Freetronics Experimenters Kit

Project Guide

Getting Started	2
Parts Guide	3
Software Setup	7
Project 1: Controlling An LED	8
Project 2: Controlling 8 LEDs	12
Project 3: Reading Digital (On/Off) Input	17
Project 4: Reading Analog (Variable) Input	21
Project 5: Dimming LEDs Using PWM	25
Project 6: Making Things Move With Servos	29
Project 7: RGB LED	31
Project 8: Drive More Outputs With A Shift Register	34
Project 9: Making Sounds	40
Project 10: Detecting Vibration And Knocks	42
Project 11: Light Input Controlling Sound Output	45
Recommended Tools For Advanced Projects	47
More Information And Getting Help	48
Reading Resistor Colour Codes	49

Revision 1.2 / September 2013

Please check for updates and corrections to this guide at:
www.freetronics.com/expkit

Getting Started

For most people, electronics is just like magic. We have little magic boxes called telephones that let us talk to each other. We have big magic boxes called televisions that let us watch moving pictures on a screen. Our cars have magic boxes hidden inside them to control how the engine runs. We have small magic boxes attached to the wall that measure the temperature and control our heating and cooling, so our houses stay comfortable.

But those magic boxes aren’t really magic at all. They’re just carefully designed machines that do a certain job, and if you want to learn how they work, or to make them work differently, or even to design your own, you can do it. By starting with some basic projects and learning the principles of how various electronic parts work, you can create your very own inventions and designs.

Most modern electronic devices are actually a combination of both hardware and software, and that’s where Arduino comes in. Arduino boards are tiny computers that let you build your own hardware projects, and then run small programs called “sketches” to make that hardware do different things.

This combination of hardware and software is what makes electronic devices seem so magical, and it’s what makes them both flexible and powerful. The hardware provides the inputs and the outputs, while the software controls how the device will behave.

This guide will help you get started with your first adventures in electronics, but there’s a whole world of opportunities to explore if you want to take things further.

Most importantly, we hope you have fun and learn lots of interesting things along the way!

Getting Help: The Freetronics Forum

If you have any questions or get stuck on any of these projects, a great place to ask for help is the Freetronics Forum:

forum.freetronics.com

Updates To This Guide

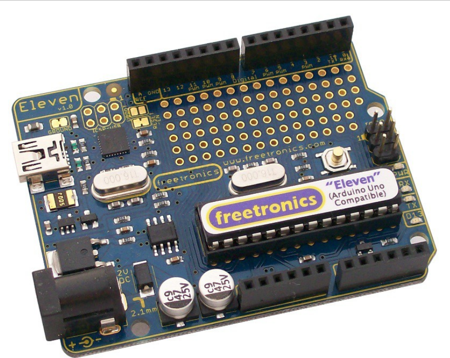
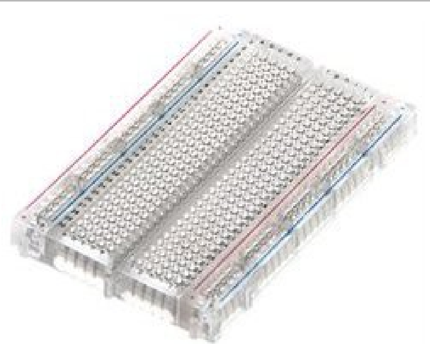
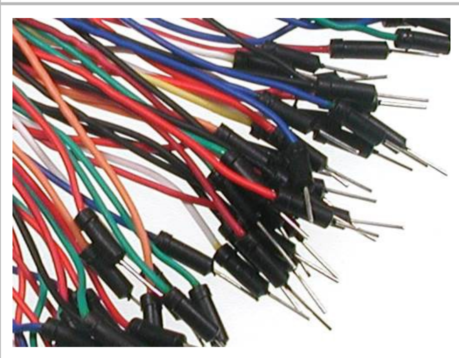
The latest version of this guide is always available online along with source code for all the projects, so make sure you check it out to see if there have been any updates or new projects added:

www.freetronics.com/expkit

“Arduino” is a registered trademark of Arduino.cc

Parts Guide

The Freetronics Experiments Kit comes with a variety of parts that will help you get started with electronics. You can use the parts for the projects in this guide, and later use them in your own projects.

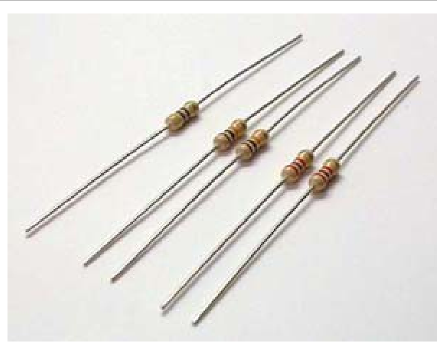
	**“Eleven” Arduino-Compatible Microcontroller Board** The brains of the projects in this guide. The Eleven is a tiny computer that can run special programs called “sketches”. You write sketches on your regular PC running Windows, MacOS, or Linux, and then upload them to the Eleven using the supplied USB cable. The Eleven has a variety of input and output connections arranged in convenient headers on the edges, allowing you to hook it up to your projects. www.freetronics.com/eleven
	**Solderless Breadboard** Great for quickly plugging parts together without needing a soldering iron. By plugging in jumper wires and component leads you can build simple circuits on a breadboard very quickly. Under the holes in the breadboard are tiny sockets that make connections to whatever you plug in, and also links them together in rows across the breadboard. By plugging two or more leads or jumper wires into the same row they will be connected together inside the breadboard. www.freetronics.com/breadboard
	**Breadboard Jumper Wires** We’ve provided a big bundle of jumper wires that you can use to connect your Arduino to your breadboard, and to make connections between different parts of the breadboard.

	Red and Green LEDs (Light Emitting Diodes) LEDs are a great way to provide visual feedback from your projects. You can use them to show status, or to show that your project has detected a certain event or input, or for lighting effects or displays. We've provided a selection of red and green LEDs, and you can also buy blue, white, yellow, orange, and other colours. LEDs can easily be burned out if you let too much current flow through them, so make sure you use a "current limiting resistor" as explained in Project #1. LEDs are "polarised", which means they only work one way around: one lead needs to be connected to positive, and the other to negative. You can tell the leads apart because the positive lead is longer. The LED also has a small flat spot on the body next to the negative lead, which is handy if the leads have been cut to the same length.
	RGB LED Most LEDs are just one colour, but RGB LEDs are special: they combine three different LEDs (one each of red, green, and blue - hence the name) into a single physical package. If you turn on just the blue part, it will glow blue. Turn on the red part, it will glow red. Turn on all three, and it will glow white. By varying the relative intensity of each of the three colours you can produce all the colours of the rainbow. Unlike regular LEDs, RGB LEDs have four leads: one common lead, plus one lead for each of the colour elements.
	Servo Motor A special type of motor that doesn't just spin, like a regular motor. Instead, it rotates to a specific position controlled by your Arduino. The Arduino sends a series of pulses to the servo, and the servo then translates those pulses into an angular position. By changing the timing of the pulses the Arduino can tell the servo to move to different positions. Servo motors have different shaped arms (often called "horns") that can be fitted on top to connect them to different things.

Pushbutton Switches

Pushbuttons (often called “button switches”, or just “buttons”) let you provide manual input or control of your projects. Buttons are available in a huge variety of configurations, including momentary vs latching, normally-open vs normally-closed, and with multiple connection options.

The buttons supplied in the Experimenters Kit are the simplest and most common type: “momentary” so that they return to their normal state when you stop pressing them, and “normally open” which means that they are usually an “open circuit” but when you press them they change to a “closed circuit”.



Resistors

One of the simplest and most useful parts you can use in your circuits. Resistors “resist” the flow of electricity, and come in a variety of different values measured in “Ohms”. They also come in different power and tolerance ratings.

Resistors in “through hole” style like the ones supplied in the Experimenters Kit use colour coded bands to show their value, so by reading the colours and looking up what numbers those particular colours correspond with you can determine the value of the resistor.

Diodes

A diode is like a one-way valve: it lets electricity flow in one direction, but not the other.

That means it’s very important that you put them in place the right way around, so diodes are marked with a tiny band near one end so you can tell which end is which.

Transistors

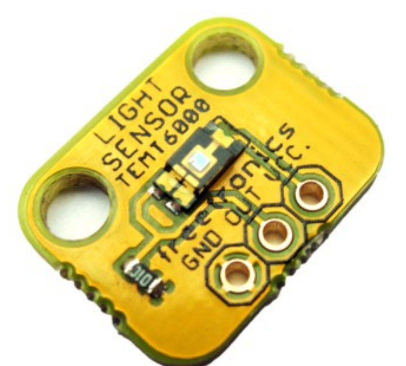
Probably one of the more tricky parts to understand when you’re first starting out with electronics, transistors allow one signal to either amplify or switch another signal.

The transistors supplied in the Experimenters Kit are a type called a “MOSFET”, and they have three leads called “gate”, “source”, and “drain”. It’s very important to connect them up the right way.

Potentiometer

"Potentiometer" is just a fancy name for a variable resistor, so it's not as complicated as it looks! They're usually just referred to as "pots" for short.

Pots have three connections so that they can be used in different ways, depending on the project. Sometimes only two connections are used to make it operate as a simple variable resistor, and sometimes all three connections are used to form what's called a "voltage divider".

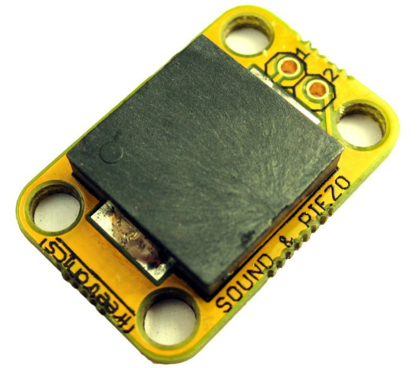


"LIGHT" Light Sensor Module

A special silicon light sensor that outputs a voltage proportional to the intensity of light falling on it.

The module has three connections: one for 0V (GND), one marked "VCC" for power (3V to 5.5V), and one for the output.

www.freetronics.com/light

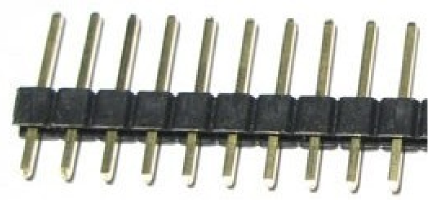


"SOUND" Sound & Piezo Module

A versatile module that can be used in several different ways, including generating sound and detecting taps or knocks.

Only two connections and they aren't polarised, so you can connect this module either way around.

www.freetronics.com/sound



Pin Header Strip

Pin header strip can be cut or broken to different lengths and soldered into the connections in the modules with the pins sticking down underneath, making it easy to plug them into a breadboard or directly into the headers on your Arduino.

74HC595 Shift Register IC

ICs ("Integrated Circuits") are complex parts each designed to do a specific job. There are thousands of different types of IC, and we've included a particularly handy one called a "shift register".

ICs have to be connected up the right way around, and at one end of the body you'll usually find a tiny notch or a printed dot. That's to indicate which end is near pin number 1.

Software Setup

You can create Arduino programs on your computer using software called the “Arduino IDE”. The Arduino IDE is available for Windows, Linux, and Mac, and is free to download and use.

Setting up the Arduino IDE

1. Download the latest version of the Arduino IDE to suit your operating system from:
   www.arduino.cc/en/Main/Software

2. Follow the specific installation instructions for your computer:
   Windows: www.arduino.cc/en/Guide/Windows
   Mac: www.arduino.cc/en/Guide/MacOSX
   Linux: www.arduino.cc/playground/Learning/Linux

Windows users please note: The USB driver for the Eleven must be downloaded and used for first-time installation of the Eleven's USB port. Full instructions are online at:
   www.freetronics.com/usbdriver

3. With the Arduino IDE installed, we're ready to do the initial board and port setup. You won't need to do this again unless the serial port number changes such as when using a different USB port on your computer.
   - In the Arduino IDE, select Tools > Board > Arduino Uno.
   - Before connecting your Eleven to the USB port, have a look at the list of ports in Tools > Serial Port. That's where your Eleven serial port is going to appear when you plug it in.
   - Connect your Eleven to the computer USB port. We supply an appropriate USB cable with the Eleven. After a short while if you look at Tools > Serial Port again you'll see a new port appear: that's the Eleven ready to be used. Select that port now with Tools > Serial Port so there is a tick mark next to it.

You're ready to go. The Arduino IDE now knows about your board and has a connection to it. Our Eleven boards ship with the "Blink" sketch preloaded so you should immediately see the blue power LED illuminate, and the red D13 LED will begin flashing on and off at 1 second intervals.

Compiling and uploading a “sketch” (program) to your Eleven

1. Open the Arduino IDE. From the top menu, select File > Examples > Digital > Blink. You'll see the code for the “Blink” sketch open in the IDE’s editor window.

2. Select Sketch > Verify/Compile. You've now “compiled” (prepared) the program ready to be loaded.

3. Select File > Upload to I/O Board. You'll see the red D13 LED flicker as the board is reset, then the green and yellow RX and TX LEDs will flash while the upload is being done.

