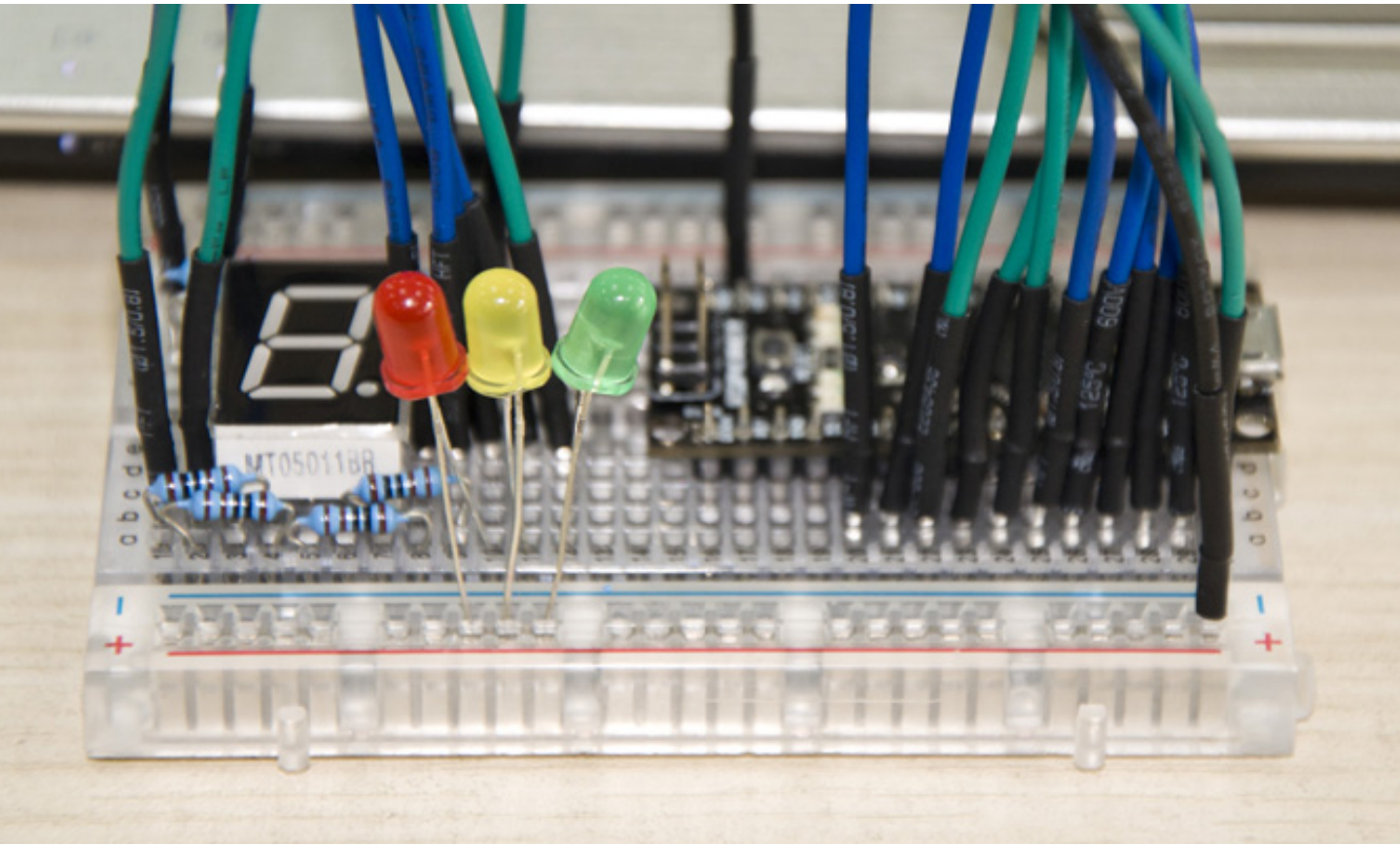
ARDUINO beginners KIT



What Exactly is an Arduino?

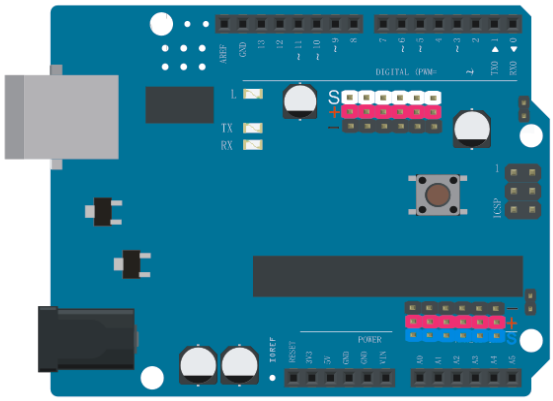


Figure 1-1. *An Arduino Uno*

Wikipedia states "*An Arduino is a single-board microcontroller and a software suite for programming it. The hardware consists of a simple open hardware design for the controller with an Atmel AVR processor and on-board I/O support. The software consists of a standard programming language and the boot loader that runs on the board.*"

To put that in layman's terms, an Arduino is a tiny computer that you can program to process inputs and outputs between the device and external components you connect to it (see Figure 1-1). The Arduino is what is known as a Physical or Embedded Computing platform, which means that it is an interactive system that can interact with its environment through the use of hardware and software. For example, a simple use of an Arduino would be to turn a light on for a set period of time, let's say 30 seconds, after a button has been pressed. In this example, the Arduino would have a lamp and a button connected to it. The Arduino would sit patiently waiting for the button to be pressed; once pressed, the Arduino would turn the lamp on and start counting. Once it had counted for 30 seconds, it would turn the lamp off and then wait for another button press. You could use this setup to control a lamp in an closet, for example.

You could extend this concept by connecting a sensor, such as a PIR, to turn the lamp on when it has been triggered. These are some simple examples of how you could use an Arduino.

The Arduino can be used to develop stand-alone interactive objects or it can be connected to a computer, a network, or even the Internet to retrieve and send data to and from the Arduino and then act on that data. In other words, it can send a set of data received from some sensors to a website, which can then be displayed in the form of a graph.

The Arduino can be connected to LEDs, dot matrix displays (see Figure 1-2), buttons, switches, motors, temperature sensors, pressure sensors, distance sensors, GPS receivers, Ethernet modules, or just about anything that outputs data or can be controlled. A look around the Internet will bring up a wealth of projects where an Arduino has been used to read data from or control an amazing array of devices.

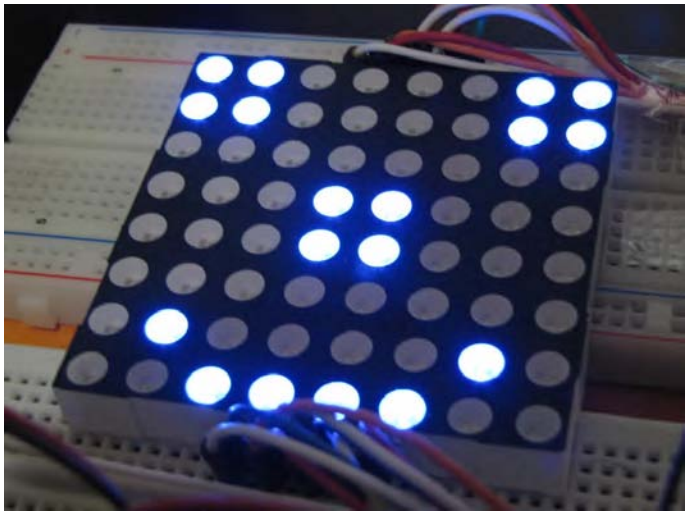


Figure 1-2. *A dot matrix display controlled by an Arduino (image courtesy of Bruno Soares)*

The Arduino board is made up of an Atmel AVR Microprocessor, a crystal or oscillator (a crude clock that sends time pulses at a specified frequency to enable it to operate at the correct speed), and a 5-volt linear regulator. Depending on what type of Arduino you have, it may also have a USB socket to connect to a PC or Mac for uploading or retrieving data. The board exposes the microcontroller's I/O (input/output) pins so that you can connect those pins to other circuits or to sensors.

The latest Arduino board, the Uno, differs from the previous versions of the Arduino in that it does not use the FTDI USB-to-serial driver chip. Instead, it uses an Atmega8U2 programmed as a USB-to-serial converter. This gives the board several advantages over its predecessor, the Duemilanove. First, the Atmega chip is a lot cheaper than the FTDI chip, bringing the prices of the boards down. Secondly, and most importantly, it enables the USB chip to have its firmware reflashed to make the Arduino show up on your PC as another device, such as a mouse or game controller. This opens up a whole array of new uses for the Arduino. Unfortunately, moving over to this new USB chip has made it a lot more difficult for clone manufacturers to make Arduino Uno clones.

To program the Arduino (make it do what you want it to) you use the Arduino IDE (Integrated Development Environment), which is a piece of free software in which you write code in the language that the Arduino understands (a language called C). The IDE lets you to write a *computer program*, which is a set of step-by-step instructions that you then upload to the Arduino. Your Arduino will then carry out these instructions and interact with whatever you have connected to it. In the Arduino world, programs are known as *sketches*.

The Arduino hardware and software are both open source, which means that the code, schematics, design, etc. can be taken freely by anyone to do what they like with them. Hence, there are many clone boards and other Arduino-based boards available to purchase or to make from a schematic. Indeed, there is nothing stopping you from purchasing the appropriate components and making your own Arduino on a breadboard or on your own homemade PCB (Printed Circuit Board). The only caveat that the Arduino team imposes is that you cannot use the word "Arduino." This name is reserved for the official board. Hence, the clone boards have names such as Freeduino, Roboduino, etc.

As the designs are open source, any clone board is 100% compatible with the Arduino and therefore any software, hardware, shields, etc. will also be 100% compatible with a genuine Arduino.

The Arduino can also be extended with the use of *shields*, which are circuit boards containing other devices (e.g. GPS receivers, LCD Displays, Ethernet modules, etc.) that you can simply connect to the top of your Arduino to get extra functionality. Shields also extend the pins to the top of its own circuit board so you still have access to all of them. You don't have to use a shield if you don't want to; you can make the exact same circuitry using a breadboard, Stripboard, Veroboard, or by making your own PCB. Most of the projects in this book are made using circuits on a breadboard.

There are many different variants of the Arduino. The latest version is the Arduino Uno. The previous version, the very popular Duemilanove (Italian for 2009), is the board you will most likely see being used in the vast majority of Arduino projects across the Internet. You can also get Mini, Nano, and Bluetooth variations of the Arduino. Another new addition to the product line is the Arduino Mega 2560; it offers increased memory and number of I/O pins. The new boards use a new bootloader called Optiboot, which frees up another 1.5k of flash memory and enables faster boot up.

Probably the most versatile Arduino, and hence the reason it is the most popular, is the Uno, or its predecessor, the Duemilanove. This is because it uses a standard 28-pin chip attached to an IC (Integrated Circuit) socket. The beauty of this system is that if you make something with an Arduino and then want to turn it into something permanent, instead of using a relatively expensive Arduino board, you can simply pop the chip out of the board and place it into your own circuit board in your custom device. By doing so, you have made a custom embedded device, which is really cool.

Then, for a couple of quid or bucks, you can replace the AVR chip in your Arduino with a new one. Note that the chip must be pre-programmed with the Arduino Bootloader (software programmed onto the chip to enable it to be used with the Arduino IDE), but you can either purchase an AVR Programmer to burn the bootloader yourself or you can buy a chip ready programmed; most of the Arduino parts suppliers provide these. It is also possible to program a chip using a second Arduino; instructions are available online for this.

Figure 1-3. *Anthros art installation by Richard V. Gilbank controlled using an Arduino*

If you do a search on the Internet for “Arduino,” you will be amazed at the large number of websites dedicated to the Arduino or that feature cool project created with an Arduino. The Arduino is an amazing device and will enable you to create anything from interactive works of art (see Figure 1-3) to robots. With a little enthusiasm for learning how to program an Arduino and make it interact with other components as well as a bit of imagination, you can build anything you can think of.

This book will give you the necessary skills needed to make a start in this exciting and creative hobby. Now that you know what an Arduino is, let’s get one hooked up to your computer and start using it.

Getting Started

This section will explain how to set up your Arduino and the IDE for the first time. The instructions for Windows and Macs (running OSX 10.3.9 or later) are given. If you use Linux, refer to the Getting Started instructions on the Arduino website at www.arduino.cc.playground/Learning/Linux. I will also presume you are using an Arduino Uno. If you have a different type of board, such as the Duemilanove (see Figure 1-4), then refer to the corresponding page in the Getting Started guide of the Arduino website.

You will also need a USB cable (A to B plug type) which is the same kind of cable used for most modern USB printers. If you have an Arduino Nano, you will need a USB A to Mini-B cable instead. Do not plug in the Arduino just yet, wait until I tell you to do so.

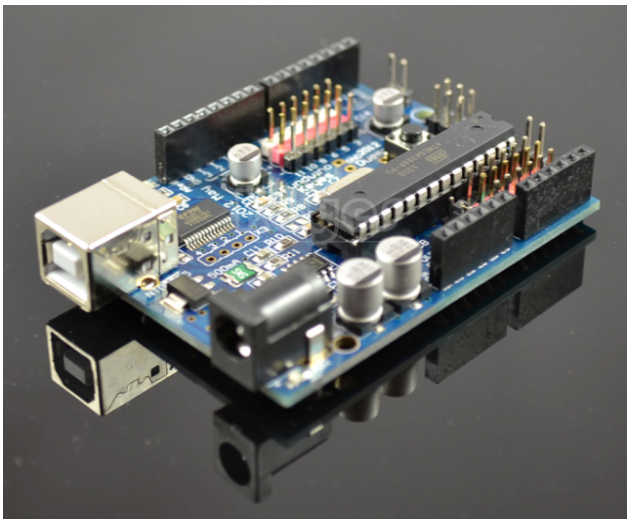


Figure 1-4. *An Arduino Duemilanove (image courtesy of Snorpey)*

Next, download the Arduino IDE. This is the software you will use to write your programs (or sketches) and upload them to your board. For the latest IDE go to the Arduino download page at http://arduino.cc/en/Main/Software and obtain appropriate the version for your OS.

Windows XP Installation

Once you have downloaded the latest IDE, unzip the file and double-click the unzipped folder to open it. You will see the Arduino files and sub-folders inside. Next, plug in your Arduino using the USB cable and ensure that the green power LED (labeled PWR) turns on. Windows will say "Found new hardware: Arduino Uno" and the Found New Hardware Wizard will appear. Click next and Windows will attempt to load the drivers. This process will fail. This is nothing to worry about; it's normal.

Next, right-click on the My Computer icon on your desktop and choose Manage. The Computer Management window will open up. Now go down to Event Manager in the System Tools list and click it. In the right hand window, you'll see a list of your devices. The Arduino Uno will appear on the list with a yellow exclamation mark icon over it to show that the device has not been installed properly. Right click on this and choose Update Driver. Choose "No, not this time" from the first page and click next. Then choose "Install from a list or specific location (Advanced)" and click next again. Now click the "Include this location in the search" and click Browse. Navigate to the Drivers folder of the unzipped Arduino IDE and click Next. Windows will install the driver and you can then click the Finish button.

The Arduino Uno will now appear under Ports in the device list and will show you the port number assigned to it (e.g. COM6). To open the IDE double-click the Arduino icon in its folder.

Windows 7 & Vista Installation

Once you have downloaded the latest IDE, unzip the file and double-click the unzipped folder to open it. You will see the Arduino files and sub-folders inside. Next, plug in your Arduino using the USB cable and ensure that the green power LED (labeled PWR) turns on. Windows will attempt to automatically install the drivers for the Arduino Uno and it will fail. This is normal, so don't worry.

Click the Windows Start button and then click Control Panel. Now click System and Security, then click System, and then click Device Manager from the list on the left hand side. The Arduino will appear in the list as a device with a yellow exclamation mark icon over it to show that it has not been installed properly. Right click on the Arduino Uno and choose "Update Driver Software."

Next, choose "Browse my computer for driver software" and on the next window click the Browse button. Navigate to the Drivers folder of the Arduino folder you unzipped earlier and then click OK and then Next. Windows will attempt to install the driver. A Windows Security box will open up and will state that "Windows can't verify the publisher of this driver software." Click "Install this driver software anyway." The Installing Driver Software window will now do its business. If all goes well, you will have another window saying "Windows has successfully updated your driver software. Finally click Close. To open the IDE double-click the Arduino icon in its folder.

Mac OSX Installation

Download the latest disk image (.dmg) file for the IDE. Open the .dmg file; it will appear like Figure 1-5.

Figure 1-5. *The Arduino .dmg file open in OSX*

Drag the Arduino icon over to the Applications folder and drop it in there. If are using an older Arduino, such as a Duemilanove, you will need to install the FTDI USB Serial Driver. Double-click the package icon and follow the instructions to do this. For the Uno and Mega 2560, there is no need to install any drivers.

To open the IDE, go into the Applications folder and click the Arduino icon.

Board and Port Selection

Once you open up the IDE, it will look similar to Figure 1-6.

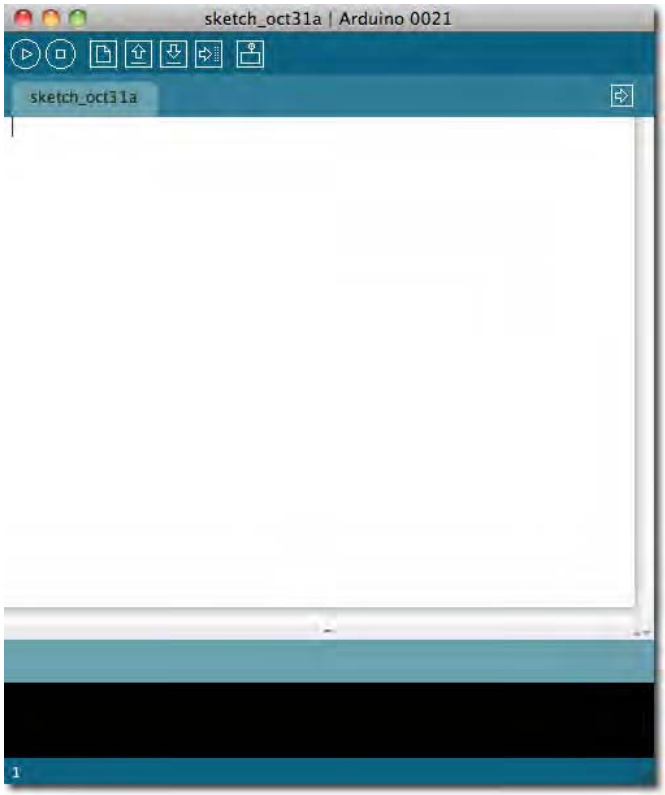


Figure 1-6. *The Arduino IDE when first opened*

Now go to the menu and click Tools. Then click Board (See Figure 1-7).

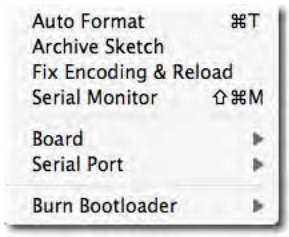


Figure 1-7. *The Arduino Tools menu*

You will now be presented with a list of boards (See Figure 1-8). If you have an Uno, choose that. If you have a Duemilanove or another Arduino variant, choose the appropriate one from the list.

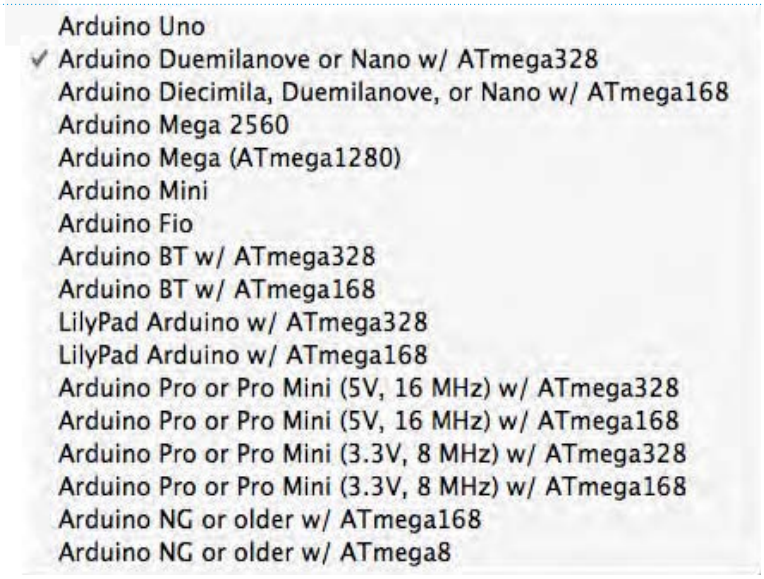


Figure 1-8. *The Arduino Boards menu*

Next, click the Tools menu again, click Serial Port, and then choose the appropriate port from the list for your Arduino (Figure 1-9). You are now ready to upload an example sketch to test that the installation has worked.

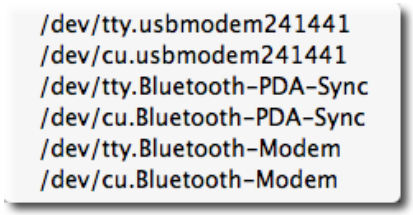


Figure 1-9. *The Serial Port list*

Upload Your First Sketch

Now that you have installed the drivers and the IDE and you have the correct board and ports selected, it's time to upload an example sketch to the Arduino to test that everything is working properly before moving on to the first project.

