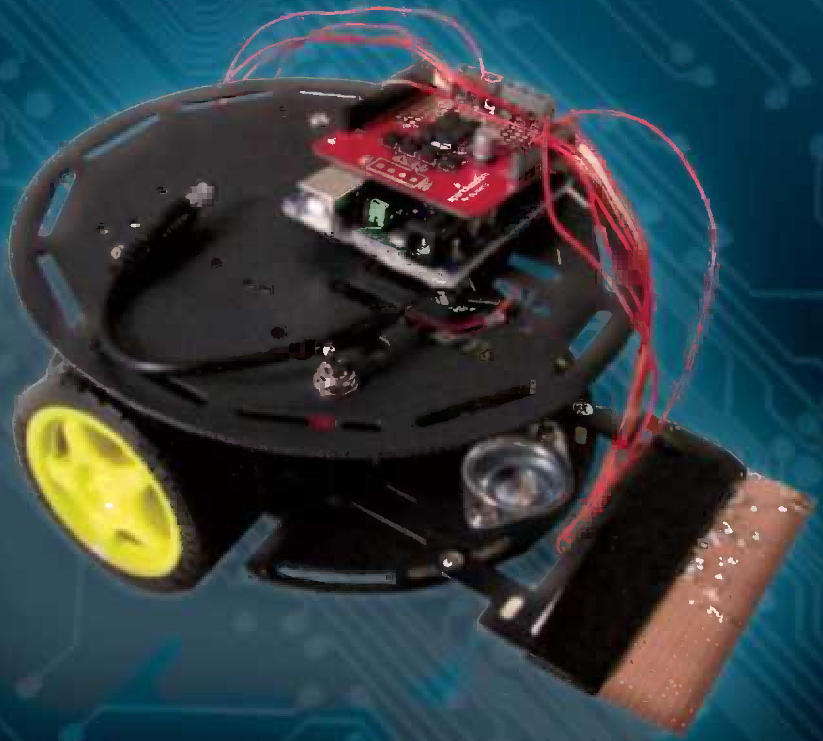
TECHNOLOGY IN ACTION™

Beginning Arduino



CREATE SIMPLE BUT PRACTICAL PROJECTS WITH ARDUINO USING STEP-BY-STEP INSTRUCTIONS AND EASY-TO-FOLLOW DIAGRAMS, NO EXPERIENCE NECESSARY

Michael McRoberts

SECOND EDITION

Contents at a Glance

Introduction

I first discovered the Arduino in 2008 when I was looking for ways to connect temperature sensors to my PC so I could make a cloud detector. I wanted to try out a cloud detection concept I'd read about on a weather forum, and as it was experimental, I didn't want to spend a lot of money on it in case it failed. There were many solutions on the market, but the Arduino appealed to me the most. Not only did it seem to be an easy and cheap way to connect the sensors I required, but it could be used for other cool things. Thousands of projects in blogs, video sites, and forums showed the amazing things people were doing with their Arduinos. There seemed to be a huge sense of community with everyone trying to help one another.

It was obvious that I could have a lot of fun with an Arduino. However, I didn't want to be trawling through websites for information. I wanted to buy a book on the subject, something I could hold in my hand and read on the train to work. After looking around, I found one book. Unfortunately, it was very basic and out of date. Worse, it didn't give me anything practical to do with the Arduino, and I didn't warm to the teaching style, either. What I wanted was a hands-on book that taught me both programming and electronics as I built things instead of having to wade through pages of theory first. Such a book just didn't exist at the time.

Then I started Earthshine Electronics to sell kits based on the Arduino. To go with the kit, I produced a small tutorial booklet to get people started. This little booklet ended up being extremely popular, and I got hundreds of queries from people asking when I would be adding more projects or if I sold a printed version. In fact, I had already thought that it would be great to produce a comprehensive beginner's book, crammed with projects and written in an easy-to-follow style. That is how this book came about. This book has proven so successful at teaching people about the Arduino that it has since been updated to this second edition with improvements and updated sections relevant to the changes in the Arduino world since I began.

I have written this book with the presumption that you have never done either computer programming or electronics before. I also presume you're not interested in reading lots of theory before you actually get down to making something with your Arduino. Hence, right from the start of the book, you will be diving right into making a simple project. From there, you will work through a total of 50 projects until you become confident and proficient at Arduino development. I believe that the best way to learn anything is by learning as you go and getting your hands dirty.

The book works like this: the first project introduces basic concepts about programming the Arduino and also about electronics. The next project builds on that knowledge to introduce a little bit more. Each project after that builds on the previous projects. By the time you have finished all 50 projects, you will be confident and proficient at making your own projects. You'll be able to adapt your new skills and knowledge to connect just about anything to your Arduino and make great projects for fun or to make your life easier.

Each project starts off with a list of required parts. I have chosen common parts that are easy to source. I also provide a circuit diagram showing exactly how to connect the Arduino and parts together using jumper wires and a breadboard. To create the parts images and breadboard diagrams for the book, I used the excellent open-source program Fritzing. The program allows designers to document their prototypes and then go on to create PCB layouts for manufacture. It is an excellent program and a brilliant way of demonstrating a breadboard circuit to others. Pop on over to `http://fritzing.org` and check it out.

After you have made your circuit, I supply a code listing to type into the Arduino's program editor (the IDE) which can then be uploaded to your Arduino to make the project work. You will very quickly have a fully working project. It is only after you have made your project and seen it working that I explain how it works. The hardware will be explained to you in such a way that you know how the components work and how to connect them to the Arduino correctly.

The code will then be explained to you step by step so you understand exactly what each section of the code does. By dissecting the circuit and the code, you will understand how the whole project works and can then apply the skills and knowledge to later projects and then to your own projects in the future.