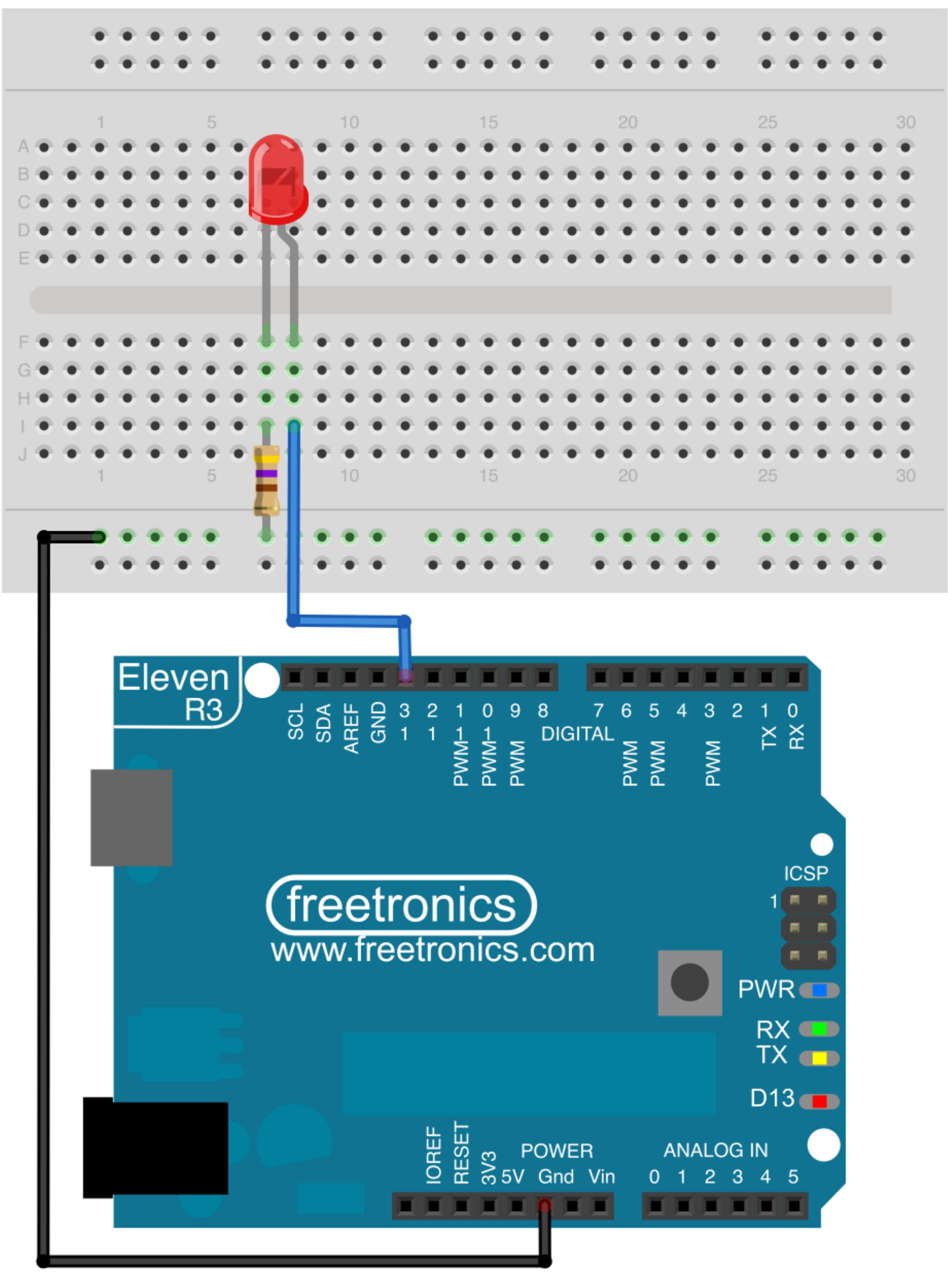
4. A few seconds later the RX and TX LEDs will go off, the board will reset, and the red D13 LED will begin flashing on and off at 1 second intervals. Congratulations! You've now compiled and uploaded your first sketch to the Eleven.

Project 1: Controlling An LED

Let’s start simple, and connect a single LED that you can control using your Arduino. It’s a very easy circuit, but it’s an important building block that we’ll expand on with the following projects so don’t skip this step: it’s a real milestone to flash your first LED!

Parts required

1 x “Eleven” Arduino-compatible microcontroller board
1 x LED (either green or red, doesn’t matter)
1 x 470 Ohm resistor (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x Solderless breadboard
2 x Jumper wires



Hardware assembly

Insert the LED into the breadboard as shown, taking care to find which lead is longer (the "+" lead) and orienting it with that pin to right. You won't damage anything if you put the LED in backwards so don't worry too much, but it won't work either!

Bend the leads of the resistor around until it's in a "U" shape, then insert it so that one resistor lead is in the same row as the "-" lead of the LED and the other resistor lead is in a sidebar of the breadboard.

Next, use a jumper wire (preferably black as a matter of convention, but it doesn't really matter) to connect the breadboard sidebar to one of the Arduino headers marked "GND". GND is an abbreviation for "ground", which means 0 Volts. All other voltages in the circuit are measured relative to GND.

Finally, use a second jumper wire to connect the breadboard row with the LED "+" lead in it to the Arduino header marked "13".

How the hardware works

What you've just done is create what is called a "series" circuit, because the LED and resistor are connected in a line: electricity has to flow through first one component, then the other. One end of that line has been connected to GND, or 0V, and the other end of the line has been connected to a digital output of the Arduino.

That means the Arduino now has the power to control whether any electricity flows through the LED. If the Arduino digital output is in a "low" (0V) state, no electricity will flow through the LED. However, if the digital output is in a "high" (5V) state, there is 5V applied across the LED and resistor pair and so the LED will light up.

Software

Now that the Arduino has the ability to control the LED, we need to give it a program that will tell it what to do. Open the Arduino IDE on your computer, and open the following menu:

File -> Examples -> 01.Basics -> Blink

The IDE will open a window containing a very simple program called "Blink", which we can use to make the LED blink on and off.

Plug your Arduino into your computer using a USB cable, and click the little round "Upload" button near the top left of the IDE window. It's the one that looks like an arrow pointing to the right. After a few seconds the program will be prepared ("compiled") and sent to the Arduino through the USB cable, and your Arduino will then start flashing your LED on and off about once per second.

If your LED is flashing, congratulations! You've passed the first hurdle and made your first electronic circuit, and even uploaded a program to control it.

If the LED isn't flashing, don't worry. It's easy to have one of the parts just a little bit out of place, so go back over your circuit and compare it with the instructions above. In particular, make sure that the LED is the right way around.

How the software works

The “Blink” sketch used in this example is a great demonstration of the basic structure of all Arduino programs. As you’ll see with the following projects, they all follow this same fundamental program structure, just with more bits added along the way!

Blink starts with an entry called a “variable declaration”, which specifies that an “int” (or integer) will be stored using the name “led”, and that its value will be set to 13. This name is used later in the program so that it will know which pin you’ve connected the LED to. The name of the variable could have been anything: “led” was used just because it’s easy to remember what it means.

```
int led = 13;
```

You’ll notice the line ends with a semicolon: that’s to tell the Arduino where the end of the command is. Every command has to end with a semicolon.

Next we have a block called “setup()”, which begins and ends with curly brackets so you can see what’s inside it. Curly brackets are like the covers of a book: they define the boundaries, and all the pages of the book are inside them. In this case the only thing inside setup() is a single operational line, which tells the Arduino to set the “pinMode” of the pin called “led” (which has the value 13, remember, because we set it a moment ago) to be an OUTPUT. That’s because the digital pins on an Arduino can be used as either inputs or outputs, so it needs to know what you want it to be. In other projects we’ll use some pins as inputs and some as outputs.

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

The setup() block is only executed once when the Arduino starts up, so it’s used for commands that you only want it to do once. In this case setting the pin mode only needs to be done once when the program starts running, so it can be inside setup().

Next we have another block of commands called “loop()”, and it also begins and ends with curly brackets. The difference is that although setup() is only executed once when the Arduino starts up, the loop() block is run over and over again. As soon as the Arduino gets to the end of the commands inside loop(), it runs right back to the start and does it all over again.

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

Let’s step through it, just like the Arduino does when it’s running the program. First is a command called “digitalWrite()”, which tells the pin referred to as “led” (pin 13) to go HIGH. Going HIGH will make the pin go to 5V, which will then make the LED connected to that pin turn on.

Next is a “delay()” command, which tells the Arduino to stop doing anything until a certain period of time is up. The number in the command tells it how many milliseconds (thousandths of a second) it should wait, so with the command “delay(1000)” it will wait for one second before moving on.

Next is another digitalWrite, but this time it sends the pin LOW (0V) so the LED will turn off.

Finally another delay(), so that the Arduino will wait another second before doing anything else.

Once that delay has finished, the Arduino scoots right back to the start of the “loop()” block and begins again.

The end result is simple: the Arduino turns on the LED, waits one second, turns off the LED, waits one second, and then does it all again, forever. Or at least until you pull out the power!

Now that you have the basics under your belt, try altering the Blink sketch a bit to make it behave differently. For example, change both of the “1000” delay values to 100 so that it blinks ten times as fast. Or make the first delay 100, and the second delay 900, so that exactly once per second it will blink for 1/10th of a second.

Each time you make a change to the sketch, click the “Upload” button again near the top left so that the new version is uploaded to your Arduino and you can see the result.

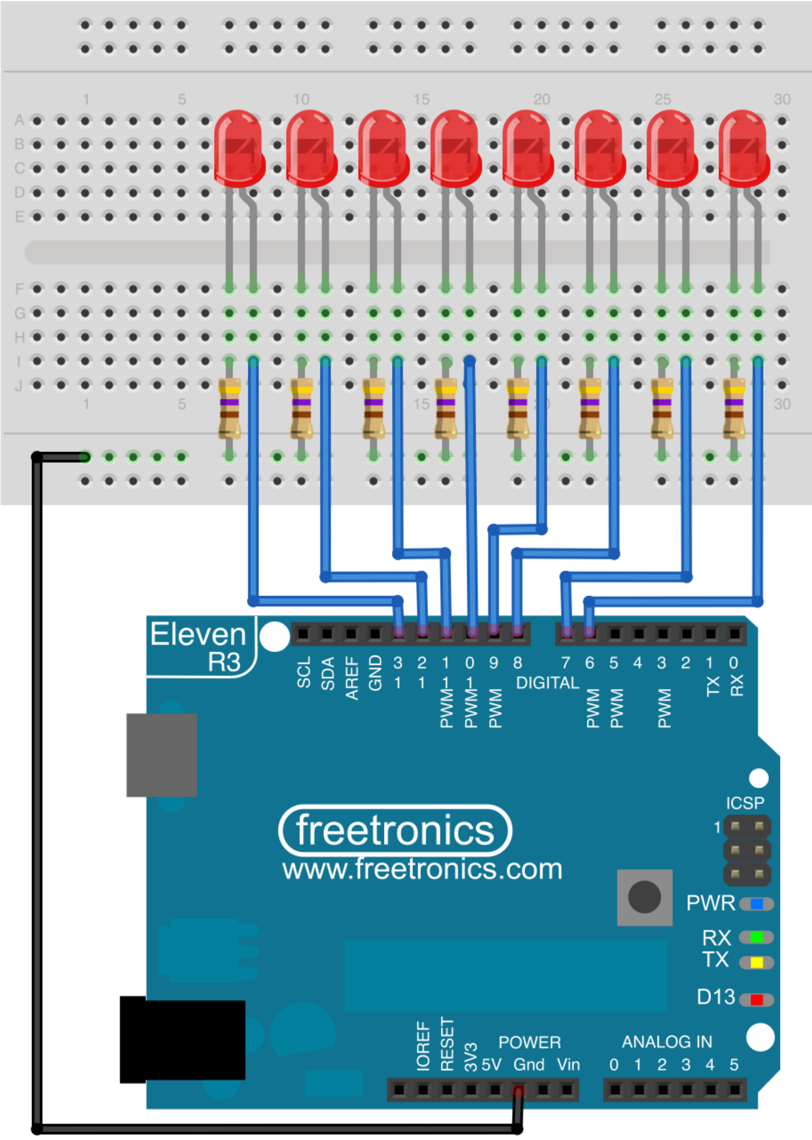
Project 2: Controlling 8 LEDs

You've already seen that LEDs can be lots of fun even when there's only one of them, and by adding more we can do even more impressive things such as controlling them to make different patterns just by changing the software. This is a great example of how a combination of the electronics (the hardware) and the sketch (the software) gives you the flexibility to achieve different end results very easily.

We'll use a sketch that makes a light "scan" along a row of LEDs, much like the Kitt scanner in the show "Knight Rider" or the red Cylon eye in Battlestar Galactica.

Parts required

1 x "Eleven" Arduino-compatible microcontroller board
8 x LEDs (either green or red, doesn't matter)
8 x 470 Ohm resistors (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x Solderless breadboard
9 x Jumper wires



Hardware assembly

This project is very similar to the first project, but repeated eight times. If you've just finished the first project, you can use that as the starting point and add the other seven LEDs and resistors.

Let's start with the LEDs. Insert a total of eight LEDs in a row along the solderless breadboard, checking that they all have the long (+) lead to the right.

Now insert a total of eight resistors, each one connecting between the short (-) pin of an LED and the common GND bar on the side of the breadboard.

Finally, use jumper wires to link each of the LED + connections to one of the digital pins on the Arduino. We used pin 13 for the first LED, so work backwards in sequence to connect the additional LEDs to pins 12, 11, 10, and so on.

