, 0);` ends in a semi-colon `;`, that tells the Arduino that instruction is finished, and to move to the next one.

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 0)
}
```

```
Compiling sketch.ino...
/sketch/sketch.ino: In function 'void loop()':
/sketch/sketch.ino:5:1: error: expected ';' before '}' token
 }
 ^
exit status 1
```

More errors!

The error this time is **error: expected ';' before '}' token**

This means we need to add a semi-colon to the end of the `writeRGB` function so the line becomes:

```
Esplora.writeRGB(255, 0, 0);
```

1.3. Arduino Functions

1.3.1. The writeRGB function

The previous program used the **`writeRGB`** function to turn on the *red* LED. Let's take a look in a bit more detail...

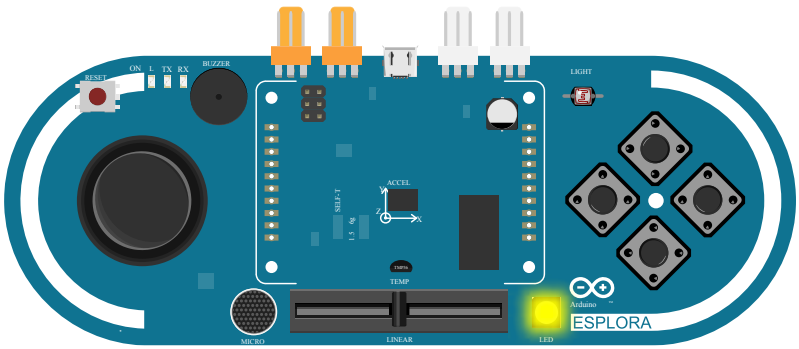
The function takes *3 numbers*, separated by a comma **`,`** to set the brightness of the *red*, *green* and *blue* LEDs, respectively.

The minimum brightness is 0 and the maximum is 255.

The reason we use those three colours is that they correspond to the photoreceptors in our eyes. These primary colours can be *added* together to produce any colour. We call this [additive colour (https://en.wikipedia.org/wiki/Additive_color)](https://en.wikipedia.org/wiki/Additive_color).

Try running the following code. You will see that the combination of red and green makes yellow!

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 255, 0);
}
```

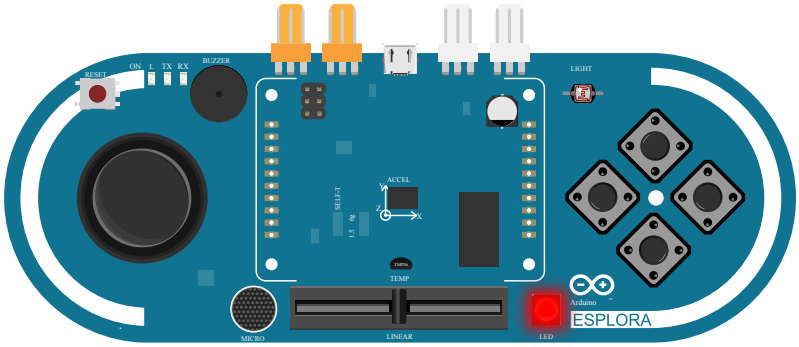


1.3.2. Writing multiple times

Say for example we wanted to flash the LED between red and green.

Try running the following code in the simulator and see what happens...

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRGB(255, 0, 0);
  Esplora.writeRGB(0, 255, 0);
}
```



It is difficult to tell what the colour is, because in the simulator they're flashing on and off really quickly.

To see each colour individually we need to pause between each colour change.

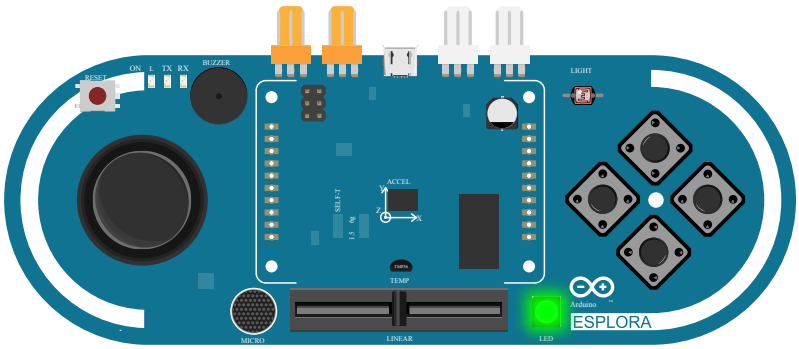
Let's do that by adding a `delay`!

1.3.3. The delay function

The `delay` function tells the Arduino Esplora to wait for a given period of time, in *milliseconds*. A millisecond is 1/1000th of a second. So it takes 1000 milliseconds to wait for one second.

Run the following code to see an example of the delay function:

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.writeRGB(255, 0, 0);
  delay(1000);
  Esplora.writeRGB(0, 255, 0);
  delay(1000);
}
```



After the final `delay(1000);` line, the code executes from the *beginning* of the loop and starts all over again.

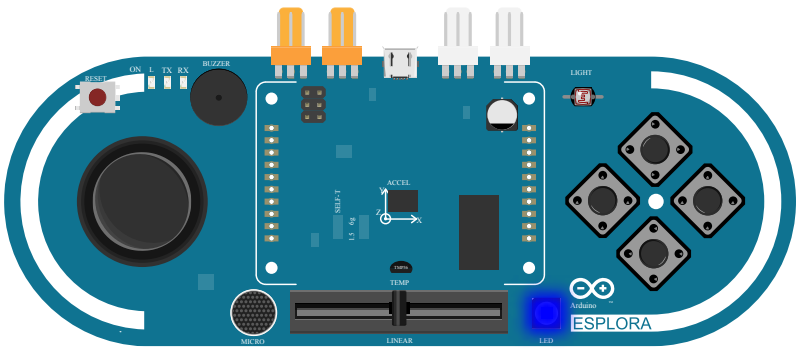
1.3.4. Problem: Flash The Siren!

The Australian Police Force has asked you to program some new lights for their siren.

Write a program to flash the LED between red and blue, with a delay of 500 milliseconds between each colour.

In order to complete this problem you are going to need the two functions we just reviewed. The `writeRGB` function, and the delay function.

Your program should behave like below:



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRGB(0, 0, 255);
  delay(100);
}
```

Testing

- [ ] Testing that the LED is on.
- [ ] Testing that the blue is off.
- [ ] Testing that the LED initially is red.
- [ ] Testing that the LED red at full brightness.
- [ ] Testing the blue is off.
- [ ] Testing the green is off.
- [ ] Testing the delay is correct.
- [ ] Testing the LED turns blue after the delay.
- [ ] Testing the LED turns blue at full brightness after the delay.
- [ ] Testing the delay after blue.
- [ ] Testing it is inside the loop.

1.3.5. Problem: CMY Colour scheme

Cyan, Magenta and Yellow are the colours that make up the CMYK colour model (https://en.wikipedia.org/wiki/CMYK_color_model), this is the subtractive colour model (https://en.wikipedia.org/wiki/Subtractive_color) that some colour printers use to produce all colours on pieces of paper.

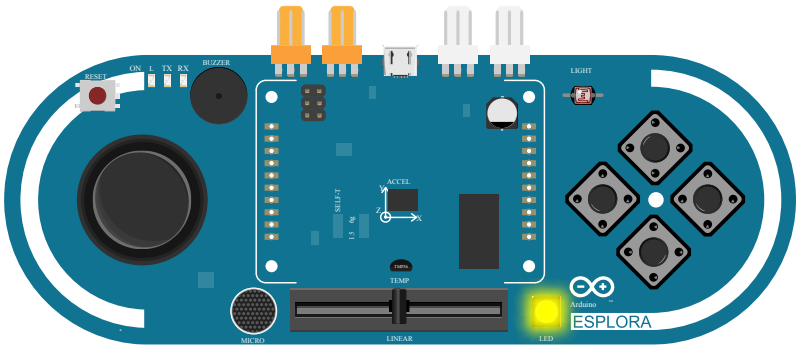
Sandra's printing service have asked you to produce an animation with the Esplora board so they can compare their ink cartriges with the colours produced by the RGB LED.

They have asked you to show each colour at full brightess, and each colour should remain on for *750 milliseconds*.

Your program should repeatedly change between the three colours while the Esplora is on in the following order

- Cyan - green and blue
- Magenta - red and blue
- Yellow - red and green

Your program should behave like below:



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // add your code here!
}
```

Testing

- [ ] Testing the LED turn on.
- [ ] Testing the green is on.
- [ ] Testing the green is on at full brightness.
- [ ] Testing the blue is also on.
- [ ] Testing cyan at full brightness.
- [ ] Testing magenta.
- [ ] Testing magenta at full brightness.

- [ ] Testing yellow.
- [ ] Testing yellow at full brightness.
- [ ] Testing the first delay.
- [ ] Testing the second delay.
- [ ] Testing the third delay.
- [ ] Testing it repeats those colours forever.

USING SENSORS

2.1. Variables

2.1.1. Variables

In computers, we store information in a *variable*. But there are different *types* of information we want to store, so we need to tell the computer (or Arduino) what type of information that we are storing.

Here are some examples:

- `int` (https://www.arduino.cc/en/Reference/Int) - a whole number, stands for **integer**
- `bool` (https://www.arduino.cc/en/Reference/BooleanVariables) - true or false, stands for **boolean**
- `float` (https://www.arduino.cc/en/Reference/Float) - a number with a decimal place, stands for **floating point**
- `String` (https://www.arduino.cc/en/Reference/String) - some text

There are more, but these are all the types we will be using in this course.

Let's take a look a single line of code to create an integer:

```
int my_integer = 500;
```

Here we have an integer, `my_integer`, that is storing a value of 500.

2.1.2. Integers

An integer is a whole number, either positive or negative. Computers have a finite amount of space, so the numbers that computers can store are also finite.

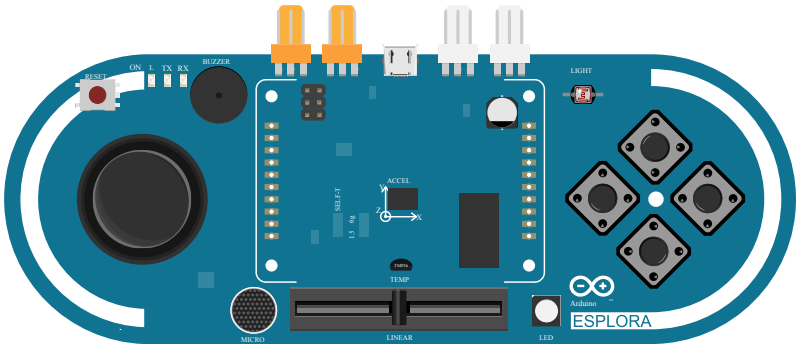
On the Arduino, an Int (https://www.arduino.cc/en/Reference/Int) has a range of values between -32,768 and 32,767.

If you want to store a larger number than that - you need to use a different data type like an unsigned int (https://www.arduino.cc/en/Reference/UnsignedInt), long (https://www.arduino.cc/en/Reference/Long) or unsigned long (https://www.arduino.cc/en/Reference/UnsignedLong).

2.1.3. Using Variables

Run the following program to flash the LED!

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRGB(255, 0, 255);
  delay(250);
  Esplora.writeRGB(0, 0, 0);
  delay(250);
}
```

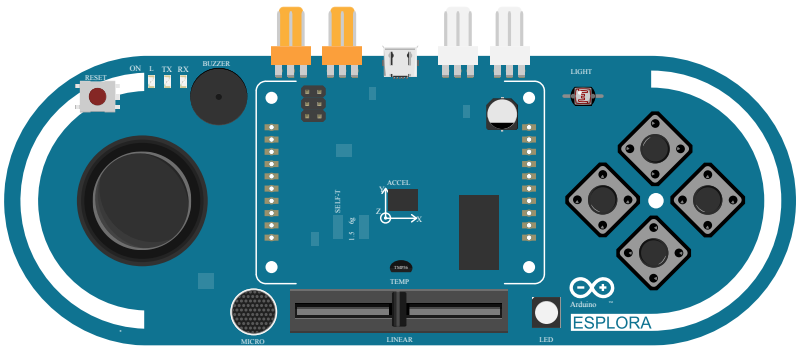


But if we wanted to change the speed at which the LED flashes on and off, it's possible to use a variable. That value will change the delay everywhere in the program that it is used.

```
int flash_delay = 250;
```

Try changing the flash_delay variable to a 500, or 5000, to see what happens to the speed the LED flashes.

```
#include <Esplora.h>
void setup() {}
void loop() {
  int flash_delay = 250;
  Esplora.writeRGB(255, 0, 255);
  delay(flash_delay);
  Esplora.writeRGB(0, 0, 0);
  delay(flash_delay);
}
```

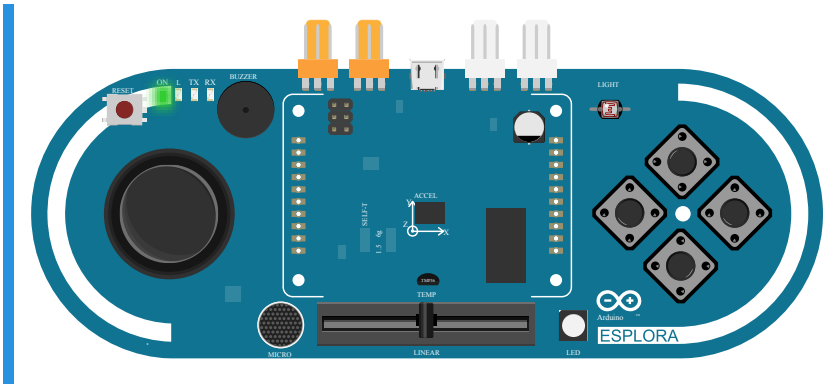


2.1.4. Problem: Flashing!

Variables can be set so a value that is repeated in your code can be changed easily.

Modify the programs that the time between the LED changing colour is half a second (500 milliseconds). To do this, just change the `flash_delay` variable.

Below is an animation of how the program should run.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  int flash_delay = 200;
  Esplora.writeRGB(255, 255, 0);
  delay(flash_delay);
  Esplora.writeRGB(0, 0, 0);
  delay(flash_delay);
}
```

Testing

- [ ] Testing the LED is on initially.
- [ ] Testing yellow initially.
- [ ] Testing yellow all full brightness.
- [ ] Testing the led turns off.
- [ ] Testing the delay.
- [ ] Testing the second delay.
- [ ] Testing the it loops forever.

2.1.5. Local Variables

Local variables are ones that are defined inside a function.

Let's try and make a counter - to count how many times the Arduino program has looped. We know the setup runs only once, so you might try and start the counter there like below:

```
void setup() {
  int counter = 0;
  // counter is visible here
}
void loop() {
  // but not here!
  counter = counter + 1;
  Serial.println(count);
  delay(50);
}
```

```
Compiling sketch.ino...
/sketch/sketch.ino: In function 'void loop()':
/sketch/sketch.ino:7:3: error: 'counter' was not declared in this scope
   counter = counter + 1;
   ^
/sketch/sketch.ino:8:18: error: 'count' was not declared in this scope
   Serial.println(count);
                  ^
exit status 1
```

The program will throw an error because the variable counter is not visible within the scope of loop().

> **Looking through the scope**
>
> In Arduino, variables are only visible within a certain scope. This means that if a variable was declared *inside* a function - it is not visible anywhere outside of that function.

Let's learn about global variables to fix this problem on the next page!

2.1.6. Global variables

To fix this problem, let's move the line where we declared the counter variable to the top of the sketch, to make it global!

```
int counter = 0;
void setup() {
  // counter is now visible here as well
}
void loop() {
  counter++;
  Serial.println(counter);
  delay(500);
}
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
1
2
3
4
5
6
```

Now the program works!

The ++ operator

It's possible to change the line from the previous page:

```
counter = counter + 1;
```

to

```
counter++;
```

They're exactly the same! But the `counter++` is a lot easier to write. It just adds one to a number!

2.1.7. Problem: Make it count!

You are teaching your younger sibling how to count, but you are far too lazy to write all the numbers down by hand.

Instead, write a program to print out all the positive numbers, starting from 0, with a *1.5 second* delay between each number. Each number should print on a new line to the serial port.

The output should look like the following:

```
0
1
2
3
4
...
```

You will need to store the `counter` as a *global variable*, and any `delay` should be at the **end** of the `loop`.

> **Printing**
>
> Use the `Serial.println` function to print and then add a new line!

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing 1 is printed.
- [ ] Testing 2 is printed.
- [ ] Testing 3 is printed.
- [ ] Testing 0 is printed.
- [ ] Testing the delay between each output.
- [ ] Testing a lot of numbers.
- [ ] Testing a hidden case.