First, click the File menu (Figure 1-10) and then click Examples.

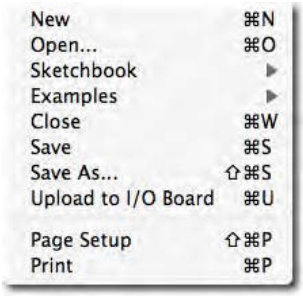


Figure 1-10. *The File menu*

You will be presented with a huge list of examples to try out. Let's try a simple one. Click on Basics, and then Blink (Figure 1-11). The Blink sketch will be loaded into the IDE.

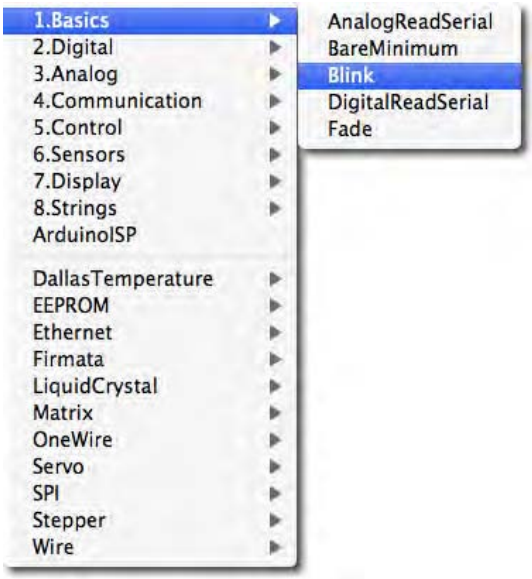


Figure 1-11. *The Examples menu*

Next, click the Upload button (sixth button from the left) and look at your Arduino. (If you have an Arduino Mini, NG, or other board, you may need to press the reset button on the board prior to pressing the Upload button.) The RX and TX lights should start to flash to show that data is being transmitted from your computer to the board. Once the sketch has successfully uploaded, the words "Done uploading" will appear in the IDE status bar and the RX and TX lights will stop flashing.

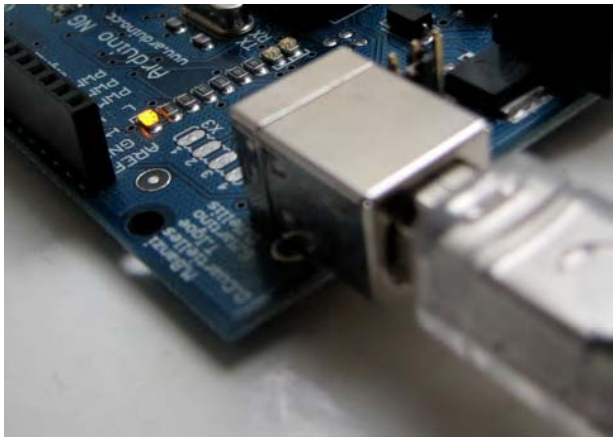


Figure 1-12. *LED 13 blinking*

After a few seconds, you should see the Pin 13 LED (the tiny LED next to the RX and TX LEDs) start to flash on and off at one second intervals. If it does, you have just successfully connected your Arduino, installed the drivers and software, and uploaded an example sketch. The Blink sketch is a very simple sketch that blinks LED 13 shown in Figure 1-12, the tiny green (or orange) LED soldered to the board (and also connected to Digital Pin 13 from the microcontroller).

Before you move onto Project 1, let's take a look at the Arduino IDE. I'll explain each part of the program.

The Arduino IDE

When you open up the Arduino IDE, it will look very similar to the image in Figure 1-13. If you are using Windows or Linux, there may be some slight differences but the IDE is pretty much the same no matter what OS you use.

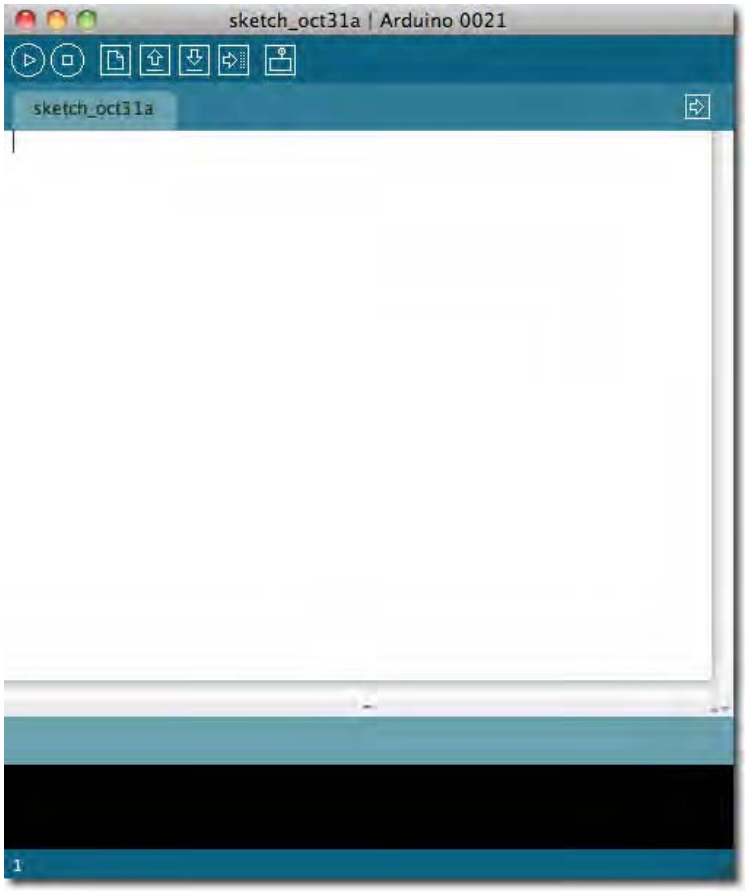


Figure 1-13. *What the IDE looks like when the application opens*

The IDE is split into three parts: the Toolbar across the top, the code or Sketch Window in the center, and the messages window in the bottom. The Toolbar consists of seven buttons. Underneath the Toolbar is a tab, or set of tabs, with the filename of the sketch within the tab. There is also one button on the far right hand side.

Along the top is the file menu with drop down menus labeled File, Edit, Sketch, Tools and Help. The buttons in the Toolbar (see Figure 1-14) provide convenient access to the most commonly used functions within this file menu.

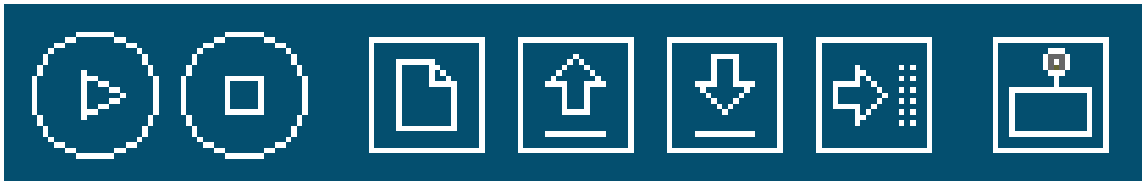


Figure 1-14. *The Toolbar*

The Toolbar buttons and their functions are listed in Table 1-1.

Table 1-1. *The Toolbar button functions*

Verify/Compile	Checks the code for errors
Stop	Stops the serial monitor, or un-highlights the other buttons
New	Creates a new blank sketch
Open	Shows a list of sketches in your Sketchbook to open
Save	Saves the current Sketch to your Sketchbook
Upload	Uploads the current Sketch to the Arduino
Serial Monitor	Displays serial data being sent from the Arduino

The Verify/Compile button is used to check that your code is correct and error free before you upload it to your Arduino board.

The Stop button stops the serial monitor from operating. It also un-highlights other selected buttons. While the serial monitor is operating, you can press the Stop button to obtain a snapshot of the serial data so far to examine it. This is particularly useful if you are sending data out to the Serial Monitor quicker than you can read it.

The New button creates a new and blank sketch ready for you to enter your code into. The IDE asks you to enter a name and a location for your sketch (try to use the default location if possible) and then gives you a blank Sketch ready to be coded. The tab at the top of the sketch shows the name you have given to your new sketch.

The Open button presents you with a list of sketches stored within your sketchbook as well as a list of example sketches that you can try out with various peripherals. The example sketches are invaluable for beginners to use as a foundation for their own sketches. Open the appropriate sketch for the device you are connecting and then modify the code for your own needs.

The Save button saves the code within the sketch window to your sketch file. Once complete, you will get a “Done Saving” message at the bottom of your code window.

The Upload to I/O Board button uploads the code within the current sketch window to your Arduino. Make sure that you have the correct board and port selected (in the Tools menu) before uploading. It is essential that you save your sketch before you upload it to your board in case a strange error causes your system to hang or the IDE to crash. It is also advisable to hit the Verify/Compile button before you upload to ensure there are no errors that need to be debugged first.

The serial monitor is a very useful tool, especially for debugging your code. The monitor displays serial data being sent out from your Arduino (USB or serial board). You can also send serial data back to the Arduino using the serial monitor. Clicking the Serial Monitor button results in a window like the one in Figure 1-15.

On the bottom right side, you can select the Baud Rate that the serial data is to be sent to/from the Arduino. The Baud Rate is the rate per second that state changes or bits (data) are sent to/from the board. The default setting is 9600 baud, which means that if you were to send a text novel over the serial communications line (in this case, your USB cable) then 1200 letters or symbols of the novel would be sent per second (9600 bits/8 bits per character = 1200 bytes or characters). Note that bits and bytes will be explained later.

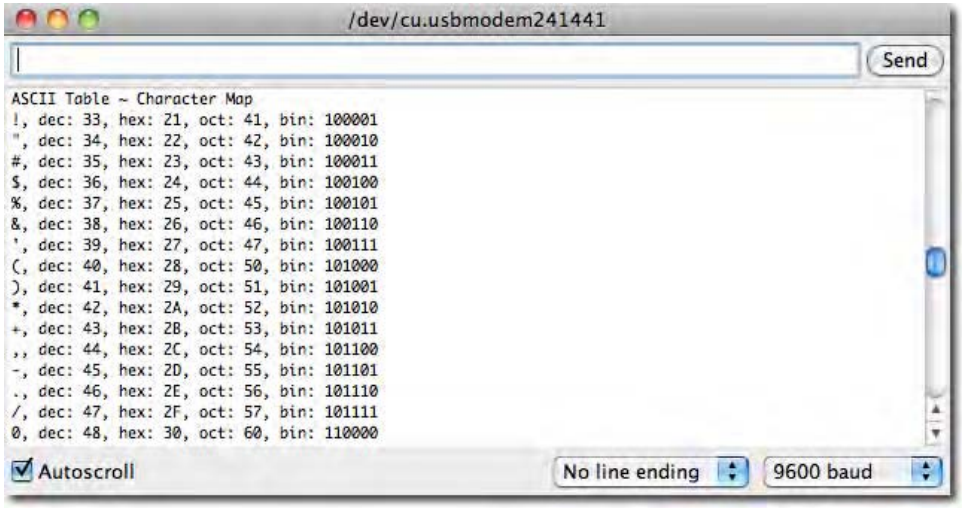


Figure 1-15. *The serial window in use*

At the top is a blank text box for you to enter text to send back to the Arduino and a Send button to make it happen. Note that the serial monitor can receive no serial data unless you have set up the code inside your sketch for it to do so. Similarly, the Arduino will not receive any data sent unless you have coded it to do so.

Finally, the black area is where your serial data will be displayed. In the image above, the Arduino is running the `ASCIITable` sketch (from the Communications example). This program outputs ASCII characters from the Arduino via serial (the USB cable) to the PC where the serial monitor then displays them.

To start the serial monitor, press the Serial Monitor button. To stop it, press the Stop button. On a Mac or in Linux, the Arduino board will reset itself (rerun the code from the beginning) when you click the Serial Monitor button.

Once you are proficient at communicating via serial to and from the Arduino, you can use other programs such as Processing, Flash, MaxMSP, etc. to communicate between the Arduino and your PC. You will make use of the serial monitor later when you read data from sensors and get the Arduino to send that data to the serial monitor in human readable form.

At the bottom of the IDE window is where you will see error messages (in red text) that the IDE will display when trying to connect to your board, upload code, or verify code. At the bottom left of the IDE you will see a number. This is the current location of the cursor within the program. If you have code in your window and you move down the lines of code (using the ↓ key on your keyboard), you will see the number increase as you move down the lines of code. This is useful for finding bugs highlighted by error messages.

Across the top of the IDE window (or across the top of your screen if you are using a Mac) you will see the various menus that you can click on to access more menu items (see Figure 1-16).

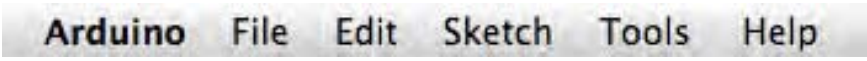


Figure 1-16. *The IDE menus*

The first menu is the Arduino menu (see Figure 1-17). The About Arduino option shows the current version number, a list of the people involved in making this amazing device, and some further information.

Figure 1-17. *The Arduino menu*

Underneath that is the Preferences option. This brings up the preferences window where you can change various IDE options, such as your default Sketchbook location, etc. The Quit option quits the program.

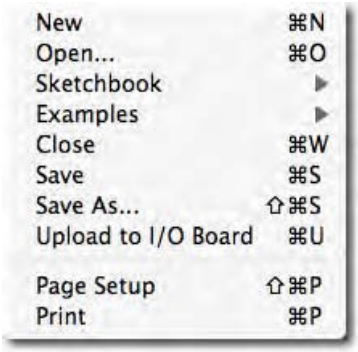


Figure 1-18. *The File menu*

The File menu (see Figure 1-18) is where can access options to create a new sketch, take a look at sketches stored in your Sketchbook (as well as the example sketches), save your sketch or use the Save As option if you want to give it a different name, upload your sketch to the I/O Board (Arduino), or print out your code.

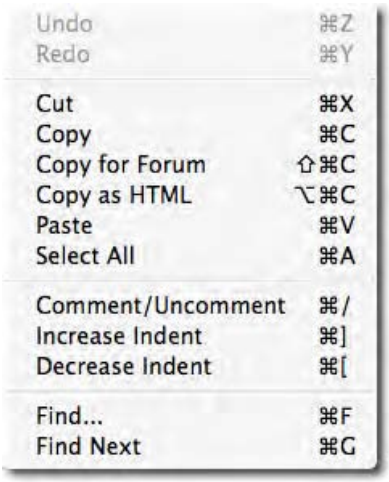


Figure 1-19. *The Edit menu*

The Edit menu (see Figure 1-19) offers options to let you to cut, copy, and paste sections of code. You can also Select All of your code or Find certain words or phrases within the code. The useful Undo and Redo options come in handy when you make a mistake.

Figure 1-20. *The Sketch menu*

The Sketch menu (see Figure 1-20) contains the Verify/Compile functions and other useful functions including the Import Library option, which brings up a list of the available libraries stored within your libraries folder.

A *library* is a collection of code that you can include in your sketch to enhance the functionality of your project. It is a way of preventing you from re-inventing the wheel; instead, you can reuse code already written by someone else for various pieces of common hardware. For example, the Stepper library is a set of functions to control a stepper motor. Somebody else has kindly already created all of the functions necessary to control a stepper motor, so by including the Stepper library into your sketch, you can use those functions to control the motor. By storing commonly used code in a library, you can re-use that code over and over in different projects. You can also hide the complicated parts of the code from the user. I will go into greater detail concerning the use of libraries later on.

The Show Sketch Folder option opens the folder where your sketch is stored. The Add File option lets you to add another source file to your sketch, which allows you to split larger sketches into smaller files and then add them to the main sketch.

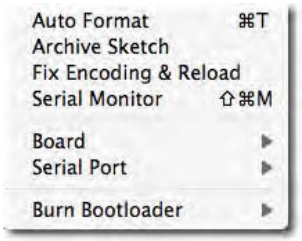


Figure 1-21. *Tools menu*

The Tools menu (see Figure 1-21) offers several options. You can select the Board and Serial Port, as you did when setting up the Arduino for the first time. The Auto Format function formats your code to make it look nicer. The Copy for Forum option copies the code within the sketch window, but in a format that, when pasted into the Arduino forum (or most other Forums for that matter), will show up the same as it is in the IDE, along with syntax coloring, etc. The Archive Sketch option lets you to compress your sketch into a ZIP file and will ask you where you want to store it. Finally, the Burn Bootloader option burns the Arduino Bootloader (the piece of code on the chip to make it compatible with the Arduino IDE) to the chip. This option can only be used if you have an AVR programmer and if you have replaced the chip in your Arduino or have bought blank chips to use in your own embedded project. Unless you plan on burning many chips, it's usually cheaper and easier to just buy an ATmega chip (see Figure 1-22) with the Arduino Bootloader already pre-programmed. Many online stores stock inexpensive pre-programmed chips.

Figure 1-22. *An Atmel ATmega chip, the heart of your Arduino. (image courtesy of Earthshine Electronics)*

The final menu, Help, is where you can find more information about the IDE or links to the reference pages of the Arduino website and other useful pages.

The Arduino IDE is pretty basic and you will learn how to use it quickly and easily as you work through the projects. As you become more proficient at using an Arduino and programming in C (the programming language used to code on the Arduino), you may find the Arduino IDE is too basic. If you want something with better functionality, you can try one of the professional IDE programs (some of which are free) such as Eclipse, ArduIDE, GNU/Emacs, AVR-GCC, AVR Studio, and even Apple's XCode.

Now that you have your Arduino software installed, the board connected and working, and you have a basic understanding of how to use the IDE, let's jump right in with Project 1 – LED Flasher.

Light 'Em Up

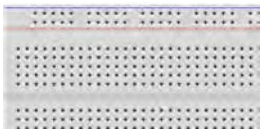
You are now going to work your way through the first four projects. These projects all use LED lights in various ways. You will learn about controlling outputs from the Arduino as well as simple inputs such as button presses. On the hardware side, you will learn about LEDs, buttons, and resistors, including pull up and pull down resistors, which are important in ensuring that input devices are read correctly. Along the way, you will pick up the concepts of programming in the Arduino language. Let's start with a "Hello World" project that makes your Arduino flash an external LED.

Project 1 – LED Flasher

For the first project, you are going to repeat the LED blink sketch that you used during your testing stage. This time, however, you are going to connect an LED to one of the digital pins rather than using LED13, which is soldered to the board. You will also learn exactly how the hardware and the software for this project works, learning a bit about electronics and coding in the Arduino language (which is a variant of C) at the same time.

Parts Required

Breadboard



5mm LED

100 ohm Resistor*

Jumper Wires

**This value may differ depending on what LED you use. The text will explain how to work it out.*

The best kind of breadboard for the majority of the projects in this book is an 840 tie-point breadboard. These are fairly standard sized breadboards, measuring approximately 16.5cm by 5.5cm and featuring 840 holes (or tie points) on the board. Usually, the boards have little dovetails on the side allowing you to connect several of them together to make larger breadboards; this is useful for more complex projects. For this project though, any sized breadboard will do.

The LED should be a 5mm one of any color. You will need to know the current and voltage (sometimes called forward current and forward voltage) of the LED so that you can calculate the resistor value needed—you will work out this value later in the project.

The jumper wires you use can either be commercially available jumper wires (usually with molded ends to make insertion into the breadboard easier) or you can make your own by cutting short strips of stiff single core wire and stripping away about 6mm from the end.

Connecting Everything

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now, take your breadboard, LED, resistor, and wires and connect everything as shown in Figure 2-1.

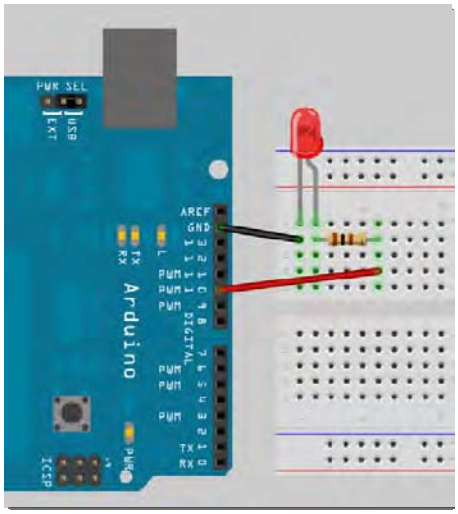


Figure 2-1. *The circuit for Project 1 – LED Flasher (see insert for color version)*

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

Make sure that your LED is connected correctly with the longer leg connected to Digital Pin 10. The long leg is the anode of the LED and must always go to the +5v supply (in this case, coming out of Digital Pin 10); the short leg is the cathode and must go to Gnd (ground).

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

Enter the Code

Open up your Arduino IDE and type in the code from Listing 2-1.

Listing 2-1. *Code for Project 1*

```
// Project 1 - LED Flasher
int ledPin = 10;
void setup() {
         pinMode(ledPin, OUTPUT);
}
void loop() {
        digitalWrite(ledPin, HIGH);
        delay(1000);
        digitalWrite(ledPin, LOW);
        delay(1000);
}
```

Press the Verify/Compile button at the top of the IDE to make sure there are no errors in your code. If this is successful, click the Upload button to upload the code to your Arduino. If you have done everything right, you should now see the red LED on the breadboard flashing on and off every second.