How the hardware works

The hardware for this project works just like the circuit in the first project, but repeated eight times. Each LED is controlled by a separate digital pin, so the Arduino can turn on any of the LEDs without altering any of the others.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int ledCount = 8;
int ledPins[] = { 6, 7, 8, 9, 10, 11, 12, 13 };
int ledDelay = 300;

void setup() {
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT);
  }
}

void loop() {
  for (int thisLed = 0; thisLed < ledCount-1; thisLed++) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    digitalWrite(ledPins[thisLed], LOW);
  }
  for (int thisLed = ledCount-1; thisLed > 0; thisLed--) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    digitalWrite(ledPins[thisLed], LOW);
  }
}
```

How the software works

This sketch introduces a couple of important concepts in programming: arrays and loops.

The “ledCount” variable specifies how many LEDs are connected, so you can use this same sketch with fewer or more LEDs if you like.

The “ledPins” variable is an “array”, which you can think of as being like a list: in this case it’s a list of the pins that the LEDs are connected to. Later in the sketch we’ll refer to the list to turn different LEDs on and off.

The “ledDelay” variable just sets the number of milliseconds to leave each LED on for. To make the scanner run faster, make the number smaller. To slow it down, make the number bigger.

The setup() function is where things get interesting. It could alternatively have been written with a series of eight nearly-identical lines, each one using pinMode() to set one of the LED pins to OUTPUT mode, but that’s not a convenient way to write longer programs. Instead it uses a “for” loop, which runs pinMode() eight times but with a different value each time.

The first argument passed to the “for” loop sets the name of the variable that will be used as the “loop counter”. In this example the loop counter is a number called “thisLed”, and it’s given a starting value of 0.

The second argument sets the terminating condition so the “for” loop will know when it should stop running. The loop will keep repeating itself until the condition fails, so in this case the loop will repeat until the value of “thisLed” is equal to the value of “ledCount”. We’ve set ledCount to 8, so it will keep running until ledPin is equal to 8.

The third and final argument is an action that the loop will perform each time it runs. The “++” operator is a shorthand way of saying to take a variable and add 1 to it, so in this case the “thisLed” variable will get bigger by 1 every time the loop executes.

When you put those three arguments together, you have a loop that will behave in a certain predictable way. It starts out with a variable with a value of 0, increases that value by 1 each time it runs, and when the value reaches 8 it stops. It’s a very concise way of saying “do this particular thing 8 times, slightly differently each time” without having to write each one out individually.

The loop itself is interesting, but what’s happening inside it is also interesting. You’ll see it references “ledPins[thisLed]”, which looks a bit confusing. You’ll remember that the variable “ledPins” is an array, or list, containing a series of values. Referencing it this way allows us to look at positions in the list without having to know what happens to be on the list in that position. Imagine you have a restaurant menu with a list of dishes, and each one has a number next to it: you don’t need to state the name of the dish, you can just say “I’ll have number 4, please” because the position (or number) of the entry is an easy way to reference that item on the menu.

In arrays, the position in the list is called the “array index”, and it’s a number that specifies what position in the array we want to examine. Unlike typical restaurant menus, though, array indices don’t start at 1: they start at 0, so the first position in the array is referenced as index 0. The second position is index 1, and so on.

So when the value of “thisLed” is 0, it’s the same as referencing “ledPins[0]”, which is position 0 (the first item) on the list. The original code looks like this:

```
pinMode(ledPins[thisLed], OUTPUT);
```

The value for thisLed is 0 on the first pass through the loop, so to understand what's going on we can imagine the variable has been replaced by its value, like this:

```
pinMode(ledPins[0], OUTPUT);
```

But of course "ledPins[0]" is really just pointing to the first position in the list. Looking back at the definition of the ledPins array, we can see that the first item on the list has a value of "6". That means the command that is really being executed the first pass is this:

```
pinMode(6, OUTPUT);
```

Then on the next pass through the loop, the value of "thisLed" has increased to 1, so we're referencing "ledPins[1]", which is the second item on the list. Then if you look at the second item on the list and substitute it in, the end result is that what will really be executed on the second pass through the loop is this:

```
pinMode(7, OUTPUT);
```

And so on.

Arrays can be a tricky concept, but they're very powerful and can make your sketches much simpler by allowing you to define lists of things and then step through the list, instead of specifying similar but slightly different operations over and over again and taking many more lines to achieve the same end result.

Next we get into the "loop()" part of the program, which will continue indefinitely. It might look cryptic at first glance, but when you break it down into its major sections it's fairly simple.

It's really just two more "for" loops, one after the other, that operate just like the "for" loop we looked at a moment ago.

The first loop increments a counter to step forwards through the list using the "++" operator explained previously. It begins at the start of the list of LEDs, and for each one it turns it on, waits for a delay period, then turns it off again before moving on to the next one.

Once the first loop finishes, the second loop begins. The second loop is almost the same, but it runs backwards! It starts at the last position in the list of LEDs, and each time through it uses the "--" operator to decrease the position value by 1. It's like reading the restaurant menu backwards, starting at the last item.

When it reaches the start of the list, the second loop finishes and the program jumps back to the start to begin the first loop all over again.

Further experiments

To help understand how the array works, try swapping some of the values in it around. You'll see that the counter will still faithfully step through the list in order, and the LEDs will turn on in the order you specify in the array.

Or you could consider other ways to structure this sketch. For example, think about how you could make it scan from side to side but only use one "for" loop instead of two. At first glance that may seem impossible, but by thinking about how the array is used as a list to step through the LEDs in sequence it's actually quite easy. For example, instead of having a simple list of pins and stepping through it first one way and then the other, you could make a longer array that specifies the complete sequence for an up-and-back scan, and just loop over it once, like this:

```
int ledCount = 14;
int ledPins[] = { 6, 7, 8, 9, 10, 11, 12, 13, 12, 11, 10, 9, 8, 7 };
int ledDelay = 300;

void setup() {
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT);
  }
}

void loop() {
  for (int thisLed = 0; thisLed < ledCount-1; thisLed++) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    digitalWrite(ledPins[thisLed], LOW);
  }
}
```

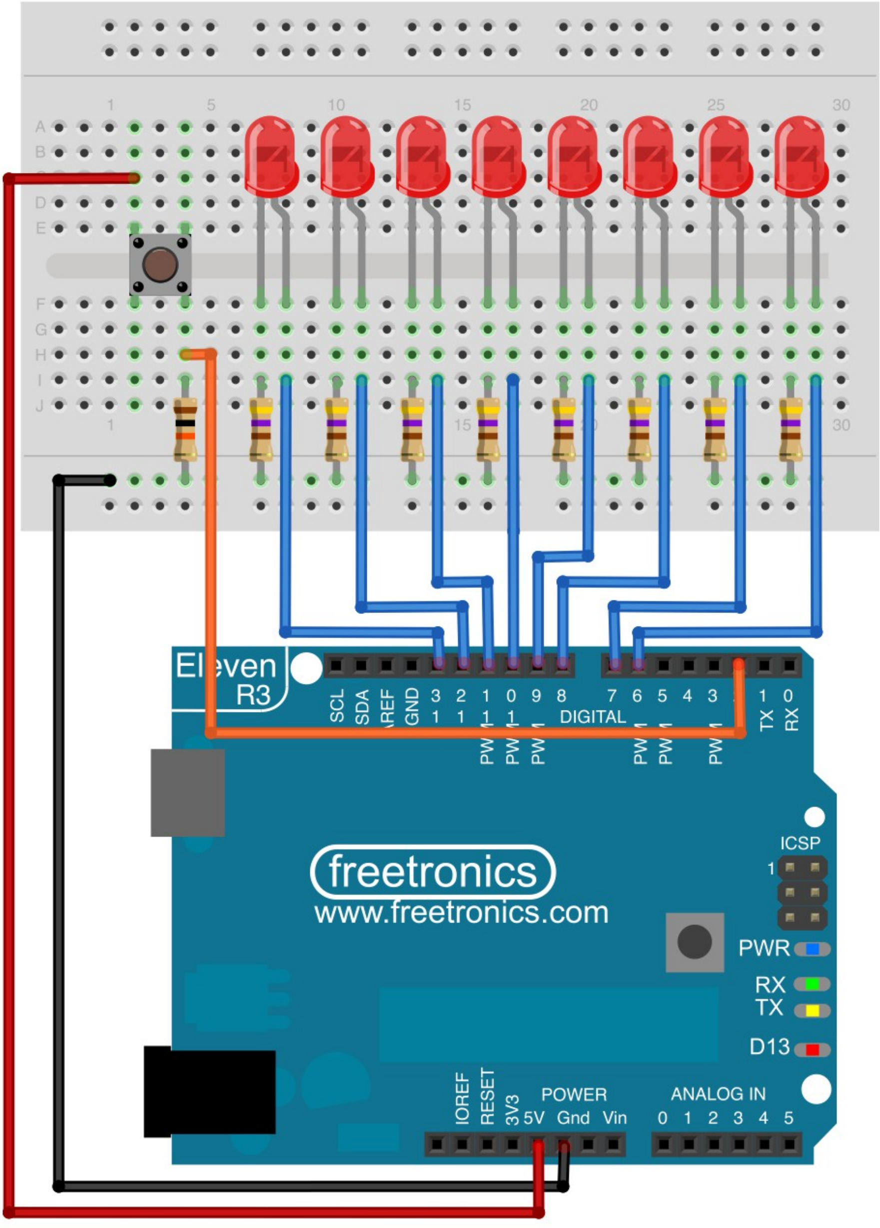
Yes, we've told the sketch that we have 14 LEDs when we really only have 8, but that doesn't matter. The 14 LED positions count up from 6 to 13 and then back down again, repeating some of the positions. Of course this means that setup() will call pinMode() multiple times on the repeated pins, but that doesn't matter. No harm will be done, and the loop can then step through the list as if it's a linear sequence when in fact the list doubles back on itself.

Project 3: Reading Digital (On/Off) Input

The first two projects operate on their own: once you start them up, they just begin doing their thing without any input or control from you. Detecting button input is the basic building block that you can use for all sorts of interaction with your projects, so we'll build on the previous projects to add a button that you can use to control the state of the display.

Parts required

1 x "Eleven" Arduino-compatible microcontroller board
8 x LEDs (either green or red, doesn't matter)
8 x 470 Ohm resistors (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x Momentary pushbutton
1 x 10K Ohm resistor (brown-black-black-red-brown, or brown-black-orange-gold)
1 x Solderless breadboard
11 x Jumper wires



Hardware assembly

Once again this project builds on the previous one, so if you've just finished the last project you can use that as the starting point. Otherwise, go back to the instructions in project #2 and assemble the 8 LEDs and their matching 470 Ohm current-limiting resistors.

Next, insert a pushbutton into the solderless breadboard as shown, paying attention to the difference in spacing between the pins. If you look closely you'll see that the pins are in two pairs, slightly further apart in one direction than the other. Even though the button has a total of four pins, that's only to make it mechanically stronger when it's mounted: it really only has two connections, which are doubled up so that it can have four pins.

Internally, the buttons pins are connected along the long sides of the button. If you orient it as shown in the breadboard layout with the long sides spanning the central gap, and then connect one wire to the left of the button and one wire to the right, it'll work just fine.

Use a jumper lead to connect from one side of the button to the "D2" digital pin on the Arduino.

Use a 10K Ohm resistor to connect from that same side of the button to the "GND" (0V) rail on the breadboard.

Use a jumper wire to connect from the other side of the button to the 5V header on the Arduino.

How the hardware works

The LEDs work just like in the previous project: each one is a simple series circuit that goes from the matching Arduino digital pin, through a current-limiting resistor, through an LED, and to GND or 0V. For more details see the previous projects.

The button, 10K Ohm resistor, and associated jumper wires form a circuit that pulls Arduino digital pin D2 down to 0V through the resistor. This is called "biasing", where a resistor is used to connect a digital pin to a certain voltage (in this case 0V) so that it tries to be at that voltage. However, because the connection goes through a resistor instead of directly through a simple piece of wire it's not pulled "hard" to that voltage, and it's possible to override it and force it to a different voltage if you want to.

That's done using the button, which is also connected to the digital input and has its other connection going directly to 5V. The result is that when the button is not being pressed, digital pin D2 is pulled down to 0V ("low"), but when the button is pressed it's pulled up to 5V ("high) through the button.

The terms "high" and "low" are commonly used (often capitalised) to describe the "state" of parts of a digital circuit, and you'll see them used in many places in Arduino sketches. You'll often see statements like "pin 13 is in a LOW state" or "pin 7 then goes HIGH" in discussions of digital circuits.

What this button circuit does is allow the sketch running on the Arduino to detect when the button is being pressed, because it can measure the state of the digital pin connected to it. If it measures the state of the pin and discovers it's low, it knows the button isn't being pressed. If the pin is high, it knows the button is currently being pressed. In a moment we'll take advantage of that to use the button to change the behavior of the LEDs.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int ledCount = 14;
int ledPins[] = { 6, 7, 8, 9, 10, 11, 12, 13, 12, 11, 10, 9, 8, 7 };
int ledDelay = 300;
int buttonPin = 2;

void setup() {
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT);
  }
  pinMode(buttonPin, INPUT);
}

void loop() {
  for (int thisLed = 0; thisLed < ledCount-1; thisLed++) {
    digitalWrite(ledPins[thisLed], HIGH);
    delay(ledDelay);
    while(digitalRead(buttonPin) == HIGH) {
      delay(10);
    }
    digitalWrite(ledPins[thisLed], LOW);
  }
}
```

How the software works

Most of the sketch is identical to the “scanner” sketch from the previous project. However, this version adds a “pause” button.

Near the top of the sketch the “buttonPin” variable is set to the value 2, because the button is connected to digital pin 2.

Inside setup(), the button pin is set to be an input so that the sketch can read from it and detect whether it’s currently at a high (5V) or low (0V) value. Remember that we have a 10K Ohm resistor biasing the input low, so most of the time it will be at 0V and will read low. When the button is pressed the input will be pulled up to 5V, and will read high.

Finally, inside the loop that scans along the LEDs, there is a new section of code:

```
while(digitalRead(buttonPin) == HIGH) {
  delay(100);
}
```

This is another type of loop, called a “while” loop. The loop has a condition declared, and as long as that condition is true it will keep repeating itself. In this sketch, the loop condition is that when reading from the pin while the button is pressed, the value must be HIGH, or 5V. If that condition isn’t met (ie: the result of that comparison is false) the loop will exit and the rest of the program will continue. But if the comparison is true, it will keep doing whatever is inside the loop.

In this case there's nothing much inside the loop at all, except a call to delay for 100 milliseconds and then go back to the start and check the button again. The result is that as long as the button is held down, the reading will be high, the condition will therefore be true, and the loop will sit there doing nothing much at all, so the program can't continue and the scanner will stop. Release the button and the condition will fail, and the scanner will resume as before.