2.2. Serial

2.2.1. Serial

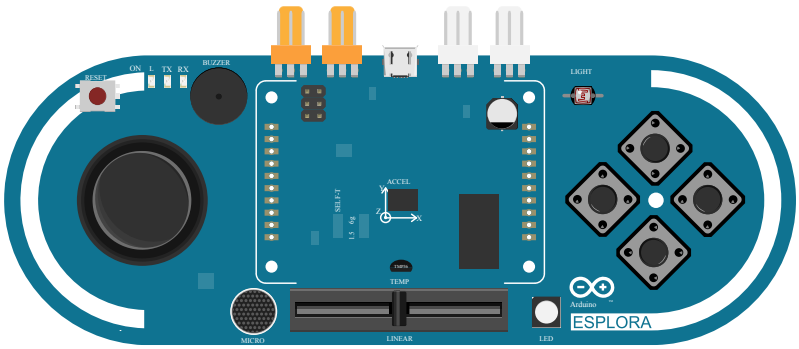
All the sensors we read from give us an integer, but how do we know what the value of the integer actually is?

To do this we can use the Serial Monitor, by using the print statement.

To activate the Serial Monitor, we add Serial.begin(9600); into the setup() function.

Run the following program to see an example in the virtual serial console.

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  int slider = Esplora.readSlider();
  Serial.println(slider);
  delay(500);
}
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
512
512
512
```

The program above will save the position of the slider to the variable slider. Then print that number to the Serial console.

Try running the program and moving the slider in your browser!

> **Adding a delay**
>
> Because Arduino loops forever, adding a delay at the end of the loop function can be a good way to ensuring your code doesn't print so much text that you can't actually read what is going on.

2.2.2. Begin

In order for an Arduino (an external device) to communicate with the computer, it needs to open the Serial connection. This can be done with the begin (https://www.arduino.cc/en/Serial/Begin) function.

```
Serial.begin(9600);
```

`9600` specifies the rate at which the data is transmitted to the computer in *bits per second*. We'll always use that number in this course.

2.2.3. Printing text

We can also write messages to the serial console, instead of writing just a number.

Try running the code below!

```
void setup() {
Serial.begin(9600);
}
void loop() {
  Serial.print("Hello!");
  delay(5000);
}
```

```
Compiling sketch.ino...
===info |||  Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info |||  Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
Hello!
```

Notice something missing?

Have you noticed that the program above does not include the Esplora library? Because we aren't using any of the sensors on the board, we don't need to include it.

2.2.4. The temperature sensor

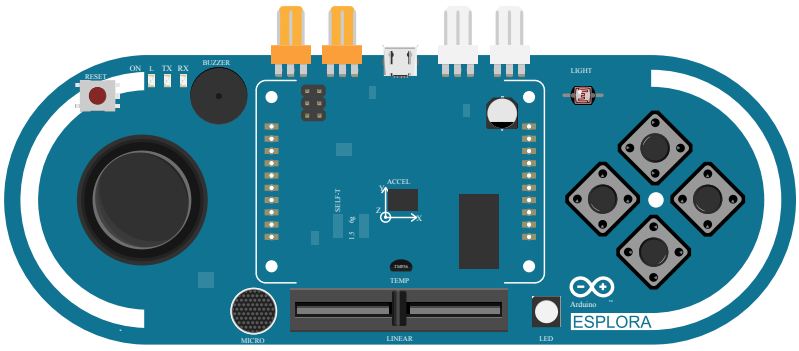
The temperature sensor reads the *ambient* temperature in a room.

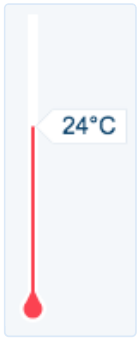
It's possible to read both the temperature in degrees celsius and fahrenheit the following way:

```
int celsius = Esplora.readTemperature(DEGREES_C);
int fahrenheit = Esplora.readTemperature(DEGREES_F);
```

Try running the program below and changing the temperature!

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  int t = Esplora.readTemperature(DEGREES_C);
  Serial.println(t);
  delay(2000);
}
```






```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
24
```

2.2.5. Problem: A temperature monitor!

At school, the air conditioning in your classroom seems a bit faulty. Despite complaints to the school office, the rooms are still too hot.

To gather some data to support your claim that the rooms are above a reasonable temperature, create a temperature monitor that prints the temperature to the serial port on a new line *every second*.

The `readTemperature` function reads the ambient temperature of the room.

The output should look exactly like the following:

```
<temperature>
<temperature>
```

Where `<temperature>` is the number read by the `readTemperature` function in degrees Celsius.

For example the room temperature is 24. If it the device starts at 24 and changes to 25, it will print:

```
24
24
24
25
25
```

Be sure to add a delay of **1 second** after every line is printed.

You will need to use the function `Esplora.readTemperature(DEGREES_C);` to read the temperature.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing something is printed.
- [ ] Testing the serial output is 24 when nothing changes.
- [ ] Testing the serial output changes to 25.
- [ ] Testing the serial output changes to 26.
- [ ] Testing when the temperature changes.
- [ ] Testing the delay.
- [ ] Testing a hidden case.

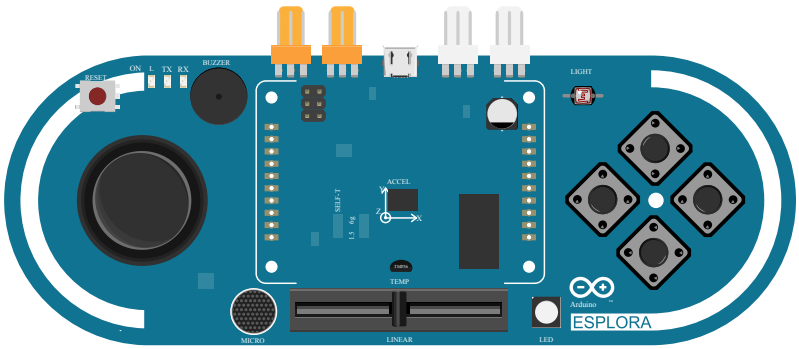
2.2.6. Printing multiple types

To make what we print to the serial console easier to understand, often we want to write some text which explains what that data means. Printing is really useful for *debugging* code, if your program behaves in a way that you don't expect.

On the Arduino, we need an additional print statement to output our text.

Run the following example to print the value of the slider.

```
#include <Esplora.h>
void setup() {
Serial.begin(9600);
}
void loop() {
  int slider = Esplora.readSlider();
  Serial.print("The slider is: ");
  Serial.println(slider);
  delay(500);
}
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
The slider is: 512
The slider is: 512
```

You will see the Serial monitor prints

```
The slider is: <slider>
```

on a new line every half a second.

The first print statement is text, the second is the integer!

> **print vs. println**
>
> You might have noticed there was a `print` and `println` function in the code above. The difference is the `println` prints a new line after it has finished. So the next thing to be printed is on the next line.

2.2.7. Problem: Printing Multiple Types

Fix up the temperature sensor to make the message a bit clearer. Change your program so that the printed output is:

```
The temperature is <temperature> degrees Celsius.
The temperature is <temperature> degrees Celsius.
The temperature is <temperature> degrees Celsius.
```

After each line is printed, add a delay of **1 second**.

If the temperature is `<temperature>` degrees. For example your program should print the following if the temperature changes from 24 to 25 degrees.

```
The temperature is 24 degrees Celsius.
The temperature is 24 degrees Celsius.
The temperature is 25 degrees Celsius.
...
```

> **Spaces**
>
> Remember to add the spaces and full stop into your sentence.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing something is printed.
- [ ] Test the output at 24 degrees.
- [ ] Test the output at 25 degrees.
- [ ] Test the output at 26 degrees.
- [ ] Testing when the temperature changes to 25.
- [ ] Testing when the temperature changes.
- [ ] Testing a hidden case.

2.3. Sensors

2.3.1. Sensors

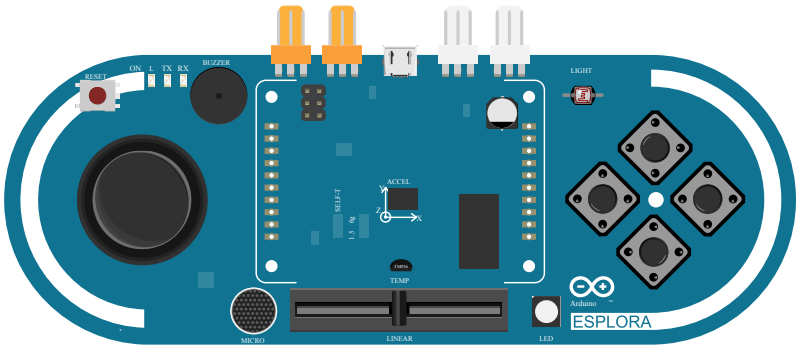
Sensors are things that respond to changes in the environment.

On the Arduino Esplora, sensors give us information in the form of a number. Based on what the number is, we can *interpret* those numbers in order to make some decision or perform an action.

Here is an example to read from the Slider:

```
int slider = Esplora.readSlider();
```

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  int slider = Esplora.readSlider();
  Serial.println(slider);
  delay(1000);
}
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
512
```

It's important to add the delay so you don't write to the console too quickly.

2.3.2. The Microphone

Just like reading from the slider, it is possible to read from the Esplora's microphone.

This can be done by calling the `readMicrophone` function. Here is the line of code that does the magic:

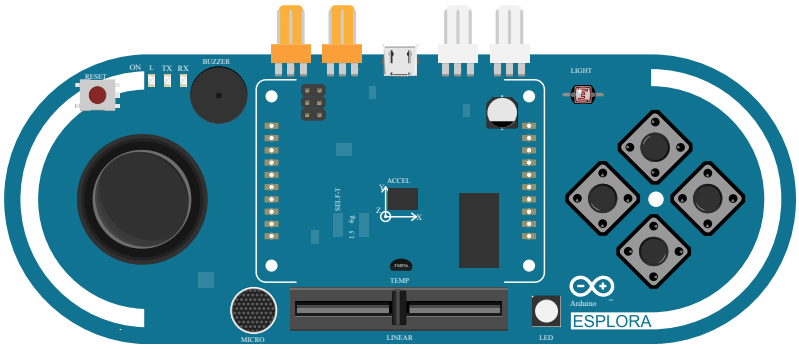
```
int mic = Esplora.readMicrophone();
```

The line of code above reads from the microphone and saves the value in the variable `mic`, which we can use later on.

On the Esplora, the microphone sensor measures the *volume* of sound that is received by the microphone.

Run the program below and click on the microphone to make a sound!

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  int mic = Esplora.readMicrophone();
  Serial.println(mic);
  delay(100);
}
```




```
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
0
0
0
0
0
0
0
0
0
0
0
0
```

2.3.3. Problem: A sound logger!

Schultz Security Services have asked you to make a sound logger, so they can tell if a sound is made in a room which is supposed to be empty.

Write a program that prints the value of the microphone to the serial monitor - with a 50ms delay at the end of the loop.

The output should come in the following format:

```
microphone: <number>
```

Where `<number>` is the number picked up by the microphone. When there is no sound, the output should look like:

```
microphone: 0
microphone: 0
microphone: 0
microphone: 0
```

Activating the microphone

Click on the microphone to generate a sound.

Ctrl+click (or cmd+click on mac) to make a louder sound!

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  // your code goes here!
}
```

Testing

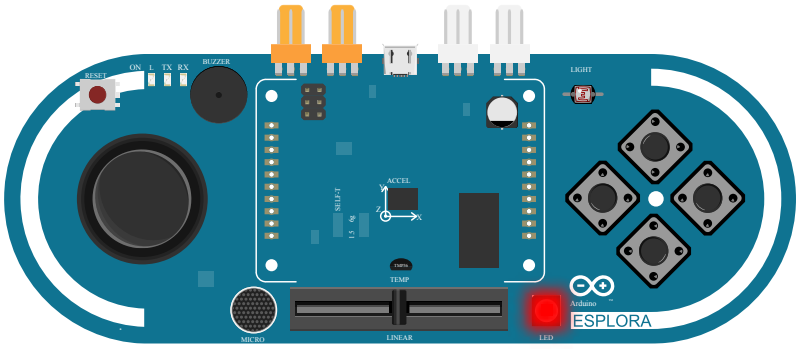
- [ ] Testing when the microphone is 0.
- [ ] Testing when the microphone is clicked - 405.
- [ ] Testing when the microphone ctrl+clicked - 608.
- [ ] Testing a range of microphone values.
- [ ] Testing a hidden case.

2.3.4. Sound-activated LEDs

Since the value received from sensors is an integer (a number), and what we send to the LEDs is also a number. It's quite easy to control the LEDs from a sensor.

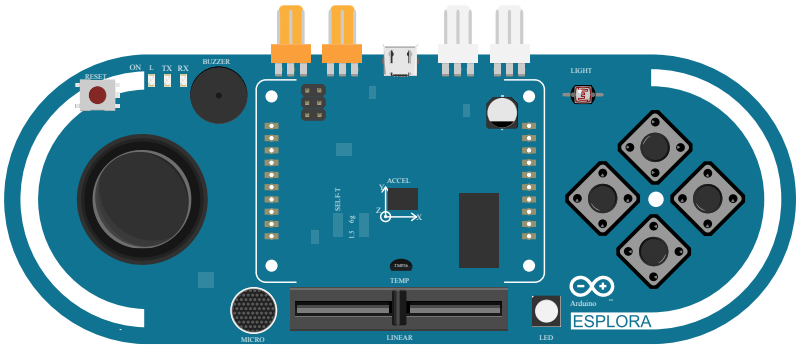
Remember how we turn on the red LED, just by writing a number between 0 and 255 into the writeRGB function, like below.

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRGB(255, 0, 0);
  delay(20);
}
```



In order to change the brightness of the red LED with the amount of sound measured by the microphone, we could do something like the following:

```
#include <Esplora.h>
void setup() {}
void loop() {
  int mic = Esplora.readMicrophone();
  Esplora.writeRGB(mic, 0, 0);
  delay(20);
}
```

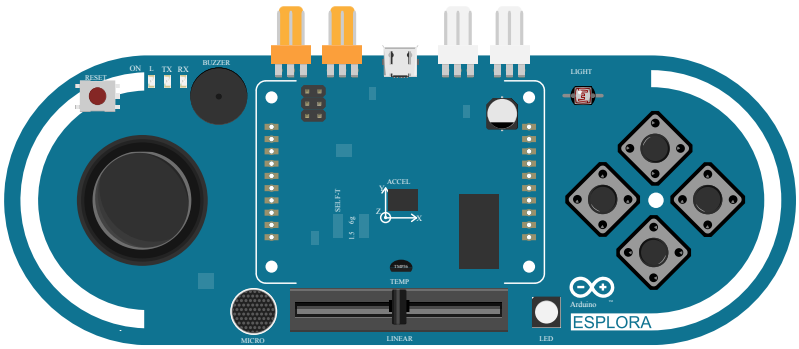


2.3.5. Problem: Sound activation!

Write a program that takes the value from the microphone, and sends it to the green LED.

Make it that the number sent to the green led is the same as the value picked up by the microphone.

To activate the microphone, click on it to make a sound, and ctrl+click to make a loud sound.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here!
}
```

Testing

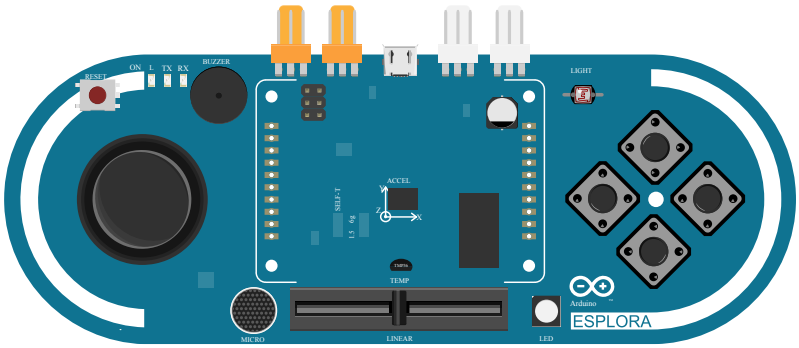
- [ ] Testing the LED is off when there is no sound.
- [ ] Testing the LED turns on when a sound is made.
- [ ] Testing the green is on.
- [ ] Testing the other colours are off.
- [ ] Testing with a low sound.
- [ ] Testing with a higher sound.
- [ ] Testing with a much higher sound.
- [ ] Testing with a range of sounds.
- [ ] Testing with a hidden case.

2.3.6. Maximum values

The maximum value for the LED is 255, but what happens if we try and send a value greater than that to it?

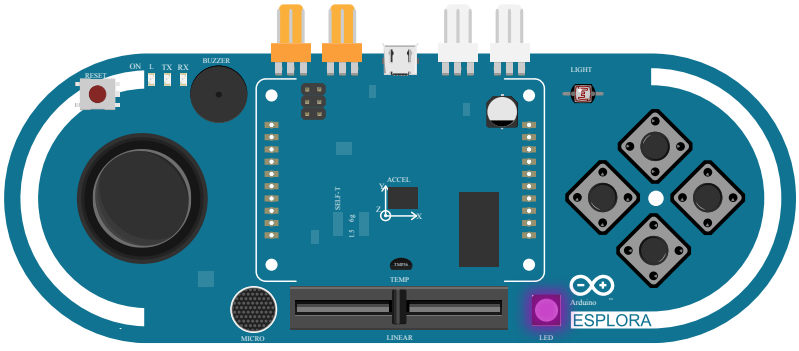
Try this with the slider:

```
#include <Esplora.h>
void setup() {}
void loop() {
  int slider = Esplora.readSlider();
  Esplora.writeRGB(slider, 0, slider);
}
```



To make it work as you would expect, we need to divide the slider by 4, as shown in the example below.