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

Project 1 – LED Flasher – Code Overview

The first line of code for this project is:

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

This is just a *comment* in your code. You can tell it's a comment because it starts with // and any text that begins this way will be ignored by the compiler. Comments are essential in your code; they help you understand how your code works. As your projects get more complex and your code expands into hundreds or maybe thousands of lines, comments will be vital in making it easy for you to see how each section functions. You may come up with an amazing piece of code, but you can't count on remembering how it works when you revisit it several days, weeks, or months later. Comments, however, will remind you of its functionality. Also, if your code is meant to be seen by other people, comments will help that person understand what is going on in your code. The whole ethos of the Arduino, and indeed the whole Open Source community, is to share code and schematics. I hope that when you start making your own cool stuff with the Arduino you will be willing to share it with the world, too.

There is another format for making comments; it is a block statement bookended by /* and */ , like so:

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

The IDE will automatically turn the color of any commented text to grey. The next line of the program is

```
int ledPin = 10;
```

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

Although you can call your variables anything, every variable name in C must start with a letter; the rest of the name can consist of letters, numbers, and underscore characters. Note that C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords like main, while, switch etc as variable names. Keywords are constants, variables, and function names that are defined as part of the Arduino language. To help you avoid naming a variable after a keyword, all keywords within the sketch will appear in red.

Imagine a variable as a small box where you can keep things. So in this sketch, you have set up an area in memory to store a number of type integer and have stored in that area the number 10.

Finally, a variable is called a variable because you can change it. Later, you will carry out mathematical calculations on variables to make your program do more advanced things.

Next is your `setup()` function:

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

An Arduino sketch must have a `setup()` and `loop()` function, otherwise it will not work. The `setup()` function runs once and once only at the start of the program and is where you will issue general instructions to prepare the program before the main loop runs, such as setting up pin modes, setting serial baud rates, etc. Basically, a function is a bunch of code assembled into one convenient block. For example, if you created your own function to carry out a series of complicated mathematics that had many lines of code, you could run that code as many times as you liked simply by calling the function name instead of writing out the code again each time. You will go into functions in more detail later when you start to create your own. In the case of this program, however, the `setup()` function only has one statement to carry out. The function starts with

```
void setup()
```

This tells the compiler that your function is called setup, that it returns no data (void), and that you pass no parameters to it (empty parenthesis). If your function returned an integer value and you also had integer values to pass to it (e.g. for the function to process), it would look something like this:

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

Here you have created a function (or a block of code) called myFunc. This function has been passed two integers called x and y. Once the function has finished, it will then return an integer value to the point after where your function was called in the program (hence `int` before the function name).

All of the code within the function is contained within the curly braces. A { symbol starts the block of code and a } symbol ends the block. Anything in between those two symbols is code that belongs to the function. (I will go into greater detail about functions later, so don't worry about them for now.)

In this program, you have two functions; the first function is called setup and its purpose is to setup anything necessary for your program to work before the main program loop runs:

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

Your setup function only has one statement and that is pinMode, which telling the Arduino that you want to set the mode of one of your pins to be Output mode, rather than Input. Within the parenthesis, you put the pin number and the mode (OUTPUT or INPUT). Your pin number is ledPin, which has been previously set to the value 10. Therefore, this statement is simply telling the Arduino that Digital Pin 10 is to be set to OUTPUT mode. As the `setup()` function runs only once, you now move onto the main function loop:

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

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

In this project, you want the LED to turn on, stay on for one second, turn off and remain off for one second, and then repeat. The commands to tell the Arduino to do this are contained within the `loop()` function because you wish them to repeat over and over. The first statement is

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

and this writes a HIGH or a LOW value to the pin within the statement (in this case ledPin, which is Digital Pin 10). When you set a pin to HIGH, you are sending out 5 volts to that pin. When you set it to LOW, the pin becomes 0 volts, or ground. This statement, therefore, sends out 5v to pin 10 and turns the LED on. After that is

```
delay(1000);
```

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

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

which will turn off the power going to Digital Pin 10 and therefore turn the LED off. Then there is another delay statement for another 1000 milliseconds and then the function ends. However, as this is your main `loop()` function, the function will start again at the beginning.

By following the program structure step by step again, you can see that it is very simple:

```
// Project 1 - LED Flasher
int ledPin = 10;
void setup() {
        pinMode(ledPin, OUTPUT);
}
void loop() {
        digitalWrite(ledPin, HIGH);
        delay(1000);
        digitalWrite(ledPin, LOW);
        delay(1000);
}
```

You start off by assigning a variable called ledPin, giving that variable a value of 10. Then you move on to the `setup()` function where you set the mode for Digital Pin 10 as an output. In the main program loop, you set Digital Pin 10 to high, sending out 5v. Then you wait for a second and then turn off the 5v to Digital Pin 10, before waiting another second. The loop then starts again at the beginning: the LED will turn on and off continuously for as long as the Arduino has power.

Now that you know this, you can modify the code to turn the LED on for a different period of time and turn it off for a different time period. For example, if you wanted the LED to stay on for 2 seconds, then go off for half a second, you could do the following:

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

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

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

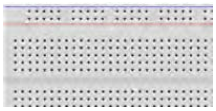
To make the LED flash on and off very fast, try this:

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

By varying the on and off times of the LED you create any effect you want (well, within the bounds of a single LED going on and off). Before you move onto something a little more exciting, let's take a look at the hardware and see how it works.

Project 1 – LED Flasher – Hardware Overview

The hardware used in Project 1:

Breadboard 

5mm LED 

100 ohm Resistor* 

Jumper Wires 

** or whatever value appropriate for your LED*

The breadboard is a reusable solderless device used to prototype an electronic circuit or for experimenting with circuit designs. The board consists of a series of holes in a grid; underneath the board these holes are connected by a strip of conductive metal. The way those strips are laid out is typically something like that in Figure 2-2.

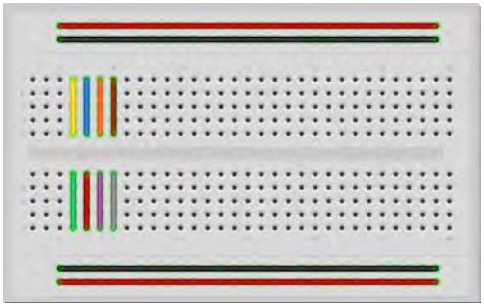


Figure 2-2. *How the metal strips in a breadboard are laid out*

The strips along the top and bottom run parallel to the board and are design to carry your power rail and your ground rail. The components in the middle of the board conveniently connect to either 5v (or whatever voltage you are using) and ground. Some breadboards have a red and a black line running parallel to these holes to show which is power (Red) and which is ground (Black). On larger breadboards,

the power rail sometimes has a split, indicated by a break in the red line. This makes it possible to send different voltages to different parts of your board. If you are using just one voltage, a short piece of jumper wire can be placed across this gap to make sure that the same voltage is applied along the whole length of the rail.

The strips in the centre run at 90 degrees to the power and ground rails in short lengths and there is a gap in the middle to allow you to put Integrated Circuits across the gap so that each pin of the chip goes to a different set of holes and therefore a different rail (see Figure 2-3).

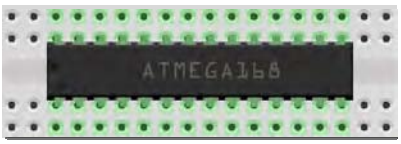


Figure 2-3. *An Integrated Circuit (or chip) plugged across the gap in a breadboard*

The next component is a resistor. A *resistor* is a device designed to cause resistance to an electric current in order to cause a drop in voltage across its terminals. You can think of a resistor as a water pipe that is a lot thinner than the pipe connected to it. As the water (the electric current) comes into the resistor, the pipe gets thinner and the water volume (current) coming out of the other end is therefore reduced. You use resistors to decrease voltage or current to other devices.

The value of resistance is known as an Ohm and its symbol is a Greek Omega symbol Ω. In this case, Digital Pin 10 is outputting 5v DC at (according to the Atmega datasheet) 40mA (milliamps), and your LEDs require (according to their datasheet) a voltage of 2v and a current of 35mA. Therefore, you need a resistor that will reduce the 5v to 2v and the current from 40mA to 35mA if you want to display the LED at its maximum brightness. If you want the LED to be dimmer, you could use a higher value of resistance.

■ **Note** NEVER use a value of resistor that is LOWER than needed. You will put too much current through the LED and damage it permanently. You could also damage other parts of your circuit.

The formula to work out what resistor you need is

$R = (V_S - V_L) / I$

where V_S is the supply voltage, V_L is the LED voltage, and I is the LED current. Your example LED has a voltage of 2v and a current of 35mA connected to a digital pin from an Arduino, which gives out 5 volts, so the resistor value needed would be

$R = (5 - 2) / 0.035$

which gives a value of 85.71.

Resistors come in standard values and the closest common value would be 100 Ω. Always choose the next standard value resistor that is HIGHER than the value needed. If you choose a lower value, too much current will flow through the resistor and will damage it.

So how do you find a 100Ω resistor? A resistor is too small to contain easily readable labeling so resistors instead use a color code. Around the resistor you will typically find 4 colored bands; by using the color code in Table 2-1 you can find out the value of a resistor. Likewise, you can find the color code for a particular resistance.

Table 2-1. *Resistor color codes*

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

According to the table, for a 100Ω resistor you need 1 in the first band, which is brown, followed by a 0 in the next band, which is black. Then you need to multiply this by 10^1 (in other words add 1 zero), which results in brown for the third band. The final band indicates the tolerance of the resistor. If your resistor has a gold band, it has a tolerance of ±5 percent; this means the actual value of the resistor varies between 95Ω and 105Ω. Therefore, if you have an LED that requires 2 volts and 35mA, you need a resistor with a Brown, Black, Brown band combination.

If you need a 10K (or 10 kilo-ohm) resistor, you need a Brown, Black, Orange combination (1, 0, +3 zeros). If you need a 570K resistor, the colors would be Green, Violet, and Yellow.

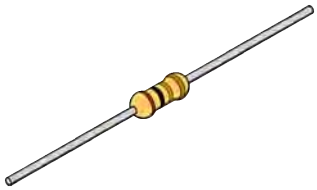


Figure 2-4*. A 10KΩ resistor with a 5 percent tolerance*

In the same way, if you found a resistor and wanted to know its value, you would do the same in reverse. So if you found the resistor in Figure 2-4 and wanted to find its value so you could store it away in your nicely labeled resistor storage box, you could look at the table to see it has a value of 220Ω.

Now that you know how the color coding works, choose the correct resistance value for the LED you have purchased to complete this project.

The final component (other than the jumper wires, but I'm sure you can figure out what they do for yourself) is the LED, which stands for Light Emitting Diode. A diode is a device that permits current to flow in only one direction; it's just like a valve in a water system, but in this case it is letting electrical current to go in one direction. If the current tries to reverse and go back in the opposite direction, the diode stops it from doing so. Diodes can be useful to prevent someone from accidently connecting the power and ground to the wrong terminals in a circuit and damaging the components.

An LED is the same thing, but it also emits light. LEDs come in all kinds of different colors and levels of brightness, including the ultraviolet and infrared part of the spectrum (like in the LEDs in your TV remote control).

If you look carefully at an LED you will notice two things: the legs are of different lengths, and one side of the LED is flattened rather than cylindrical (see Figure 2-5). These are clues as to which leg is the Anode (positive) and which is the Cathode (negative): the longer leg (Anode) gets connected to the positive supply (3.3v) and the leg with the flattened side (Cathode) goes to ground.

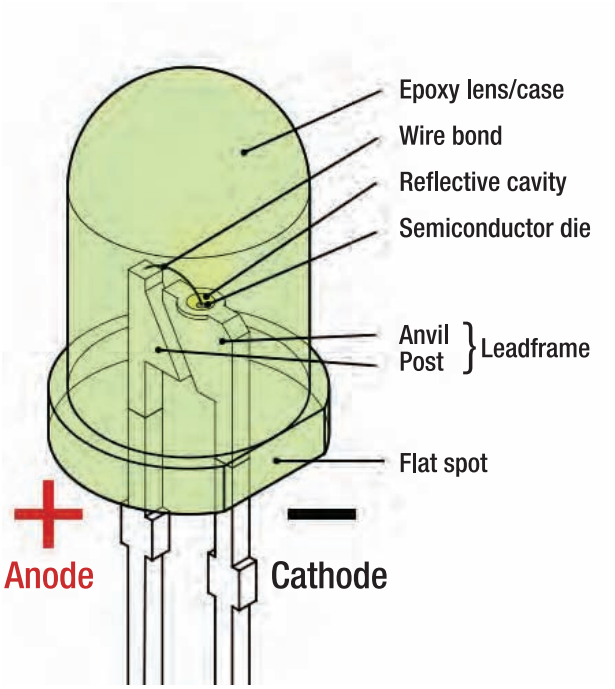


Figure 2-5. *The parts of an LED (image courtesy of Inductiveload from Wikimedia Commons)*

If you connect the LED the wrong way, it will not damage it (unless you put very high currents through it). However, it's essential that you always put a resistor in series with the LED to ensure that the correct current gets to the LED. You can permanently damage the LED if you fail to do this.

Note that you can also obtain bi-color and tri-color LEDs. These have several legs coming out of them. An RGB LED has a red, green, and blue (hence RGB) LED in one package. This LED has four legs; one will be a common anode or cathode (common to all three LEDs) and other legs will go to the anode or cathode of an individual LED. By adjusting the brightness values of the R, G and B channels of the RGB LED, you can get any color you want (the same effect can be obtained if you used three separate red, green and blue LEDs).

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

Project 2 – S.O.S. Morse Code Signaler

For this project, you are going to reuse the circuit set up from Project 1 (so no need for a Hardware Overview), but you'll use different code to make the LED signal the letters S.O.S., which is the International Morse Code distress signal. Morse Code is a type of character encoding that transmits letters and numbers using patterns of on and off. It is therefore nicely suited to your digital system as you can turn an LED on and off in the necessary pattern to spell out a word or a series of characters. In

this case, the S.O.S. pattern is three dits (short flash), followed by three dahs (long flash), followed by three dits again.

To flash the LED on and off in this pattern, signaling SOS, use the code in Listing 2-2.

Listing 2-2. *Code for Project 2*

```
// LED connected to pin 10
int ledPin = 10;

// run once, when the sketch starts
void setup()
{
        // sets the pin as output
        pinMode(ledPin, OUTPUT);
}

// run over and over again
void loop()
{
        // 3 dits
        for (int x=0; x<3; x++) {
                digitalWrite(ledPin, HIGH);   // sets the LED on
                delay(150);                  // waits for 150ms
                digitalWrite(ledPin, LOW);    // sets the LED off
                delay(100);               // waits for 100ms
        }

        // 100ms delay to cause slight gap betyouen letters
        delay(100);
        // 3 dahs
        for (int x=0; x<3; x++) {
                digitalWrite(ledPin, HIGH);   // sets the LED on
                delay(400);                  // waits for 400ms
                digitalWrite(ledPin, LOW);    // sets the LED off
                delay(100);               // waits for 100ms
        }

        // 100ms delay to cause slight gap betyouen letters
        delay(100);

  // 3 dits again
        for (int x=0; x<3; x++) {
                digitalWrite(ledPin, HIGH);   // sets the LED on
                delay(150);                  // waits for 150ms
                digitalWrite(ledPin, LOW);    // sets the LED off
                delay(100);               // waits for 100ms
        }

        // wait 5 seconds before repeating the SOS signal
        delay(5000);
}
```

Create a new sketch and then type in the code from Listing 2-2. Verify that your code is error free and then upload it to your Arduino. If all goes well, you will see the LED flash the Morse Code SOS signal, wait 5 seconds, then repeat.

If you were to rig up a battery operated Arduino to a very bright light and place the whole assembly into a waterproof and handheld box, it could be used to control an SOS emergency strobe light for used on boats, while mountain climbing, etc.

Let's figure out how this code works.

Project 2 – S.O.S. Morse Code Signaler – Code Overview

The first part of the code is identical to the last project where you initialize a variable and then set Digital Pin 10 to be an output. In the main code loop, you can see the same kind of statements to turn the LEDs on and off for a set period of time. This time, however, the statements are within three separate code blocks.

The first block is what outputs the three dits:

```
for (int x=0; x<3; x++) {
        digitalWrite(ledPin, HIGH);
        delay(150);
        digitalWrite(ledPin, LOW);
        delay(100);
  }
```

You can see that the LED is turned on for 150ms and then off for 100ms; you can also see that those statements are within a set of curly braces and are therefore in a separate code block. But, when you run the sketch you can see the light flashes three times, not just once.

This is done using the `for` loop:

```
 for (int x=0; x<3; x++) {
```

This statement is what makes the code within the code block execute three times. There are three parameters you need to give to the `for` loop. These are initialization, condition, and increment. The **initialization** happens first and exactly once. Each time through the loop, the **condition** is tested; if it's true, the statement block and the **increment** is executed, then the **condition** is tested again. When the **condition** becomes false, the loop ends.

So, first you need to initialize a variable as the start number of the loop. In this case, you set up variable X and set it to zero:

```
int x=0;
```

You then set a condition to decide how many times the code in the loop will execute:

```
x<3;
```

In this case, the code will loop if x is smaller than (<) 3. The code within a `for` loop will always execute once no matter what the condition is set to.

The < symbol is what is known as a *comparison operator*. They are used to make decisions within your code and to compare two values. The symbols used are:

== (equal to)

!= (not equal to)

< (less than)

> (greater than)

<= (less than or equal to)

>= (greater than or equal to)

In your code, you are comparing x with the value of 3 to see if it is smaller than 3. If x is smaller than 3, the code in the block will repeat again.

The final statement

```
x++
```

is a statement to increase the value of x by 1. You could also have typed in x = x + 1, which would assign to x the value of x + 1. Note there is no need to put a semi-colon after this final statement in the `for` loop.

You can do simple mathematics by using the symbols +, -, * and / (addition, subtraction, multiplication and division). For example:

1 + 1 = 2

3 - 2 = 1

2 * 4 = 8

8 / 2 = 4

So, your `for` loop initializes the value of x to 0, then runs the code within the block (curly braces). It then increases the increment (in this case, adds 1 to x). Finally, it checks that the condition is met, which is that x is smaller than 3 and if so repeats.

Now that you know how the `for` loop works, you can see that there are three `for` loops in your code: one that loops three times and displays the dits, one that repeats three times and displays the dahs, and then there is a repeat of the dits again.

It must be noted that the variable x has a local *scope*, which means it can only be seen by the code within its own code block, unless you initialize it before the `setup()` function, in which case it has *global scope* and can be seen by the entire program. If you try to access x outside the `for` loop, you will get an error.

In between each `for` loop is a small delay to make a tiny visible pause between letters of SOS. Finally, the code waits for 5 seconds before the main program loop starts again from the beginning.

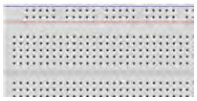
Now let's move onto using multiple LEDs.

Project 3 – Traffic Lights

You are now going to create a set of traffic lights that will change from green to red, via amber, and back again, after a set length of time using the four-state UK system. This project could be used to make a set of working traffic lights for a model railway or for a child's toy town. If you're not from the UK, you can

modify the code and colors to make them work like the traffic lights in your own country. First, though, make the project as it is and change it once you know how it works.

Parts Required

Breadboard 

Red Diffused LED 

Yellow Diffused LED 

Green Diffused LED 

3 x 150 ohm Resistors* 

Jumper Wires 

**or whatever value you require for your type of LED*

Connect It Up

Connect your circuit as shown in Figure 2-6. This time you connect three LEDs with the anode of each one going to Digital Pins 8, 9 and 10 via a 150Ω resistor (or whatever value you require) each.

Take a jumper wire from ground of the Arduino to the ground rail at the top of the breadboard; a ground wire goes from the Cathode leg of each LED to the common ground rail via a resistor—this time connected to the cathode. (For this simple circuit, it doesn't matter if the resistor is connected to the anode or cathode).

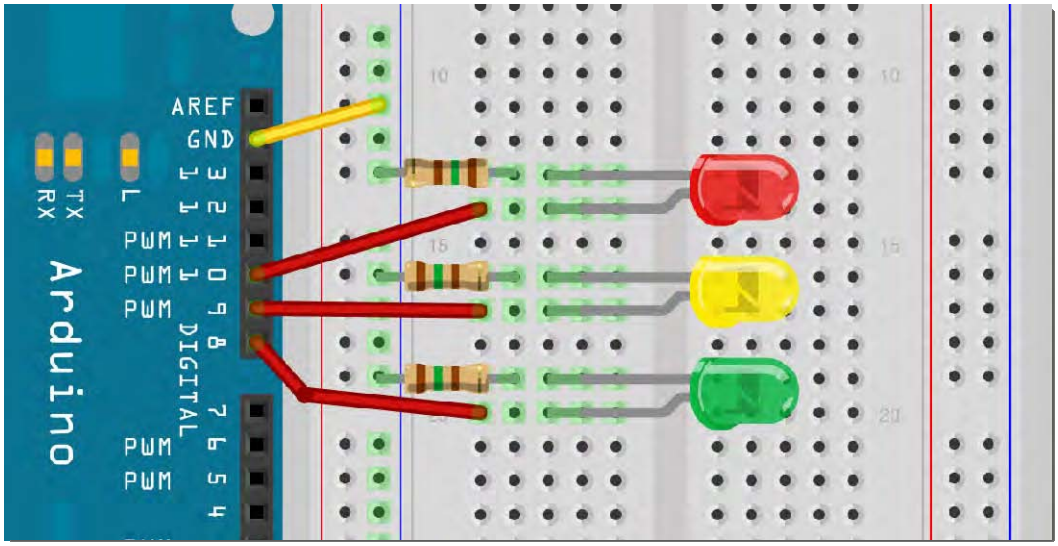


***Figure 2-6**. The circuit for Project 3 – Traffic Lights (see insert for color version)*

Enter the Code

Enter the code from Listing 2-3, check it, and upload to your Arduino. The LEDs will now move through four states that simulate the UK traffic light system, as seen in Figure 2-7. If you have followed Projects 1 and 2, both the code and the hardware for Project 3 will be self-explanatory. I shall leave you to examine the code and figure out how it works.