Further experiments

This project is controlled by a pushbutton that provides simple "yes/no" value, but it's not just buttons and switches that behave this way. Many other input devices have simple on/off values that you can read just like a button, so with the skills you've learned so far you could already replace the button with a variety of other devices such as a security system motion detector, or a thermal switch, or a tilt switch.

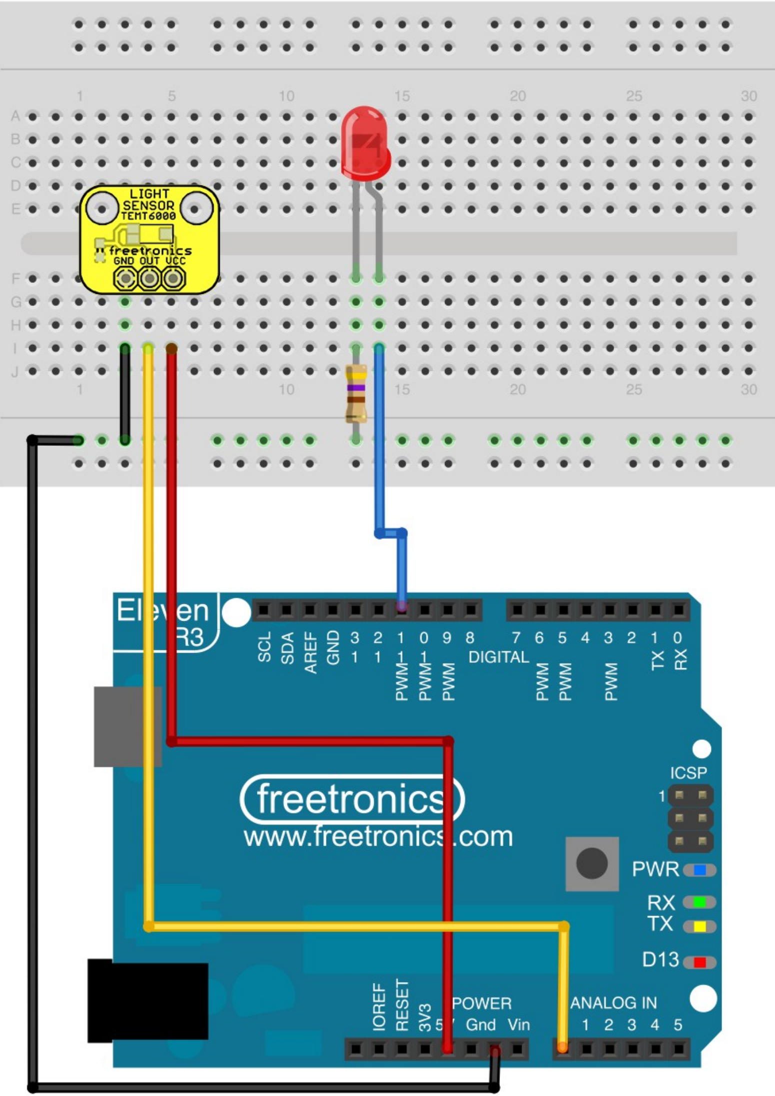
Project 4: Reading Analog (Variable) Input

So far we've already made great progress: we can control digital outputs to turn things on and off, and we can read from digital inputs to detect simple "yes or no" events like button presses. For many projects that's all you need, but sometimes you need to do more than just measure simple low/high states: you need to be able to measure across a range, and measure the level of something that's not totally off and not totally on.

In this project we'll measure the level of light in the room, and show how to send that value to your PC so you can read it and also use it to change how the sketch operates.

Parts required

1 x "Eleven" Arduino-compatible microcontroller board
1 x LED (either green or red, doesn't matter)
1 x 470 Ohm resistor (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x "LIGHT" Light level sensor module
1 x Solderless breadboard
5 x Jumper wires



Hardware assembly

If you've finished the previous project you can leave the rest of the circuit in place if you like, but for this project we'll go back to using just one LED. It'll be connected up just like in project #1, but instead of digital pin D13, this time we'll use digital pin D11.

Connect a jumper wire from D11 through a 470 Ohm resistor, then the LED, then to GND, as shown in the diagram.

The LIGHT module has three connections. They are designed to have a header or wires soldered directly to them, so to make it easy for you to use the module in your solderless breadboard we've pre-fitted header pins. That way you can just plug them in and use jumpers to link them to your Arduino.

The three module pads need to be connected to three specific places on your Arduino:

- The "GND" connection on the left needs to link to GND on the Arduino.
- The "OUT" connection in the middle needs to link to Analog input A0 down near the bottom of the Arduino.
- The "VCC" connection on the right needs to link to 5V on the Arduino.

How the hardware works

The LED connection should be very familiar to you now, because it's the same as used in the first three projects: an LED and a resistor in series, connected to a digital pin at one end and GND at the other so that when the pin is high the LED illuminates, and when the pin is low the LED is extinguished.

The interesting part is the connections to the LIGHT module. The module needs GND and 5V connections as a power supply so that it can operate, so that's easy. Then, when it is powered up, it reads the intensity of the light that is falling on it and changes the voltage given on the OUT pin proportionally. If the module is in complete darkness, it will output very close to 0V on the OUT pin. When the module is in intense light, it will output very close to 5V on the OUT pin. When it's in dim light, it will be somewhere around the middle at about 2.5V, and so on.

So by measuring the voltage of the OUT pin in the range of 0V to 5V, we can tell how much light is falling on the module.

Remember in the previous project we measured whether a button was being pressed or not, by reading one of only two states: high and low. That was a digital value that was being read, because it could only be in one of those two states. This is different, because the output from the LIGHT module could be 0V, or 5V, but it could also be anywhere in between. We can't read it as a digital input because it could be at 2.865V - what does that mean? Would we read it as high or low?

A signal that can have values anywhere across a range is called "analog" and the Arduino has a block of 6 inputs near the bottom right specifically designed for analog signals.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site, and then open the serial monitor with the baud rate set to 38,400bps:

```
int led = 11;
int lightLevel;

void setup()
{
  Serial.begin(38400);
  pinMode(led, OUTPUT);
}

void loop()
{
  lightLevel = analogRead(A0);
  Serial.print("Light level: ");
  Serial.println(lightLevel, DEC);
  digitalWrite(led, HIGH);
  delay(lightLevel);
  digitalWrite(led, LOW);
  delay(lightLevel);
}
```

How the software works

The interesting part of this sketch is the use of “analogRead()” to measure the voltage currently being detected by analog input A0. The analog inputs on the arduino return values in the range from 0 to 1023. If the voltage is at minimum (0V) it will return a value of 0. If the voltage is at maximum (5V) it will return a value of 1023. If the voltage is exactly in the middle (2.5V) it will return a value of 511.

Right now you’re probably thinking “Why is the maximum value 1023? And why is the middle reading 511? That doesn’t make any sense! Why couldn’t they make it read from 0 to 100, or 0 to 1000, or something sensible like that?”.

It may not make much sense when you first see it, but there’s a very good reason the values read from 0 through to 1023. The explanation is quite complicated, but it’s to do with the fact that the special circuitry inside the analog inputs has to take a continuously-varying level and break it down into a series of distinct steps, and figure out which step the voltage is closest to. There is a very detailed explanation of the process in the book “Practical Arduino” by Jonathan Oxer and Hugh Blemings if you want to look into the details.

For now, just accept that 0 is the lowest reading, 1023 is the highest, and the rest of the range is spread evenly between those two limits.

Having read the light level from the module and obtaining a value somewhere between 0 and 1023, the sketch then uses that value to decide how long to wait each time it turns the LED on and off. This gives us an LED that will flash at different rates depending on how bright the light is, because if the light level is low the delay will be shorter and if the light level is high the delay will be longer.

Try putting your hand over the light sensor and see if the flash rate of the LED gets faster. Try shining a bright torch on it or putting it in direct sunlight and see if it gets slower.

As well as varying the delay depending on the light level, the reading is also transmitted out the USB port to your PC as a serial connection so that you can see the value that the analog input is reading. This is an extremely useful technique that you'll use in many Arduino projects.

The data connection to your PC is configured inside the setup() routine, including the communication speed. In this example it's set to 38,400bps, so with the sketch running on your Arduino and your PC still connected by the USB cable you can click the "Serial Monitor" button in the top right of the IDE to open a communications window to see messages sent to you by the sketch. Make sure the baud rate setting at the bottom is set to 38,400 to match the setting in the sketch.

Inside the loop() routine, you'll see the functions "Serial.print()" and "Serial.println()" used. Those functions both send data to the PC: the first sends just the raw message, and the second sends the message and then also adds a "new line" character to the end so the messages don't all run together on the same line in the Serial Monitor window.

Further experiments

This example makes the flashing get slower as the brightness increases. How would you reverse the behavior, so that the flashing gets faster as brightness increases? One simple way to do it would be to read the value from the sensor, then instead of using it directly you could subtract it from 1023:

```
delay(1023 - lightLevel);
```

A reading of 0 will then give a delay of 1023, and a reading of 1023 will give a delay of 0. Other intermediate values will likewise be inverted.

Rather than simply altering the flash rate of a single LED, you could combine this project with project #3 to drive 8 LEDs and have the delay duration vary with the reading from the light sensor. Then the scanning speed will get faster as the light gets dimmer.

What if you want to control the input manually, rather than based on light level? For that you can use the variable resistor included in the Experimenters Kit. Connect one of the end connections to GND, the other end connection to 5V, and the middle connection to the A0 analog input. You can adjust the position of the variable resistor, and the voltage on the middle pin will vary proportionally.

Project 5: Dimming LEDs Using PWM

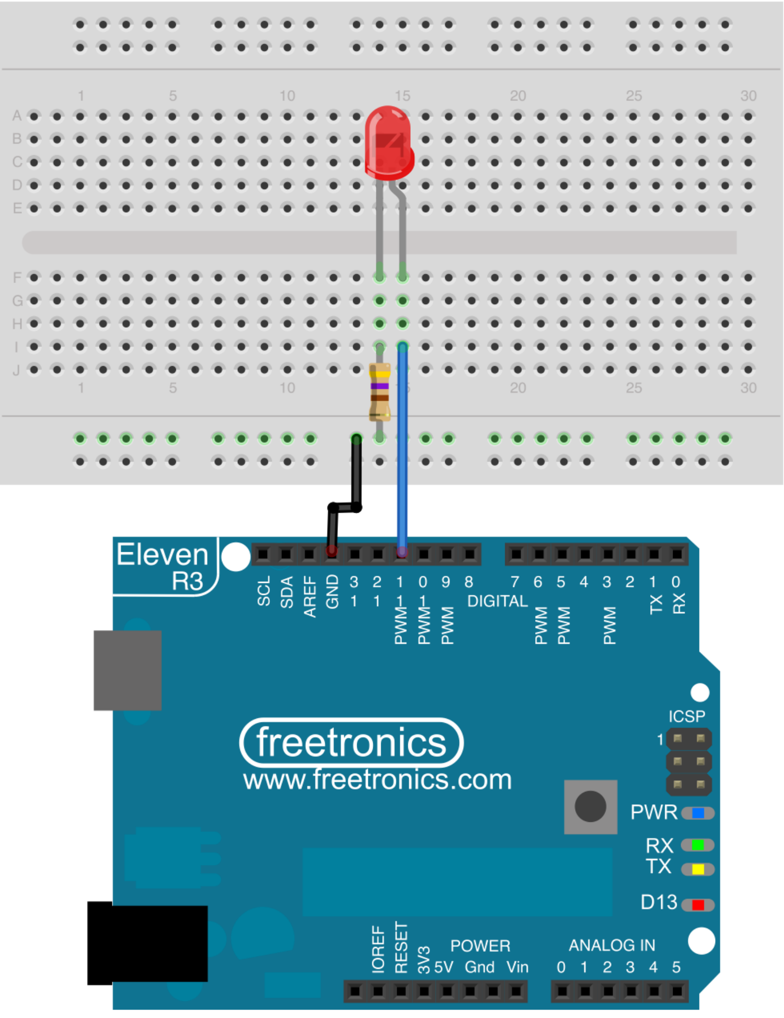
We've now done digital output, digital input, and analog input. What's left? Analog output!

Sometimes you don't want things like LEDs to be either fully on or fully off. You want them to be half on, or a quarter on. But that's harder than it sounds, because to a computer, everything is "yes or no", "high or low", "on or off" values: everything is black or white, computers don't like grey! This is called "binary" logic, where everything is either 0 or 1. The outputs from the Arduino are the same. They can be either off or on, but not part way in between.

The solution is a trick called "Pulse Width Modulation", usually referred to simply as PWM. In this project we'll use PWM to control the brightness of an LED so that it looks like it's breathing: slowly pulsing on and off.

Parts Required

1 x "Eleven" Arduino-compatible microcontroller board
1 x LED (any colour)
1 x 470 Ohm resistor (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x Solderless breadboard
2 x Jumper wires



Hardware assembly

You can do this project with the exact same hardware as the previous project, so if you've just finished that you can leave it exactly the same. We won't use the light sensor module for the basic example, but later we'll give you some ideas about how you can incorporate that as well and there's no harm leaving it connected.

How the hardware works

The circuit is extremely simple, just like the first project in this guide: a digital pin sending current through an LED, with a current-limiting resistor to prevent the LED burning out.

The interesting part is the magic inside the microcontroller that does the PWM output.

PWM is a sneaky way to use a digital output and make it pretend to be partly on, even when it can't be. Like most magic, the trick is in the timing: if you have a digital output going to an LED, and you turn it on and off very fast, the LED will be fully on for some of the time and fully off for some of the time. If you do it fast enough, the human eye can't see it flashing on and off. The human eye has a slow response time which results in a phenomenon known as “persistence of vision” or the “flicker fusion threshold”, so if you make an LED flash on and off more than about 30 times per second you won't be able to detect the individual flashes. To you it will just look like the LED is dimmer than normal.