```
#include <Esplora.h>
void setup() {}
void loop() {
  // divides the slider by 4, so the maximum value is 255
  int slider = Esplora.readSlider()/4;
  Esplora.writeRGB(slider, 0, slider);
}
```



Now try moving the slider to control the brightness of the LED!

3

MAKING DECISIONS

3.1. The If Statement

3.1.1. Making Decisions

We often want to make a decision when something has occurred.

It could be when the temperature reaches a certain value, or a sound has been detected, or a button has been pushed - when these things occur we want to perform some action.

All of this can be done with an if statement (https://www.arduino.cc/en/reference/if), which is the most basic form of decision-making a computer does.

In this module, you'll learn how to write if statements and learn how they can be applied to various situations.

3.1.2. If

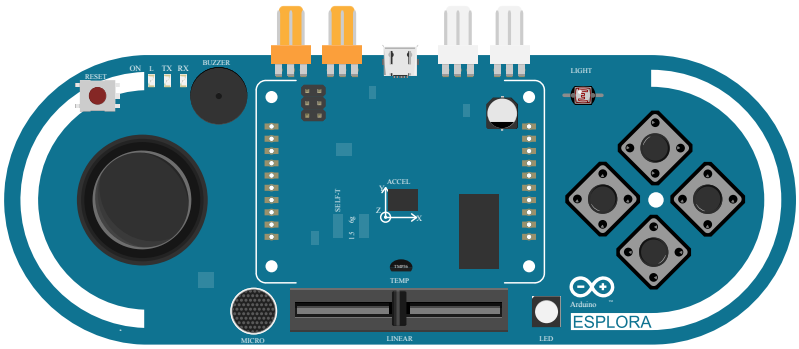
The if statement (https://www.arduino.cc/en/reference/if) decides whether a block of code should be executed.

It can be written using the following syntax:

```
if (condition) {
  // block of code to run if condition is true
}
```

In the above code - if the condition is `true`, then the block of code that is indented will run!

```
void setup() {
  Serial.begin(9600);
  int x = 5;
  if (x == 5) {
    Serial.println("x is 5!!");
  }
  delay(2000);
}
void loop() { }
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
x is 5!!
```

All we're asking is *yes* or *no* questions!

The == symbol *compares* one side with the other. If they are both the same, the result is `true`. So in this case, x is 5, so the block of code executes.

Comparison vs assignment

A single = sign, is an assignment operator. Which assigns a particular value.

```
int my_value = 42;
```

So the following will not do what you expect (note the single equals sign):

```
if (x = 8) {
  // do something
}
```

The above line will set x to be 8, not check if it is 8, and the result will always evaluate to be true, so the block of code the if statement controls will always execute. Fix it with:

```
int x = 8;
if (x == 8) {
  // do something
}
```

3.1.3. Using If Statements

Let's take a look at how we can use the if statement on the Esplora.

To check if one of the buttons has been pressed down - we can use the `Esplora.readButton` function. We can send it a number to check which button we would like to read. For example, to read from button 1:

```
if (Esplora.readButton(1) == PRESSED) {
  Serial.println("Button 1 was pressed!");
}
```

In the example above, we use the `PRESSED` keyword, to tell when the button has been pressed.

If statement syntax

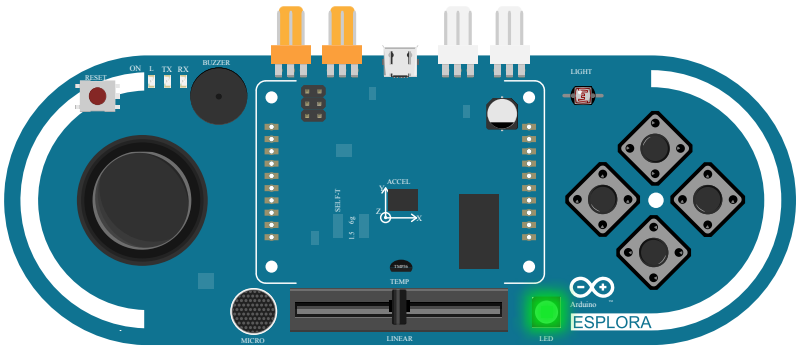
It is important to note that there is no semi-colon ; on the if statement!

3.1.4. Problem: What if...

Write a program to turn the LED green at a brightness of **200** when button **2** has been pressed.

Make sure all the other colours are set to **0** (off).

Below is an animation of the device working when the button is pressed.



Remember: To turn on the RGB LED use the line: `Esplora.writeRGB(RED, GREEN, BLUE);`

> **Pressing buttons**
>
> You can press the buttons by clicking on the button, or by pressing 1, 2, 3 and 4 on the keyboard!

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED is off initially.
- [ ] Testing that the LED turns on after a button has been pressed.
- [ ] Testing that the LED turns on after button 2 is pressed.
- [ ] Testing that the green LED turns on after button 2 is pressed.
- [ ] Testing the green is at least 200.
- [ ] Testing the green is 200.
- [ ] Testing that the other colours are the correct value.
- [ ] Testing that the LED stays on.

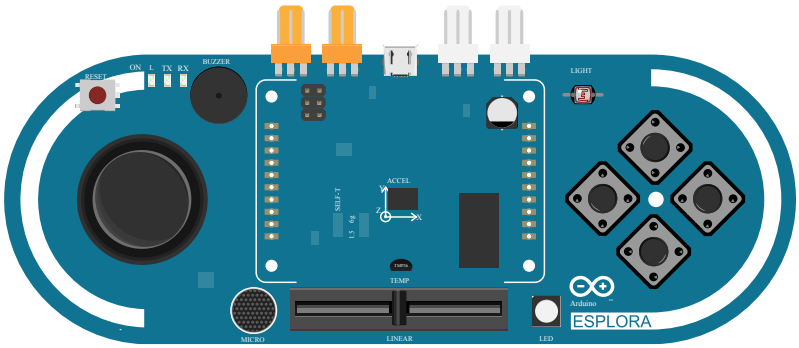
3.1.5. Two if statements

After the `if-statement` has finished, the code just keeps executing through the loop.

So it's possible to add *multiple* if statements if we want to check multiple things!

Let's add a second if statment to turn all the LEDs off again by pressing *button 2*.

```
#include <Esplora.h>
void setup() { }
void loop() {
  if (Esplora.readButton(1) == PRESSED) {
    Esplora.writeRGB(0, 255, 0);
  }
  if (Esplora.readButton(2) == PRESSED) {
    Esplora.writeRGB(0, 0, 0);
  }
}
```

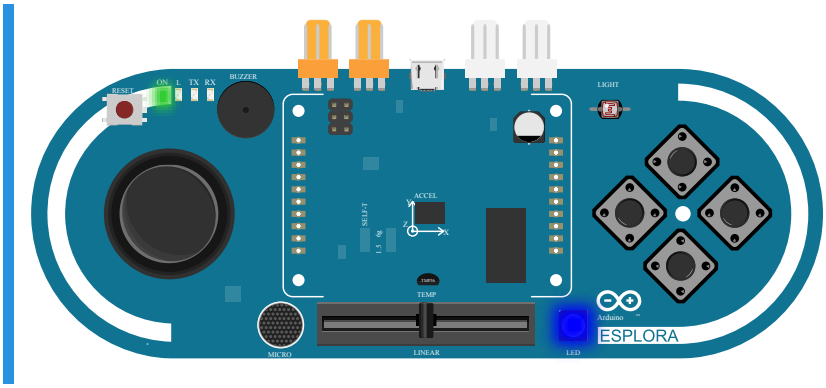


3.1.6. Problem: Manual Siren

The AFP want to press a button to control how the siren changes colour!

Write a program that turns the LED *red* when *button 1* is pressed, and *blue* when *button 2* is pressed. In both instances the value should be 255.

Below is an animation of what should happen when the buttons have been pressed.



Remember: To turn on the RGB LED use the line:

```
Esplora.writeRGB(RED, GREEN, BLUE);
```

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED is off initially.
- [ ] Testing the LED turns on when a button is pressed.
- [ ] Testing the red turns on when pressing button 1.
- [ ] Testing the red is 255 when pressing button 1.
- [ ] Testing the other colours are off when button 1 is pressed.
- [ ] Testing the LED is blue when button 2 is pressed.
- [ ] Testing the LED is blue at full brightness when button 2 is pressed.
- [ ] Testing red and green are 0 when button 2 is pressed.
- [ ] Testing the colours are correct for both buttons

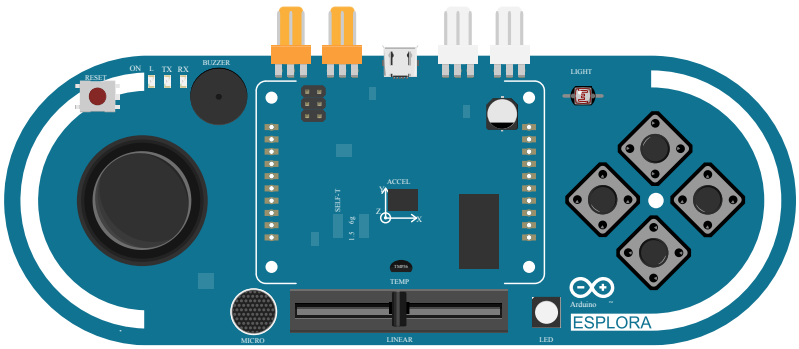
3.2. Else Statments

3.2.1. The Else Statement

It's all well and good being able to make decisions, but sometimes we need to perform some *other* action if the condition we are testing is *not* true.

This is possible by using the else (https://www.arduino.cc/en/reference/else) statement. Let's take a look at an example.

```
#include <Esplora.h>
void setup() {}
void loop() {
  if (Esplora.readButton(1) == PRESSED) {
    // runs if button 1 is pressed
    Esplora.writeRGB(255, 0, 255);
  }
  else {
    // runs if button 1 is NOT pressed
    Esplora.writeRGB(0, 0, 0);
  }
}
```



The else statement catches everything *else*, so it *must* be paired with a corresponding *if-statement*.

Else syntax

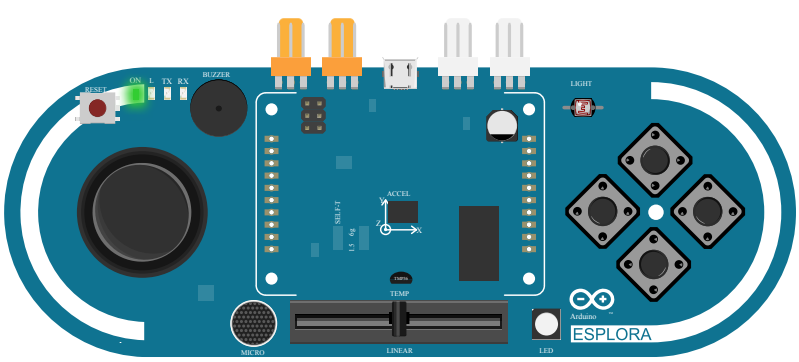
Just like the if statement, the else statement also does not have a semi-colon ; after it.

3.2.2. Problem: Mr Difficult

Mr Difficult wants a light-control system different from the others. He wants the light only to turn on when *button 4* is pressed.

Implement this by using an `if-else` statement to make your LED turn *blue* at *full brightness** when *button 4* is pressed, and off when it is *not* pressed.

Below is an animation of the button being pressed and released.



Remember: To turn on the RGB LED use the line: `Esplora.writeRGB(RED, GREEN, BLUE);`

Interactive solution

You can check the program working in the demo program by pressing on button 4 with your mouse pointer.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing that the LED is off initially.
- [ ] Testing that the LED turns on when a button is pressed.
- [ ] Testing that the LED turns on when button 4 is pressed.
- [ ] Testing that the LED turns blue when 4 is pressed.
- [ ] Testing that the LED turns off when button 4 is released.
- [ ] Testing multiple button presses.

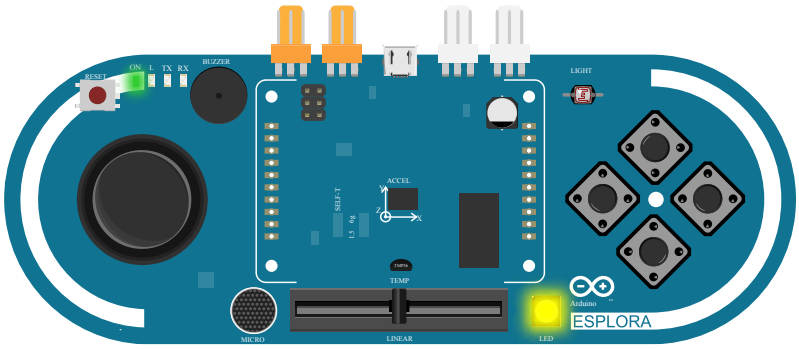
3.2.3. Problem: Mrs Opposite

Mrs Opposite was a new approach to lighting systems. She wants to trial a new system where the light is *on* by default, and only turns off when the correct button is pressed.

Protoype this new system for her which behaves the following way:

- No buttons pressed - the LED should be **yellow** at full brightness
- **button 3 is pressed** - LED should be OFF

Below is an animation of the prototype running when the button has been pressed.



Hint

You will need an *if-else* block to complete this question.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing that the LED is on initially.
- [ ] Testing that the LED is yellow initially.
- [ ] Testing that the LED is yellow at full brightness.
- [ ] Testing the LED turns off when any button is pressed.
- [ ] Testing the LED turns off when button 3 is pressed.
- [ ] Testing the LED turns on again when button 3 is released.
- [ ] Testing multiple button presses.

3.3. Conditions

3.3.1. Conditions

What type of conditions can we test? Remember in an if statement, it executes if the result is true. That means, we are asking yes or no questions.

For example, is x equal to y? If the answer is yes, then the result is true. Otherwise, the result is false.

Whether something is true or false, just comes down to a yes or no question.

We've already shown we can test if two things are equal (using the == operator), but there are also others:

Operation	Operator
equal to	==
not equal to	!=
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=

> **Numbers**
>
> You might notice that most of those comparisons are mathematical operations. In Arduino, the most common comparison you make is with numbers.
>
> For example, we set the RGB LED to be a number, and it turns on. This is the case for all our inputs and outputs.

3.3.2. Not Equal

It's possible to use conditions in order to decide to do something with the Arduino!

Let's take a look at how to check if something is *NOT* equal:

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
  int x = 8;
  int y = 10;
  if (x != y) {
    Serial.println("x is NOT equal to y");
    x = 10;
  }
}
void loop() { }
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
x is NOT equal to y
```

The `if-statement` used the `!=` symbol to test if x is *NOT* equal to y. Previously we used the `==` symbol to check if the two side *are* equal.

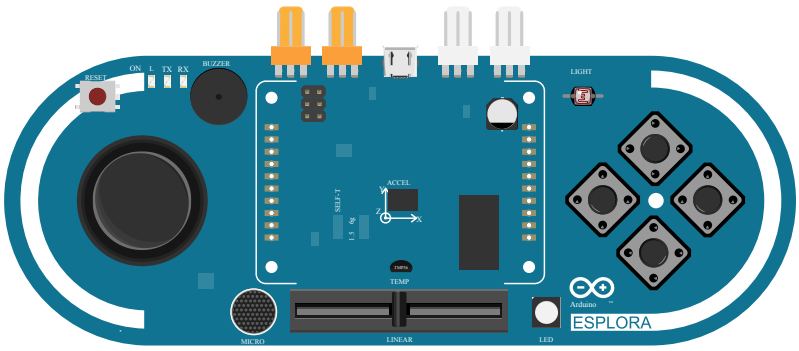
3.3.3. Inequalities

Inequalities test a range of values - instead of testing if something is exactly equal. The following are inequalities:

Operation	Operator
less than	<
less than or equal to	<=
greater than	>
greater than or equal to	>=

On the Arduino, the slider gives us a value of 1023 when it is all the way to the *left*, and 0 when it is all the way to the *right*. The following program turn on the RED LED when the slider is on the right-half, and off otherwise.

```
#include <Esplora.h>
void setup() { }
void loop() {
  if (Esplora.readSlider() < 512) {
    Esplora.writeRGB(255, 0, 0);
  }
  else {
    Esplora.writeRGB(0, 0, 0);
  }
}
```



Run the program while moving the slider to see!