***Listing 2-3.** Code for Project 3*

```
// Project 3 - Traffic Lights

int ledDelay = 10000; // delay in between changes
int redPin = 10;
int yellowPin = 9;
int greenPin = 8;

void setup() {
        pinMode(redPin, OUTPUT);
        pinMode(yellowPin, OUTPUT);
        pinMode(greenPin, OUTPUT);
}
```

```
void loop() {

        digitalWrite(redPin, HIGH); // turn the red light on
        delay(ledDelay); // wait 5 seconds

        digitalWrite(yellowPin, HIGH); // turn on yellow
        delay(2000); // wait 2 seconds

        digitalWrite(greenPin, HIGH); // turn green on
        digitalWrite(redPin, LOW); // turn red off
        digitalWrite(yellowPin, LOW); // turn yellow off
        delay(ledDelay); // wait ledDelay milliseconds

        digitalWrite(yellowPin, HIGH); // turn yellow on
        digitalWrite(greenPin, LOW); // turn green off
        delay(2000); // wait 2 seconds

        digitalWrite(yellowPin, LOW); // turn yellow off
        // now our loop repeats

}
```

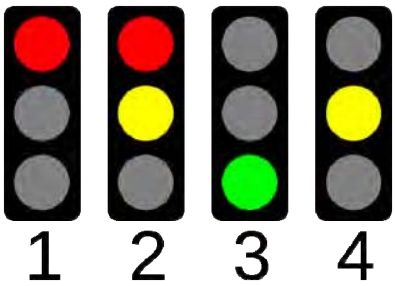


***Figure 2-7**. The four states of the UK traffic light system (image by Alex43223 from WikiMedia) (see insert for color version)*

Project 4 – Interactive Traffic Lights

This time you are going to extend the previous project to include a set of pedestrian lights and a pedestrian push button to request to cross the road. The Arduino will react when the button is pressed by changing the state of the lights to make the cars stop and allow the pedestrian to cross safely.

This is the first time you are going to interact with the Arduino and cause it to do something when you change the state of a button that the Arduino is watching. In this project, you will also learn how to create your own functions in code.

From now on, I will no longer list the breadboard and jumper wires in the parts required list. Note that you will always need these basic components.

Parts Required

2 Red Diffused LEDs 

Yellow Diffused LED 

2 Green Diffused LEDs 

150 ohm Resistor 

4 Resistors 

Pushbutton 

Choose the appropriate value resistor for the LEDs you are using in your project. The 150Ω resistor is for the pushbutton; it's known as a *pull down resistor* (which I will define later). The pushbutton is sometimes referred to by suppliers as a *tactile switch* and is ideal for breadboard use.

Connect It Up

Connect your circuit as shown in Figure 2-8. Double-check your wiring before providing any power to your Arduino. Remember to have your Arduino disconnected to the power while wiring up the circuit.

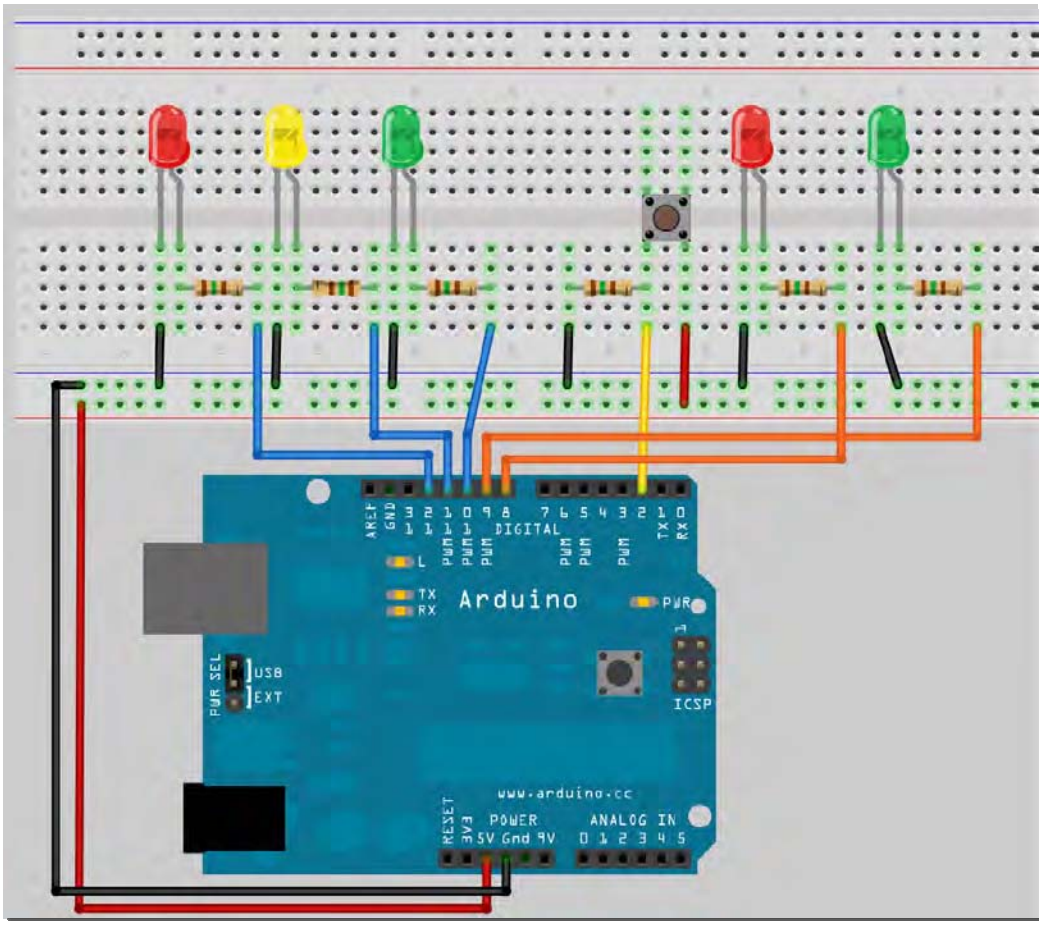


***Figure 2-8**. The circuit for Project 4 - Traffic light system with pedestrian crossing and request button (see insert for color version)*

Enter the Code

Enter the code in Listing 2-4, verify, and upload it. When you run the program, it begins with the car traffic light on green to allow cars to pass and the pedestrian light on red.

When you press the button, the program checks that at least 5 seconds have gone by since the last time the lights changed (to allow traffic to get moving), and if so, passes code execution to the function you have created called `changeLights()`. In this function, the car lights go from green to amber to red, and then the pedestrian lights go green. After the period of time set in the variable crossTime (time enough to allow the pedestrians to cross), the green pedestrian light flash on and off as a warning to the pedestrians to hurry because the lights are about to change to red. Then the pedestrian light changes to red, the vehicle lights go from red to amber to green, and the traffic flow resumes.

Listing 2-4. *Code for Project 4*

```
// Project 4 - Interactive Traffic Lights

int carRed = 12; // assign the car lights
int carYellow = 11;
int carGreen = 10;
int pedRed = 9; // assign the pedestrian lights
int pedGreen = 8;
int button = 2; // button pin
int crossTime = 5000; // time alloyoud to cross
unsigned long changeTime; // time since button pressed

void setup() {
        pinMode(carRed, OUTPUT);
        pinMode(carYellow, OUTPUT);
        pinMode(carGreen, OUTPUT);
        pinMode(pedRed, OUTPUT);
        pinMode(pedGreen, OUTPUT);
        pinMode(button, INPUT); // button on pin 2
        // turn on the green light
        digitalWrite(carGreen, HIGH);
        digitalWrite(pedRed, HIGH);
}

void loop() {
        int state = digitalRead(button);
        /* check if button is pressed and it is over 5 seconds since last button press */
        if (state == HIGH && (millis() - changeTime) > 5000) {
                // Call the function to change the lights
                changeLights();
        }
}


void changeLights() {
        digitalWrite(carGreen, LOW); // green off
        digitalWrite(carYellow, HIGH); // yellow on
        delay(2000); // wait 2 seconds

        digitalWrite(carYellow, LOW); // yellow off
        digitalWrite(carRed, HIGH); // red on
        delay(1000); // wait 1 second till its safe

        digitalWrite(pedRed, LOW); // ped red off
        digitalWrite(pedGreen, HIGH); // ped green on
        delay(crossTime); // wait for preset time period
```

```
        // flash the ped green
        for (int x=0; x<10; x++) {
                digitalWrite(pedGreen, HIGH);
                delay(250);
                digitalWrite(pedGreen, LOW);
                delay(250);
        }
        // turn ped red on
        digitalWrite(pedRed, HIGH);
        delay(500);

        digitalWrite(carYellow, HIGH); // yellow on
        digitalWrite(carRed, LOW); // red off
        delay(1000);
        digitalWrite(carGreen, HIGH);
        digitalWrite(carYellow, LOW); // yellow off

        // record the time since last change of lights
        changeTime = millis();
        // then return to the main program loop
}
```

Project 4 – Code Overview

You will understand and recognize most of the code in this project from previous projects. I'll just point out the new keywords and concepts:

```
unsigned long changeTime;
```

Here is a new data type for a variable. Previously, you created integer data types, which can store a number between -32,768 and 32,767. This time you created a data type of *long*, which can store a number from -2,147,483,648 to 2,147,483,647. However, you have specified an *unsigned long*, which means the variable cannot store negative numbers, so the range is from 0 to 4,294,967,295. If you use an integer to store the length of time since the last change of lights, you would only get a maximum time of 32 seconds before the integer variable reached a number higher than it could store.

As a pedestrian crossing is unlikely to be used every 32 seconds, you don't want your program crashing due to your variable "overflowing" when it tries to store a number too high for the variable data type. So you use an unsigned long data type to get a huge length of time in between button presses:

4294967295 * 1ms = 4294967 seconds

4294967 seconds = 71582 minutes

71582 minutes - 1193 hours

1193 hours - 49 days

It's pretty inevitable that a pedestrian crossing button will be pressed at least once in 49 days, so you shouldn't have a problem with this data type.

So why isn't there just one data type that can store huge numbers all the time? Well, because variables take up space in memory; the larger the number, the more memory is used up for storing variables. On your home PC or laptop, you won't have to worry about it much at all, but on a small microcontroller like the Arduino's Atmega32, it's essential that you use only the smallest variable data type necessary for your purpose.

Table 2-2 lists the various data types you can use in your sketches.

Table 2-2. *Data types*

Data type	RAM	Number Range
void keyword	N/A	N/A
boolean	1 byte	0 to 1 (True or False)
byte	1 byte	0 to 255
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 65,535
word	2 byte	0 to 65,535
long	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long	4 byte	0 to 4,294,967,295
float	4 byte	-3.4028235E+38 to 3.4028235E+38
double	4 byte	-3.4028235E+38 to 3.4028235E+38
string	1 byte + x	Arrays of chars
array	1 byte + x	Collection of variables

Each data type uses up a certain amount of memory: some variables use only 1 byte of memory and others use 4 or more (don't worry about what a byte is for now; I will discuss this later). Note that you can't copy data from one data type to another. In other words, if x was an int and y was a string, x = y would not work because the two data types are different.

The Atmega168 has 1Kb (1000 bytes) and the Atmega328 has 2Kb (2000 bytes) of SRAM; this is not a lot of memory. In large programs with lots of variables, you could easily run out of memory if you do not optimize your usage of the correct data types. As you have used int (which uses up 2 bytes and can store

a number up to 32,767) to store the number of your pin, which will only go as high as 13 on your Arduino (and up to 54 on the Arduino Mega), you have used up more memory than was necessary. You could have saved memory by using the *byte* data type, which can store a number between 0 and 255—more than enough to store the number of an I/O pin.

Next you have

```
pinMode(button, INPUT);
```

which tells the Arduino that you want to use Digital Pin 2 (button = 2) as an INPUT. You are going to use Digital Pin 2 to listen for button presses so its mode needs to be set to input.

In the main program loop, you check the state of pin 2 with this statement:

```
int state = digitalRead(button);
```

This initializes an integer (yes, it's wasteful and you should use a boolean) called state and then sets the value of state to be the value of Digital Pin 2. The `digitalRead` statement reads the state of the pin within the parenthesis and returns it to the integer you have assigned it to. You can then check the value in state to see if the button has been pressed or not:

```
if (state == HIGH && (millis() - changeTime) > 5000) {
   // Call the function to change the lights
   changeLights();
 }
```

The `if` statement is an example of a control structure and its purpose is to check if a certain condition has been met or not. If so, it executes the code within its code block. For example, if you wanted to turn an LED on if a variable called x rose above the value of 500, you could write the following:

```
if (x>500) {digitalWrite(ledPin, HIGH);
```

When you read a pin using the `digitalRead` command, the state of the pin will either be `HIGH` or `LOW`. So the `if` command in your sketch looks like this:

```
if (state == HIGH && (millis() - changeTime) > 5000)
```

What you are doing here is checking that two conditions have been met. The first is that the variable called state is high. If the button has been pressed, state will be high because you have already set it to be the value read in from Digital Pin 2. You are also checking that the value of `millis()`-changeTime is greater than 5000 (using the logical AND command &&). The `millis()` function is one built into the Arduino language, and it returns the number of milliseconds since the Arduino started to run the current program. Your changeTime variable will initially hold no value, but after the changeLights) function runs, you set it at the end of that function to the current `millis()` value.

By subtracting the value in the changeTime variable from the current `millis()` value, you can check if 5 seconds have passed since changeTime was last set. The calculation of `millis()`- changeTime is put inside its own set of parenthesis to ensure that you compare the value of state and the result of this calculation, and not the value of `millis()` on its own.

The symbol && in between

```
state == HIGH
```

and the calculation is an example of a Boolean Operator. In this case, it means AND. To see what this means, let's take a look at all of the Boolean Operators:

&& - Logical AND

|| - Logical OR

! - NOT

These are logic statements and can be used to test various conditions in `if` statements.

&& means true if both operands are true, so this `if` statement will run its code only if x is 5 and y is 10:

```
if (x==5 && y==10)  {....
```

|| means true if either operand is true; for example, this `if` statement will run if x is 5 or if y is 10:

```
if (x==5 || y==10) {.....
```

The ! or NOT statement means true if the operand is false, so this `if` statement will run if x is false, i.e. equals zero:

```
if (!x) {.......
```

You can also *nest* conditions with parenthesis, for example:

```
if (x==5 && (y==10 || z==25)) {.......
```

In this case, the conditions within the parenthesis are processed separately and treated as a single condition and then compared with the second condition. So, if you draw a simple truth table (see Table 2-3) for this statement, you can see how it works.

Table 2-3. *Truth table for the condition (x==5 && (y==10 || z==25))*

x	y	z	True/False?
4	9	25	FALSE
5	10	24	TRUE
7	10	25	FALSE
5	10	25	TRUE

The command within the `if` statement is

```
changeLights();
```

and this is an example of a function call. A *function* is simply a separate code block that has been given a name. However, functions can be passed parameters and/or return data, too. In this case, you have not passed any data to the function nor have you had the function return any date. I will go into more detail later on about passing parameters and returning data from functions.

When `changeLights()` is called, the code execution jumps from the current line to the function, executes the code within that function, and then returns to the point in the code after where the function was called.

In this case, if the conditions in the if statement are met, then the program executes the code within the function and returns to the next line after `changeLights()` in the if statement.

The code within the function simply changes the vehicles lights to red, via amber, then turns on the green pedestrian light. After a period of time set by the variable crossTime, the light flashes a few time to warn the pedestrian that his time is about to run out, then the pedestrian light goes red and the vehicle light goes from red to green, via amber, thus returning to its normal state.

The main program loop simply checks continuously if the pedestrian button has been pressed or not, and, if it has and (&&) the time since the lights last changed is greater than 5 seconds, it calls the `changeLights()` function again.

In this program, there was no benefit from putting the code into its own function, apart from making the code look cleaner and to explain the concept of functions. It is only when a function is passed parameters and/or returns data that their true benefits come to light; you will take a look at that later when you use functions again.

Project 4 – Interactive Traffic Lights - Hardware Overview

The new piece of hardware introduced in Project 4 is the button, or tactile switch. As you can see by looking at the circuit, the button is not directly connected between the power line and the input pin; there is a resistor going between the button and the ground rail. This is what is known as a pull-down resistor and it is essential to ensure the button works properly. I will take a little diversion to explain pull-up and pull-down resistors.

Logic States

A *logic circuit* is one designed to give an output of either on or off, which are represented by the binary numbers 1 and 0. The off (or zero) state is a voltage near to zero volts at the output; a state of on (or 1) is represented by a higher level, closer to the supply voltage. The simplest representation of a logic circuit is a switch (see Figure 2-9).

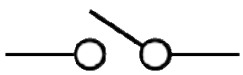


Figure 2-9. *The electronic symbol for a switch*

When the switch is open, no current can flow through it and no voltage can be measured at the output. When you close the switch, the current can flow through it, thus a voltage can be measured at the output. The open state can be thought of as a 0 and the closed state as a 1 in a logic circuit.

In a logic circuit, if the expected voltage to represent the on (or 1) state is 5v, it's important that when the circuit outputs a 1 that the voltage is as close to 5v as possible. Similarly, when the output is a zero (or off), it is important that the voltage is as close to zero volts as possible. If you do not ensure that the states are close to the required voltages, that part of the circuit may be considered to be *floating* (it is neither in a high or low state). The floating state is also known as electrical noise, and noise in a digital circuit may be interpreted as random 1's and 0's.

This is where pull up or pull down resistors can be used to ensure the state is high or low. If you let that node in the circuit float, it may be interpreted as either a zero or a one, which is not desirable. It's better to force it towards a desired state.

Pull-Down Resistors

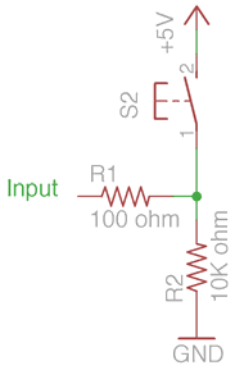


Figure 2-10. *A pull-down resistor circuit*

Figure 2-10 shows a schematic where a pull-down resistor being used. If the button is pressed, the electricity takes the path of least resistance and moves between the 5v and the input pin (there is a 100 ohm resistor on the input pin and a 10K ohm resistor on ground). However, when the button is not pressed, the input is connected to the 100K ohm resistor and is pulled towards ground. Without this pull to ground, the pin would not be connected to anything when the button was not depressed, thus it would float between zero and 5v. In this circuit, the input will always be pulled to ground, or zero volts, when the button is not pressed and it will be pulled towards 5v when the button is pressed. In other words, you have ensured that the pin is not floating between two values. Now look at Figure 2-11.

Pull-Up Resistors

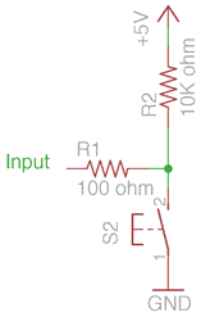


Figure 2-11. *A pull-up resistor circuit*

In this circuit, you have swapped the pull-down resistor and the switch. The resistor now becomes a pull-up resistor. As you can see, when the button is not pressed, the input pin is pulled towards the 5v, so it will always be high. When the button is pressed, the path of least resistance is towards the ground and so the pin is pulled to ground or the low state. Without the resistor between 5v and ground, it would be a short circuit, which would damage your circuit or power supply. Thanks to the resistor, it is no longer a short circuit as the resistor limits the amount of current. The pull-up resistor is used more commonly in digital circuits.

With the use of simple pull-up or pull-down resistors you can ensure that the state of an input pin is always either high or low, depending on your application.

In Project 4, you use a pull-down resistor to ensure a button press will register correctly by the Arduino. Let's take a look at the pull-down resistor in that circuit again (see Figure 2-12).

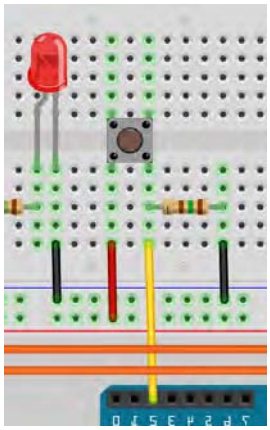


Figure 2-12. *A pull-down resistor from Project 4 (see insert for color version)*

This circuit contains a push button. One pin of the button is connected directly to 5v and the other is connected directly to Digital Pin 2. It is also connected directly to ground via a pull-down resistor. This means that when the button is not pushed, the pin is pulled to ground and therefore reads a zero or low state. When the button is pressed, 5 volts flows into the pin and it is read as a 1 or a high state. By detecting if the input is high or low, you can detect if the button is pressed or not. If the resistor was not present, the input pin wire would not be connected to anything and would be floating. The Arduino could read this as either a HIGH or a LOW state, which might result in it registering false button presses.