So if you want to make an LED look like it's at about half normal brightness, it's easy: just turn it on for a set period of time, then turn it off for the same period of time, and so on. If it's on about half the time that's what engineers call a “50% duty cycle”, and the result is that the LED will be about half as bright as normal.

But what about other brightness levels? What if you want to make the LED look like it's at ¼ brightness? Easy, just change the relative times of the “on” and “off” periods. That's why it's called “pulse **width** modulation” - it's the width (duration) of the pulse that changes to give different output levels. To achieve ¼ output (or 25%) you just change the timing so that it's turned on ¼ of the time, and turned off ¾ of the time. For example, you could turn on the output for ten milliseconds, then turn it off for thirty milliseconds, on for ten, off for thirty, and so on.

That sounds like a lot of work, and it can make your sketch quite complicated if you try to control all the timing yourself, but luckily the Arduino has a special feature that makes it really easy to use PWM on certain outputs. If you look closely at the labels near the headers on your Eleven, you'll see that some of the pins have a tiny “PWM” label. You can use those special outputs to control things like the brightness of LEDs using a very simple command in your sketch.

Inside the microcontroller is circuitry attached to those pins that accepts a setting to tell it what duty cycle to run at, and then it just keeps turning the pin on and off very fast to maintain that duty cycle. Once your sketch has told the pin what duty cycle it should run at, the sketch can continue on doing other things and the microcontroller will keep the pin going at that same duty cycle until it's told otherwise.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int led = 11;
int brightness = 0;
int delayTime = 10;
void setup() {
  pinMode(led, OUTPUT);
}
void loop() {
  while(brightness < 255)
  {
    analogWrite(led, brightness);
    delay(delayTime);
    brightness = brightness + 1;
  }
  while(brightness > 0)
  {
    analogWrite(led, brightness);
    delay(delayTime);
    brightness = brightness - 1;
  }
}
```

How the software works

By now most of the code in this sketch should be quite familiar to you. There are some variables declared at the start, the pin mode is set in setup(), and then there are two loops: first it steps the brightness value up from 0 to 255, then it steps the brightness back down again to 0.

The new part is the lines that use “analogWrite()”, which tell the PWM-capable pin to pulse with a certain timing. With a value of 0, the pin will be permanently off: a 0% duty cycle. With a value of 255, the pin will be permanently on: a 100% duty cycle. With a value of 127, the pin will be on half the time: a 50% duty cycle. By stepping the value up from 0 to 255 the pin will rise slowly from not on at all to fully on all the time, and the LED will also start totally off and gradually illuminate until it’s totally on.

To the human eye the result is very simple, with a smooth transition from off to on in 256 steps. What’s going on behind the scenes is far more complicated!

Further experiments

Combining this project with the light sensor, you could make a night-light that varies the illumination level depending on how bright the background light level is. Start with the sketch from the previous project, and restructure it so that the loop isn’t delaying different amounts depending on the light level: instead, it sends different values to analogWrite() to change the brightness of the LED.

Remember though that analogWrite() needs values between 0 and 255, while analogRead() gives values between 0 and 1023. That means you need to proportionally scale back the reading from analogRead() by dividing it by 4 before you can use it to set the LED brightness.

A sketch like this should do the job:

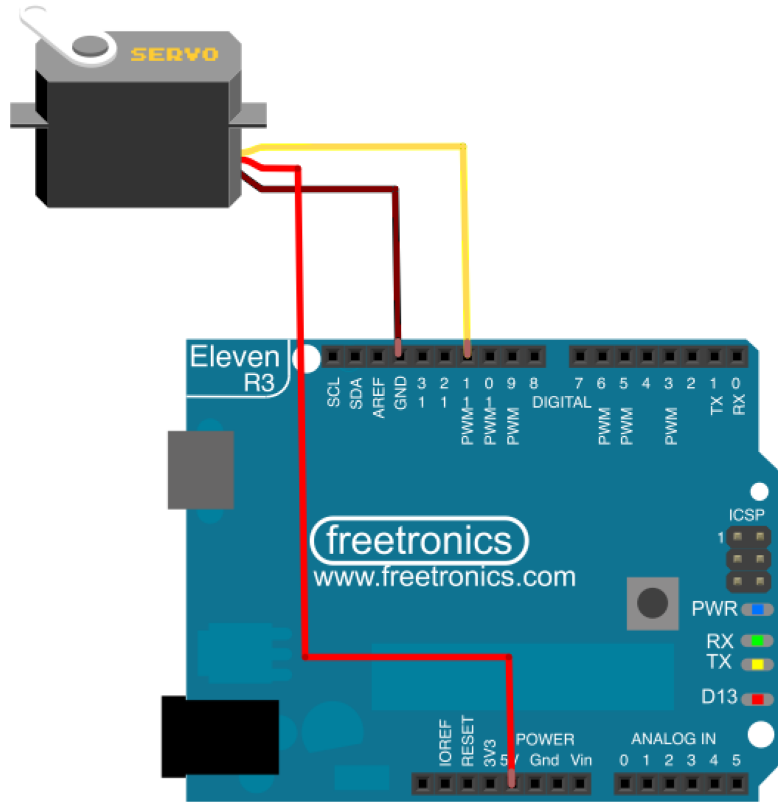
```
int led = 11;
int lightLevel;
int ledLevel;

void setup()
{
  Serial.begin(38400);
  pinMode(led, OUTPUT);
}

void loop()
{
  lightLevel = analogRead(A0);
  ledLevel = lightLevel / 4;
  Serial.print("Light level: ");
  Serial.println(lightLevel, DEC);
  analogWrite(led, ledLevel);
  delay(100);
}
```

Project 6: Making Things Move With Servos

Blinking LEDs are fun, but for any really cool robotics project you need to be able to make things move. Controlling servo motors is much easier than you may think, and it opens up all sorts of possibilities for projects that let your Arduino control the physical world.

Parts required

1 x "Eleven" Arduino-compatible microcontroller board
1 x Servo motor
3 x jumper wires



Hardware assembly

Assembly for this project is very easy. Simply plug the three jumper wires into the connector on the servo, and then plug the other ends into the appropriate locations on your Arduino. You don't even need a breadboard.

Servos generally have three connections: "+", "-", and "signal". The trick is knowing which wire is which, because different manufacturers often use their own colour codes and connection order. Luckily the "big 4" servo manufacturers decided to all use the same connections to reduce confusion, which in turn has forced many of the smaller servo manufacturers to follow their lead. You have to be careful though because sometimes you'll come across servos that use different connections, so you need to check the manufacturer's specifications just to be safe.

The servos in the Experimenters Kit use brown, red, and orange wires, which is a very common colour combination among servo manufacturers.

Wire Colour	Use	Connection
Brown	- (Negative supply)	GND on Arduino
Red	+ (Positive supply)	5V on Arduino
Orange	Signal	D11 on Arduino

User jumper wires to connect the servo to your Arduino as shown.

How the hardware works

Servo motors may look very simple, and they're easy to interface with, but that's because they're actually much smarter than they look. Inside each servo is a tiny PCB with electronics that processes control signals and moves the motor to the requested position.

The positive and negative supply connections simply provide power to the servo motor and its control electronics.

The interesting part is the "signal" connection, which your Arduino uses to tell the servo what angle to move to. Servos work on a PWM (pulse width modulation) signal, just like the technique used in the last project to dim LEDs. You'll remember that PWM can vary from 0% to 100% duty cycle, and the Arduino does it in steps from 0 to 255. The servo reads the PWM signal and converts it into an angular setting as a percentage of the total possible rotation for that particular servo. Most servos can turn approximately 180 degrees, so a PWM duty cycle of 0% will typically set it to 0 degrees and a duty cycle of 100% will set it to 180 degrees. Other duty cycles will set it to other proportional positions within its range of motion.

Software

If you still have the sketch loaded from the previous project, give that a try! It'll work just as well turning a servo as changing the brightness of an LED.

Try moving your hand over the light sensor to change the illumination and watch the servo change position to suit.

How the software works

Just like when driving an LED, the Arduino uses PWM to vary the duty cycle on the special PWM-capable outputs. The servo reads the PWM duty cycle and translates it into an angular position.

Further experiments

Replace the light sensor with a variable resistor, and control the servo position manually by changing the resistor position.

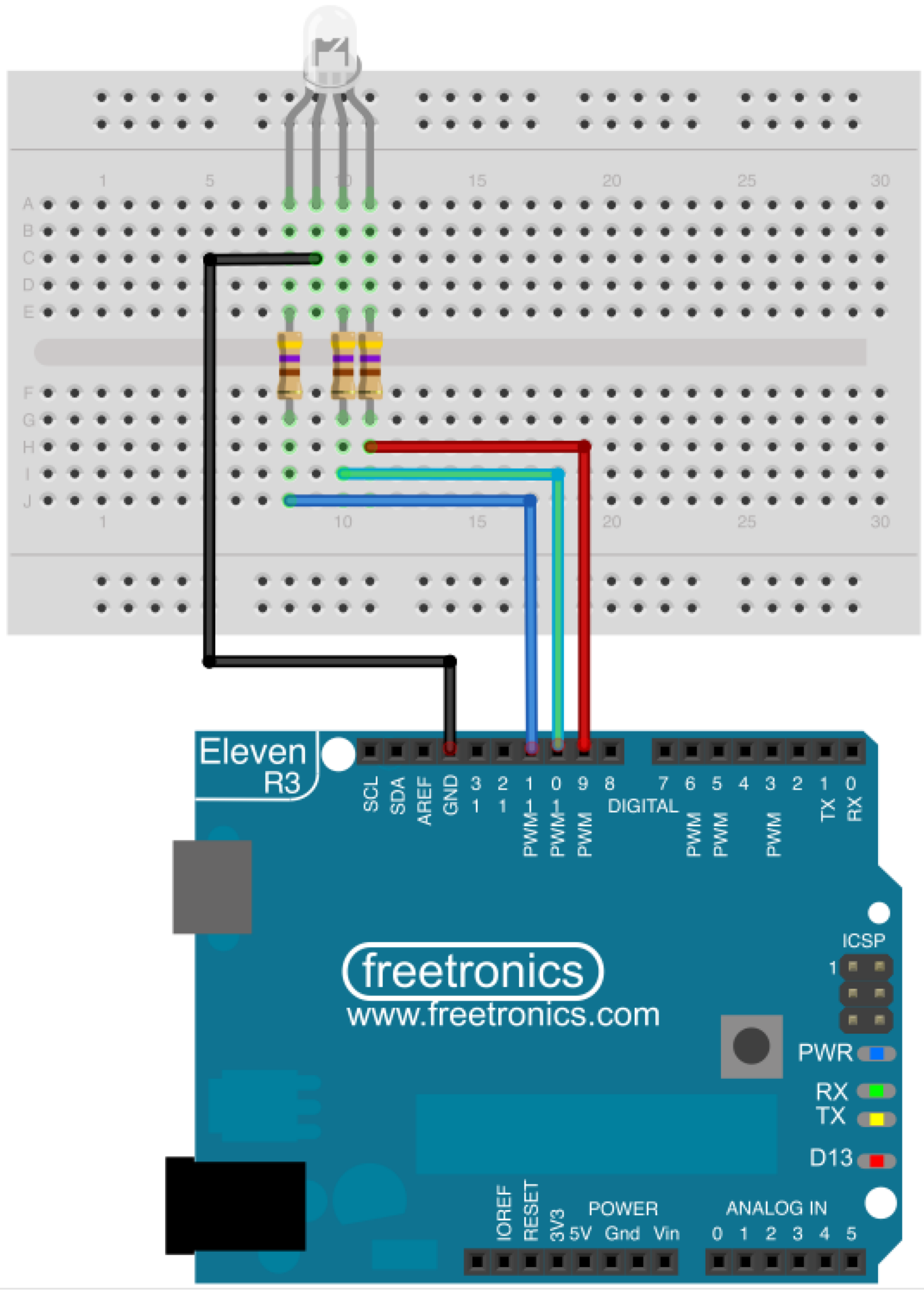
Or read values sent to the sketch via USB from the Serial Monitor in the Arduino IDE, so that you can type in a value and have the servo move to that position.

Project 7: RGB LED

Single-colour LEDs are great for most purposes, but an RGB LED gives you the flexibility to display any colour you like.

Parts required

1 x “Eleven” Arduino-compatible microcontroller board
1 x RGB LED (common-cathode type)
3 x 470 Ohm resistors (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x Solderless breadboard
4 x Breadboard jumper wires



Hardware assembly

If you look carefully at the four leads of the RGB LED you'll see that one is longer than the others. That lead is the common cathode that each of the three LED elements inside the RGB LED shares with its siblings.

Spread the LED legs apart just enough that all four can be inserted into adjacent rows of the solderless breadboard.

Place the three resistors into the breadboard, so that one end of each resistor is connected to a colour pin of the LED and the other end is on a row of its own.

Use one of the jumper wires to connect from the common cathode (long) pin of the LED to the GND connection on the Arduino.

Use the other three jumper wires to connect from the ends of the resistors to Arduino digital pins D9, D10, and D11.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int redLedPin = 9;
int greenLedPin = 10;
int blueLedPin = 11;

void setup() {
  pinMode(redLedPin, OUTPUT);
  pinMode(greenLedPin, OUTPUT);
  pinMode(blueLedPin, OUTPUT);
}

void loop()  {
  analogWrite(blueLedPin, random(0, 255));
  analogWrite(greenLedPin, random(0, 255));
  analogWrite(redLedPin, random(0, 255));

  delay(500);
}
```

How the software works

The particular pins we picked are PWM-capable, which provides the flexibility to vary the brightness on each colour element. That means we can write to each of them using the analogWrite() function, which sets them to a duty cycle between 0% (0 value) and 100% (255 value) per colour.