Esplora functions

You might be wondering - how do we know that the function to read from the slider is `Esplora.readSlider`? The answer is that all the functions we use are in the Arduino reference (https://www.arduino.cc/en/Reference/EsploraLibrary)!

3.3.4. The Light Sensor

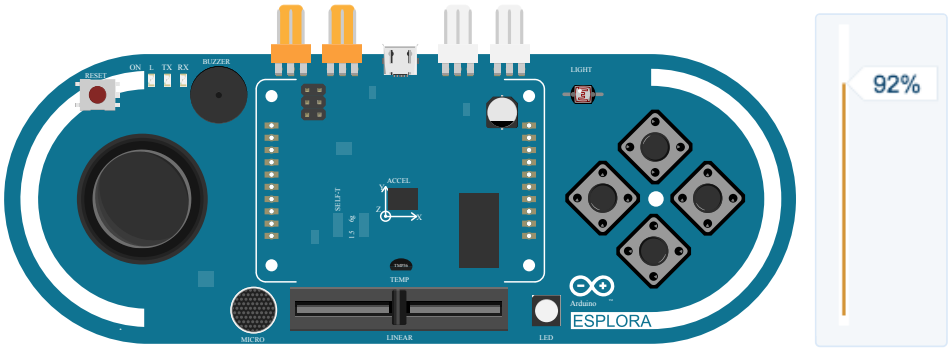
The Esplora has lots of different sensors, all of them give you an integer.

The light sensor can be accessed using the `Esplora.readLightSensor` function.

```
int light = Esplora.readLightSensor();
```

The numbers it gives us go from 0 to 1023. Run the program and see what the number is compared to the percentage.

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  int light = Esplora.readLightSensor();
  Serial.println(light);
  delay(500);
}
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
941
941
```

3.3.5. Problem: The automatic torch!

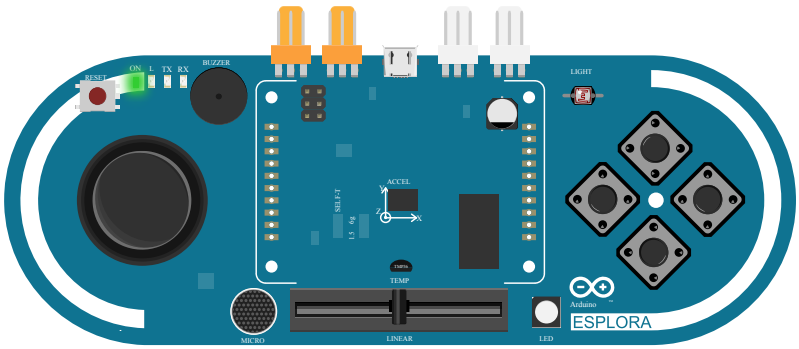
It is too dark to see at night, so you need a torch to light the way.

But your boss at Schultz Security Services has decided that pressing buttons is far too much work for her.

Make her an automatic torch that turns the LED to *white* at *full brightness* once the light levels are below 80% (at 818).

That means the torch will turn on automagically after it gets dark!

Below is an animation when the light level changes.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED is off initially.
- [ ] Testing the LED turns on when the light is 0.
- [ ] Testing the LED is white at full brightness.
- [ ] Testing the LED is off at a high number.
- [ ] Testing the LED is on at 817.
- [ ] Testing the LED is off at 818.
- [ ] Testing the LED turns on and off when the light level changes.
- [ ] Testing a hidden case.

3.3.6. Problem: Complimentary colours

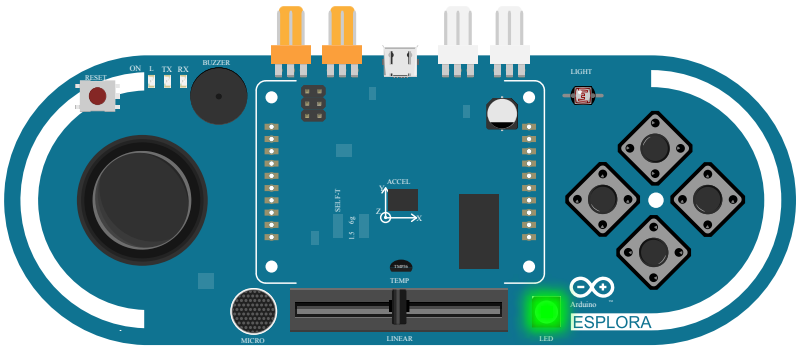
Complimentary colours (https://en.wikipedia.org/wiki/Complementary_colors) are colours that are opposite on the colour wheel.

Write a program to show the opposite colours magenta and green on the rgb LED by using the slider.

When the slider is on the left (greater than 512), make the LED **magenta** at full brightness.

When the slider on on the right (less than or equal to 512), make the LED **green** at full brightness.

See an animation of the program working when the slider moves left and right.



Slider

Remember to use the `Esplora.readSlider()` function to access the value of the slider.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED is green when the slider is 0.
- [ ] Testing the other colours are set to 0.
- [ ] Testing the slider greater than 512.
- [ ] Testing the slider exactly 512.
- [ ] Testing a range of values.

3.4. Multiple statements

3.4.1. Multiple statements

If you want to test *multiple* conditions, we need to add more statements! This can be done with *nesting*.

```
int x = 10;
if (x < 5) {
  Serial.println("x is less than 5.");
}
else {
  if (x == 5) {
    Serial.println("x is 5");
  }
  else {
    Serial.println("x is greater than 5")
  }
}
```

This can get complicated really quickly! A nice way is to use the `else if` statement, which works like this:

```
int x = 10;
if (x < 5) {
  Serial.println("x is less than 5.");
}
else if (x == 5) {
  Serial.println("x is 5");
}
else {
  Serial.println("x is greater than 5")
}
```

You can add as many conditions to test as you like with more `else if` statements.

3.4.2. Or else

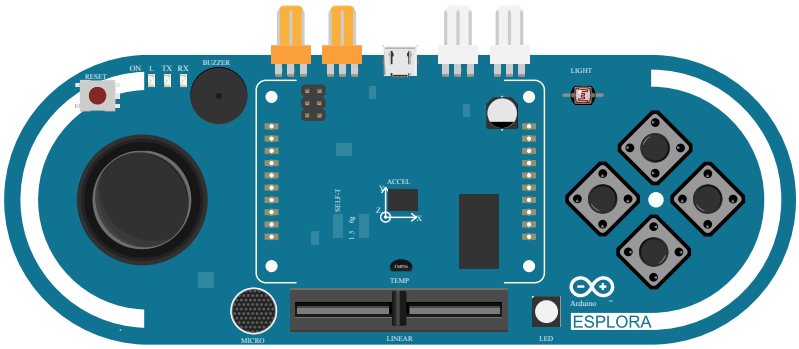
Let's look at an example where we might need this.

The `else` statement catches *everything* not run in the `if` statement. So it can be thought of as catching every other situation.

We can have multiple `if` and `else` checks in the same program, to test different things.

Let's take a look at an example where we want button 1 pressed to make the LED green and button 2 to make the LED yellow. Using what we've learned so far - you might try the following:

```
#include <Esplora.h>
void setup() {}
void loop() {
  if (Esplora.readButton(1) == PRESSED) {
    Esplora.writeRGB(255, 0, 0);
  }
  else {
    Esplora.writeRGB(0, 0, 0);
  }
  if (Esplora.readButton(2) == PRESSED) {
    Esplora.writeRGB(255, 255, 0);
  }
  else {
    Esplora.writeRGB(0, 0, 0);
  }
}
```



Did you notice that whenever you press a button, the LED turns on and off really fast?

Let's explore this flashing behaviour: When we press button 1, the LED turns RED. But because button2 is not pressed, the second else clause applies, which turns off the LED. In turn, when we press button 2, the LED turns GREEN, but because button 1 is not pressed, the first else clause will switch the LED off in the following loop.

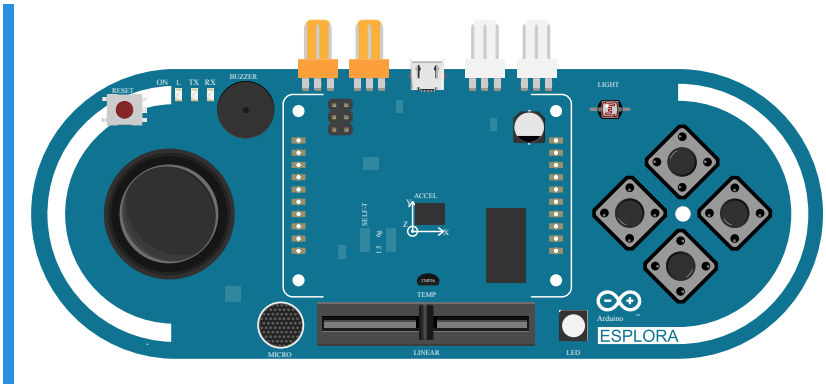
We can fix this issue in the next section, with the `else if` statement!

3.4.3. If-Else-If-Else

Let's take a look at an example:

If we want to test if either button 1 or button 2 has been pressed, we can do the following:

```
#include <Esplora.h>
void setup() {}
void loop() {
  if (Esplora.readButton(1) == PRESSED) {
    Esplora.writeRGB(255, 0, 0);
  }
  else if (Esplora.readButton(2) == PRESSED) {
    Esplora.writeRGB(0, 255, 0);
  }
  else {
    Esplora.writeRGB(0, 0, 0);
  }
}
```



Note the final else statement to turn OFF the LED if neither condition is true.

Try pressing both buttons at the same time by pressing 1 and 2 on the keyboard.

3.4.4. Problem: Colour selecta

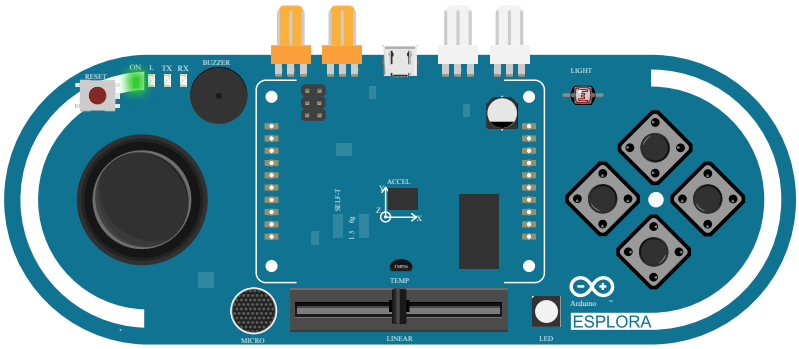
Your arts teacher is giving a lesson on *primary colours* in computers, and has asked you to create a device where they can select the correct colour by pressing a button.

Your program should behave the following way when the buttons are pressed:

- **button 1** - red
- **button 2** - green
- **button 3** - blue

In all cases the LED should be on at full brightness. If multiple buttons are pressed, the button with the *lower number* takes precedence. So if buttons 1 **and** 2 are pressed, the LED will be red.

Below is an animation of the buttons being pushed, one after another.



Pressing buttons

It's possible to press buttons 1, 2, 3 and 4 on the keyboard to press down that button in the Arduino Esplora simulator.

Give it a go!

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

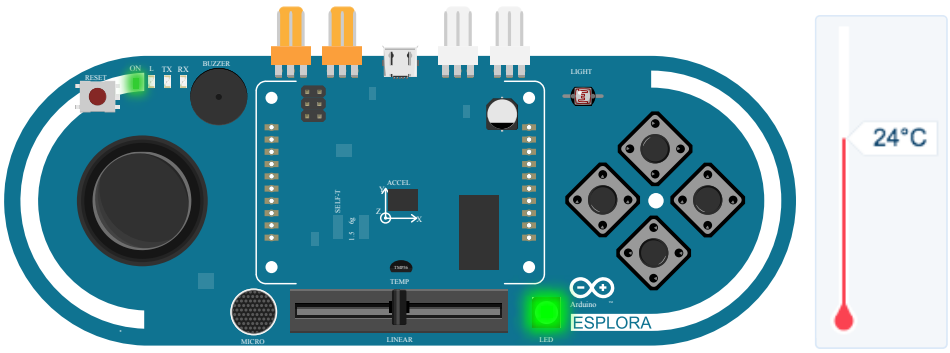
- [ ] Testing the LED is off initally.
- [ ] Testing the LED is red when button 1 is pressed.
- [ ] Testing the LED is green when button 2 is pressed.
- [ ] Testing the LED is blue when button 3 is pressed.
- [ ] Testing the LED turns off after button 1 is released.
- [ ] Testing the LED turns off after button 2 is released.
- [ ] Testing the LED turns off after button 3 is released.
- [ ] Testing multiple button presses.

3.4.5. Problem: Temperature indicator

You have been asked to create a temperature monitor for your workshop, using the LED to indicate the temperature. It should behave the following way:

- When the temperature is **greater than** 30, the LED should be red
- When the temperature is from 19 to 30, the LED should be green
- When it's 18 or below, the LED should turn blue.

Below is an animation playing while temperature changes.



When a condition is satisfied, it will run that block of code and jump to the *end* of the if-else statements. To check if a number is *less than* 1000 but *greater than* 500, we could do so like this:

```
if (number > 1000) {
 // number is greater than 1000
}
else if (number > 500) {
 // number is greater than 500,
 // but less than or equal to 1000
}
```

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED is on regardless of the temperature.
- [ ] Testing at 35 degrees.
- [ ] Testing at 25 degrees.
- [ ] Testing at 10 degrees.
- [ ] Testing the first threshold.
- [ ] Testing the second threshold.
- [ ] Testing a range of values.

- [ ] Testing a hidden case.

4

USING NUMBERS

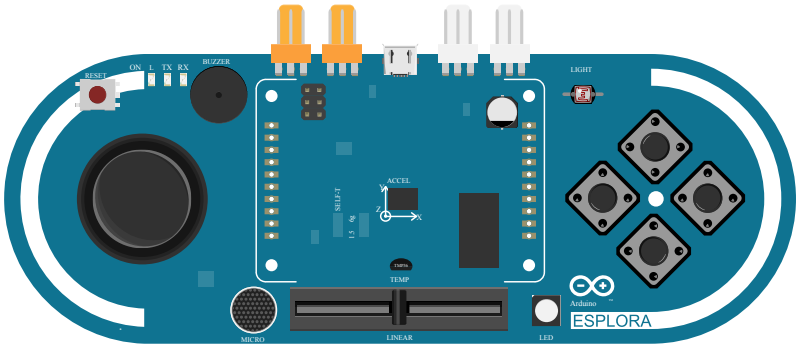
4.1. Making music

4.1.1. Playing tones

Let's make some music!

We can emit a tone using the `Esplora.tone` function. Let's look at an example!

```
#include <Esplora.h>
void setup() { }
void loop() {
  if (Esplora.readButton(1) == PRESSED) {
    Esplora.tone(1000);
  }
  else {
    Esplora.noTone();
  }
}
```



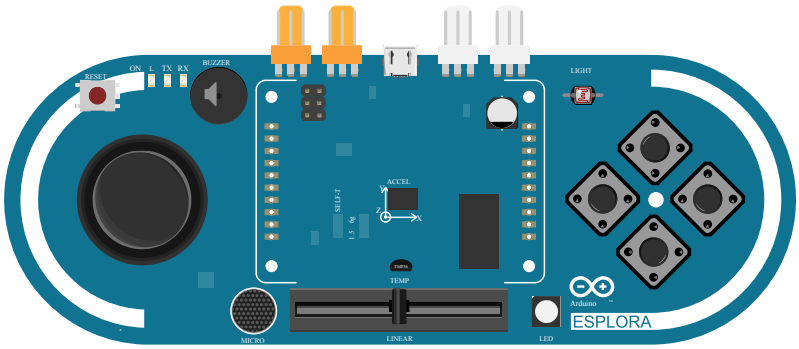
When you hold down *button 1*, a tone of 1000 Hz will be emitted by the speaker. Try changing the number to adjust the pitch of the sound that is emitted!

4.1.2. Problem: Make an instrument!

You have been asked by your music teacher to make a musical instrument using the Esplora.

Your instrument should behave in the following way:

- When button 1 is pressed, a tone should be emitted by an amount equal to the slider
- There should be no tone when no buttons are pressed.



Pressing buttons

Remember you can press buttons on the Esplora by pressing the keys 1, 2, 3 and 4.

This way you can press the button at the same time as moving the slider.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing no tone happens initially.
- [ ] Testing a tone is emitted when any button is pressed.
- [ ] Testing a tone is emitted when button 1 is pressed.
- [ ] Testing the tone when the slider is 500.
- [ ] Testing the tone when the slider is 1000.
- [ ] Testing the tone when the slider is 0.
- [ ] Testing the tone a range of slider values.
- [ ] Testing the tone when button is released.
- [ ] Testing multiple button presses.
- [ ] Testing a hidden case.

4.1.3. Performing Calculations

It's often much easier for a computer to perform calculations, rather than us doing them by hand.

Arduinos are really good calculators! They can do anything a calculator can do. Let's look at some basic algebra.

Symbol	Name
+	add
-	subtract
*	multiply
/	divide

Let's look at an example!