Pull-up resistors are often used in digital circuits to ensure an input is kept high. For example, the 74HC595 Shift Register IC (Integrated Circuit) that you will be using later on in the book has a Master Reset pin. This pin resets the chip when it is pulled low. As a result, it's essential that this pin is kept high at all times, unless you specifically want to do a reset; you can hold this pin high by using a pull-up resistor at all times. When you want to reset it, you pull it low using a digital output set to LOW; at all other times, it will remain high. Many other IC's have pins that must be kept high for most of the time and only pulled low for various functions to be activated.

The Arduino's Internal Pull-Up Resistors

Conveniently, the Arduino contains pull-up resistors that are connected to the pins (the analog pins have pull-up resistors also). These have a value of 20K ohms and need to be activated within software to use them. To activate an internal pull-up resistor on a pin, you first need to change the `pinMode` of the pin to an INPUT and then write a HIGH to that pin using a `digitalWrite` command:

```
pinMode(pin, INPUT);
digitalWrite(pin, HIGH);
```

If you change the pinMode from INPUT to OUTPUT after activating the internal pull-up resistors, the pin will remain in a HIGH state. This also works in reverse: an output pin that was in a HIGH state and is subsequently switched to an INPUT mode will have its internal pull-up resistors enabled.

Summary

Your first four projects covered a lot of ground. You now know the basics of reading inputs and turning LEDs on and off. You are beginning to build your electronic knowledge by understanding how LEDs and resistors work, how resistors can be used to limit current, and how they can be used to pull an input high or low according to your needs. You should also now be able to pick up a resistor and work out its value in ohms just by looking at its colored bands. Your understanding of the Arduino programming language is well underway and you have been introduced to a number of commands and concepts.

The skills learned in Chapter 2 are the foundation for even the most complex Arduino project. In Chapter 3, you will continue to use LEDs to create various effects, and in doing so will learn a huge number of commands and concepts. This knowledge will set you up for the more advanced subjects covered later in the book.

Subjects and Concepts Covered in Chapter 2:

- The importance of comments in code
- Variables and their types
- The purpose of the `setup()` and `loop()` functions
- The concept of functions and how to create them
- Setting the pinMode of a digital pin
- Writing a HIGH or LOW value to a pin
- How to create a delay for a specified number of milliseconds
- Breadboards and how to use them
- What a resistor is, its value of measurement, and how to use it to limit current
- How to work out the required resistor value for an LED
- How to calculate a resistor's value from its colored bands
- What an LED is and how it works
- How to make code repeat using a `for` loop
- The comparison operators
- Simple mathematics in code
- The difference between local and global scope
- Pull up and pull down resistors and how to use them
- How to read a button press
- Making decisions using the `if` statement
- Changing a pins mode between INPUT and OUTPUT
- The `millis()` function and how to use it
- Boolean operators and how to use them to make logical decisions

LED Effects

In Chapter 2 you learned the basics of input and output, some rudimentary electronics, and a whole bunch of coding concepts. In this chapter, you're going to continue with LEDs, making them produce some very fancy effects. This chapter doesn't focus much on electronics; instead, you will be introduced to many important coding concepts such as arrays, mathematic functions, and serial communications that will provide the necessary programming skills to tackle the more advanced projects later in this book.

Project 5 – LED Chase Effect

You're going to use a string of LEDs (10 in total) to make an LED chase effect, similar to that used on the car KITT on Knight Rider or on the face of the Cylons in Battlestar Galactica. This project will introduce the concept of arrays.

Parts Required

10 5mm RED LEDs

10 Current Limiting Resistors

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now, use your breadboard, LEDs, resistors, and wires to connect everything as shown in Figure 3-1. Check your circuit thoroughly before connecting the power back to the Arduino.

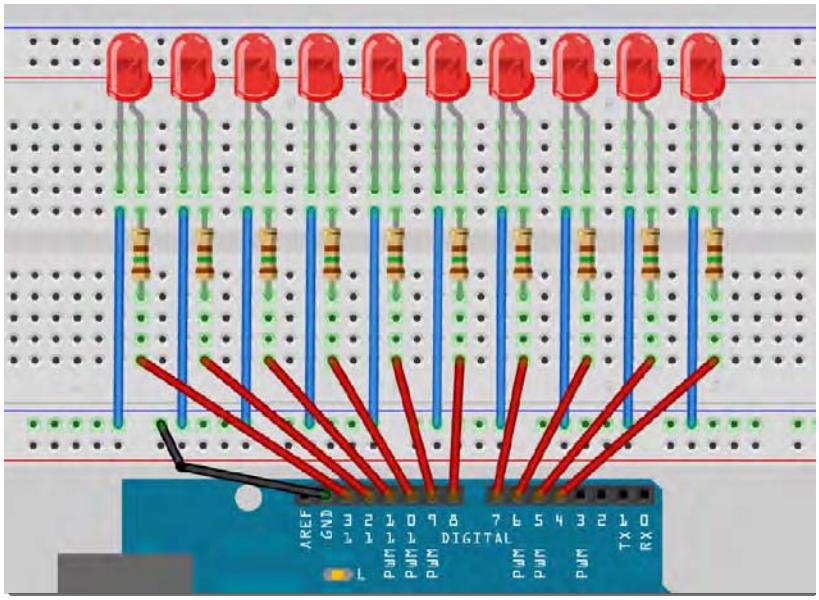


Figure 3-1. *The circuit for Project 5 – LED Chase Effect (see insert for color version)*

Enter the Code

Open up your Arduino IDE and type in the code from Listing 3-1.

Listing 3-1. *Code for Project 5*

```
// Project 5 - LED Chase Effect
byte ledPin[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13};   // Create array for LED pins
int ledDelay(65);  // delay between changes
int direction = 1;
int currentLED = 0;
unsigned long changeTime;

void setup() {
        for (int x=0; x<10; x++) {      // set all pins to output
                pinMode(ledPin[x], OUTPUT); }
        changeTime = millis();
}
```

```
void loop() {
        if ((millis() - changeTime) > ledDelay) {      // if it has been ledDelay ms since
last change
                changeLED();
                changeTime = millis();
        }
}

void changeLED() {
        for (int x=0; x<10; x++) {              // turn off all LED's
                 digitalWrite(ledPin[x], LOW);
        }
        digitalWrite(ledPin[currentLED], HIGH);         // turn on the current LED
        currentLED += direction;        // increment by the direction value
        // change direction if we reach the end
        if (currentLED == 9) {direction = -1;}
        if (currentLED == 0) {direction = 1;}
}
```

Press the Verify/Compile button at the top of the IDE to make sure there are no errors in your code. If this is successful, click the Upload button. If you have done everything right, the LEDs will appear to move along the line then bounce back to the start.

I haven't introduced any new hardware in this project so there's no need to take a look at that. However, I have introduced a new concept in the code for this project in the form of arrays. Let's take a look at the code for Project 5 and see how it works.

Project 5 – LED Chase Effect – Code Overview

The first line in this sketch

```
byte ledPin[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

is a declaration of a variable of data type array. An array is a collection of variables that are accessed using an index number. In your sketch, you declare an array of data type byte and called it `ledPin`. Then, you initialize the array with 10 values (Digital Pins 4 through to 13). To access an element of the array, you simply refer to the index number of that element. Arrays are zero indexed, which means that the first index starts at zero and not 1. So, in your 10 element array, the index numbers are 0 to 9.In this case, element 3 (`ledPin[2]`) has the value of 6 and element 7 (`ledPin[6]`) has a value of 10.

You have to tell the size of the array if you don't initialize it with data first. In your sketch, you didn't explicitly choose a size because the compiler is able to count the values you have assigned to the array to work out that the size is 10 elements. If you had declared the array but not initialized it with values at the same time, you would need to declare a size. For example, you could have done this

```
byte ledPin[10];
```

and then loaded data into the elements later. To retrieve a value from the array, you would do something like this:

```
x = ledpin[5];
```

In this example, x would now hold a value of 8.

To get back to your program, you have started off by declaring and initializing an array that stores 10 values, which are the digital pins used for the outputs to your 10 LEDs.

In your main loop, you check that at least `ledDelay` milliseconds have passed since the last change of LEDs; if so, it passes control to your function. The reason you pass control to the `changeLED()` function in this manner, rather than using `delay()` commands, is to allow other code to run in the main program loop, if needed (as long as that code takes less than `ledDelay` to run).

The function you create is

```
void changeLED() {
        // turn off all LED's
        for (int x=0; x<10; x++) {
                digitalWrite(ledPin[x], LOW);
         }
        // turn on the current LED
        digitalWrite(ledPin[currentLED], HIGH);
        // increment by the direction value
        currentLED += direction;
        // change direction if we reach the end
        if (currentLED == 9) {direction = -1;}
        if (currentLED == 0) {direction = 1;}
}
```

and the job of this function is to turn all LEDs off and then turn on the current LED (this is done so fast you will not see it happening), which is stored in the variable `currentLED`.

This variable then has direction added to it. As direction can only be either a 1 or a -1, the number will either increase (+1) or decrease by one (`currentLED` +(-1)).

Then there's an `if` statement to see if you have reached the end of the row of LEDs; if so, you reverse the direction variable.

By changing the value of `ledDelay` you can make the LED ping back and forth at different speeds. Try different values to see what happens.

Note that you have to stop the program, manually change the value of `ledDelay,` and then upload the amended code to see any changes. Wouldn't it be nice to be able to adjust the speed while the program is running? Yes, it would, so let's do exactly that in the next project. You'll learn how to interact with the program and adjust the speed using a potentiometer.

Project 6 – Interactive LED Chase Effect

Leave your circuit board intact from Project 5. You're just going to add a potentiometer to this circuit, which will allow you to change the speed of the lights while the code is running.

Parts Required

All of the parts for Project 5 plus....

4.7KΩ Rotary Potentiometer

Image courtesy of Iain Fergusson.

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now, add the potentiometer to the circuit so it is connected as shown in Figure 3-2 with the left leg going to the 5v on the Arduino, the middle leg going to Analog Pin 2, and the right leg going to ground.

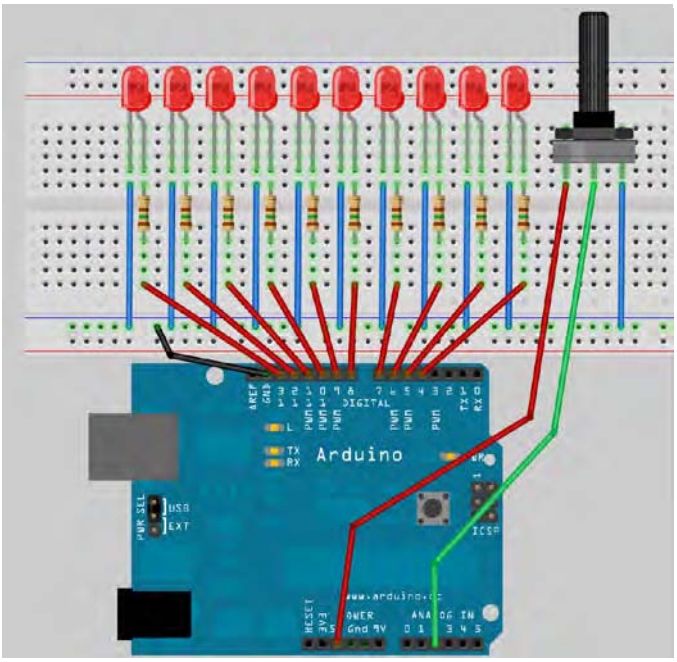


Figure 3-2. *The circuit for Project 6 – Interactive LED Chase Effect (see insert for color version)*

Enter The Code

Open up your Arduino IDE and type in the code from Listing 3-2.

Listing 3-2. *Code for Project 6*

```
byte ledPin[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13};    // Create array for LED pins
int ledDelay; // delay between changes
int direction = 1;
int currentLED = 0;
unsigned long changeTime;
int potPin = 2;    // select the input pin for the potentiometer

void setup() {
for (int x=0; x<10; x++) {   // set all pins to output
                pinMode(ledPin[x], OUTPUT); }
        changeTime = millis();
}

void loop() {
ledDelay = analogRead(potPin); // read the value from the pot
        if ((millis() - changeTime) > ledDelay) {      // if it has been ledDelay ms since
                                                          last change
                changeLED();
                changeTime = millis();
        }
}

void changeLED() {
        for (int x=0; x<10; x++) {   // turn off all LED's
                digitalWrite(ledPin[x], LOW);
        }
        digitalWrite(ledPin[currentLED], HIGH); // turn on the current LED
        currentLED += direction; // increment by the direction value
        // change direction if we reach the end
        if (currentLED == 9) {direction = -1;}
        if (currentLED == 0) {direction = 1;}
}
```

When you verify and upload your code, you should see the lit LED appear to bounce back and forth between each end of the string of lights as before. But, by turning the knob of the potentiometer, you will change the value of `ledDelay` and speed up or slow down the effect.

Let's take a look at how this works and find out what a potentiometer is.

Project 6 – Interactive LED Chase Effect – Code Overview

The code for this Project is almost identical to the previous project. You have simply added a potentiometer to your hardware and the code additions enable it to read the values from the potentiometer and use them to adjust the speed of the LED chase effect.

You first declare a variable for the potentiometer pin

```
int potPin = 2;
```

because your potentiometer is connected to Analog Pin 2. To read the value from an analog pin, you use the analogRead command. The Arduino has six analog input/outputs with a 10-bit analog to digital convertor (I will discuss *bits* later). This means the analog pin can read in voltages between 0 to 5 volts in integer values between 0 (0 volts) and 1,023 (5 volts). This gives a resolution of 5 volts / 1024 units or 0.0049 volts (4.9mV) per unit.

Set your delay using the potentiometer so that you can use the direct values read in from the pin to adjust the delay between 0 and 1023 milliseconds (or just over 1 second). You do this by directly reading the value of the potentiometer pin into `ledDelay`. Notice that you don't need to set an analog pin to be an input or output (unlike with a digital pin):

```
ledDelay = analogRead(potPin);
```

This is done during your main loop and therefore it is constantly being read and adjusted. By turning the knob, you can adjust the delay value between 0 and 1023 milliseconds (or just over a second) and thus have full control over the speed of the effect.

Let's find out what a potentiometer is and how it works.

Project 6 – Interactive LED Chase Effect – Hardware Overview

The only additional piece of hardware used in this project is the 4K7 (4700Ω) potentiometer.

You know how resistors work. Well, the potentiometer is simply an adjustable resistor with a range from 0 to a set value (written on the side of the pot). In this project, you're using a 4K7 (4,700Ω) potentiometer, which means its range is from 0 to 4700 Ohms.

The potentiometer has three legs. By connecting just two legs, the potentiometer becomes a variable resistor. By connecting all three legs and applying a voltage across it, the pot becomes a voltage divider. The latter is how you going to use it in your circuit. One side is connected to ground, the other to 5v, and the center leg to your analog pin. By adjusting the knob, a voltage between 0 and 5v will be leaked from the center pin; you can read the value of that voltage on Analog Pin 2 and use it to change the delay rate of the light effect.

The potentiometer can be very useful in providing a means of adjusting a value from 0 to a set amount, e.g. the volume of a radio or the brightness of a lamp. In fact, dimmer switches for your home lamps are a kind of potentiometer.

EXERCISES

You have all the necessary knowledge so far to adjust the code to enable you to do the following:

- Exercise 1: Get the LEDs at BOTH ends of the strip to start as on, then move towards each other, appear to bounce off each other, and then move back to the end.
- Exercise 2: Make a bouncing ball effect by turning the LEDs so they are vertical, then make an LED start at the bottom, then "bounce" up to the top LED, then back to the bottom, then only up to the 9th LED, then back down, then up to the 8th, and so on to simulate a bouncing ball losing momentum after each bounce.

Project 7 – Pulsating Lamp

You are now going try a more advanced method of controlling LEDs. So far, you have simply turned the LED on or off. Would you like to adjust the brightness of an LED? Can you do that with an Arduino? Yes, you can.

Time to go back to basics.

Parts Required

Green Diffused 5mm LED 

Current Limiting Resistor 

Connect It Up

The circuit for this project is simply a green LED connecting, via a current limiting resistor, between ground and Digital Pin 11 (see Figure 3-3).

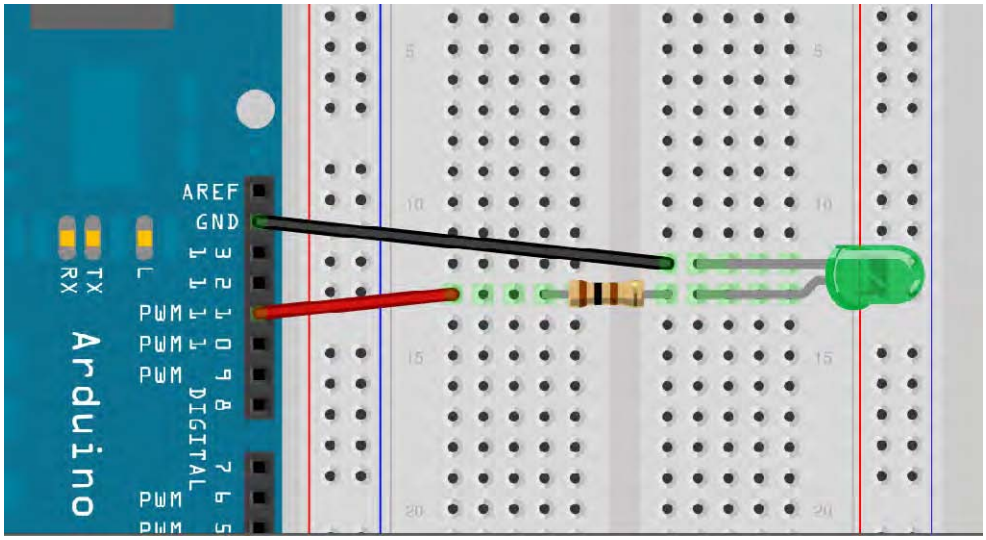


Figure 3-3. *The circuit for Project 7 – Pulsating Lamp (see insert for color version)*

Enter the Code

Open up your Arduino IDE and type in the code from Listing 3-3.

Listing 3-3. *Code for Project 7*

```
// Project 7 - Pulsating lamp
int ledPin = 11;
float sinVal;
int ledVal;

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

void loop() {
        for (int x=0; x<180; x++) {
        // convert degrees to radians then obtain sin value
        sinVal = (sin(x*(3.1412/180)));
        ledVal = int(sinVal*255);
        analogWrite(ledPin, ledVal);
        delay(25);
        }
}
```

Verify and upload. You will see your LED pulsate on and off steadily. Instead of a simple on/off state, however, you're going to adjust the brightness. Let's find out how this works.

Project 7 – Pulsating Lamp – Code Overview

The code for this project is very simple, but it requires some explanation.

First, you set the variables for the LED Pin, a float (floating point data type) for a sine wave value, and `ledVal` which will hold the integer value to send out to Digital PWM Pin 11.

The concept here is that you are creating a sine wave and having the brightness of the LED follow the path of that wave. This is what makes the light pulsate instead of just flare to full brightness and fade back down again.

You use the sin() function, which is a mathematical function, to work out the sine of an angle. You need to give the function the degree in radians. So, you have a `for` loop that goes from 0 to 179; you don't want to go past halfway as this will take you into negative values and the brightness value can only be from 0 to 255.

The sin() function requires the angle to be in radians and not degrees so the equation of x*(3.1412/180) will convert the degree angle into radians. You then transfer the result to `ledVal`, multiplying it by 255 to get the value. The result from the `sin()` function will be a number between -1 and 1, so multiply it by 255 for the maximum brightness. You *cast* the floating point value of `sinVal` into an integer by the use of `int()` in the following statement:

```
ledVal = int(sinVal*255);
```

Then you send that value out to Digital PWM Pin 11 using the statement:

```
analogWrite(ledPin, ledVal);
```

Casting means you have converted the floating point value into an integer (effectively throwing away whatever is after the decimal point). But, how can you send an analog value to a digital pin? Well, take a look at your Arduino. If you examine the digital pins, you can see that six of them (3, 5, 6, 9, 10 & 11) have PWM written next to them. These pins differ from the remaining digital pins in that they are able to send out a PWM signal.

PWM stands for Pulse Width Modulation, which is a technique for getting analog results from digital means. On these pins, the Arduino sends out a square wave by switching the pin on and off very fast. The pattern of on/offs can simulate a varying voltage between 0 and 5v. It does this by changing the amount of time that the output remains high (on) versus off (low). The duration of the on time is known as the *pulse width*.

For example, if you were to send the value 0 out to Digital PWM Pin 11 using `analogWrite()`, the ON period would be zero, or it would have a 0 percent duty cycle. If you were to send a value of 64 (25 percent of the maximum of 255) the pin would be ON for 25 percent of the time and OFF for 75 percent of the time. The value of 191 would have a duty cycle of 75 percent; a value of 255 would have a duty cycle of 100 percent. The pulses run at a speed of approximately 500Hz or 2 milliseconds each.

So, in your sketch, the LED is being turned on and off very fast. If the Duty Cycle was 50 percent (a value of 127), the LED would pulse on and off at 500Hz and would display at half the maximum brightness. This is basically an illusion that you can use to your advantage by allowing the digital pins to output a simulated analog value to your LEDs.

Note that even though only six of the pins have the PWM function, you can easily write software to give a PWM output from all of the digital pins if you wish.

Later, you'll revisit PWM to create audible tones using a piezo sounder.

Project 8 – RGB Mood Lamp

In the last project, you learned how to adjust the brightness of an LED using the PWM capabilities of the Atmega chip. You'll now take advantage of this capability by using a red, green, and blue LED and mixing these colors to create any color you wish. From that, you'll create a mood lamp similar to those seen in stores nowadays.

Parts Required

This time you are going to use three LEDs: one red, one green, and one blue.

Red Diffused 5mm LED

Green Diffused 5mm LED

Blue Diffused 5mm LED

3 Current Limiting Resistors

Connect It Up

Connect the three LEDs as shown in Figure 3-4. Get a piece of letter-size paper, roll it into a cylinder, and tape it to secure it. Place the cylinder over the top of the three LEDs. This will diffuse the light from each LED and merge the colors into one.

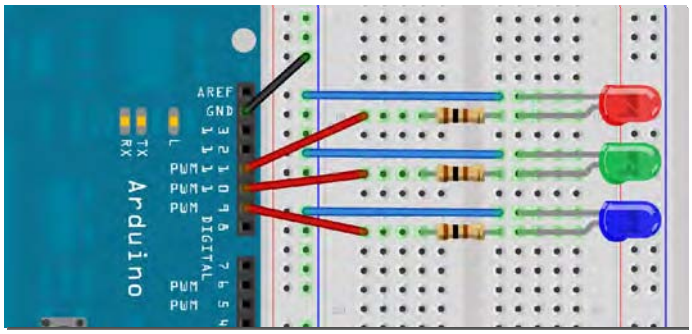


Figure 3-4. *The circuit for Project 8 – RGB Mood Lamp (see insert for color version)*

Enter the Code

Open up your Arduino IDE and type in the code from Listing 3-4.

Listing 3-4. *Code for Project 8*

```
// Project 8 - Mood Lamp
float RGB1[3];
float RGB2[3];
float INC[3];

int red, green, blue;

int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;

void setup()
{
        randomSeed(analogRead(0));

        RGB1[0] = 0;
        RGB1[1] = 0;
        RGB1[2] = 0;

        RGB2[0] = random(256);
        RGB2[1] = random(256);
        RGB2[2] = random(256);
}

void loop()
{
        randomSeed(analogRead(0));

        for (int x=0; x<3; x++) {
                INC[x] = (RGB1[x] - RGB2[x]) / 256; }

        for (int x=0; x<256; x++) {
                red = int(RGB1[0]);
                green = int(RGB1[1]);
                blue = int(RGB1[2]);

                analogWrite (RedPin, red);
                analogWrite (GreenPin, green);
                analogWrite (BluePin, blue);
                delay(100);

                RGB1[0] -= INC[0];
                RGB1[1] -= INC[1];
                RGB1[2] -= INC[2];
        }
```

```
        for (int x=0; x<3; x++) {
                RGB2[x] = random(556)-300;
                RGB2[x] = constrain(RGB2[x], 0, 255);
                delay(1000);
        }
}
```

When you run this, you will see the colors slowly change. You've just made your own mood lamp!

Project 8 – RGB Mood Lamp – Code Overview

The LEDs that make up the mood lamp are red, green, and blue. In the same way that your computer monitor is made up of tiny red, green, and blue (RGB) dots, you can generate different colors by adjusting the brightness of each of the three LEDs in such a way to give you a different RGB value.

Alternatively, you could have used an RGB LED. This is a single 5mm LED, with 4 legs (some have more). One leg is either a common anode (positive) or common cathode (negative); the other three legs go to the opposite terminal of the red, green, and blue LEDs inside the lamp. It is basically three colored LEDs squeezed into a single 5mm LED. These are more compact, but more expensive.

An RGB value of 255, 0, 0 is pure red. A value of 0, 255, 0 is pure green and 0, 0, 255 is pure blue. By mixing these, you can get any color. This is the additive color model (see Figure 3-5). Note that if you were just turning the LEDs ON or OFF (i.e. not trying out different brightness) you would still get different colors.

Table 3-5. *Colors available by turning LEDs ON or OFF in different combinations*

Red	Green	Blue	Color
255 0		0	Red
0 255		0	Green
0 0 255			Blue
255 255	0		Yellow
0 255		255	Cyan
255 0		255	Magenta
255 255	255		White

Diffusing the light with the paper cylinder mixes the colors nicely. The LEDs can be placed into any object that will diffuse the light; another option is to bounce the light off a reflective diffuser. Try putting the lights inside a ping-pong ball or a small white plastic bottle (the thinner the plastic the better).

By adjusting the brightness using PWM, you can get every other color in between, too. By placing the LEDs close together and mixing their values, the light spectra of the three colors added together make a single color (see Figure 3-5). The total range of colors available using PWM with a range of 0 to 255 is 16,777,216 colors (256x256x256).

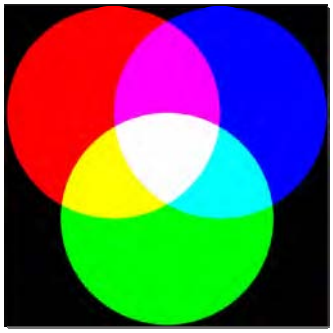


Figure 3-5. *Mixing R, G, and B to get different colors (see insert for color version)*

In the code, you begin by declaring some floating point arrays and some integer variables to store your RGB values as well as an increment value, like so:

```
float RGB1[3];
float RGB2[3];
float INC[3];

int red, green, blue;
```

In the setup function, you have the following:

```
randomSeed(analogRead(0));
```

The randomSeed command creates random (actually pseudo-random) numbers. A computer chip is not able to produce truly random numbers so it looks at data in a part of its memory that may differ or it looks at a table of different values and uses those as pseudo-random numbers. By setting a *seed*, you can tell the computer where in memory or in that table to start counting from. In this case, the value you assign to randomSeed is a value read from Analog Pin 0. Because there's nothing connected to Analog Pin 0, all it will read is a random number created by analog noise.

Once you have set a seed for your random number, you can create one using the `random()` function. You then have two sets of RGB values stored in a three element array. RGB1 contains the RGB values you want the lamp to start with (in this case, all zeros, or off):

```
  RGB1[0] = 0;
  RGB1[1] = 0;
  RGB1[2] = 0;
```

The RGB2 array is a set of random RGB values that you want the lamp to transition to:

```
  RGB2[0] = random(256);
  RGB2[1] = random(256);
  RGB2[2] = random(256);
```

In this case, you have set them to a random number set by random(256) which will give you a number between 0 and 255 inclusive (as the number will always range from zero upwards).

If you pass a single number to the random() function, it will return a value between 0 and 1 less than the number, e.g. random(1000) will return a number between 0 and 999. If you supply two numbers as the parameters, it will return a random number between the lower number inclusive and the maximum number (-1), e.g. random(10,100) will return a random number between 10 and 99.

In the main program loop, you first take a look at the start and end RGB values and work out what value is needed as an increment to progress from one value to the other in 256 steps (as the PWM value can only be between 0 and 255). You do this with the following:

```
for (int x=0; x<3; x++) {
       INC[x] = (RGB1[x] - RGB2[x]) / 256; }
```

This `for` loop sets the INCrement values for the R, G and B channels by working out the difference between the two brightness values and dividing that by 256.

You have another `for` loop

```
for (int x=0; x<256; x++) {

       red = int(RGB1[0]);
       green = int(RGB1[1]);
       blue = int(RGB1[2]);

       analogWrite (RedPin, red);
       analogWrite (GreenPin, green);
       analogWrite (BluePin, blue);
       delay(100);

       RGB1[0] -= INC[0];
       RGB1[1] -= INC[1];
       RGB1[2] -= INC[2];
 }
```

that sets the red, green, and blue values to the values in the RGB1 array; writes those values to Digital Pins 9, 10 and 11; deducts the increment value; and repeats this process 256 times to slowly fade from one random color to the next. The delay of 100ms in between each step ensures a slow and steady progression. You can, of course, adjust this value if you want it slower or faster; you can also add a potentiometer to allow the user to set the speed.

After you have taken 256 slow steps from one random color to the next, the RGB1 array will have the same values (nearly) as the RGB2 array. You now need to decide upon another set of three random values ready for the next time. You do this with another `for` loop:

```
for (int x=0; x<3; x++) {
       RGB2[x] = random(556)-300;
       RGB2[x] = constrain(RGB2[x], 0, 255);
       delay(1000);
}
```

The random number is chosen by picking a random number between 0 and 556 (256+300) and then deducting 300. In this manner, you are trying to force primary colors from time to time to ensure that you don't always just get pastel shades. You have 300 chances out of 556 in getting a negative number and therefore forcing a bias towards one or more of the other two color channels. The next command makes sure that the numbers sent to the PWM pins are not negative by using the `constrain()` function.

The `constrain()` function requires three parameters: x, a, and b where x is the number you want to constrain, a is the lower end of the range, and b is the higher end. So, the `constrain()` function looks at the value of x and makes sure it is within the range of a to b. If it is lower than a, it sets it to a; if it is higher than b, it sets it to b. In your case, you make sure that the number is between 0 and 255 which is the range of your PWM output.

As you use random(556)-300 for your RGB values, some of those values will be lower than zero; the constrain function makes sure that the value sent to the PWM is not lower than zero.

EXERCISE

See if you can change the code to make the colors cycle through the colors of the rainbow rather than between random colors.

Project 9 – LED Fire Effect

Project 9 will use LEDs and a flickering random light effect, via PWM again, to mimic the effect of a flickering flame. If you place these LEDs inside a house on a model railway, for example, you can make it look like the house is on fire, or you can use it in a fireplace in your house instead of wood logs. This is a simple example of how LEDs can be used to create special effects for movies, stage plays, model dioramas, model railways, etc.

Parts Required

This time we are going to use three LEDs: one red and two yellow.

Red Diffused 5mm LED	
2 Yellow Diffused 5mm LED	
3 Current Limiting Resistor	

Connect It Up

Power down your Arduino, then connect your three LEDs as shown in Figure 3-6. This is essentially the same circuit as in Project 8, but using one red and two yellow LEDs instead of a red, green, and blue. Again, the effect is best seen when the light is diffused using a cylinder of paper, or when bounced off a white card or mirror onto another surface.

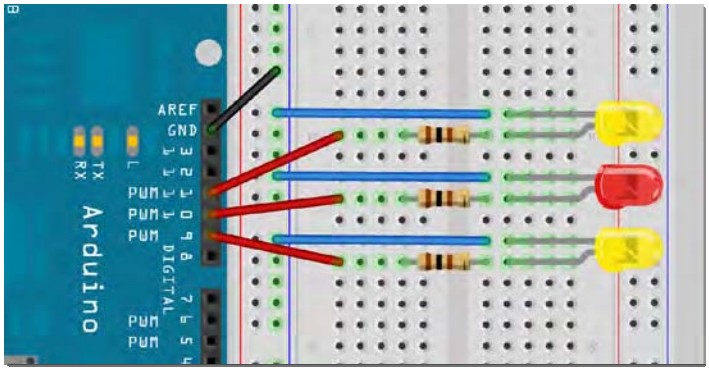


Figure 3-6. *The circuit for Project 9 – LED Fire Effect (see insert for color version)*

Enter the Code

Open up your Arduino IDE and type in the code from Listing 3-5.

Listing 3-5. *Code for Project 9*

```
// Project 9 - LED Fire Effect
int ledPin1 = 9;
int ledPin2 = 10;
int ledPin3 = 11;

void setup()
{
        pinMode(ledPin1, OUTPUT);
        pinMode(ledPin2, OUTPUT);
        pinMode(ledPin3, OUTPUT);
}

void loop()
{
        analogWrite(ledPin1, random(120)+135);
        analogWrite(ledPin2, random(120)+135);
        analogWrite(ledPin3, random(120)+135);
delay(random(100));
}
```

Press the Verify/Compile button at the top of the IDE to make sure there are no errors in your code. If this is successful, click the Upload button.

If you have done everything right, the LEDs will flicker in a random manner to simulate a flame or fire effect.

Project 9 – LED Fire Effect – Code Overview

Let's take a look at the code for this project. First, you declare and initialize some integer variables that will hold the values for the digital pins you are connecting your LEDs to:

```
int ledPin1 = 9;
int ledPin2 = 10;
int ledPin3 = 11;
```

Then, set them to be outputs:

```
pinMode(ledPin1, OUTPUT);
pinMode(ledPin2, OUTPUT);
pinMode(ledPin3, OUTPUT);
```

The main program loop sends out a random value between 0 and 120; add 135 to it to get full LED brightness for the PWM Pins 9, 10, and 11:

```
analogWrite(ledPin1, random(120)+135);
analogWrite(ledPin2, random(120)+135);
analogWrite(ledPin3, random(120)+135);
```

Lastly, you have a random delay between ON and 100ms:

```
delay(random(100));
```

The main loop then starts again, causing the flicker effect. Bounce the light off a white card or a mirror onto your wall and you will see a very realistic flame effect.

The hardware is simple and you should understand it by now, so let's move on to Project 10.

EXERCISE

Try out the following two exercises:

- Exercise 1: Using a blue and/or white LED or two, see if you can recreate the effect of the flashes of light from an arc welder.
- Exercise 2: Using blue and red LEDs, recreate the effect of the lights of an emergency vehicle.

Project 10 – Serial Controlled Mood Lamp

For Project 10, you will revisit the circuit from Project 8 — RGB Mood Lamp, but you'll now delve into the world of serial communications. You'll control your lamp by sending commands from the PC to the

Arduino using the Serial Monitor in the Arduino IDE. Serial communication is the process of sending data one bit at a time across a communication link.

This project also introduces how to manipulate text strings. So, set up the hardware as you did in Project 8 and enter the new code.

Enter the Code

Open up your Arduino IDE and type in the code from Listing 3-6.

Listing 3-6. *Code for Project 10*

```
// Project 10 - Serial controlled mood lamp
char buffer[18];
int red, green, blue;

int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;

void setup()
{
        Serial.begin(9600);
        Serial.flush();
        pinMode(RedPin, OUTPUT);
        pinMode(GreenPin, OUTPUT);
        pinMode(BluePin, OUTPUT);
}

void loop()
{
        if (Serial.available() > 0) {
                int index=0;
                delay(100); // let the buffer fill up
                int numChar = Serial.available();
                if (numChar>15) {
                        numChar=15;
                 }
                while (numChar--) {
                        buffer[index++] = Serial.read();
                }
                splitString(buffer);
        }
}

void splitString(char* data) {
        Serial.print("Data entered: ");
        Serial.println(data);
        char* parameter;
        parameter = strtok (data, " ,");
        while (parameter != NULL) {
```

```
                setLED(parameter);
                parameter = strtok (NULL, " ,");
}

         // Clear the text and serial buffers
        for (int x=0; x<16; x++) {
                buffer[x]='\0';
         }
        Serial.flush();
}

void setLED(char* data) {
        if ((data[0] == 'r') || (data[0] == 'R')) {
                int Ans = strtol(data+1, NULL, 10);
                Ans = constrain(Ans,0,255);
                analogWrite(RedPin, Ans);
                Serial.print("Red is set to: ");
                Serial.println(Ans);
        }
        if ((data[0] == 'g') || (data[0] == 'G')) {
                int Ans = strtol(data+1, NULL, 10);
                Ans = constrain(Ans,0,255);
                analogWrite(GreenPin, Ans);
                Serial.print("Green is set to: ");
                Serial.println(Ans);
        }
        if ((data[0] == 'b') || (data[0] == 'B')) {
                int Ans = strtol(data+1, NULL, 10);
                Ans = constrain(Ans,0,255);
                analogWrite(BluePin, Ans);
                Serial.print("Blue is set to: ");
                Serial.println(Ans);
        }
}
```

Once you've verified the code, upload it to your Arduino.

Note when you upload the program nothing seems to happen. This is because the program is waiting for your input. Start the Serial Monitor by clicking its icon in the Arduino IDE taskbar.

In the Serial Monitor text window, you'll enter the R, G, and B values for each of the three LEDs manually. The LEDs will change to the color you have input.

If you enter R255, the Red LED will display at full brightness. If you enter R255, G255, both the red and green LEDs will display at full brightness. Now enter R127, G100, B255. You get a nice purplish color. Typing r0, g0, b0 will turn off all of the LEDs.

The input text is designed to accept a lowercase or uppercase R, G, and B and then a value from 0 to 255. Any value over 255 will be dropped down to 255 by default. You can enter a comma or a space between parameters and you can enter one, two, or three LED values at any once; for example:

```
r255 b100

r127 b127 g127

G255, B0

B127, R0, G255
```

Project 10 – Serial Controlled Mood Lamp – Code Overview

This project introduces a several new concepts, including serial communication, pointers, and string manipulation. Hold on to your hat; this will take a lot of explaining.

First, you set up an array of char (characters) to hold your text string that is 18 characters long, which is longer than the maximum of 16 allowed to ensure that you don't get "buffer overflow" errors.

```
char buffer[18];
```

You then set up the integers to hold the red, green, and blue values as well as the values for the digital pins:

```
int red, green, blue;

int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;
```

In your setup function, you set the three digital pins to be outputs. But, before that, you have the Serial.begin command:

```
void setup()
{
        Serial.begin(9600);
        Serial.flush();
        pinMode(RedPin, OUTPUT);
        pinMode(GreenPin, OUTPUT);
        pinMode(BluePin, OUTPUT);
}
```

Serial.begin tells the Arduino to start serial communications; the number within the parenthesis (in this case, 9600) sets the baud rate (symbols or pulses per second) at which the serial line will communicate.

The Serial.flush command will flush out any characters that happen to be in the serial line so that it is empty and ready for input/output.

The serial communications line is simply a way for the Arduino to communicate with the outside world, which, in this case, is to and from the PC and the Arduino IDE's Serial Monitor.

In the main loop, you have an `if` statement

```
if (Serial.available() > 0) {
```

that is using the Serial.available command to check to see if any characters have been sent down the serial line. If any characters have been received, the condition is met and the code within the `if` statements code block is executed:

```
if (Serial.available() > 0) {
        int index=0;
        delay(100); // let the buffer fill up
        int numChar = Serial.available();
        if (numChar>15) {
                numChar=15;
        }
        while (numChar--) {
                buffer[index++] = Serial.read();
        }
        splitString(buffer);
}
```

An integer called index is declared and initialized as zero. This integer will hold the position of a pointer to the characters within the char array.

You then set a delay of 100. The purpose of this is to ensure that the serial buffer (the place in memory where the received serial data is stored prior to processing) is full before you process the data. If you don't do that, it's possible that the function will execute and start to process the text string before you have received all of the data. The serial communications line is very slow compared to the execution speed of the rest of the code. When you send a string of characters, the Serial.available function will immediately have a value higher than zero and the `if` function will start to execute. If you didn't have the `delay(100)` statement, it could start to execute the code within the `if` statement before all of the text string had been received, and the serial data might only be the first few characters of the line of text entered.

After you have waited for 100ms for the serial buffer to fill up with the data sent, you then declare and initialize the numChar integer to be the number of characters within the text string.

So, if we sent this text in the Serial Monitor

```
R255, G255, B255
```

the value of numChar would be 17. It is 17, and not 16, because at the end of each line of text there is an invisible character called a NULL character that tells the Arduino when it has reached the end of the line of text.

The next `if` statement checks if the value of numChar is greater than 15; if so, it sets it to be 15. This ensures that you don't overflow the array char buffer[18].

Next is a `while` command. This is something you haven't come across before, so let me explain.

You have already used the `for` loop, which will loop a set number of times. The `while` statement is also a loop, but one that executes only while a condition is true.

The syntax is as follows:

```
while(expression) {
 //        statement(s)
}
```

In your code, the `while` loop is:

```
while (numChar--) {
      buffer[index++] = Serial.read();
    }
```

The condition it is checking is numChar. In other words, it is checking that the value stored in the integer numChar is not zero. Note that numChar has -- after it. This is a post-decrement: the value is decremented *after* it is used. If you had used –numChar, the value in numChar would be decremented (have one subtracted from it) before it was evaluated. In your case, the `while` loop checks the value of numChar and then subtracts 1 from it. If the value of numChar was not zero before the decrement, it then carries out the code within its code block.

numChar is set to the length of the text string that you have entered into the Serial Monitor window. So, the code within the `while` loop will execute that many times.

The code within the `while` loop is

```
buffer[index++] = Serial.read();
```

and this sets each element of the buffer array to each character read in from the Serial line. In other words, it fills up the buffer array with the letters you entered into the Serial Monitor's text window.

The `Serial.read()` command reads incoming serial data, one byte at a time. So now that your character array has been filled with the characters you entered in the Serial Monitor, the `while` loop will end once numChar reaches zero (i.e. the length of the string).

After the `while` loop you have

```
splitString(buffer);
```

which is a call to one of the two functions you created and called `splitString()`. The function looks like this:

```
void splitString(char* data) {
        Serial.print("Data entered: ");
        Serial.println(data);
        char* parameter;
        parameter = strtok (data, " ,");
        while (parameter != NULL) {
                setLED(parameter);
                parameter = strtok (NULL, " ,");
        }