However, we're not hard-coding values to send to analogWrite(): instead we call the "random()" function to generate a random number between 0 and 255, and send that instead. By randomly setting each of the three colour elements individually and pausing for half a second each time, the RGB LED will flicker to a different random colour twice per second. Funky!

Further experiments

You could use a light sensor as a simple proximity sensor, assuming the light readings get dimmer as you move your hand closer to it and cast more of a shadow, and use the value read from it to smoothly go from “cold” blue when your hand is far away to “hot” red when it’s close.

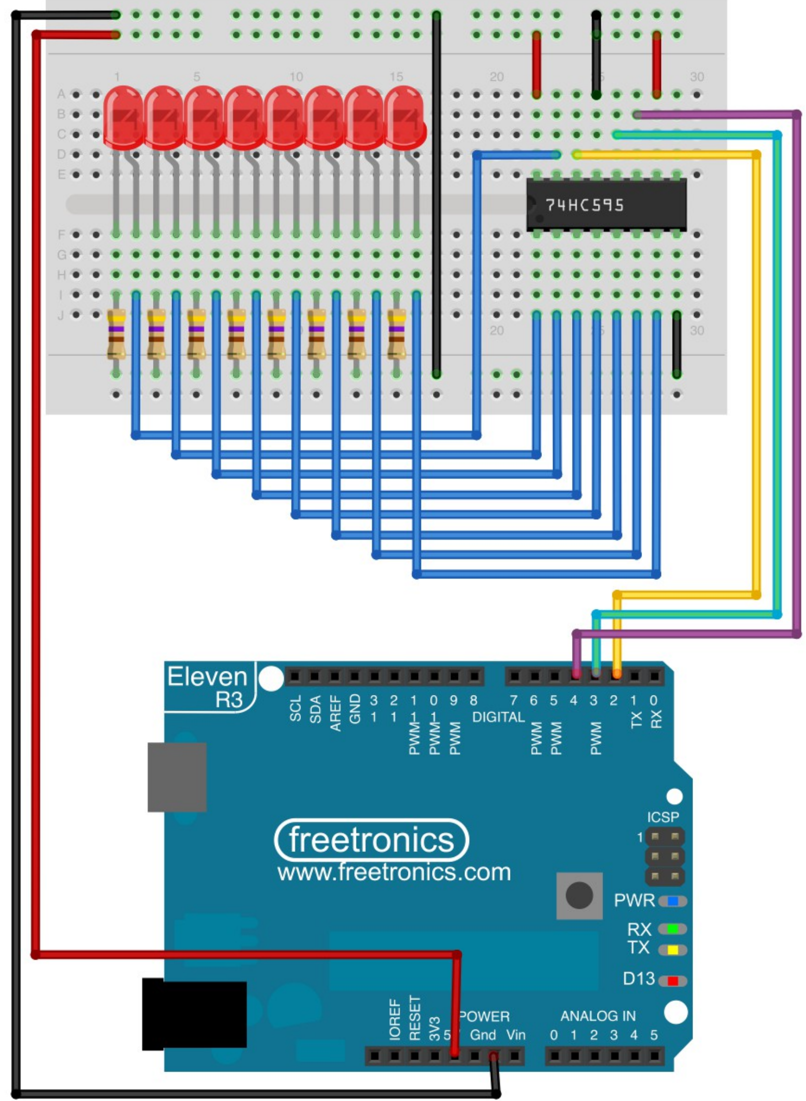
Project 8: Drive More Outputs With A Shift Register

Microcontrollers have a limited number of input and output pins available for use in your projects, and it's not unusual to run out of pins when you want to control many things at once. A simple solution is a device called a "shift register", which has many outputs that can be controlled individually using just three pins from your Arduino. By sending different values to the shift register you can turn its outputs on or off at will.

You can even connect multiple shift registers together in a row called a "daisy chain", giving you the ability to control a huge number of outputs while still using just three pins on the Arduino.

Parts Required

1 x "Eleven" Arduino-compatible microcontroller board
1 x 74HC595 shift-register IC
8 x LEDs (any colour)
8 x 470 Ohm resistors (yellow-violet-black-black-brown, or yellow-violet-brown-gold)
1 x Solderless breadboard
18 x Breadboard jumper wires



Hardware Assembly

If this is your first time using an IC (Integrated Circuit) it may seem a bit scary because it has so many pins, but working with ICs is an important skill to learn.

Before you begin assembly there is one very important thing to remember about most ICs: they can be damaged by invisible static electricity that builds up on your skin, clothes, and workbench. Simply touching the IC in the wrong way can be enough to kill it forever. Some ICs aren't susceptible to static electricity damage, but many are, so the safest thing to do is to assume that all ICs are static-sensitive unless specified otherwise.

Don't worry, it's not as bad as it sounds. As long as you follow some simple precautions you'll be fine. Professional electronics labs often have special work surfaces and anti-static wrist straps, but for experiments on your kitchen table you don't need to go that far. The important thing is to minimise contact with the pins of the IC once it has been removed from its protective foam. Grasp the IC by the ends of the package, and don't touch the pins to anything that may have a static charge.

Once you've removed the 74HC595 from its protective foam you'll notice something annoying: the pins aren't straight! IC pins are made to splay out slightly so that they can be inserted into circuit boards with a special tool that squeezes the pins together, and then when the tool is released the pins spring back out a bit to hold it firmly in place prior to soldering. That means you'll need to bend the pins a bit before you can fit the IC into a solderless breadboard.

To bend the pins, hold the IC package side-on against a hard surface such as a wooden table top and push down hard. They'll be quite difficult to bend but you don't want to bend them too far, so the trick is pushing hard enough but not too hard! Try to bend both sides in just enough that they are parallel to each other.

Next, check the orientation of the IC and then push it into the solderless breadboard so that the two rows of pins are on either side of the centre line and the dimple or dot near one end is at the top. This will allow you to easily connect jumper wires to any pin of the IC.

The 74HC595 shift register has a few different features that we're not going to use, so the first thing to do is connect various pins to either GND or 5V to make the chip operate correctly.

With the chip oriented as described above, you'll notice that the bottom left pin connects to GND and the top right pin connects to 5V. This is a loose standard used by most ICs that have two rows of pins. Use jumper wires to connect those pins to the GND and 5V rails on the solderless breadboard.

There are two more pins we need to set to specific values: RESET and OUTPUT ENABLE. Use another jumper wire to connect RESET to 5V, and OUTPUT ENABLE to GND.

Now we're ready to connect up all the LEDs to the outputs of the chip. The outputs are labelled A through H, and each needs to connect to the + (anode) side of an LED. The - (cathode) side of each LED then needs to connect to a 470 Ohm resistor (yellow-violet-black-black-brown, or yellow-violet-brown-gold) which also connects to GND.

The final three connections are the control lines from the Arduino, the vital links that will tell the shift register which outputs should be turned on.

IN (input) to Arduino digital I/O pin D2.
LCLK (latch clock) to Arduino digital I/O pin D3.
CLK (clock) to Arduino digital I/O pin D4.

Once it's all connected up you'll notice there is one remaining pin on the shift register that doesn't have anything connected to it: OUT. That's OK, it's meant to remain unconnected for this project.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int dataPin = 2;
int latchPin = 3;
int clockPin = 4;

void setup() {
  pinMode(dataPin, OUTPUT);
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
}

void loop() {
  for (int currentValue = 0; currentValue < 256; currentValue++) {
    // Disable the latch while we clock in data
    digitalWrite(latchPin, LOW);

    // Send the value as a binary sequence to the module
    shiftOut(dataPin, clockPin, MSBFIRST, currentValue);

    // Enable the latch again to set the output states
    digitalWrite(latchPin, HIGH);

    delay(200);
  }
}
```

How the software works

The explanation behind this sketch may sound a bit brain-bending at first, but once you understand what's going on you'll see it's not really that complicated.

The short version is that the 8 outputs of the shift register represent the individual "bit" values of a single "byte" of data. A "byte" is a value that consists in binary form of 8 bits, with each bit having double the place value of the previous bit when read from right to left.

Sound confusing? OK, let’s break it down.

One “byte” can be represented in binary form as 8 bits. Each bit can have one of only two values: either 0 or 1, off or on. Just like with regular decimal numbers, the position of each digit changes its value: the number “50” in decimal means “five tens”, and it’s a bigger number than “5”, which means “five ones”. In just the same way, the binary number “100” (which is not one hundred in decimal, by the way!) has a different value to the binary number “10”, because the place-value of the digits is different.

Position	8	7	6	5	4	3	2	1
Decimal place value	10000000	1000000	100000	10000	1000	100	10	1
Binary place value	128	64	32	16	8	4	2	1

Since a byte has eight bits, and each bit can have a maximum value of 1, the biggest possible number that can fit inside one byte is the binary number “11111111”, which is equivalent to a decimal value of 255: 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1. One byte can therefore store any number between 0 and 255 decimal.

Now stop for a second and look at that binary number again. That’s eight bits, all in a row, and every bit can have the value of either 0 or 1. Now imagine that each one of those bits is wired up to one of the eight outputs (A through H) of the shift register, so that if that particular bit is set to 1 then the output is on, and if that particular bit is set to 0 then the output is off.

Got it? Well, that’s exactly how a shift register works! The eight outputs are just physical representations of the state of each of the eight bits inside that byte. And by sending a different value for the byte, we can manipulate which outputs are turned on and which are turned off.

Remember that the bits are counted from the right in increasing place value (just like decimal numbers) so to set a specific set of outputs on, you just work out the bits to represent those outputs.

Want to turn on outputs A, C, and E, and leave all the others off? Easy, just map it out like this, remembering that digits to the left have higher place value so we’ll label it starting with “A” on the right and working to the left like we would with a decimal number:

H	G	F	E	D	C	B	A
0	0	0	1	0	1	0	1

That’s binary “00010101”, and from the previous table we can add up the place values of those bits (16 + 4 + 1) to give a decimal value of “21”. So if you send a value of 21 to the shift register, it will turn on outputs A, C, and E, and turn all the others off.

You can use this same technique to figure out how to turn on any combination of outputs, and then map it down to a single value in either decimal or binary that will achieve that end result.

Now with those principles in mind, look at the sketch again and it may make a bit more sense.

First it sets up the connections to the shift register input, and to the CLK (clock) and LCLK (latch clock) pins.

Then, in the main loop, it uses a “for” loop to repeat over an incrementing number. The “for” loop starts with a variable called “currentValue” which begins with a value of 0, and keeps repeating as long as the value is less than 256. Each time it repeats, the value increases by 1, so it will go through with a value of 0, then a value of 1, then a value of 2, then a value of 3, and so on until the value reaches 255. On the next time through the value will reach 256, which hits the limit that has been set and causes the program to move on from the “for” loop.

After the “for” loop has finished there are no more commands, so the main loop then just goes back to the start, and begins all over again running the “for” loop from scratch with a starting value of 0 again. That just repeats forever, so what the sketch does is count from 0 to 255, then go back and do it again.

But the interesting bit is what the sketch does with that value each time it runs through the “for” loop. Inside the “for” loop are three commands: first to change the state of the “LCKL” (latch clock) pin, then a command to send the current value to the register using the IN pin, then to change the state of LCLK back again.

Those three commands, in that sequence, will cause the Arduino to take the value of the “currentValue” variable and send it to the shift register to set the state of all the outputs.

Using the "shiftOut" command we can clock a byte of data (value 0 to 255) to the module, setting the state of each output appropriately. For example, sending a value of "0" decimal will set all the outputs LOW. Sending a value of "255" decimal will set all of the outputs HIGH. Sending a value of 1, 2, 4, 8, 16, 32, 64, or 128 decimal will set only output A, B, C, D, E, F, G, or H HIGH respectively.

The sketch below counts from 0 to 255 and sends each consecutive value to the shift register

Further Experiments

If you want to be able to manipulate individual outputs from within your sketch without affecting the state of other outputs, the most convenient method is probably to store the current value of the shift register as a 1-byte variable and apply bitwise operators to it. The Arduino website has a good introduction to bitwise operators at http://www.arduino.cc/playground/Code/BitMath.

Sometimes eight outputs aren’t enough! You can connect multiple shift registers together in a row (called a “daisy chain”) so that you can pass values down the chain to each of the modules individually. That way you can control 16, 24, 32, or even more separate outputs with the same three outputs from your Arduino.

To daisy-chain multiple shift registers, begin with the simple example above. Then connect a second shift register so that the VCC, GND, RST, OE, CLK, and LCLK connections are all linked to the same place as the first shift register. Finally, run a jumper wire from the OUT connection on the first shift register to the IN connection on the second shift register.

Once you have two shift registers daisy-chained together, setting their outputs is just as easy as addressing a single register except that you have to send two bytes of data instead of one. The first byte sent will pass down the chain to the last register, and the second byte sent will then go to the

first register: think of it as pushing messages down a pipe. The first one you push in will go towards the end, and the second message will go in right behind it.

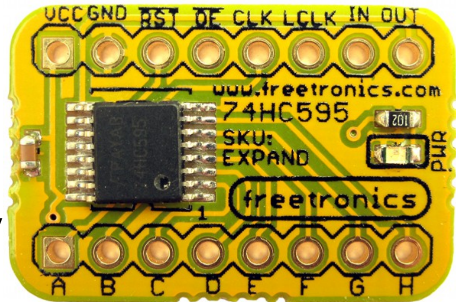
Using three or more shift registers follows the exact same principle: simply daisy-chain more together linking the OUT connection of each into the IN connection of the next in the chain, and send as many bytes of data as you have shift registers in the chain.

Addendum: Understanding Shift Register Connections

Serial OUT: Passes serial data back out to another module. Can be connected to the Serial IN of another module to daisy-chain them together and control even more outputs. Typically left unconnected.
Serial IN: Data sent from the microcontroller to set the state of the output pins. Connect to a digital I/O pin of your microcontroller.
Latch: Also sometimes referred to as the "storage clock" or "latch clock". On a LOW-to-HIGH transition this pin causes the values loaded into the shift register to be applied to the outputs. Connect to a digital I/O pin of your microcontroller.
Clock: On a LOW-to-HIGH transition this pin causes the value currently applied to the Serial IN pin to be read into the shift register, and all other values moved along one position.
Output Enable: Active LOW. When pulled HIGH, all outputs are disabled and placed in a high-impedance state. When pulled LOW, all outputs are enabled. For minimal pin usage this can be tied to GND to permanently enable the outputs, but this will have the effect that they may start in an unknown state at power-up. Connect to a digital I/O pin of your microcontroller if you want to explicitly enable outputs from software, otherwise connect to GND.
Reset: Active LOW. When pulled LOW, the shift register is cleared and all stored values are lost. Pull HIGH for normal operation. For minimal pin usage this can be permanently tied to 5V to disable reset, but this will require you to explicitly set all 8 bits in order to reset the outputs if they are not in a known state. Connect to a digital I/O pin of your microcontroller if you want to be able to reset the state with a single pulse, but generally it's simplest to connect to 5V.
GND: Connect to GND (0V) on your microcontroller.
VCC: Connect to 5V on your microcontroller.
Outputs A to H: Outputs to your other devices. Can be driven LOW, HIGH, or be set into a high-impedance state by setting Output Enable to LOW.

Shift Register / Expansion Module

In the Experimenters Kit we've supplied a 74HC595 shift register as a loose IC to give you the opportunity to learn how to work directly with bare ICs. As an alternative, we also have the EXPAND Shift Register / Expansion Module with a 74HC595 fitted to it along with a power LED and power supply smoothing capacitor, with all the control lines and outputs broken out to handy connection terminals.

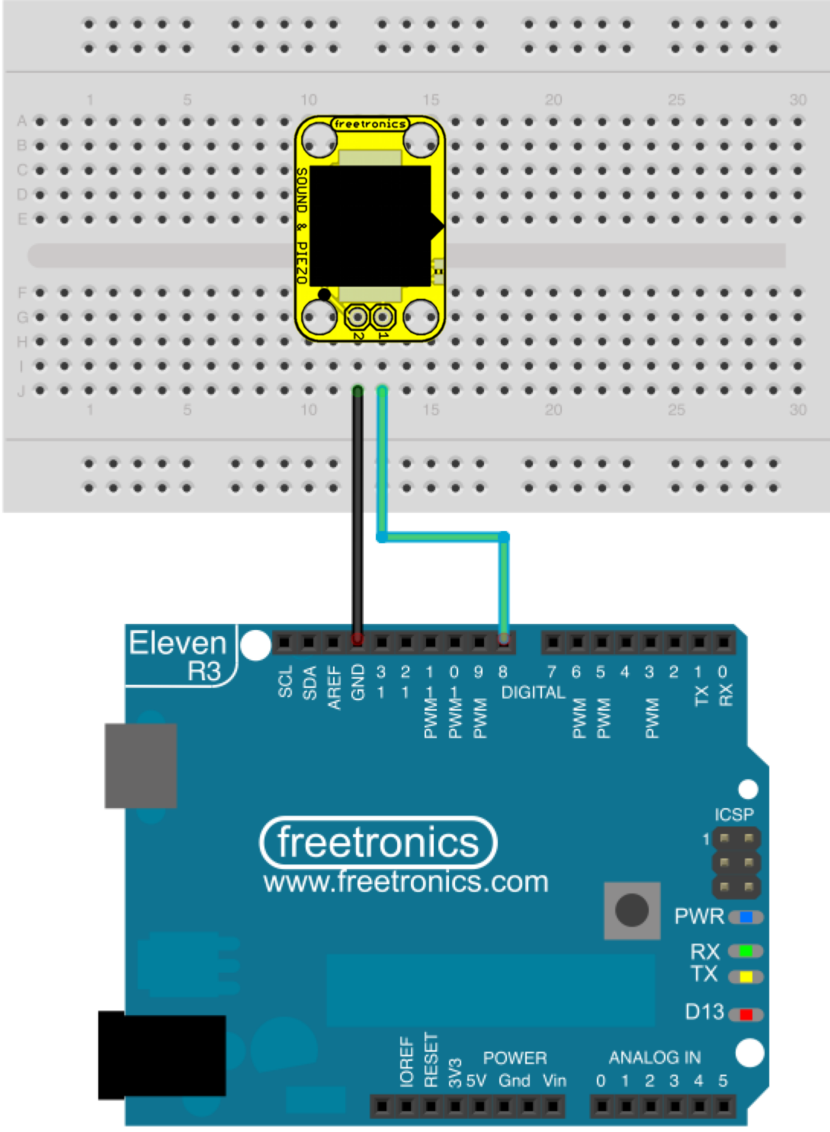


Project 9: Making Sounds

The sound and piezo module can be used for creating a variety of sounds by applying an electrical current. In this project we'll create sounds that are relative to musical notes.

Parts Required

1 x "Eleven" Arduino-compatible microcontroller board
1 x "SOUND" sound and piezo module
1 x Solderless breadboard
2 x Breadboard jumper wires



Hardware assembly

Connecting your module is very easy, as it is not polarised. Use one of the jumper wires to connect a pin of the module to the GND on the Arduino, and the second jumper wire between the other module pin and the Arduino digital pin D8.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int piezo = 8;
int duration = 250;

int notes[] =
{261, 293, 329, 349, 392, 440, 493, 523, 587, 659, 698, 783, 880};
// frequencies for musical notes - from middle C, D, E, F, G, A, B, C, D, E, F, G,
A

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

void loop()
{
  for (int i = 0; i < 13; i++)
  {
    tone(piezo, notes[i], duration);
    delay(duration);
  }
  for (int i = 11; i > 0; --i)
  {
    tone(piezo, notes[i], duration);
    delay(duration);
  }
}
```