```
void setup() {
  int x = 5;
  Serial.println((10 + x) * 3);
}
void loop () { }
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
45
```

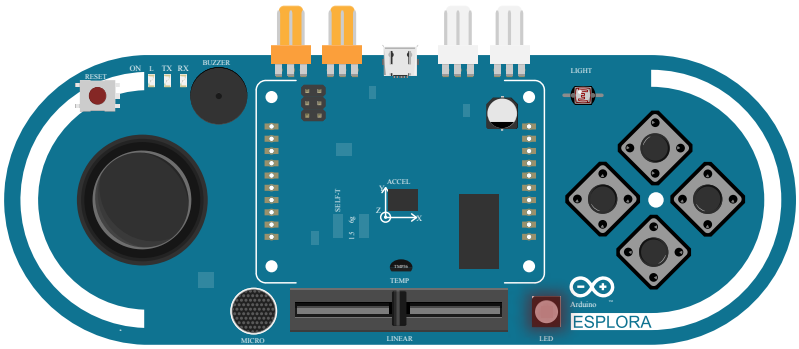
The number printed is 60, which does the calulation `(10 + 5) * 3`, just like in maths we use the brackets to say which operations go first.

4.1.4. Multiplying

We can use the Esplora to manipulate numbers. You probably remember that our sensors just produce *numbers*. So instead of just using the sensor value, we can do some math with it first, which we use as output. This way we can produce some interesting output on our LED or speaker.

Run the code below!

```
#include <Esplora.h>
void setup() { }
void loop() {
  int colour = 120;
  if (Esplora.readButton(1) == PRESSED) {
    Esplora.writeRGB(colour * 2, 0, 0);
  }
  else {
    Esplora.writeRGB(colour / 2, 0, 0);
  }
}
```



When button 1 is pressed, the red will be at 240, almost full brightness. But if it's not pressed, the brightness will be 60.

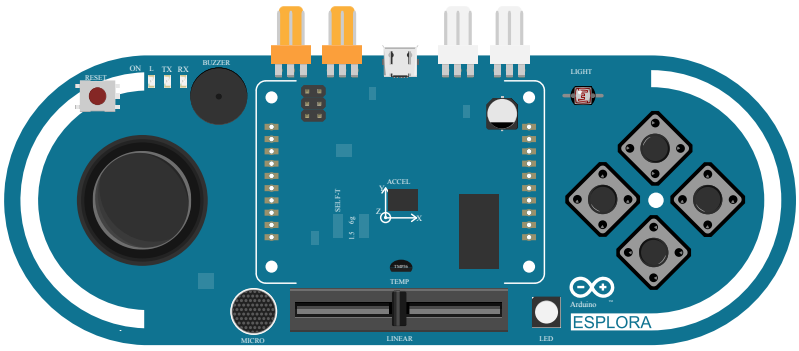
4.1.5. Notes

Let's try and do the same thing with tones!

Run the example below and change the `pitch` variable to change the frequency.

Try pressing *both* buttons down at once by pressing 1 and 2 on the keyboard and see what happens!

```
#include <Esplora.h>
void setup() { }
void loop() {
  int pitch = 440;  // note A4
  if (Esplora.readButton(1) == PRESSED) {
    Esplora.tone(pitch);
  }
  else if (Esplora.readButton(2) == PRESSED) {
    Esplora.tone(pitch * 2);
  }
  else {
    Esplora.noTone();
  }
}
```



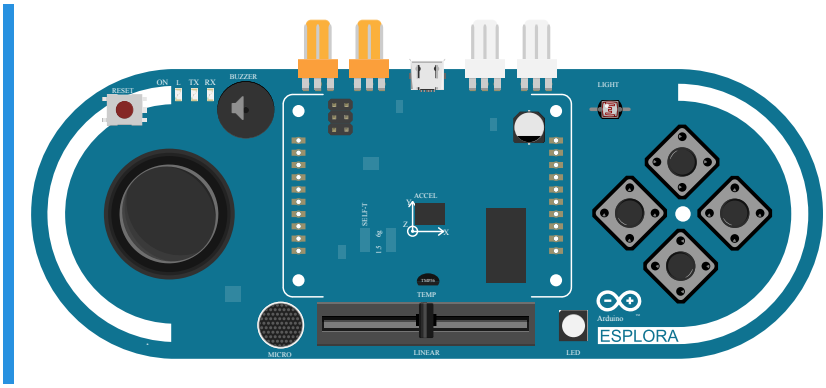
4.1.6. Problem: Double the notes

Your music teacher now asked you to extend the previous instrument and make it better by playing more notes!

Modify your previous instrument so it now behaves in the following way:

- When button 1 is pressed, a tone should be emitted by an amount equal to the slider
- When button 2 is pressed, the tone should be **2 times the slider**
- There should be no tone when no buttons are pressed.
- If both button 1 and 2 are pressed, the tone is equal to the slider

Below is an animation of the device operating when the slider is moved and either button 1 or 2 are pressed.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing no tone happens initially.
- [ ] Testing a tone is emitted when any button is pressed.
- [ ] Testing a tone is emitted when button 1 is pressed.
- [ ] Testing the tone when the slider is 500.
- [ ] Testing the tone when the slider is 1000.
- [ ] Testing the tone a range of slider values.
- [ ] Testing a tone is emitted when button 2 is pressed.
- [ ] Testing the tone when the slider is 500.
- [ ] Testing the tone when the slider is 1000.
- [ ] Testing a range of slider values on button 2.
- [ ] Testing the tone when button is released.

- [ ] Testing button 1 and 2 pressed at the same time.
- [ ] Testing multiple button presses.
- [ ] Testing a hidden case.

4.2. Logic

4.2.1. Tone timer

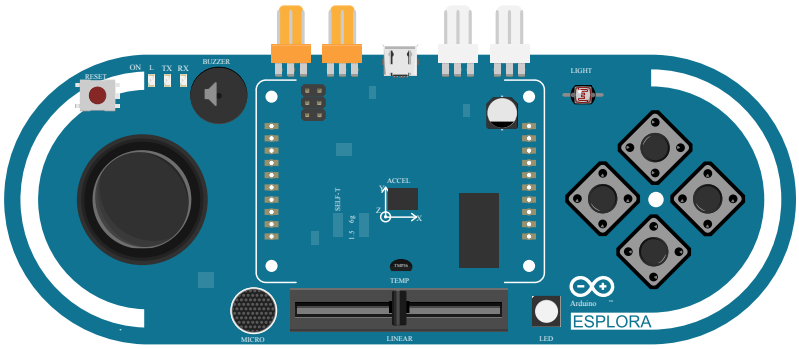
It's possible to make a tone last for a given amount of time, by adding a second number in the tone function.

The first number is tone, and the second optional number is the duration in *milliseconds*.

```
Esplora.tone(frequency, duration);
```

Run the example below where every *5 seconds*, a *500 Hertz* tone is emitted for *1 second*.

```
#include <Esplora.h>
void setup() { }
void loop() {
  Esplora.tone(500, 1000);
  delay(5000);
}
```



Change the numbers around to see the how the duration works!

4.2.2. Logical operators

Sometimes we want to make multiple decisions, we can use logical operations to make that process easier. These are sometimes referred to as boolean operations (https://www.arduino.cc/en/Reference/Boolean).

The logical operations we'll use are:

- **AND** - If something **and** something else is true.
- **OR** - If something **or** something else is true.

In code, the **AND** symbol is `&&`, the **OR** symbol is `||`.

4.2.3. AND

Let's assume that we wanted to perform some kind of operation when *button 1* is pressed **and** the temperature is greater than 30 degrees C. There are two conditions to check, which we can join together in a single `if` statement.

This can be done with the **and statement** - `&&` - to check multiple conditions in an `if` statement.

```
if (Esplora.readButton(1) == PRESSED && Esplora.readTemperature() > 30) {
  // both conditions satisfied
}
```

Using logical expressions makes it easier to make more sophisticated decisions.

More decisions

It is possible to have more decisions than just two with the and statement.

```
if (cond1 && cond2 && cond3) {
  // all 3 conditions have been satisfied
}
```

It's possible to add as many **&&** expressions as you need.

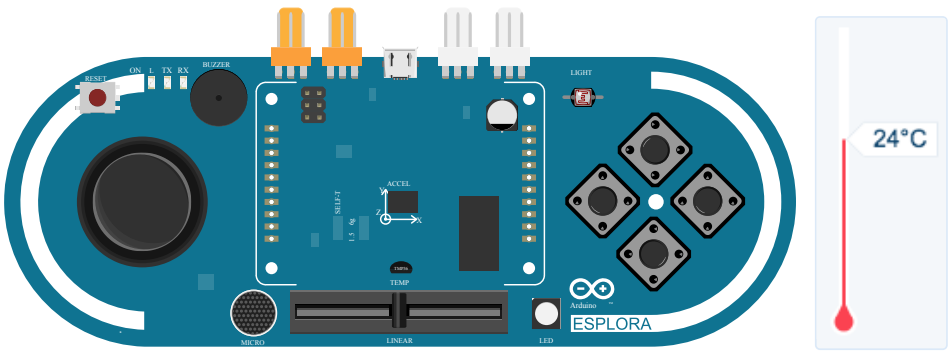
4.2.4. AND again

Let's combine our music playing with the **and** statement to emit a tone when multiple conditions are satisfied.

The following program is a temperature alarm, which the slider can control whether it activates.

The Esplora emits a tone when the temperature is over 30 degrees **and** the slider is on the left, to turn it on.

```
#include <Esplora.h>
void setup() { }
void loop() {
  int slider = Esplora.readSlider();
  int temp = Esplora.readTemperature(DEGREES_C);
  if (slider > 1000 && temp > 30) {
    Esplora.tone(500, 1000);
  }
}
```



Move the slider back and forth after the temperature is over 30 to see if the tone still plays.

4.2.5. Problem: Sense of Security

Schultz Security Services are prototyping a new device that will monitor if there is sound in a yard, but they only want the device to be active at night.

They have asked you to protoype their device using the Arduino Esplora.

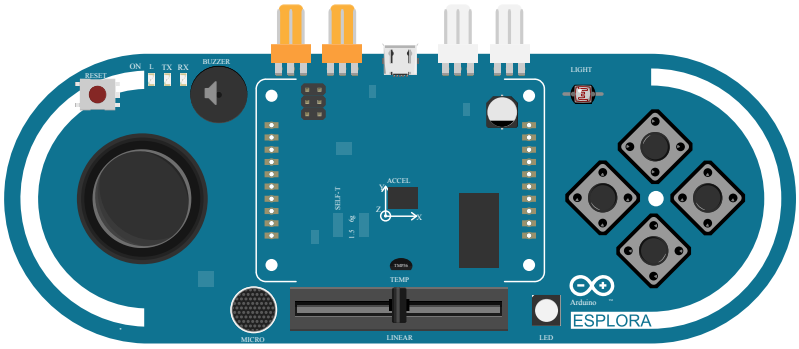
The device should activate when:

- It is dark outside (the light is *less* than 700) **and** it detects a sound *greater* than 300.

Activate means:

- Turning the LED to *white* at full brightness
- Emitting a tone of 800 Hz, for **2 seconds**.

After it has activated, it should delay for **3 second** before rechecking for sound, if there is no sound then turn the LED and tone off.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing nothing happens initially.
- [ ] Testing the LED turns on.
- [ ] Testing the microphone threshold.
- [ ] Testing the light sensor threshold.
- [ ] Testing the LED is the correct colour.
- [ ] Testing the LED is white at full brightness.
- [ ] Testing the there is a tone.
- [ ] Testing the tone is 800Hz.
- [ ] Testing the tone lasts for 2 seconds.
- [ ] Testing the LED is on for 3 seconds.

- [ ] Testing one changes.
- [ ] Testing when the other sensor changes.
- [ ] Testing a hidden case.

4.2.6. OR

The other type of logic we often need is the **OR** statement.

The **OR** statement will check if one condition *or* another is true. The or statement consists of two vertical bars `||`. They can be entered by pressing `Shift+\` (the key above enter).

```
if (condition1 == true || condition2 == true) {
  // either condition1 OR condition2 is true
}
```

Chaining decisions

Just like with the AND operator, it's possible to add many OR `||` conditions to an `if` statement.

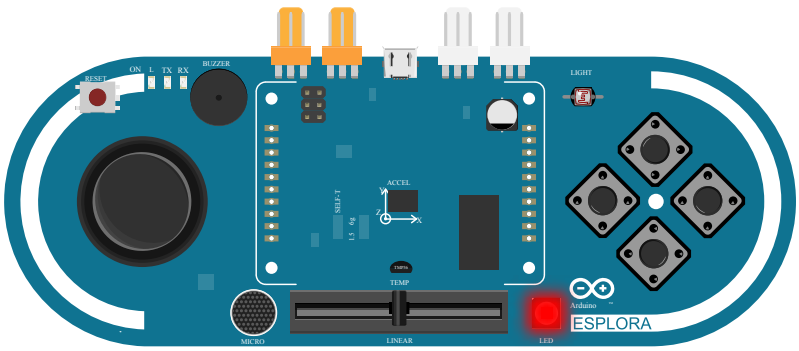
4.2.7. Problem: Sitting on the fence

Sitting on the fence (https://en.wikipedia.org/wiki/Sitting_on_the_fence) is when someone can't decide which side to be on. Make the LED red when a fence sitter is detected, otherwise make the LED green.

Decide whether the slider is *fence sitting* by checking if it is halfway *plus or minus 150*, which is 362 to 662.

That means if the slider is less than **362** *or* greater than **662**, the LED should be green at full brightness. Otherwise, the LED should be red, to indicate a fence-sitter.

Below is an animation with a simulated slider that moves.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the red in on when the slider is 512.
- [ ] Testing the red is full brightness when the slider at 512.
- [ ] Testing the colour is correct when the slider is at 1023.
- [ ] Testing the slider at 0.
- [ ] Testing the slider at upper threshold.
- [ ] Testing the slider at the lower threshold.
- [ ] Testing a range of slider values.

4.3. Floats

4.3.1. Data types

Computers store data internally in **binary** and they have different ways of representing different *types* of data. For each variable, we have to tell the computer what *type* of data it represents.

So far we've only been using *integers*, but doing mathematical operations means the answer may have a *decimal place*.

But integers are whole numbers, they can't store decimal places, so they just chop off everything after the decimal point.

Run the code to see an example.

```
void setup() {
  Serial.begin(9600);
  int chop_chop = 10 / 3;
  Serial.println(chop_chop);
}
void loop () { }
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
3
```

4.3.2. Float

The method of storing decimal places is with the `float` data type.

Creating a `float` is almost exactly the same as creating an integer.

```
float number = 35.33;
```

Instead of the word *int*, use the word `float` to tell the Arduino that we need to store a decimal place.

4.3.3. Calculation with float

Dividing with Arduino is a bit tricky, because the default type of a number is an *integer*, and the Arduino will perform an integer calculation. If we want a float calculation, we indicate this by changing one of the numbers to a float. We have done this in the following example by changing 5 to 5.0.