        // Clear the text and serial buffers
        for (int x=0; x<16; x++) {
                buffer[x]='\0';
        }
        Serial.flush();
}
```

The function returns no data, hence its data type has been set to void. You pass the function one parameter, a char data type that you call data. However, in the C and C++ programming languages, you are not allowed to send a character array to a function. You get around that limitation by using a pointer. You know it's a pointer because an asterisk has been added to the variable name *data.

Pointers are an advanced subject in C, so I won't go into too much detail about them. If you need to know more, refer to a book on programming in C. All you need to know for now is that by declaring data as a pointer, it becomes a variable that points to another variable.

You can either point it to the address at which the variable is stored within memory by using the & symbol, or in your case, to the value stored at that memory address using the * symbol. You have used it to cheat the system, because, as mentioned, you aren't allowed to send a character array to a function. However, you are allowed to send a pointer to a character array to your function. So, you have declared a variable of data type Char and called it data, but the * symbol before it means that it is pointing to the value stored within the buffer variable.

When you call `splitString()`, you sent it the contents of buffer (actually a pointer to it, as you saw above):

```
splitString(buffer);
```

So you have called the function and passed it the entire contents of the buffer character array.

The first command is

```
Serial.print("Data entered: ");
```

and this is your way of sending data back from the Arduino to the PC. In this case, the print command sends whatever is within the parentheses to the PC, via the USB cable, where you can read it in the Serial Monitor window. In this case, you have sent the words "Data entered: ". Note that text must be enclosed within quotes. The next line is similar

```
Serial.println(data);
```

and again you have sent data back to the PC. This time, you send the char variable called data, which is a copy of the contents of the buffer character array that you passed to the function. So, if your text string entered is

```
R255 G127 B56
```

then the

```
Serial.println(data);
```

command will send that text string back to the PC and print it out in the Serial Monitor window. (Make sure you have enabled the Serial Monitor window first.)

This time the print command has ln on the end to make it println. This simply means "print with a linefeed."

When you print using the print command, the cursor (the point at where the next symbol will appear) remains at the end of whatever you printed. When you use the println command, a linefeed command is issued, so the text prints and then the cursor drops down to the next line:

```
Serial.print("Data entered: ");
Serial.println(data);
```

So if you look at your two print commands, the first one prints out "Data entered: " and then the cursor remains at the end of that text. The next print command will print data (which is the contents of the array called buffer) and then issue a linefeed, which drops the cursor down to the next line. If you

issue another print or println statement after this, whatever is printed in the Serial Monitor window will appear on the next line.

You then create a new char data type called parameter

```
Char* parameter;
```

and as you are using this variable to access elements of the data array, it must be the same type, hence the * symbol. You cannot pass data from one data type variable to another; the data must be converted first. This variable is another example of one that has *local scope*. It can be seen only by the code within this function. If you try to access the parameter variable outside of the `splitString()` function, you will get an error.

You then use a strtok command, which is a very useful command for manipulating text strings. Strtok gets its name from String and Token because its purpose is to split a string using tokens. In your case, the token it is looking for is a space or a comma; it's being used to split text strings into smaller strings.

You pass the data array to the strtok command as the first argument and the tokens (enclosed within quotes) as the second argument. Hence

```
parameter = strtok (data, " ,");
```

and it splits the string at that point, which is a space or a comma.

So, if your text string is

```
R127 G56 B98
```

then after this statement the value of parameter will be

```
R127
```

because the strtok command splits the string up to the first occurrence of a space of a comma.

After you have set the d variable parameter to the part of the text string you want to strip out (i.e. the bit up to the first space or comma), you then enter a while loop with the condition that the parameter is not empty (i.e. you haven't reached the end of the string):

```
while (parameter != NULL) {
```

Within the loop we call our second function:

```
setLED(parameter);
```

(We will look at this one in detail later.) Then you set the variable parameter to the next part of the string up to the next space or comma. You do this by passing to strtok a NULL parameter, like so:

```
parameter = strtok (NULL, " ,");
```

This tells the strtok command to carry on where it last left off.

So this whole part of the function

```
char* parameter;
parameter = strtok (data, " ,");
while (parameter != NULL) {
        setLED(parameter);
        parameter = strtok (NULL, " ,");
  }
```

is simply stripping out each dpart of the text string that is separated by spaces or commas and sending that part of the string to the next function called `setLED()`.

The final part of this function simply fills the buffer array with NULL character, which is done with the /0 symbol, and then flushes the serial data out of the serial buffer so that it's ready for the next set of data to be entered:

```
// Clear the text and serial buffers
for (int x=0; x<16; x++) {
     buffer[x]='\0';
}
Serial.flush();
```

The `setLED()` function is going to take each part of the text string and set the corresponding LED to the color you have chosen. So, if the text string you enter is

```
G125 B55
```

the `splitString()`function splits that into the two separate components

```
G125
B55
```

and send that shortened text string onto the `setLED()` function, which will read it, decide what LED you have chosen, and set it to the corresponding brightness value.

Let's go back to the second function called `setLED()`:

```
void setLED(char* data) {
        if ((data[0] == 'r') || (data[0] == 'R')) {
                int Ans = strtol(data+1, NULL, 10);
                Ans = constrain(Ans,0,255);
                analogWrite(RedPin, Ans);
                Serial.print("Red is set to: ");
                Serial.println(Ans);
        }
        if ((data[0] == 'g') || (data[0] == 'G')) {
                int Ans = strtol(data+1, NULL, 10);
                Ans = constrain(Ans,0,255);
                analogWrite(GreenPin, Ans);
                Serial.print("Green is set to: ");
                Serial.println(Ans);
        }
```

```
        if ((data[0] == 'b') || (data[0] == 'B')) {
                int Ans = strtol(data+1, NULL, 10);
                Ans = constrain(Ans,0,255);
                analogWrite(BluePin, Ans);
                Serial.print("Blue is set to: ");
                Serial.println(Ans);
        }
}
```

This function contains three similar `if` statements, so let's pick one to examine:

```
if ((data[0] == 'r') || (data[0] == 'R')) {
    int Ans = strtol(data+1, NULL, 10);
    Ans = constrain(Ans,0,255);
    analogWrite(RedPin, Ans);
    Serial.print("Red is set to: ");
    Serial.println(Ans);
  }
```

The `if` statement checks that the first character in the string data[0] is either the letter r or R (upper case and lower case characters are totally different as far as C is concerned. You use the logical OR command (the symbol is ||) to check if the letter is an r OR an R, as either will do.

If it is an r or an R, the `if` statement knows you wish to change the brightness of the red LED, and the code within executes. First, you declare an integer called Ans (which has scope local to the setLED function only) and use the strtol (String to long integer) command to convert the characters after the letter R to an integer. The strtol command takes three parameters: the string you are passing it, a pointer to the character after the integer (which you won't use because you have already stripped the string using the strtok command and hence pass a NULL character), and the base (in your case, it's base 10 because you are using normal decimal numbers as opposed to binary, octal or hexadecimal, which would be base 2, 8 and 16 respectively). In summary, you declare an integer and set it to the value of the text string after the letter R (or the number bit).

Next, you use the constrain command to make sure that Ans goes from 0 to 255 and no more. You then carry out an analogWrite command to the red pin and send it the value of Ans. The code then sends out "Red is set to: " followed by the value of Ans back to the Serial Monitor. The other two `if` statements do exactly the same but for the green and blue LEDs.

You have covered a lot of ground and many new concepts in this project. To make sure you understand exactly what is going on in this code, I have set the project code (which is in C, remember) side by side with pseudo-code (essentially, the computer language described in more detail via whole words and thoughts). See Table 3-7 for the comparison.

Table 3-7. *An explanation for the code in Project 10 using pseudo-code*

The C Programming Language

```
// Project 10 - Serial controlled RGB Lamp
char buffer[18];
int red, green, blue;
int RedPin = 11;
int GreenPin = 10;
int BluePin = 9;

void setup()
{
        Serial.begin(9600);
        Serial.flush();
        pinMode(RedPin, OUTPUT);
        pinMode(GreenPin, OUTPUT);
        pinMode(BluePin, OUTPUT);
}

void loop()
{

  if (Serial.available() > 0) {
        int index=0;
        delay(100); // let the buffer fill up
        int numChar = Serial.available();
        if (numChar>15) {
                numChar=15;
        }
        while (numChar--) {
          buffer[index++] = Serial.read();
        }
        splitString(buffer);
  }
}

void splitString(char* data) {
        Serial.print("Data entered: ");
        Serial.println(data);
        char* parameter;
        parameter = strtok (data, " ,");
        while (parameter != NULL) {
            setLED(parameter);
            parameter = strtok (NULL, " ,");
        }

 // Clear the text and serial buffers
  for (int x=0; x<16; x++) {
        buffer[x]='\0';
  }
  Serial.flush();
}
```

Pseudo-Code

```
A comment with the project number and name
Declare a character array of 18 letters
Declare 3 integers called red, green and blue
An integer assigning a certain pin to the Red
LED
An integer assigning a certain pin to the Green
LED
An integer assigning a certain pin to the Blue
LED

The setup function

Set serial comms to run at 9600 chars per
second
Flush the serial line
Set the red led pin to be an output pin
Same for green
And blue


The main program loop


If data is sent down the serial line...
Declare integer called index and set to 0
Wait 100 millseconds
Set numChar to the incoming data from serial
If numchar is greater than 15 characters...
   Make it 15 and no more

While numChar is not zero (subtract 1 from it)
Set element[index] to value read in (add 1)

Call splitString function and send it data in
buffer



The splitstring function references buffer data
Print “Data entered: “
Print value of data and then drop down a line
Declare char data type parameter
Set it to text up to the first space or comma
While contents of parameter are not empty..
         Call the setLED function
Set parameter to next part of text string


Another comment
Do the next line 16 times
```

```
void setLED(char* data) {
  if ((data[0] == 'r') || (data[0] == 'R')) {
        int Ans = strtol(data+1, NULL, 10);
        Ans = constrain(Ans,0,255);
        analogWrite(RedPin, Ans);
        Serial.print("Red is set to: ");
        Serial.println(Ans);
  }
  if ((data[0] == 'g') || (data[0] == 'G')) {
        int Ans = strtol(data+1, NULL, 10);
        Ans = constrain(Ans,0,255);
        analogWrite(GreenPin, Ans);
        Serial.print("Green is set to: ");
        Serial.println(Ans);
  }
  if ((data[0] == 'b') || (data[0] == 'B')) {
        int Ans = strtol(data+1, NULL, 10);
        Ans = constrain(Ans,0,255);
        analogWrite(BluePin, Ans);
        Serial.print("Blue is set to: ");
        Serial.println(Ans);
  }
}
```

```
Set each element of buffer to NULL (empty)


Flush the serial comms


A function called setLED is passed buffer
If first letter is r or R...
Set integer Ans to number in next part of text
Make sure it is between 0 and 255
Write that value out to the red pin
Print out “Red is set to: “
And then the value of Ans

If first letter is g or G...
Set integer Ans to number in next part of text
Make sure it is between 0 and 255
Write that value out to the green pin
Print out “Green is set to: “
And then the value of Ans

If first letter is b or B...
Set integer Ans to number in next part of text
Make sure it is between 0 and 255
Write that value out to the blue pin
Print out “Blue is set to: “
And then the value of Ans
```

Hopefully, the pseudo-code will help you understand exactly what is going on within the code.

Summary

Chapter 3 introduced many new commands and concepts in programming. You've learned about arrays and how to use them, how to read analog values from a pin, how to use PWM pins, and the basics of serial communications. Knowing how to send and read data across a serial line means you can use your Arduino to communicate with all kinds of serial devices and other devices with simple communication protocols. You will revisit serial communications later in this book.

Subjects and concepts covered in Chapter 3:

- Arrays and how to use them
- What a potentiometer (or variable resistor) is and how to use it
- Reading voltage values from an analog input pin
- How to use the mathematical sine (`sin`) function
- Converting degrees to radians
- The concept of casting a variable to a different type
- Pulse Width Modulation (PWM) and how to use it with `analogWrite()`

- Creating colored lights using different RGB values
- Generating random numbers using `random()` and `randomSeed()`
- How various lighting effects can be generated with the same circuit but different code
- The concept of serial communications
- Setting the serial baud rate using `Serial.begin()`
- Sending commands using the Serial Monitor
- Using an array to create text strings
- Flushing the serial buffer using `Serial.flush`
- Checking if data is sent over the serial line using `Serial.available`
- Creating a loop while a condition is met with the `while()` command
- Reading data from the serial line using `Serial.read()`
- The basic concept of pointers
- Sending data to the Serial Monitor using `Serial.print()` or `Serial.println()`
- Manipulating text strings using the `strtok()` function
- Converting a string to a long integer using `strtol()`
- Constraining a variables value using the `constrain()` function

Simple Sounders and Sensors

This chapter is going to get noisy. You're going to attach a piezo sounder to your Arduino in order to add alarms, warning beeps, alert notifications, etc. to the device you are creating. As of version 0018 of the Arduino IDE, tones can be added easily thanks to a new command. You will also find out how to use the piezo as a sensor and learn how to read voltages from it. Finally, you'll learn about light sensors.

Let's start with a simple car alarm and the `tone()` command to make sounds from your Arduino.

Project 11 – Piezo Sounder Alarm

By connecting a piezo sounder to a digital output pin, you can create a wailing alarm sound. It's the same principle that you used in Project 7 when creating a pulsating lamp via a sine wave, but this time you replace the LED with a piezo sounder or piezo disc.

Parts Required

Piezo Sounder (or piezo disc) 

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now take the piezo sounder and screw its wires into the screw terminal. Connect the screw terminal to the breadboard and then connect it to the Arduino, as in Figure 4-1. Now, connect your Arduino back to the USB cable and power it up.

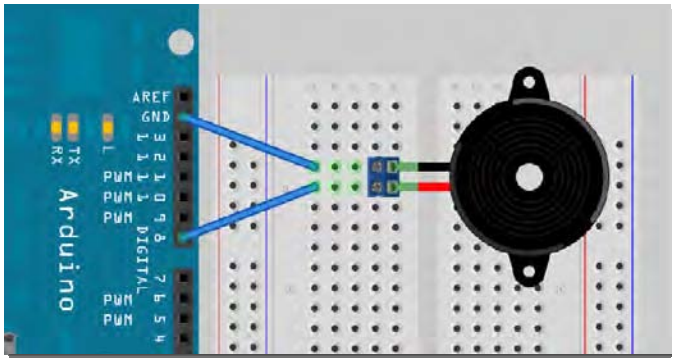


Figure 4-1. *The circuit for Project 11 – Piezo Sounder Alarm (see insert for color version)*

Enter the Code

Open up your Arduino IDE and type in the code from Listing 4-1.

Listing 4-1. *Code for Project 11*

```
// Project 11 - Piezo Sounder Alarm

float sinVal;
int toneVal;

void setup() {
        pinMode(8, OUTPUT);
}

void loop() {
        for (int x=0; x<180; x++) {
                // convert degrees to radians then obtain sin value
                sinVal = (sin(x*(3.1412/180)));
                // generate a frequency from the sin value
                toneVal = 2000+(int(sinVal*1000));
                tone(8, toneVal);
                delay(2);
        }
}
```

After you upload the code, there will be a slight delay and then your piezo will start emitting sounds. If everything is working as planned, you'll hear a rising and falling siren type alarm, similar to a car alarm. The code for Project 11 is almost identical to the code for Project 7; let's see how it works.

Project 11 – Piezo Sounder Alarm – Code Overview

First, you set up two variables:

```
float sinVal;
int toneVal;
```

The `sinVal` float variable holds the sin value that causes the tone to rise and fall in the same way that the lamp in Project 7 pulsated. The `toneVal` variable takes the value in `sinVal` and converts it to the frequency you require.

In the setup function, you set Digital Pin 8 to an output:

```
void setup() {
        pinMode(8, OUTPUT);
}
```

In the main loop, you set a `for` loop to run from 0 to 179 to ensure that the sin value does not go into the negative (as you did in Project 7):

```
for (int x=0; x<180; x++) {
```

You convert the value of x into radians (again, as in Project 7):

```
sinVal = (sin(x*(3.1412/180)));
```

Then that value is converted into a frequency suitable for the alarm sound:

```
toneVal = 2000+(int(sinVal*1000));
```

You take 2000 and add the sinVal multiplied by 1000. This supplies a good range of frequencies for the rising and falling tone of the sine wave.

Next, you use the `tone()` command to generate the frequency at the piezo sounder:

```
tone(8, toneVal);
```

The `tone()` command requires either two or three parameters, thus:

```
tone(pin, frequency)
tone(pin, frequency, duration)
```

The pin is the digital pin being used to output to the piezo and the frequency is the frequency of the tone in hertz. There is also the optional duration parameter in milliseconds for the length of the tone. If no duration is specified, the tone will keep on playing until you play a different tone or you use the `noTone(pin)` command to cease the tone generation on the specified pin.

Finally, you run a delay of 2 milliseconds between the frequency changes to ensure the sine wave rises and falls at the speed you require:

```
delay(2);
```

If you are wondering why you didn't put the 2 milliseconds into the duration parameter of the `tone()` command like this

```
tone(8, toneVal, 2);
```

it's because the `for` loop is so short that it will change the frequency in less than 2 milliseconds anyway, thus rendering the duration parameter useless. Therefore, a delay of 2 milliseconds is put in after the tone is generated to ensure that it plays for at least 2 milliseconds before the `for` loop repeats and the tone changes again.

You could use this alarm generation principle later when you learn how to connect sensors to your Arduino. Then you could activate an alarm when a sensor threshold has been reached, such as if someone gets too close to an ultrasonic detector or if a temperature gets too high.

If you change the values of 2000 and 1000 in the `toneVal` calculation and the length of the delay, you can generate different alarm sounds. Have some fun and see what sounds you can make!

Project 11 – Piezo Sounder Alarm – Hardware Overview

There are two new components in this project: a screw terminal and a piezo sounder. You use the screw terminal because the wires from your piezo sounder or disc are too thin and soft to insert into the breadboard. The screw terminal has pins on it that allow you to push it into a breadboard.

The piezo sounder or piezo disc (see Figure 4-2) is a simple device made up of a thin layer of ceramic bonded to a metallic disc.

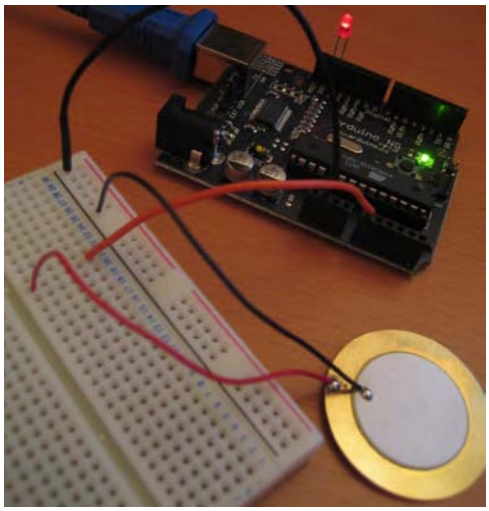


Figure 4-2. *A piezo disc and Arduino (Image courtesy of Patrick H. Lauke/splintered.co.uk)*

Piezoelectric materials, which are made up of crystals and ceramics, have the ability to produce electricity when mechanical stress is applied to them. The effect finds useful applications such as the production and detection of sound, generation of high voltages, electronic frequency generation, microbalances, and ultra fine focusing of optical assemblies.

The effect is also reversible; if an electric field is applied across the piezoelectric material, it will cause the material to change shape (by as much as 0.1 percent in some cases).

To produce sounds from a piezo disc, an electric field is turned on and off very fast to make the material change shape; this causes an audible "click" as the disc pops out and back in again (like a tiny drum). By changing the frequency of the pulses, the disc will deform hundreds or thousands of times per second, causing the buzzing sound. By changing the frequency of the clicks and the time in between them, specific notes can be produced.

You can also use the piezo's ability to produce an electric field to measure movement or vibrations. In fact, piezo discs are used as contact microphones for guitars or drum kits. You will use this feature of a piezo disc in Project 13 when you make a knock sensor.

Project 12 – Piezo Sounder Melody Player

Rather than using the piezo to make annoying alarm sounds, why not use it to play a melody? You are going to get your Arduino to play the chorus of "Puff the Magic Dragon." Leave the circuit exactly the same as in Project 11; you are just changing the code.

Enter the Code

Open up your Arduino IDE and type in the code from Listing 4-2.

Listing 4-2. *Code for Project 12*

```
// Project 12 - Piezo Sounder Melody Player

#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
#define NOTE_G4  392
#define NOTE_GS4 415
```