How the software works

We use the tone() function to send a signal to the module which creates a sound at a certain frequency. The three parameters are the digital pin the module is connected to, the frequency of tone required, and the duration in milliseconds. If playing tones one after another, you need a delay() after the tone() with a matching duration.

In the sketch we have defined the pin for the module as 8, and also filled the array “notes” with frequencies for the main musical notes from middle C. You can find out more note frequencies from the website http://bit.ly/notefrequency.

After connecting the hardware and uploading the sketch, the tones will be played repeatedly in an ascending then descending order.

Further experiments

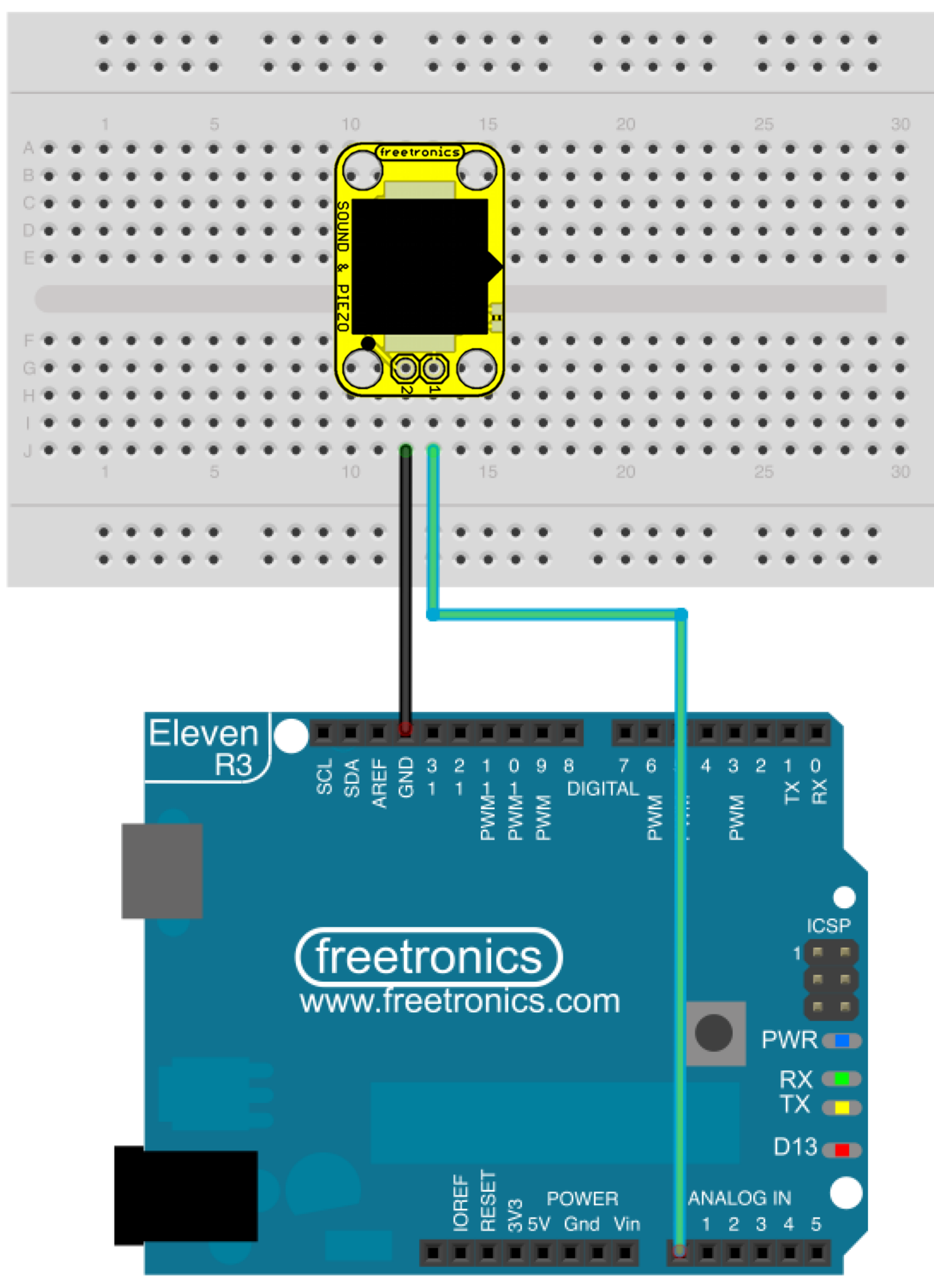
Just about any project can benefit from sound output, to be used as a warning, alarm tone, or to play a song for a game or as a pleasant notifier.

Project 10: Detecting Vibration and Knocks

We've already used the Sound & Piezo Module to generate sounds: now we'll use the same module as a form of input, by measuring vibration and knocks using an analog input pin on your Arduino.

Parts required

1 x "Eleven" Arduino-compatible microcontroller board
1 x "SOUND" sound and piezo module
1 x Solderless breadboard
2 x Breadboard jumper wires



Hardware assembly

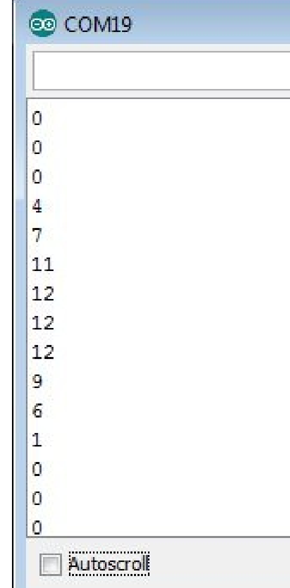
This time use one of the jumper wires to connect a pin of the module to the GND on the Arduino, and the second jumper wire between the other module pin and the Arduino analog pin A0.

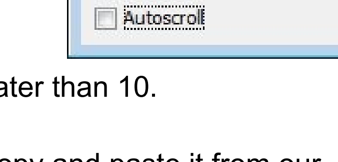
Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int knock = 0;

void setup()
{
  Serial.begin(38400);
}

void loop()
{
  knock = analogRead(0);
  Serial.println(knock);
}
```

How the software works





As a piezo can generate mild electrical currents, we measure these using the analogue input pin A0, then display the value on the serial monitor. After uploading the sketch, open the serial monitor window and set the speed to 38400.

The numbers displayed are a representation of the level of vibration or shock felt by the piezo. If you tap the module using various strengths, you can see the numbers increase then decrease. The example in the screenshot shows the effect of tapping the module with a finger.

Further experiments

Using the values returned from the sketch, you can now use your Arduino and the module to make decisions based on the strength of knock received by the module. We demonstrate this using the following sketch - when a knock is received, the onboard LED will light for half a second. In the sketch we consider a “knock” to occur when the value from the analogue input is greater than 10.

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int knock = 0;

void setup()
{
  pinMode(13, OUTPUT);
```

```
}

void loop()
{
  knock = analogRead(0);
  if (knock > 10)
  {
    digitalWrite(13, HIGH);
    delay(500);
    digitalWrite(13, LOW);
  }
}
```