```
void setup() {
  Serial.begin(9600);
  float chopped_off = 9 / 5;
  float expected = 9 / 5.0;
  Serial.print("Not rounded: ");
  Serial.println(chopped_off);
  Serial.print("Rounded: ");
  Serial.println(expected);
}
void loop() { }
```

```
Compiling sketch.ino...
===info |||  Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info |||  Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
Not rounded: 1.00
Rounded: 1.80
```

4.3.4. Calculating percentage

The formula to calculate percentage is the following:

$$\text{Percent} = \frac{\text{value}}{\text{max value}} \times 100$$

If we set the LED to be at a brightness of **155**, we can calculate the percentage the following way. We already know that the maximum brightness is **255**.

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
  int bright = 150;
  float percent = (bright/255.0)*100;     // will work
  float wrong_percent = (bright/255)*100; // wont work
  Serial.print("Percentage is: ");
  Serial.println(percent);
  Serial.print("But the percentage is not: ");
  Serial.println(wrong_percent);
  Serial.println("Because we need to add a decimal place after 255.");
}
void loop() { }
```

```
Compiling sketch.ino...
===info |||  Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info |||  Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
Percentage is: 58.82
But the percentage is not: 0.00
Because we need to add a decimal place after 255.
```

4.3.5. Problem: Calculate the percentage

Write a program to calculate the percentage the slider has moved!

Use formula for percentage, where the value for `max slider` is 1023.0.

$$\text{Percent} = \frac{\text{slider}}{\text{max slider}} \times 100$$

Your program should output the following when the slider is at 512:

```
50.05
```

When the slider is *all* the way to the left, the output should be 100 percent.

```
100.00
```

You should print the result to the serial console *every second* on a new line.

> **Hint!**
> Remember to make the denominator (the number under the fraction) have a decimal place.

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing something is printed.
- [ ] Testing the first example.
- [ ] Testing the second example.
- [ ] Testing when the slider is at 0.
- [ ] Testing when the slider is at 255.
- [ ] Testing when the slider is at 750.
- [ ] Testing a range of values.
- [ ] Testing a hidden case.

4.3.6. Problem: Farenheit to celsius

An American tourist is visiting Australia, and is having trouble converting Fahrenheit to Celsius. Write a program that helps him to do this.

Your program should: 1) read from the temperate sensor in degrees Fahrenheit, then 2) print it as a *float*, then 3) convert the temperature to degrees Celsius and 4) print it as a *float*. Print all outputs on the same line. Your program should print a new line *every second*.

The formula to convert Fahrenheit to Celsius is:

$$C = (F - 32) \times \frac{5.0}{9.0}$$

Your program should output in the form:

```
F: <fahrenheit> C: <celsuis>
```

Where `<fahrenheit>` and `<celsius>` are the values printed.

For example the default temperature of 26 degress Fahrenheit will print 24.44.

```
F: 76.00 C: 24.44
F: 76.00 C: 24.44
```

If the temperature changes to 79 degress Fahrenheit, the following is printed.

```
F: 76.00 C: 24.44
F: 79.00 C: 26.11
```

Brackets in calculations

It's helpful to add brackets to calculations to specify the order, and remember to store the variables as a `float`!

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  // your code goes here!
}
```

Testing

- [ ] Testing the first example.
- [ ] Testing the second example.
- [ ] Testing 101 degrees fahrenheit.
- [ ] Testing 33 degrees fahrenheit.
- [ ] Testing a range of values.
- [ ] Testing a hidden case.

5

LOOPING

5.1. Loops

5.1.1. The for loop

When we write computer programs, often we want to do things repeatedly.

For example on the Arduino Esplora - if we wanted to fade the LED up we could do something like this:

```
#include <Esplora.h>
void setup() {}
void loop() {
  Esplora.writeRed(0);
  Esplora.writeRed(1);
  Esplora.writeRed(2);
  Esplora.writeRed(3);
  // insert 250 more lines
  Esplora.writeRed(254);
  Esplora.writeRed(255);
}
```

That takes a lot of typing, and takes up a lot of lines of code that do essentially the same thing. Here's how we do the same thing in 3 lines.

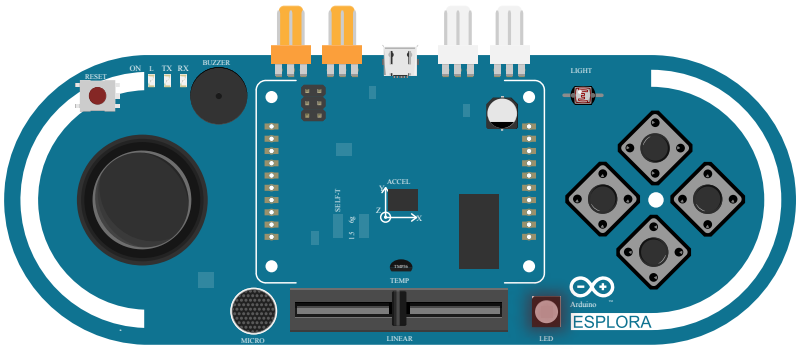
```
for (int i = 0; i < 256; i++) {
  Esplora.writeRed(i);
}
```

On the next slide we'll see what all these parts mean.

5.1.2. Fade up led

This is an example of running the for loop to fade up the red LED's brightness. Run the code and see it working on the simulated Esplora!

```
#include <Esplora.h>
void setup() {}
void loop() {
  for (int i = 0; i < 256; i++) {
    Esplora.writeRed(i);
    delay(15);
  }
}
```



Let's take a look at what each part of the for loop statement actually means...

- `int i = 0` - this is the **start condition** of the for loops, creates a new integer, `i` and sets it to `0`.
- `i < 256` - exit condition, while this condition is `true`, the for loop will continue looping
- `i++` - add 1 to `i` at the end of each iteration

The for loop will loop 256 times (where `i` is `0` to `255`).

5.1.3. More operators

In the last slide we used the symbol `i++` to add 1 to i at the end of each for loop iteration. These operators (https://www.arduino.cc/en/Reference/IncrementCompound) are really useful way to change a variable.

There are more, but the ones we use in this course are below.

Operator	Function	Usage	Meaning
`++`	add 1	`i++`	`i = i+1`
`--`	subract 1	`i--`	`i = i-1`
`+=`	add amount	`i += 5`	`i = i+5`
`-=`	subtract amount	`i -= 10`	`i = i-10`

Run the example to see them working!

```
void setup() {
  Serial.begin(9600);
}
int going_up = 0;
int doooown = 1500;
void loop() {
  going_up += 5;
  doooown -= 12;
  Serial.print("up: ");
  Serial.print(going_up);
  Serial.print("\tdown: ");
  Serial.println(doooown);
  delay(200);
}
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
up: 5 down: 1488
up: 10  down: 1476
up: 15  down: 1464
up: 20  down: 1452
up: 25  down: 1440
up: 30  down: 1428
up: 35  down: 1416
up: 40  down: 1404
up: 45  down: 1392
up: 50  down: 1380
```

Special characters

The code above uses the `\t` character to print a horizontal tab.

```
Serial.print("\tdown: ");
```

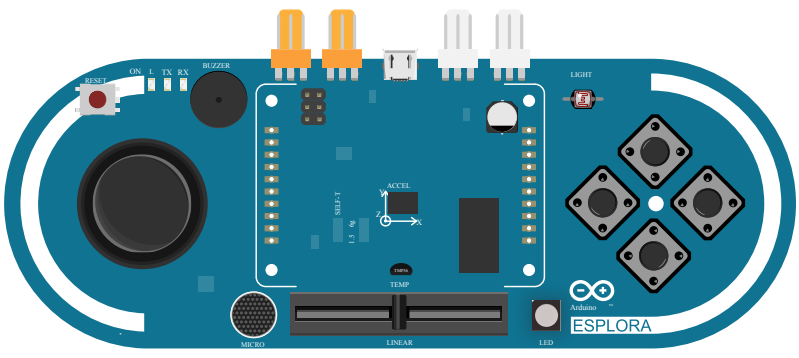
There are other special characters that are useful, like `\n` to print a new line.

5.1.4. For loop example

The for loop works exactly the same as an if statement in regards to the brackets.

It works on a whole *block of code*, so the for loop can perform operations on multiple lines!

```
#include <Esplora.h>
void setup() {}
void loop() {
  for (int i = 0; i < 256; i++) {
    // every line between these brackets
    // is inside the for loop
    Serial.print("Setting the red led to: ");
    Serial.println(i);
    Esplora.writeRed(i);
    delay(20);
  }
}
```



```
Setting the red led to: 241
Setting the red led to: 242
Setting the red led to: 243
Setting the red led to: 244
Setting the red led to: 245
Setting the red led to: 246
Setting the red led to: 247
Setting the red led to: 248
Setting the red led to: 249
Setting the red led to: 250
Setting the red led to: 251
Setting the red led to: 252
Setting the red led to: 253
Setting the red led to: 254
Setting the red led to: 255
```

If you run the following code, it also prints out the variable `i` to the serial console, so you can see how many times the for loop has already run.

5.1.5. Problem: Made to fade!

Jim wants to make a new light for his entertainment room, and has asked you to prototype an LED that slowly fades up in brightness when activated.

Write a `for` loop to fade up the LED in **magenta** from 0 to 255 *after button 1 is pushed*. Each iteration should have a delay of *20 milliseconds*.

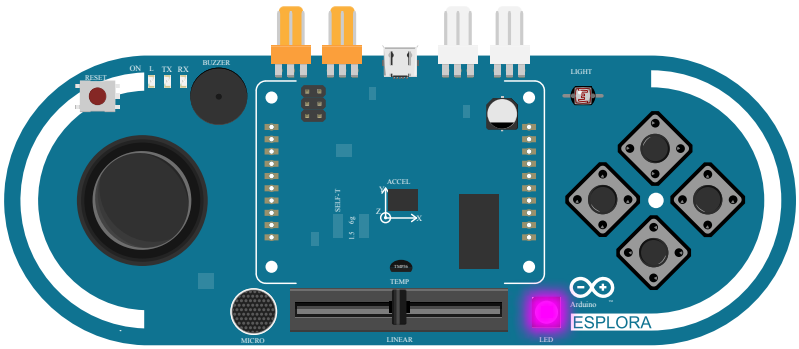
Remember the function of the for loop:

```
for (int i = 0; i < 500; i++) {
  // will loop 500 times
  // i starts at 0 and goes to 499
  delay(20);
}
```

There are **256** possible brightness levels on the LED.

Once the led has faded up to full brightness, it should stay on until the button is pressed again, when it should fade up again.

An animation of the device working is below.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED is off to start with.
- [ ] Testing the LED turns on when a button is pressed.
- [ ] Testing the LED turns on when button 1 is pressed.
- [ ] Testing the right colour turns on.
- [ ] Testing the start condition.
- [ ] Testing the loop.
- [ ] Testing the delay.

- [ ] Testing end condition.
- [ ] Testing multiple button presses.

5.1.6. Double for!

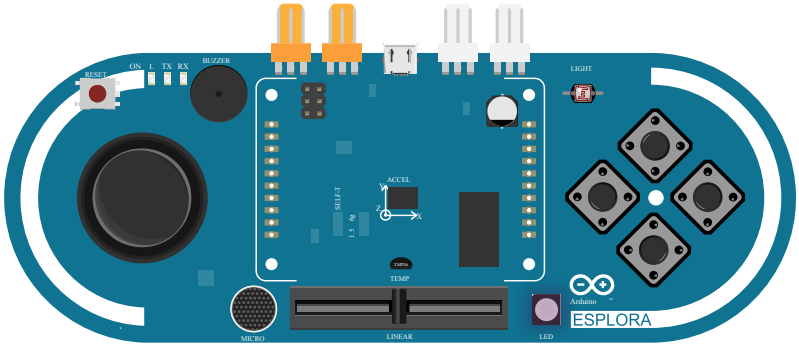
Once a **`for loop`** has completed, the code continues executing until it hits the end of the **`loop`** function.

It's possible to add multiple **`for loops`** to perform different operations.

Look at the example below that fades up *two* colours, magenta and yellow.

Run the code below to see it in action!

```
#include <Esplora.h>
void setup() { }
void loop() {
  for (int i = 0; i < 256; i++) {
    Esplora.writeRGB(i, 0, i);
    delay(20);
  }
  for (int i = 0; i < 256; i++) {
    Esplora.writeRGB(i, i, 0);
    delay(20);
  }
}
```



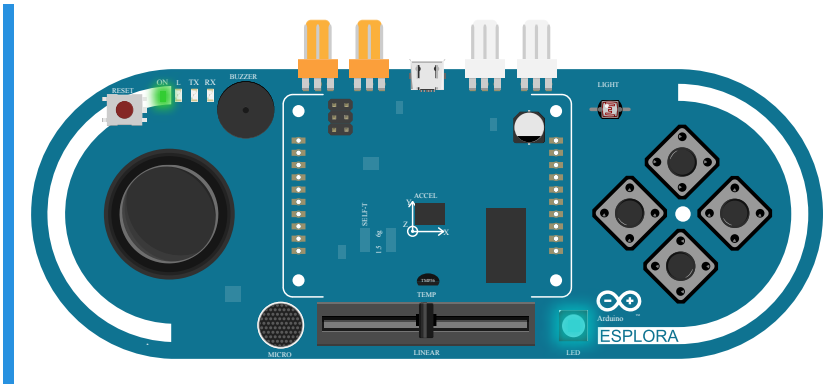
5.1.7. Problem: Triple-for!

Your task is to write an LED animation using for-loops.

Your program should fade up the colour yellow, then cyan, then blue. Each colour should loop from 0 to 255, and once all the colours have completed it should start over again.

The delay between each update to `writeRGB` should be 20 milliseconds.

Here's an animation of the program running:



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the LED turns yellow.
- [ ] Testing yellow start condition.
- [ ] Testing the yellow loop and the delay is not too short.
- [ ] Testing yellow end condition.
- [ ] Testing the yellow delay.
- [ ] Testing the LED turns cyan.
- [ ] Testing cyan start condition.
- [ ] Testing the cyan loop.
- [ ] Testing cyan end condition.
- [ ] Testing the cyan delay.
- [ ] Testing the LED turns blue.
- [ ] Testing blue start condition.
- [ ] Testing the blue loop.
- [ ] Testing blue end condition.

- [ ] Testing the blue delay.
- [ ] Testing it loops again.

5.2. Absolutely!

5.2.1. The abs function

In maths, the *absolute value* just means to *remove the negative sign*. On the Arduino we can do this with the **abs** function (https://www.arduino.cc/en/Reference/Abs).

If a number has a negative sign, the abs function removes it. If it's positive already, the number stays the same.

```
void setup() {
  Serial.begin(9600);
  int x = -50;
  Serial.print("x: ");
  Serial.print(x);
  Serial.print(" abs(x): ");
  Serial.println(abs(x));
}
void loop() { }
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
x: -50 abs(x): 50
```

5.2.2. Using your abs

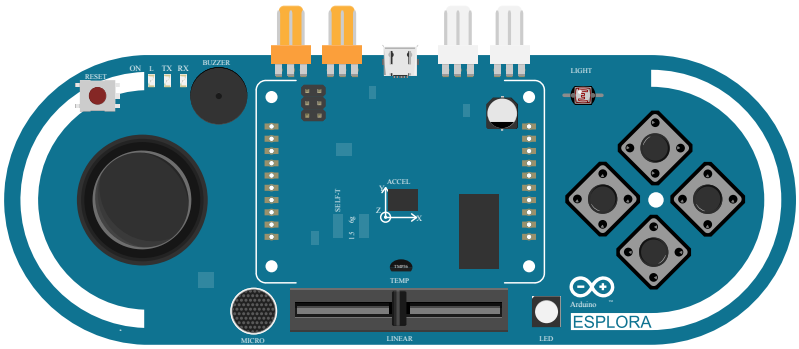
The **abs** function can be useful when determining the *magnitude* of a number, which is an indication of how far away from 0 a number is.

100 is 100 away from zero. But -100 is also 100 away from 0. See how this is the same as removing the negative sign?

The joysticks on the Esplora gives both a positive *and* negative number, which makes it easy to tell which way the joystick has moved.

If you want to just check that *x* **or** *y* has moved more than 100 from 0, do so the following way.

```
#include <Esplora.h>
void setup() { }
void loop() {
  int x = Esplora.readJoystickX();
  int y = Esplora.readJoystickY();
  if (abs(x) > 100 || abs(y) > 100) {
    Serial.print("The joystick moved! x: ");
    Serial.print(x);
    Serial.print(" y: ");
    Serial.println(y);
  }
  delay(100);
}
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
```

Move the joystick around to print its value!

5.3. Looping the other way

5.3.1. Going backwards

It's possible to make for loops go in the opposite direction.

This can be done by setting the **start condition** to be a number that isn't 0, and set the last value to subtract instead. Let's see an example!