```
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494

#define WHOLE 1
#define HALF 0.5
#define QUARTER 0.25
#define EIGHTH 0.125
#define SIXTEENTH 0.0625

int tune[] = { NOTE_C4, NOTE_C4, NOTE_C4, NOTE_C4, NOTE_C4, NOTE_B3, NOTE_G3, NOTE_A3,↩
 NOTE_C4, NOTE_C4, NOTE_G3, NOTE_G3, NOTE_F3, NOTE_F3, NOTE_G3, NOTE_F3, NOTE_E3, NOTE_G3,↩
 NOTE_C4, NOTE_C4, NOTE_C4, NOTE_C4, NOTE_A3, NOTE_B3, NOTE_C4, NOTE_D4};

float duration[] = { EIGHTH, QUARTER+EIGHTH, SIXTEENTH, QUARTER, QUARTER, HALF, HALF,↩
 HALF, QUARTER, QUARTER, HALF+QUARTER, QUARTER, QUARTER, QUARTER, QUARTER+EIGHTH, EIGHTH,↩
 QUARTER, QUARTER, QUARTER, EIGHTH, EIGHTH, QUARTER, QUARTER, QUARTER, QUARTER,↩
 HALF+QUARTER};

int length;

void setup() {
        pinMode(8, OUTPUT);
        length = sizeof(tune) / sizeof(tune[0]);
}

void loop() {
        for (int x=0; x<length; x++) {
                tone(8, tune[x]);
                delay(1500 * duration[x]);
                noTone(8);
        }
        delay(5000);
}
```

After you upload the code, there will be a slight delay and then your piezo will start to play a tune. Hopefully you will recognize it as part of the chorus of "Puff the Magic Dragon." Now, let's look at the new concepts from this project.

Project 12 – Piezo Sounder Melody Player – Code Overview

The first thing you see when looking at the code for Project 12 is the long list of define directives. The *define directive* is very simple and very useful. `#define` simply defines a value and its token. For example,

```
#define PI 3.14159265358979323846264338327950288419716939937510
```

will allow you to substitute PI in any calculation instead of having to type out pi to 50 decimal places. Another example,

```
#define TRUE  1
#define FALSE 0
```

means that you can put a TRUE or FALSE into your code instead of a 0 or a 1. This makes logical statements easier for a human to read.

Let's say that you wrote some code to display shapes on an LED dot matrix display and the resolution of the display was 8 x 32. You could create define directives for the height and width of the display thus:

```
#define DISPLAY_HEIGHT 8
#define DISPLAY_WIDTH 32
```

Now, whenever you refer to the height and width of the display in your code you can put `DISPLAY_HEIGHT` and `DISPLAY_WIDTH` instead of the numbers 8 and 32.

There are two main advantages to doing this instead of simply using the numbers. Firstly, the code becomes a lot easier to understand as you have changed the height and width values of the display into tokens that make these numbers clearer to a third party. Secondly, if you change your display at a later date to a larger resolution, say a 16 × 64 display, all you need to do is changed the two values in the define directives instead of having to change numbers in what could be hundreds of lines of code. By changing the values in the define directive at the start of the program the new values are automatically used throughout the rest of the code.

In Project 12, you create a whole set of define directives where the tokens are the notes C3 through to B4 and the values are the frequencies required to create that note. The first note of your melody is C4 and its corresponding frequency is 262 Hz. This is middle C on the musical scale. (Not all of the notes defined are used in your melody, but I have included them in case you wish to write your own tune.)

The next five define directives are for the note lengths. The notes can be a whole bar, half, quarter, eighth, or a sixteenth of a bar in length. The numbers are what we will use to multiply the length of the bar in milliseconds to get the length of each note. For example, a quarter note is 0.25 (or one quarter of one); therefore, multiply the length of the bar (in this case, 1500 milliseconds) by 0.25 to get the length of a quarter note:

1500 × QUARTER = 375 milliseconds

Define directives can also be used for creating macros; more on macros in a later chapter.

Next, you define an integer array called `tune[]` and fill it with the notes for "Puff the Magic Dragon" like so:

```
int tune[] = { NOTE_C4, NOTE_C4, NOTE_C4, NOTE_C4, NOTE_C4, NOTE_B3, NOTE_G3, NOTE_A3,↩
 NOTE_C4, NOTE_C4, NOTE_G3, NOTE_G3, NOTE_F3, NOTE_F3, NOTE_G3, NOTE_F3, NOTE_E3, NOTE_G3,↩
 NOTE_C4, NOTE_C4, NOTE_C4, NOTE_C4, NOTE_A3, NOTE_B3, NOTE_C4, NOTE_D4};
```

After that, you create another array, a float that will hold the duration of the each note as it is played:

```
float duration[] = { EIGHTH, QUARTER+EIGHTH, SIXTEENTH, QUARTER, QUARTER, HALF, HALF,↩
 HALF, QUARTER, QUARTER, HALF+QUARTER, QUARTER, QUARTER, QUARTER, QUARTER+EIGHTH, EIGHTH,↩
 QUARTER, QUARTER, QUARTER, EIGHTH, EIGHTH, QUARTER, QUARTER, QUARTER, QUARTER,↩
 HALF+QUARTER};
```

As you can see by looking at these arrays, the use of the define directives to define the notes and the note lengths makes reading and understanding the array a lot easier than if it were filled with a series of numbers. You then create an integer called length

```
int length;
```

which will be used to calculate and store the length of the array (i.e. the number of notes in the tune).

In your setup routine, you set Digital Pin 8 to an output

```
pinMode(8, OUTPUT);
```

then initialize the integer length with the number of notes in the array using the `sizeof()` function:

```
length = sizeof(tune) / sizeof(tune[0]);
```

The sizeof function returns the number of bytes in the parameter passed to it. On the Arduino, an integer is made up of two bytes. A *byte* is made up of 8 bits. (This is delving into the realm of binary arithmetic and for this project you do not need to worry about bits and bytes. You will come across them later in the book and all will be explained.) Your tune just happens to have 26 notes in it, so the `tunes[]` array has 26 elements. To calculate that we get the size (in bytes) of the entire array

```
sizeof(tune)
```

and divide that by the number of bytes in a single element

```
sizeof(tune[0])
```

which, in this case, this is equivalent to

```
26 / 2 = 13
```

If you replace the tune in the project with one of your own, length will be calculated as the number of notes in your tune.

The `sizeof()` function is useful in working out the lengths of different data types and is particularly useful if you were to port your code over to another device where the length of the datatypes may differ from those on the Arduino.

In the main loop, you set up a `for` loop that iterates the number of times there are notes in the melody

```
for (int x=0; x<length; x++) {
```

then play the next note in the `tune[]` array on Digital Pin 8

```
tone(8, tune[x]);
```

then wait the appropriate amount of time to let the note play

```
delay(1500 * duration[x]);
```

The delay is 1500 milliseconds multiplied by the note length (0.25 for a quarter note, 0.125 for an eighth note, etc.).

Before the next note is played you cease the tone generated on Digital Pin 8:

```
noTone(8);
```

This is to ensure that when two identical notes are played back to back they can be distinguished as individual notes. Without the `noTone()` function, the notes would merge into one long note instead.

Finally, after the `for` loop is complete, you run a delay of 5 seconds before repeating the melody over again:

```
delay(5000);
```

To create the notes for this tune, I found some public domain sheet music for "Puff the Magic Dragon" on the Internet and typed the notes into the `tune[]` array, followed by the note lengths in the `duration[]` array. Note that I have added note lengths to get dotted notes (e.g. `QUARTER+EIGHTH`). By doing something similar you can create any tune you want.

If you wish to speed up or slow down the pace of the tune, change the value of 1500 in the delay function to something higher or lower.

You can also replace the piezo in the circuit with a speaker or headphones, as long as you put a resistor in series with it to ensure that the maximum current for the speaker is not exceeded.

You are going to use the piezo disc for another purpose—its ability to produce a current when the disc is squeezed or knocked. Utilizing this feature, you are going to make a Knock Sensor in Project 13.

Project 13 – Piezo Knock Sensor

A piezo disc works when an electric current is passed over the ceramic material in the disc, causing it to change shape and hence make a sound (a click). The disc also works in reverse: when the disc is knocked or squeezed, the force on the material causes the generation of an electric current. You can read that current using the Arduino and you are going to do that now by making a Knock Sensor.

Parts Required

Piezo Sounder (or piezo disc) 

5mm LED (any color) 

1MΩ Resistor 

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Then connect up your parts so you have the circuit in Figure 4-3. Note that a piezo disc works better for this project than a piezo sounder.

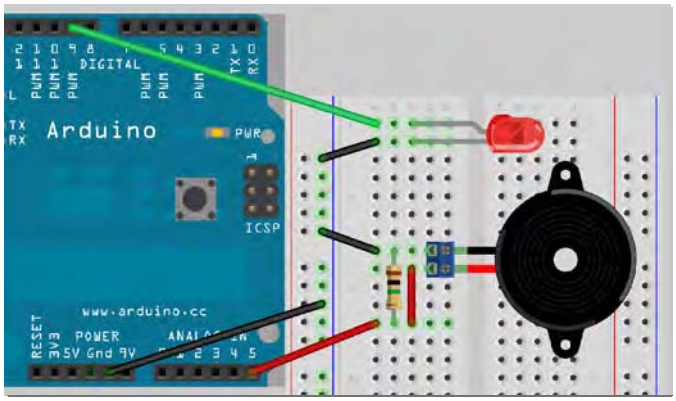


Figure 4-3. *The circuit for Project 13 – Piezo Knock Sensor (see insert for color version)*

Enter the Code

Open up your Arduino IDE and type in the code from Listing 4-3.

Listing 4-3. *Code for Project 13*

```
// Project 13 - Piezo Knock Sensor

int ledPin = 9; // LED on Digital Pin 9
int piezoPin = 5; // Piezo on Analog Pin 5
int threshold = 120; // The sensor value to reach before activation
int sensorValue = 0;  // A variable to store the value read from the sensor
float ledValue = 0; // The brightness of the LED


void setup() {
        pinMode(ledPin, OUTPUT); // Set the ledPin to an OUTPUT
        // Flash the LED twice to show the program has started
        digitalWrite(ledPin, HIGH); delay(150); digitalWrite(ledPin, LOW); delay(150);
        digitalWrite(ledPin, HIGH); delay(150); digitalWrite(ledPin, LOW); delay(150);
}
```

```
void loop() {
        sensorValue = analogRead(piezoPin);  // Read the value from the sensor
        if (sensorValue >= threshold) { // If knock detected set brightness to max
                ledValue = 255;
        }
        analogWrite(ledPin, int(ledValue) ); // Write brightness value to LED
        ledValue = ledValue - 0.05; // Dim the LED slowly
        if (ledValue <= 0) { ledValue = 0;}  // Make sure value does not go below zero
}
```

After you have uploaded your code, the LED will flash quickly twice to indicate that the program has started. You can now knock the sensor (place it flat on a surface first) or squeeze it between your fingers. Every time the Arduino detects a knock or squeeze, the LED will light up and then gently fade back down to off. (Note that the threshold value in the code was set for the specific piezo disc I used when building the project. You may need to set this to a higher or lower value depending on the type and size of piezo you have used for your project. Lower is more sensitive and higher is less.)

Project 13 – Piezo Knock Sensor – Code Overview

There aren't any new code commands in this project, but I'll go over how it works anyway.

First, set up the necessary variables for your program; these are self explanatory:

```
int ledPin = 9; // LED on Digital Pin 9
int piezoPin = 5; // Piezo on Analog Pin 5
int threshold = 120; // The sensor value to reach before activation
int sensorValue = 0;  // A variable to store the value read from the sensor
float ledValue = 0; // The brightness of the LED
```

In the setup function, the ledPin is set to an output and, as noted, the LED is flashed quickly twice as a visual indication that the program has started working:

```
void setup() {
        pinMode(ledPin, OUTPUT);
        digitalWrite(ledPin, HIGH); delay(150); digitalWrite(ledPin, LOW); delay(150);
        digitalWrite(ledPin, HIGH); delay(150); digitalWrite(ledPin, LOW); delay(150);
}
```

In the main loop, you first read the analog value from Analog Pin 5, which the piezo is attached to:

```
sensorValue = analogRead(piezoPin);
```

Then the code checks if that value is greater than or equal to (>=) the threshold you have set, i.e. if it really is a knock or squeeze. (The piezo is very sensitive as you will see if you set the threshold to a very low value). If yes, then it sets ledValue to 255, which is the maximum voltage out of Digital PWM Pin 9:

```
if (sensorValue >= threshold) {
        ledValue = 255;
}
```

You then write that value to Digital PWM Pin 9. Because `ledValue` is a float, you cast it to an integer, as the analogWrite function can only accept an integer and not a floating value

```
analogWrite(ledPin, int(ledValue) );
```

and then reduce the value of `ledValue`, which is a float, by 0.05

```
ledValue = ledValue - 0.05;
```

You want the LED to dim gently, hence you use a float instead of an integer to store the brightness value of the LED. This way you can deduct its value by a small amount (in this case 0.05), so it will take a little while as the main loop repeats for the value of `ledValue` to reach zero. If you want the LED to dim slower or faster, increase or decrease this value.

Finally, you don't want `ledValue` to go below zero as Digital PWM Pin 9 can only output a value from 0 to 255, so you check if it is smaller or equal to zero, and if so, change it back to zero:

```
if (ledValue <= 0) { ledValue = 0;}
```

The main loop then repeats, dimming the LED slightly each time until the LED goes off or another knock is detected and the brightness is set back to maximum.

Now let's introduce a new sensor, the Light Dependent Resistor or LDR.

Project 14 – Light Sensor

This project introduces a new component known as a Light Dependent Resistor, or LDR. As the name implies, the device is a resistor that depends on light. In a dark environment, the resistor has a very high resistance. As photons (light) land on the detector, the resistance decreases. The more light, the lower the resistance. By reading the value from the sensor, you can detect if it is light, dark, or anywhere between. In this project, you use an LDR to detect light and a piezo sounder to give audible feedback of the amount of light detected.

This setup could be used as an alarm that indicates when a door has been opened, for example. Alternatively, you could use it to create a musical instrument similar to a theremin.

Parts Required

Piezo Sounder (or piezo disc) 

Light Dependent Resistor 

10kΩ Resistor 

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Then connect up your parts so you have the circuit shown in Figure 4-3. Check all of your connections before reconnecting the power to the Arduino.

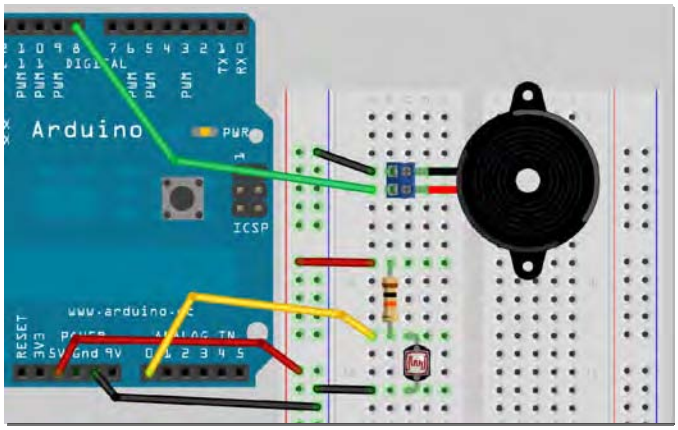


Figure 4-4. *The circuit for Project 14 – Light Sensor (see insert for color version)*

The LDR can be inserted any way because it does not have polarity. I found a 10kΩ resistor worked well for my LDR but you may need to try different resistor settings until you find one suitable for your LDR. A value between 1kΩ and 10kΩ should do the trick. Having a selection of different common resistor values in your component box will always come in handy.

Enter the Code

Now fire up your Arduino IDE and enter the short and simple code in Listing 4-4.

Listing 4-4. *Code for Project 13*

```
// Project 14 - Light Sensor

int piezoPin = 8;  // Piezo on Pin 8
int ldrPin = 0;   // LDR on Analog Pin 0
int ldrValue = 0;  // Value read from the LDR

void setup() {
  // nothing to do here
}
```

```
void loop() {
        ldrValue = analogRead(ldrPin); // read the value from the LDR
        tone(piezoPin,1000); // play a 1000Hz tone from the piezo
        delay(25);  // wait a bit
        noTone(piezoPin);  // stop the tone
        delay(ldrValue); // wait the amount of milliseconds in ldrValue
}
```

When you upload this code to the Arduino, the Arduino makes short beeps. The gap between the beeps will be long if the LDR is in the shade and will be short if bright light shines on the LDR, giving it a Geiger counter type effect. You may find it more practical to solder a set of long wires to the LDR to allow you to keep your breadboard and Arduino on the table while moving the LDR around to point it at dark and light areas. Alternatively, shine a flashlight on the sensor and move it around.

The code for Project 14 is very simple and you should be able to work out how it works yourself without any help. I will, however, show you how an LDR works and why the additional resistor is important.

Project 14 – Light Sensor – Hardware Overview

The new component in this project is a Light Dependent Resistor (LDR), otherwise known as a CdS (Cadmium-Sulfide) or a photoresistor. LDRs come in and shapes and sizes (see Figure 4-5) and in different ranges of resistance.

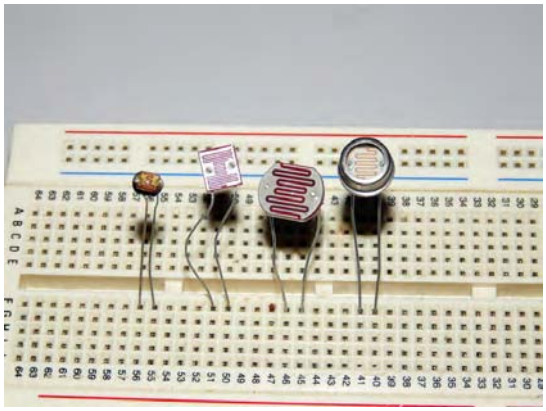


Figure 4-5. *Different kinds of LDR (image by cultured_society2nd)*

Each of the legs on the LDR goes to an electrode. Between a darker material, making a squiggly line between the electrodes, is the photoconductive material. The component has a transparent plastic or glass coating. When light hits the photoconductive material, it loses its resistance, allowing more current to flow between the electrodes. LDRs can be used in all kinds of interesting projects; for example, you could fire a laser into an LDR and detect when a person breaks the beam, triggering an alarm or a shutter on a camera.

The next new concept in your circuit is a voltage divider (also known as a potential divider). This is where the resistor comes in. By using two resistors and taking the voltage across just one of them you can reduce the voltage going into the circuit. In your case, you have a resistor of a fixed value (10kΩ or thereabouts) and variable resistor in the form of a LDR. Let's take a look at a standard voltage divider circuit using resistors and see how it works. Figure 4-6 shows a voltage divider using two resistors.

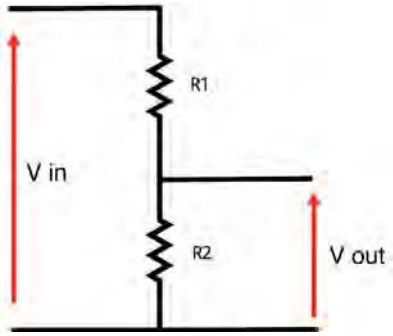


Figure 4-6. *A voltage divider*

The voltage in (Vin) is connected across both resistors. When you measure the voltage across one of the resistors (Vout) it will be less (divided). The formula for working out what the voltage at Vout comes out when measured across R2 is:

$$Vout = \frac{R2}{R2 + R1} \times Vin$$

So, if you have 100Ω resistors (or 0.1kΩ) for both R1 and R2 and 5v going into Vin, your formula is:

$$\frac{0.1}{0.1 + 0.1} \times 5 = 2.5 \text{ volts}$$

Let's do it again with 470Ω resistors:

$$\frac{0.47}{0.47 + 0.47} \times 5 = 2.5 \text{ volts}$$

Again, you get 2.5 volts. This demonstrates that the value of the resistors is not important, but the ratio between them is. Let's try a 1kΩ and a 500Ω resistor:

$$\frac{0.5}{0.5 + 1} \times 5 = 1.66 \text{ volts}$$

With the bottom resistor half the value of the top one, you get 1.66 volts, which is a third of the voltage going in. Let's make the bottom resistor twice the value of the top at 2kΩ

$$\frac{2}{2+1} \times 5 = 3.33 \text{ volts}$$

which is two-thirds of the voltage going in. So, let's apply this to the LDR. You can presume that the LDR has a range of around 10kΩ when in the dark and 1kΩ in bright light. Table 4-1 shows what voltages you will get out of your circuit as the resistance changes.

Table 4-1. *Vout values for a LDR with 5v as Vin*

R1	R2 (LDR)	Vout	Brightness
10kΩ	100kΩ	4.54v Dark	est
10kΩ	73kΩ	4.39v 25%	
10kΩ	45kΩ	4.09v 50%	
10kΩ	28kΩ	3.68v 75%	
10kΩ	10kΩ	2.5v Bri	ghtest

As you can see, as the brightness increases, the voltage at Vout decreases. As a result, the value you read at the sensor gets less and the delay after the beep gets shorter, causing the beeps to occur more frequently. If you were to switch the resistor and LDR, the voltage would increase as more light fell onto the LDR. Either way will work; it just depends how you want your sensor to be read.

Summary

In Chapter 6, you learned how to make music, alarm sounds, warning beeps, etc, from your Arduino. These sounds have many useful applications. You can, for example, make your own alarm clock. By using a piezo sounder in reverse to detect voltages from it and use that effect to detect a knock or pressure on the disc, you can make a musical instrument. Finally, by using an LDR to detect light, you can turn on a night light when ambient light falls below a certain threshold.

Subjects and Concepts covered in Chapter 4:

- What a piezoelectric transducer is and how it works
- How to create sounds using the `tone()` function
- How to stop tone generation using the `noTone()` function
- The `#define` command and how it makes code easier to debug and understand
- Obtaining the size of an array (in bytes) using the `sizeof()` function
- What an LDR (Light Dependent Resistor) is, how it works, and how to read values from it
- The concept of voltage dividers and how to use them