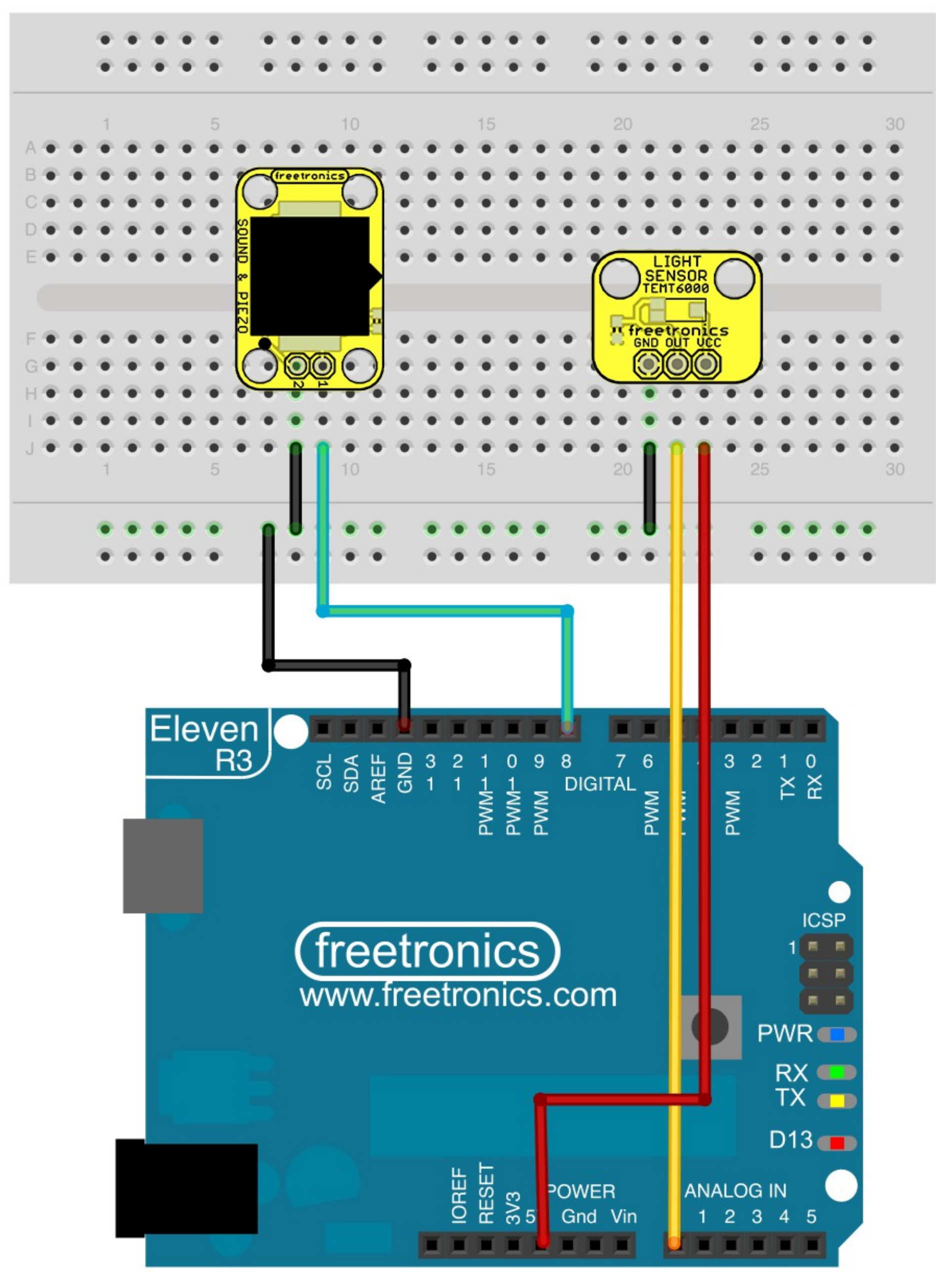
The ability to detect vibrations and knocks opens up many possibilities, including alarm and movement sensors, making your own “knock lock” that activates after receiving a knock of a certain strength, or just another form of input.

Project 11: Light Input Controlling Sound Output

The light sensor module is very easy to use - it returns an electrical signal that has a strength relative to the amount of light on the sensor. By reading this signal with an analog input pin, you can program the Arduino to make decisions based on the light levels. We'll do just that in this project, by creating a device that outputs a tone whose frequency is relative to the ambient light level.

Parts required

1 x "Eleven" Arduino-compatible microcontroller board
1 x "SOUND" sound and piezo module
1 x "LIGHT" light sensor module
1 x Solderless breadboard
5 x Breadboard jumper wires



Hardware assembly

First use one of the jumper wires to connect a pin of the sound module to the GND on the Arduino, and the second jumper wire between the other module pin and the Arduino digital pin D8. Next, connect the 5V pin from the light module to the Arduino 5V pin, the light module GND to Arduino GND and finally the light module OUT (the centre pin) to Arduino analogue pin A0.

Software

Open a new sketch in the Arduino IDE and type in the following code, or copy and paste it from our site:

```
int lightLevel;
int piezo = 8;
int duration = 300;

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

void loop()
{
  lightLevel = analogRead(A0);
  tone(piezo, lightLevel, duration);
  delay(duration);
}
```

How the software works

In the same method as the knock sensor in project #10, the current from the light sensor is measured using analog pin A0 and then stored in the variable lightLevel. This is then passed to the tone() function which plays a tone with the piezo at a frequency which is the light level value.

After assembling the hardware and uploading the sketch, move your hand over the light sensor. You should be able to hear the frequency or pitch of the sound increase with the amount of light falling onto the sensor.

Further experiments

As the light levels measured by the sensor are converted to numbers your Arduino can work with, the possibilities are almost endless. Create a nightlight, light-activated burglar alarm, or perhaps a "cricket" that only squeaks when the lights are off.

Recommended Tools For Advanced Projects

To get started with the Experimenters Kit for Arduino you don't need any tools at all, but once you get to the more advanced projects or want to work on your own designs there are a few handy items that will make things much easier for you.

Sidecutters

Sidecutters are the number one tool of any electronics hobbyist: you can use them to cut wire, strip insulation, trim component leads, and many other uses.

Like most tools, sidecutters can range from very cheap to ridiculously expensive, mostly based on the strength of the steel and how well the blades hold a sharp edge. When getting started you can get by with a cheap pair, but eventually you will probably want some good quality cutters with hard blades.

Multimeter

A multimeter is a multi-purpose test instrument for measuring all sorts of electrical properties, including the three fundamentals: voltage, current, and resistance. Many advanced multimeters can also measure capacitance, diode voltage drop, transistor gain, and other properties, but most of the time the critical feature is simply measuring voltage. All those other features are nice to have but even a $10 bargain-bin special from your local electronics store will be better than nothing when you're just getting started.

Multimeters have two leads: black (negative) and red (positive) that can be connected to your circuit in different ways depending on what you are measuring. The most common way to use a multimeter for hobbyist projects is to connect the negative test probe to a "ground" (0V) point somewhere on the circuit, and then use the positive probe to measure the voltage at other locations in the circuit.

Soldering Iron and Solder

Most of the examples in this guide can be completed without doing any soldering, but when you want to do more advanced projects you'll need a soldering iron and solder. Check your local electronics shop for a basic hobbyist-level soldering iron and a small roll of thin solder. Remember, soldering irons get very hot - safety first!

For an introduction to soldering techniques, see:

www.freetronics.com/how-to-solder

More Information And Getting Help

Experimenter’s Kit Page

Check out the web page for the Experimenter’s Kit to see updates and handy links, including easy access to all the source code used in this guide:

www.freetronics.com/expkit

Freetronics Blog

Regular stories each day about amazing things people have built using Arduino. Great for a daily dose of inspiration:

www.freetronics.com/blog/news

Freetronics Forums

Discuss your projects or ask for help on the Freetronics Forum. It’s filled with helpful hackers and Freetronics staff, so no matter what your electronics problem is you’re likely to find someone here to help figure it out:

forum.freetronics.com

Book “Practical Arduino” by Jonathan Oxer and Hugh Blemings

If you’re feeling adventurous, the 456-page book “Practical Arduino” by Freetronics co-founder Jonathan Oxer and his pal Hugh Blemings will show you substantial projects and advanced project-building skills:

www.freetronics.com/practical-arduino

Book “Arduino Workshop” by John Boxall

The best way to learn is by doing things yourself, and the book “Arduino Workshop” by Freetronics engineer John Boxall takes you through dozens of projects showing you a wide variety of techniques:

www.arduinoworkshop.com

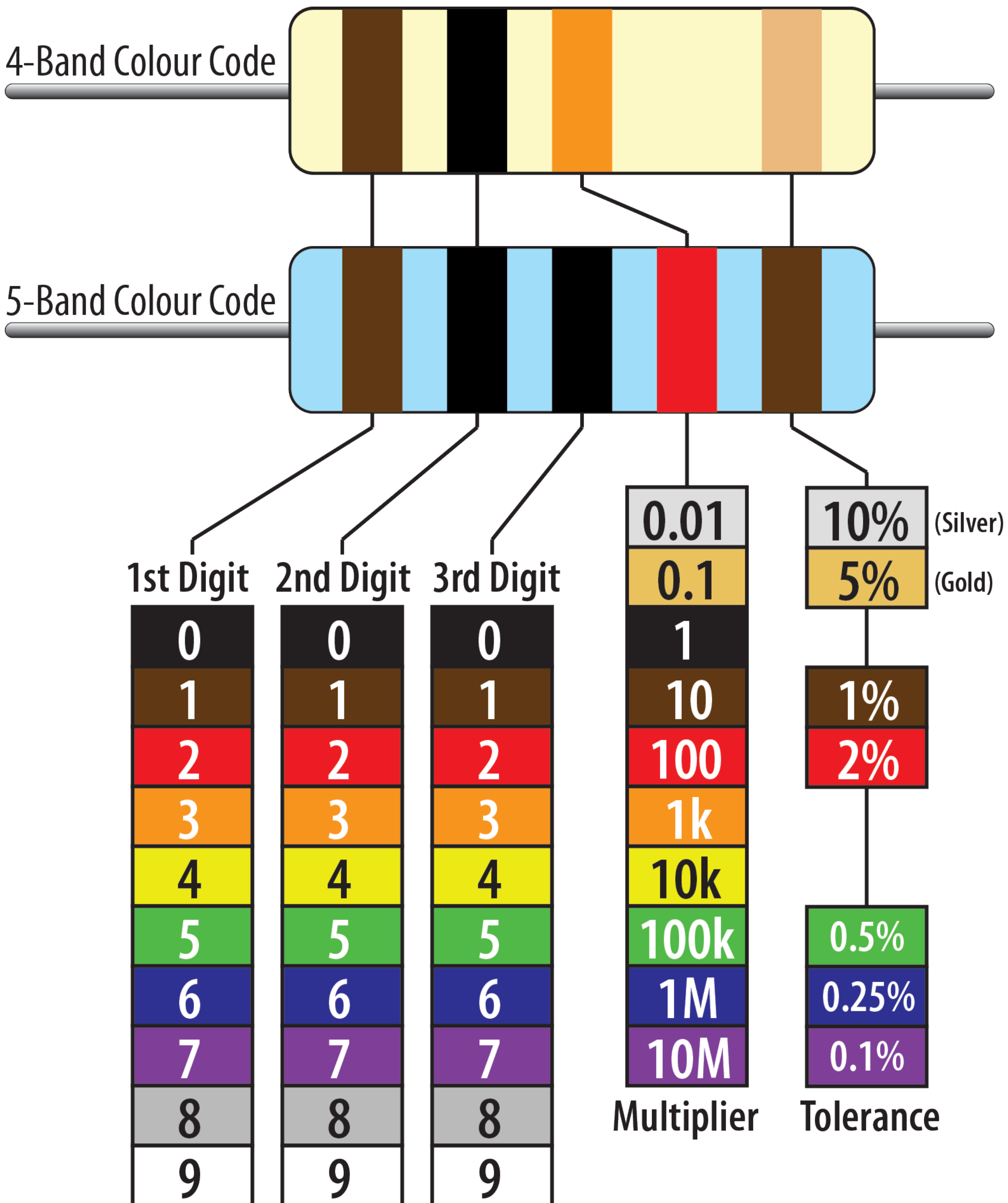
Find A Local Hackerspace

Hackerspaces are amazing places where people meet to work on projects, learn new skills, use tools, and help each other achieve their creative dreams. There are hundreds of hackerspaces all over the world, and it’s likely there’s one near you:

www.hackerspaces.org

Reading Resistor Colour Codes

The resistors supplied in the Experimenters Kit use colour codes to designate their values. You'll come across colour codes quite frequently as you tackle more so keep this reference handy to help you decode them. You can stick it on the wall over your workbench so you can refer to it any time!



In the example, the 5-band resistor has the first band brown (1), second band black (0), third band black (0), 4th band is red (x100 multiplier), so the resistor value is 10,000 Ohms: usually written as 10k.

The easiest way to understand the multiplier is to look at what number that colour represents (red is 2) and just add that many zeros after the other digits.

For more resistor colour code information, see www.freetronics.com/resistors

The Experimenter's Kit for Arduino is just the start of the adventure. See www.freetronics.com for dozens of other Arduino-compatible boards, shields, sensor modules, output modules, power supplies, displays, books, and other accessories.

Arduino-Compatible Boards

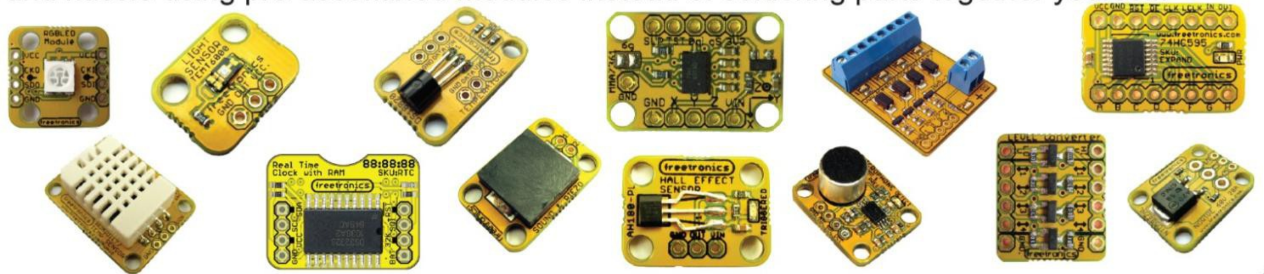
With Arduino-compatible boards ranging from the super-tiny to the super-powerful, we've got you covered.

Expansion Shields

Quick and easy expansion for your Arduino: our shields simply plug in on top, giving you extra features to make it easy to finish your project.

Modules

Our modules act as handy building blocks for you to create your own functionality. Save time and hassle using pre-assembled modules instead of soldering parts together yourself.



Parts, Books, and Accessories

Brilliant displays, in-depth books, and all the parts you need to bring your inventions to life.