```
for (int i = 10; i >= 0; i--) {
  // will loop from 10 to 0
}
```

Notice how the last value has `i--` to *decrement* `i` at the end of each iteration. Run the code below to see it working!

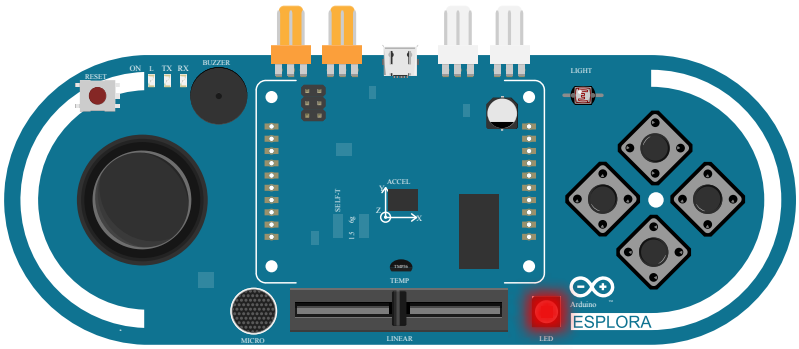
```
void setup() {
  Serial.begin(9600);
  for (int i = 10; i >= 0; i--) {
    Serial.println(i);
  }
}
void loop() { }
```

```
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
10
9
8
7
6
5
4
3
2
1
0
```

5.3.2. Fade down

For example if we wanted to fade an LED from bright to dark we could do the following:

```
#include <Esplora.h>
void setup() {}
void loop() {
  for (int i = 255; i >= 0; i--) {
    Esplora.writeRed(i);
    delay(15);
  }
}
```



- `int i = 255` start condition is 255
- `i >= 0` continue while `i` is greater than or equal to 0
- `i--` subtract 1 from `i` each time

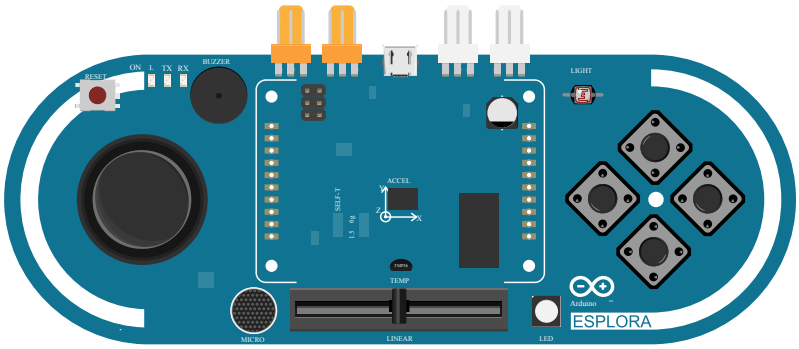
5.3.3. Problem: Bring it down

Sometimes we want to fade-out something. This means when something starts at a *maximum* value, then slowly reduces in value.

This can be done with a `for` loop.

Write a program to fade down the green LED from full brightness down to 0. Apply a **20 millisecond** after each time the brightness of the LED is changed.

Activate fade-out only after the *joystick y* moves 100 from the 0 position in either direction.



Looping down

Making a `for` loop decrease can be done in the following way:

```
for (int i = 1000; i >= 0; i--) {
  // loops from 1000 to 0
}
```

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing nothing happens initially.
- [ ] Testing the LED turns on when the joystick is moved.
- [ ] Testing the colour is correct.
- [ ] Testing the LED stays off when joystick X is moved.
- [ ] Testing the LED turns on when joystick Y is moved.
- [ ] Testing the joystick positive threshold.
- [ ] Testing the joystick negative threshold.
- [ ] Testing the loop start condition.
- [ ] Testing the loop.

- [ ] Testing the loop end condition.
- [ ] Testing the delay.
- [ ] Testing it loops only once when the joystick is moved.
- [ ] Testing it loops again.

5.3.4. Changing the loop

A for loop doesn't have to always start at 0 and go up by 1 each time. We can set other values as the *start condition* and the *increment* of a for loop.

Let's look at starting the for loop from a number other than 0.

```
void setup() {
  Serial.begin(9600);
  for (int i = 500; i < 550; i++) {
    Serial.println(i);
  }
}
void loop() { }
```

```
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
```

Try changing the numbers to see how the for loop runs!

5.3.5. Incrementing

It's also possible to change how a for loop increments and decrements at the end of each iteration.

It's possible to use the following operators instead of `i++` or `i--`

Operator	Function	Usage	Meaning
`+=`	add amount	`i += 5`	`i = i + 5`
`-=`	subtract amount	`i -= 10`	`i = i - 10`

Going up...

```
void setup() {
  Serial.begin(9600);
  for (int i = 500; i < 1000; i+=50) {
    Serial.println(i);
  }
}
void loop() { }
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
500
550
600
650
700
750
800
850
900
950
```

...and down...

```
void setup() {
  Serial.begin(9600);
  for (int i = 700; i >= 0; i-=70) {
    Serial.println(i);
  }
}
void loop() { }
```

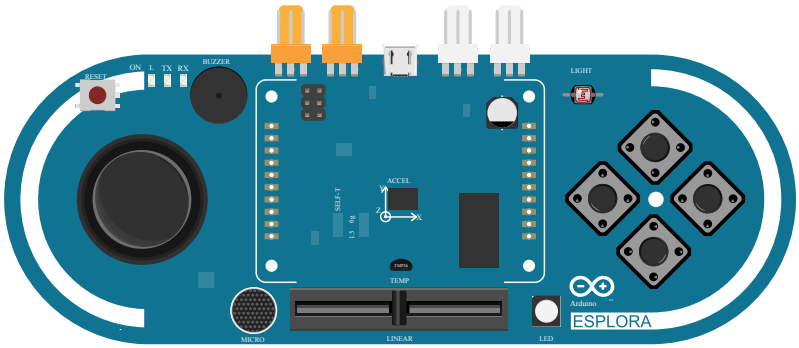
```
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
700
630
560
490
420
350
280
210
140
70
0
```

5.3.6. Making a tone

To make a sweeping tone, we can use a for loop and change the start and increment values.

Change some of the numbers in the program below!

```
#include <Esplora.h>
void setup() { }
void loop() {
  if (Esplora.readButton(1) == PRESSED) {
    for (int i = 500; i < 700; i+=10) {
      Esplora.tone(i);
      delay(10);
    }
    Esplora.noTone();
  }
}
```



5.3.7. Problem: Make a siren!

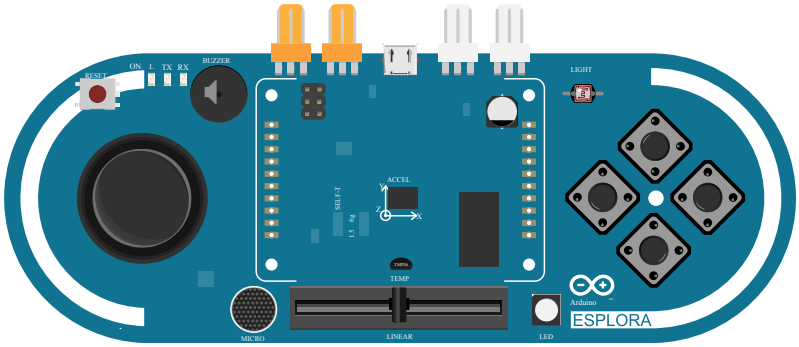
The AFP were very happy with your work on their siren lights, they've now asked you to make a sound for it as well!

Write a siren that starts at 700 Hz, and goes to 995 Hz. It should then loop from 1000 Hz back to 700 Hz.

In both cases it should change by 5Hz each time, and each note should last for *20 milliseconds* before the next note starts.

The siren should activate when *joystick x* moves over 100 from it's start position, and when the joystick is in the 0 position there should be no tone after the siren has completed.

Here is what it should sound like at the end. Click play to run through the tone it should create once.



You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing nothing happens initially.
- [ ] Testing a tone is emitted when the joystick moves.
- [ ] Testing a tone is emitted when joystick X moves.
- [ ] Testing the joystick threshold.
- [ ] Testing a negative joystick X value.
- [ ] Testing the joystick negative threshold.
- [ ] Testing the first tone is 700 Hz.
- [ ] Testing increment is not 1.
- [ ] Testing increment is not 2.
- [ ] Testing increment is not 3.
- [ ] Testing increment is not 4.
- [ ] Testing increment is 5.

- [ ] Testing the first loop.
- [ ] Testing the delay.
- [ ] Testing the end.
- [ ] Testing the second loop start condition.
- [ ] Testing second loop.
- [ ] Testing second loop delay.
- [ ] Testing there is no tone after the joystick is at 0 again.
- [ ] Testing it loops again.

6

ARRAYS

6.1. Arrays

6.1.1. Arrays

Say for example we wanted to store some notes to play on the Arduino Esplora. With everything we've learnt so far, you might think to do it this way:

```
int note_1 = 349; // F
int note_2 = 370; // F#
int note_3 = 392; // G
int note_4 = 415; // G#
int note_5 = 440; // A
...
```

So you would be storing a new variable for each note.

But what if we wanted to store 20 or 100 notes? That would be a lot of typing!

There is a simpler way. We could use an array. An array enables us to store a number of values of the same type.

6.1.2. Declaration

An array is another variable that can store a collection of data of a given type. So you can store an array of integers, or an array of any type of data.

We can declare an array the following way:

```
// delcare an array of integers with 4 elements
int notes[4];
```

We can also initialise the value of each element like this:

```
int notes[4] = {370, 392, 415, 440};
```

It is possible to *access* each element in the array by using its index:

Index	Element	Access
0	370	`notes[0]`
1	392	`notes[1]`
2	415	`notes[2]`
3	440	`notes[3]`

Run the following program to access and print each element in the array notes:

```
int notes[4] = {370, 392, 315, 440};
void setup() {
  Serial.begin(9600);
  Serial.println("The elements of notes are:");
  Serial.println(notes[0]);
  Serial.println(notes[1]);
  Serial.println(notes[2]);
  Serial.println(notes[3]);
}
void loop() {
  delay(5000);
}
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
The elements of notes are:
370
392
315
440
```

6.1.3. Using a loop!

It's even simpler to access each element in an array using a *for loop*!

```
int notes[4] = {370, 392, 415, 440};
for (int i = 0; i < 4; i++) {
  Serial.println(notes[i]);
}
```

Remember how the `for` loop works. Let's see it running!

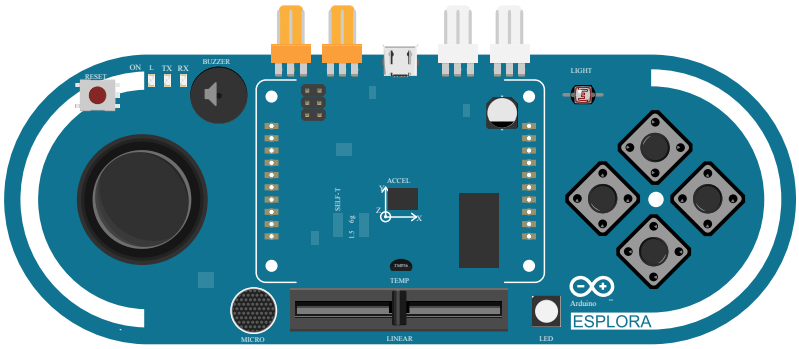
```
int notes[4] = {370, 392, 415, 440};

void setup() {
  Serial.begin(9600);
  Serial.println("The notes are:");
  for (int i = 0; i < 4; i++) {
    Serial.println(notes[i]);
  }
}
void loop() { }
```

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
The notes are:
370
392
415
440
```

You can also play the notes exactly the same way!

```
#include <Esplora.h>
int notes[4] = {370, 392, 415, 440};
void setup() {
  for (int i = 0; i < 4; i++) {
    Esplora.tone(notes[i]);
    delay(100);
  }
}
void loop() { }
```



6.1.4. Problem: Play some notes!

Your music teacher wants to test some notes on the banjo, and wants to compare the *fundamental* frequency to the sounds produced by the Esplora board.

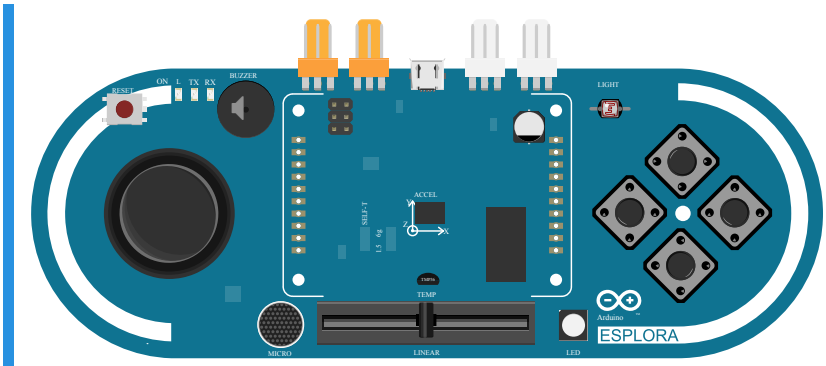
Write a program so that when you press *button 1*, all 6 notes in the array **`notes`** are played with a delay of *500 milliseconds* before the next note starts.

The last note is 0, which should turn off the speaker.

The notes to play are stored in the **`notes`** array:

```
int notes[6] = {262, 277, 294, 311, 330, 0};
```

Below is an animation of the device after a button is pressed.



You'll need

sketch.ino

```
#include <Esplora.h>
int notes[6] = {262, 277, 294, 311, 330, 0};
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing nothing happens to start with.
- [ ] Test any tone emitted after a button is pressed.
- [ ] Test any tone emitted after button 1 is pressed.
- [ ] Testing the first tone.
- [ ] Testing the second tone.
- [ ] Testing the third tone.
- [ ] Testing the fourth tone.
- [ ] Testing the fifth tone.
- [ ] Testing the last tone.
- [ ] Testing all tones.
- [ ] Testing the delay.

6.1.5. Short tone

The `tone` function has an optional argument for how long the tone is to be played, i.e. the *duration*. For example, it is possible to make a tone last for one second, while the program keeps executing. The following code demonstrates how to do this:

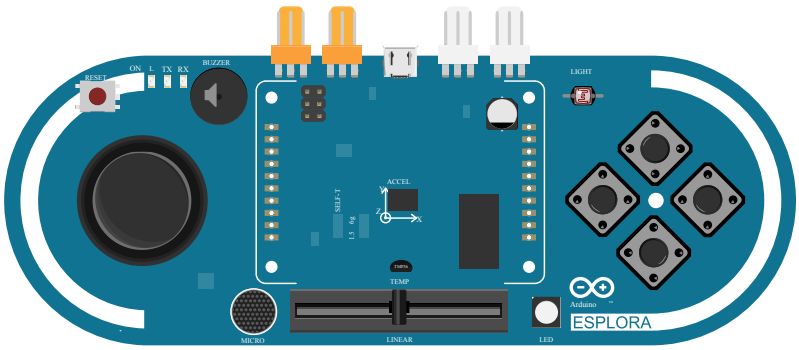
```
Esplora.tone(500, 1000);
```

The line above will emit a tone of 500Hz for 1000 milliseconds, or one second.

6.1.6. Problem: Sound-activated scales!

Octave 4 on the musical scale start at note middle C (262 Hz) and finishes at middle B (494 Hz). Write a program to play through the scale when you click your fingers. The notes are given to you in the array, `notes`.

Once you make a sound greater than **300**, you should play through the scales stored in the notes array. Each note should play for *250 milliseconds*, and have a delay of *half a second* before the next note starts.



Activate microphone

Remeber to click the microphone to simulate making a sound your device output on the right. The box above is an animation, and does not enable this functionality.

You'll need

sketch.ino

```
#include <Esplora.h>
int notes[12] = {262, 277, 294, 311, 330, 349, 370, 392, 415, 440, 466, 494};
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the first note is played.
- [ ] Testing the second note.
- [ ] Testing the third note.
- [ ] Testing the fourth note.
- [ ] Testing the fifth note.
- [ ] Testing all notes.
- [ ] Testing sound activation.
- [ ] Testing the notes change every half second.
- [ ] Testing each note lasts 250 ms.
- [ ] Testing multiple sounds.

6.2. Choosing colours

6.2.1. Looping with maths

Sometimes we need to loop a given number of times, but the thing we want to do in the loop is not always exactly the same as the iteration number.

Let's take the example of accessing an array - but choosing each element in an array with one of the buttons. The values we need are in the table below.

Index	Button
0	1
1	2
2	3
3	4

6.2.2. Button selection

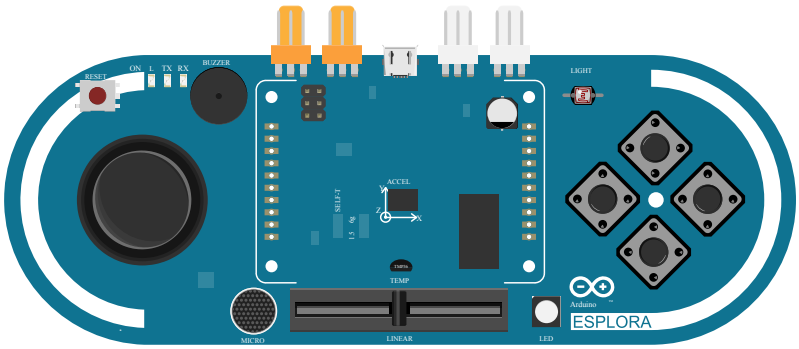
Let's take a look at the following `for` loop to access each element in the array with the buttons.

```
for (int i = 0; i < 4; i++) {
  // will print 1, 2, 3 and 4 on a new line
  Serial.println(i+1);
}
```

The above loop prints 1 to 4, but the `i` variable is from 0 to 3.

Run the code below to see this for loop in action.

```
#include <Esplora.h>
int numbers[] = {100, 200, 300, 400};
void setup() {
  Serial.begin(9600);
}
void loop() {
  for(int i = 0; i < 4; i++) {
    // i is 0, 1, 2 then 3
    // i+1 will be 1, 2, 3, then 4
    if (Esplora.readButton(i+1) == PRESSED) {
      Serial.print("Element ");
      Serial.print(i);
      Serial.print(" is ");
      Serial.println(numbers[i]);
    }
  }
}
```




```
Compiling sketch.ino...
===info |||  Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info |||  Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
```

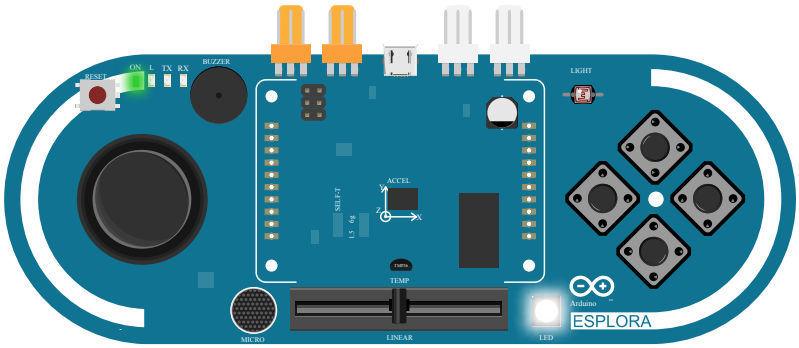
6.2.3. Problem: Select the brightness!

You've been asked to created a torch with different levels of brightness, selected by the different buttons according to the table below. The colour of the LED should be **white**.

Button	Brightness
1	0
2	100
3	200
4	255

Once a button has been pressed, the LED should remain at that brightness until another button is pressed.

Write a program so that each button chooses the brightness from the array! Below is an animation of the different brightness levels.



You'll need

sketch.ino

```
#include <Esplora.h>
int colours[4] = {0, 100, 200, 255};
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

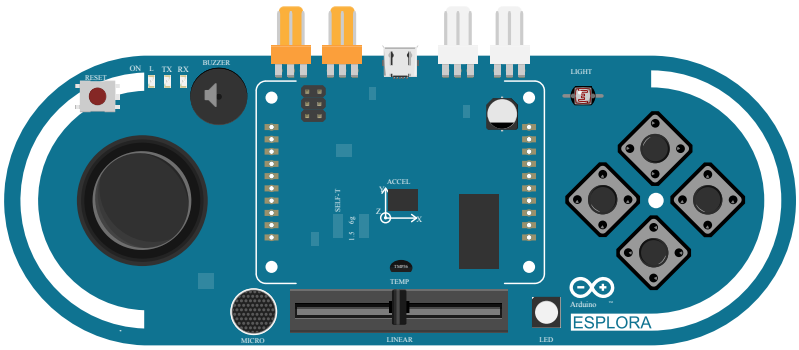
- [ ] Testing the torch is off initally.
- [ ] Testing the torch is off when button 1 is pressed.
- [ ] Testing the torch is on when any other button is pressed.
- [ ] Testing the colour.
- [ ] Testing the brightness when button 2 is pressed.
- [ ] Testing the brightness when button 3 is pressed.
- [ ] Testing the brightness when button 4 is pressed.
- [ ] Testing multiple button presses.

6.2.4. Problem: Choose the note!

For music class, you have been asked to create an instrument that plays the following notes when a specific button has been pressed. It should play a tone when the slider is *greater than* 750, otherwise no music should play. The notes are already in the program" in the `notes` array!

Button	Note
1	277
2	311
3	370
4	415

Once a button has been pressed, your program should continue playing the tone until another button is pressed or the slider moves to less than or equal to 750.



Note!

Greater than 750 is on the left hand side of the slider. Move the slider all the way to the left for the notes to play.

You'll need

sketch.ino

```
#include <Esplora.h>
int notes[4] = {277, 311, 370, 415};
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the device does nothing when no buttons are pressed.
- [ ] Testing for sound when any button is pushed and the slider has moved.
- [ ] Testing the slider threshold.
- [ ] Testing button 1.
- [ ] Testing button 2.
- [ ] Testing button 3.
- [ ] Testing button 4.

- [ ] Testing multiple button presses.

6.3. The map function

6.3.1. Accessing arrays

To access an array, remember we need to get the desired *index*.

So to access element 5 from the array, `my_array`, would be done like this:

```
my_array[4];
```

But sometimes it's there are more elements than sensors. For example, if there are more than 4 elements, we can't use the buttons.

Remember how some of the sensors give numbers from 0 to 1023?

We can scale this down to the number we need with the **`map`** function (https://www.arduino.cc/en/Reference/Map)!

6.3.2. The map function

The map function, takes a *value* and converts from a give *input range* to an *output range*.

The map function takes 5 numbers.

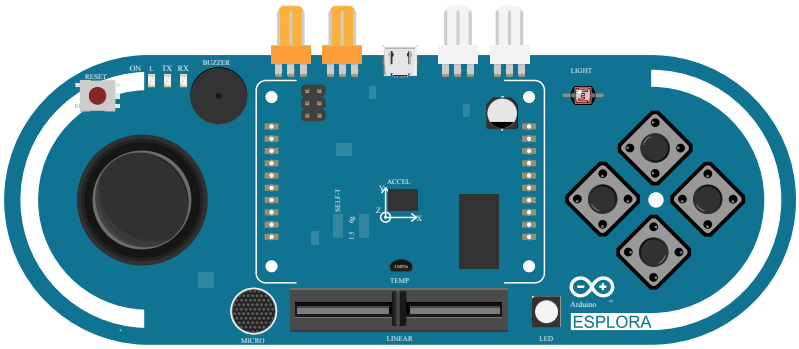
Argument	Meaning
1	input value
2	min input value
3	max input value
4	min output value
5	max output value

```
map(value, in_min, in_max, out_min, out_max);
```

Let's run an example with the slider! We take the input value, 0 to 1023, and convert it from 0 to 6, to access the desired element in the **`numbers`** array!

Move the slider to see the Esplora print the **`numbers`** from the array!

```
#include <Esplora.h>
int numbers[7] = {111, 222, 333, 444, 555, 666, 777};
void setup() {
  Serial.begin(9600);
}
void loop() {
  int slider = Esplora.readSlider();
  int index = map(slider, 0, 1023, 0, 6);
  Serial.println(numbers[index]);
  delay(500);
}
```



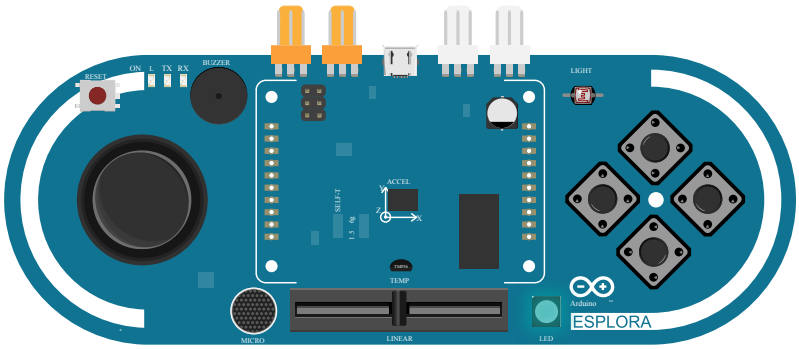

```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
444
444
```

6.3.3. Seeing it work

Let's take a look at another example which uses the slider to set the LED to be CYAN of a given colour.

In this example, moving the slider will set the LED to a predefined brightness of CYAN by accessing different elements in the `bright` array!

```
#include <Esplora.h>
int bright[] = {0, 50, 100, 150, 200, 255};
void setup() { }
void loop() {
  int slider = Esplora.readSlider();
  int index = map(slider, 0, 1023, 0, 5);
  int brightness = bright[index];
  Esplora.writeRGB(0, brightness, brightness);
}
```



6.3.4. Problem: Automatic torch upgrade!

Schultz Security Serivces want to give the automatic torch an upgrade! They found the light sensor does not respond at the right rate to the amount of light, and have asked you to fix it!

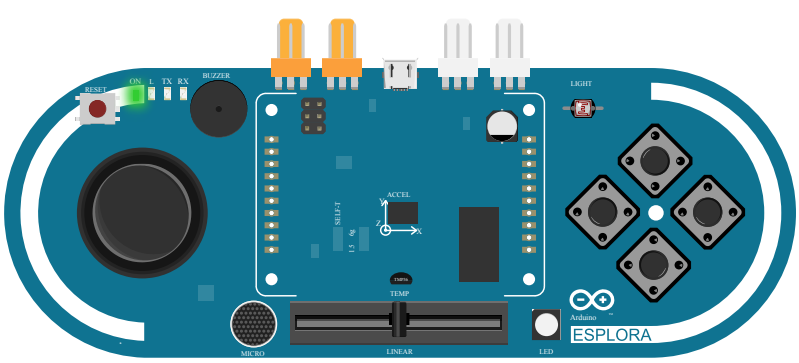
The light from the light sensor is from 0 to 1023, they've asked you to use the `map` function to divide the incoming light to 7 sections. The following array stores the brightness of the torch for each section.

```
int torch[7] = {255, 255, 255, 230, 190, 70, 0};
```

Note, the automatic torch means that when there is a *lot* of light (a high number from the light sensor), the torch will be not be very bright.

But when there is not much light, the torch will be brighter.

Here an animation of the light changing as the light reduces.



Hint

Use the `readLightSensor` function to read from the light sensor.

`map` has the following inputs:

```
map(value, in_min, in_max, out_min, out_max)
```

You'll need

sketch.ino

```
#include <Esplora.h>
int torch[7] = {255, 255, 255, 230, 190, 70, 0};
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing the led is off at maximum light.
- [ ] Testing the led is on with no light.
- [ ] Testing the colour is correct.
- [ ] Testing the torch is full brightness with no light.
- [ ] Testing section light in section 3.
- [ ] Testing section light in section 5.

- [ ] Testing section light in section 2.
- [ ] Testing section light in section 1.
- [ ] Testing section light in section 4.
- [ ] Testing section light in section 7.
- [ ] Testing section light in section 6.
- [ ] Testing a range of light values.

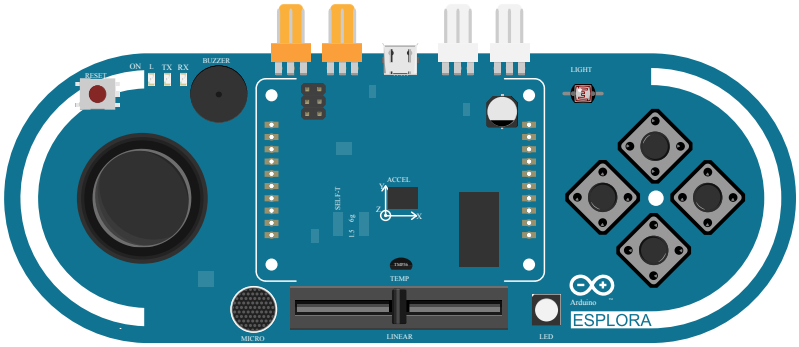
6.3.5. Flip the slider!

It's possible to *flip* the slider around by changing the order of the numbers.

To flip the slider, swap the second and third numbers around. This has been done for you in the program below.

Move the slider and look how the numbers change!

```
#include <Esplora.h>
void setup() {
  Serial.begin(9600);
}
void loop() {
  int slider = Esplora.readSlider();
  int index = map(slider, 0, 1023, 0, 20);
  int flipped = map(slider, 1023, 0, 0, 20);
  Serial.print("normal: ");
  Serial.print(index);
  Serial.print(" flipped: ");
  Serial.println(flipped);
  delay(500);
}
```




```
Compiling sketch.ino...
===info ||| Sketch uses {0} bytes ({2}%%) of program storage space. Maximum is {1} by
===info ||| Global variables use {0} bytes ({2}%%) of dynamic memory, leaving {3} byt
Compiling simulator...
Running simulator...
normal: 10 flipped: 9
normal: 10 flipped: 9
```

7

PROJECT: MAKING MUSIC

7.1. Project: Making Music

7.1.1. Making Music

We've learnt a lot of different concepts so far, nice work!

Now it's time to put them together and make our own musical instrument! You'll be able to choose what notes you play, and amaze your friends by programming your very own instrument!

7.1.2. Problem: Project: Make an instrument!

The fourth octave starts at the note C4 - 262 Hz (middle C), and finishes at B4 which has a frequency of 494 Hz. All the notes and their frequencies can be found here (http://pages.mtu.edu/%7Esuits/notefreqs.html).

The notes in the fourth octave are stored in the **notes** array.

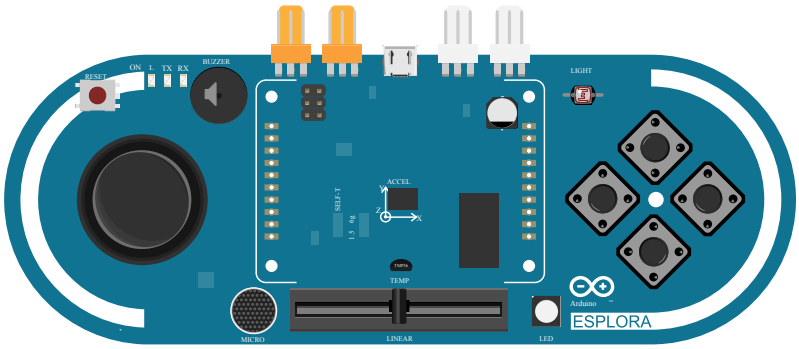
```
int notes[12] = {262, 277, 294, 311, 330, 349, 370, 392, 415, 440, 466, 494};
```

You have been asked to create a musical instrument that plays all the notes from the fourth octave, and the *slider* can be moved to select the correct note. The *left* most position of the slider should play the *first* note (262 Hz), and the *right* most position should play the last note (494 Hz).

Music should only play when *button 1* is pressed, and should stop when the button has been released. If *button 2* is pressed, the note should play the *fifth* octave by *doubling* the frequency of the note that would play if button 1 was pressed. If **both** buttons are pressed, button 1 takes precedence.

You should use the **map** function to scale the slider to select the correct note.

An animation of the slider moving from left to right can be played by clicking play!



You'll need

sketch.ino

```
#include <Esplora.h>
int notes[12] = {262, 277, 294, 311, 330, 349, 370, 392, 415, 440, 466, 494};
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] Testing nothing happens initially.
- [ ] Testing a sound is emitted when any button is pressed.
- [ ] Testing a sound is emitted when button 1 is pressed.
- [ ] Testing the tone turns off when the button is released.
- [ ] Testing the slider at 512.
- [ ] Testing the slider at 0.
- [ ] Testing the slider at 1023.

- [ ] Testing the slider at 250.
- [ ] Testing the slider at 750.
- [ ] Testing a range of slider values.
- [ ] Testing the tone when button 2 is pressed.
- [ ] Testing a range of slider values on button 2.
- [ ] Testing buttons pressed and released.
- [ ] Testing a range a values with multiple button presses.
- [ ] Testing a hidden case.

7.1.3. Congratulations!

Congratulations on finishing the Arduino course!!

You've come a long way, and just between you and me you've managed to finish the hardest course by the Australian Computing Academy, so well done!

We hope you enjoyed this course, and we hope it gives you a bit of an idea as to how devices we use everyday are made!

7.1.4. Problem: Arduino Playground

What's this? You thought you were finished!

Have a go at writing your own instrument! You can write whatever code you like in this question.

> **Save or submit your code**
>
> There are no points to be earned for this question, so you can submit whatever code you like. Make sure you save programs that you want to keep!

You'll need

sketch.ino

```
#include <Esplora.h>
void setup() {}
void loop() {
  // your code goes here
}
```

Testing

- [ ] **This is the Arduino Playground, there is no right or wrong answer!**