The style of teaching in this book is very easy to follow. Even if you have absolutely no experience of either programming or electronics, you will be able to follow along easily and understand the concepts as you go. More importantly, you will have fun. The Arduino is a great and fun open source product. With the help of this book, you'll discover just how easy it is to get involved in physical computing to make your own devices that interact with their environment.

—Mike McRoberts

Downloading the Code

The code for the examples shown in this book is available on the Apress web site, www.apress.com. A link can be found on the book's information page under the Source Code/Downloads tab. This tab is located underneath the Related Titles section of the page.

Contacting the Author

Should you have any questions or comments—or even spot a mistake you think I should know about—you can contact me at mike@earthshineelectronics.com, on Twitter as @TheArduinoGuy or on Google+ as Mike McRoberts. I am available, upon request (when I am free) to run Arduino Workshops or demonstrations at Hackerspaces and other organisations.

CHAPTER 1

Getting Started

Since the Arduino Project started in 2005, over 500,000 boards have been sold worldwide to date. The number of unofficial clone boards sold no doubt outweighs the number of official boards, and it's likely that over a million Arduino boards or its variants are out in the wild. Its popularity is ever increasing as more and more people realize the amazing potential of this incredible open source project and its ability to create cool projects quickly and easily with a relatively shallow learning curve.

The biggest advantage of the Arduino over other microcontroller development platforms is the ease of use in which non-"techie" people can pick up the basics and create their own projects in a relatively short amount of time. Artists in particular seem to find it the ideal way to create interactive works of art quickly and without specialist knowledge of electronics. There is a huge community of people using Arduinos and sharing their code and circuit diagrams for others to copy and modify. Most of this community is also very willing to help others and to provide guidance and the Arduino Forum is the place to go if you want answers quickly.

However, despite the huge amount of information available on the Internet for beginners, most of this information is spread across various sources, making it tricky for beginners to obtain the information they want. This is where this book fits in. Within the pages you are about to read are 50 projects that are all designed to take you step by step through the world of electronics and programming your Arduino in an easy to follow manner. I believe that the best way to learn anything is to jump in and just do it. That is why this book will not bore you with pages and pages of theory before you start to use your Arduino. I know what it is like when you first get an Arduino, or any new gadget: you want to plug it in, connect an LED, and get it flashing right away, not read through pages of manuals first. This author understands that excitement to get going and that is why we will dive right into connecting things to our Arduino, uploading code, and getting started right away. This is, I believe, the best way to learn a subject and especially a subject like physical computing, which is what the Arduino is all about.

How to Use This Book

The book starts with an introduction to the Arduino, how to set up the hardware, install the software, upload your first sketch, and ensure that your Arduino and the software are working correctly. We then explain the Arduino IDE (integrated development environment) and how to use it before we dive right into some projects, progressing from very basic stuff through to advanced topics. Each project will start with a description of how to set up the hardware and what code is needed to get it working. We will then separately describe the code and the hardware and explain in some detail how it works. Everything will be explained in clear and easy-to-follow steps. The book contains a lot of diagrams and photographs to make it as easy as possible to check that you are following along with the project correctly.

You will come across some terms and concepts in the book that you may not understand at first. Don't worry; these will become clear as you work your way through the projects.

What You Will Need

In order to follow along with the projects in this book, you will need various components. To carry out all of the projects will require purchasing a lot of parts first. This could be expensive, so I suggest that you start by purchasing the components for the projects in the first few chapters and obtain the parts listed at the start of the project pages. As you progress through the book, you can obtain the parts needed for subsequent projects.

There are a handful of other items you will need or may find useful. Of course, you will need to obtain an Arduino board or one of the many clone boards on the market such as the Freeduino, Seeeduino (yes, there really are three *eee*s), Boarduino, Sanguino, Roboduino or any of the other "duino" variants. These are all fully compatible with the Arduino IDE, Arduino Shields and everything else that you can use with an official Arduino Board. Remember that the Arduino is an Open Source project; therefore anyone is free to make a clone or other variant of the Arduino. However, if you wish to support the development team of the original Arduino board, get an official board from one of the recognized distributors. For the projects in this book, we will be using an Arduino Uno, although any of the available Arduino boards will work just as well.

You will need access to the Internet to download the Arduino IDE, the software used to write your Arduino code and upload it to the board, and also to download the Code Samples within this book (if you don't want to type them out yourself), as well as any code libraries that may be necessary to get your project working.

You will also need a well-lit table or other flat surface to lay out your components; this will need to be next to your desktop or laptop PC to enable you to upload the code to the Arduino. Remember that you are working with electricity (although it is low voltage DC), and therefore metal tables or surfaces will need to be covered in a non-conductive material such as a tablecloth or paper before laying out your materials. Also of some benefit, although not essential, may be a pair of wire cutters, a pair of long-nosed pliers, and a wire stripper. A notepad and pen will also come in handy for drawing out rough schematics and working out concepts and designs.

Finally, the most important thing you will need is enthusiasm and a willingness to learn. The Arduino is designed as a simple and cheap way to get involved in microcontroller electronics and nothing is too hard to learn if you are willing to give it a go. This book will help you on that journey, and introduce you to this exciting and creative hobby.

What Exactly Is an Arduino?

Wikipedia states "**Arduino** is a single-board microcontroller designed to make the process of using electronics in multidisciplinary projects more accessible. The hardware consists of a simple open-source hardware board designed around an 8-bit Atmel AVRmicrocontroller, though a new model has been designed around a 32-bit Atmel ARM. The software consists of a standard programming language compiler and a boot loader that executes on the microcontroller."

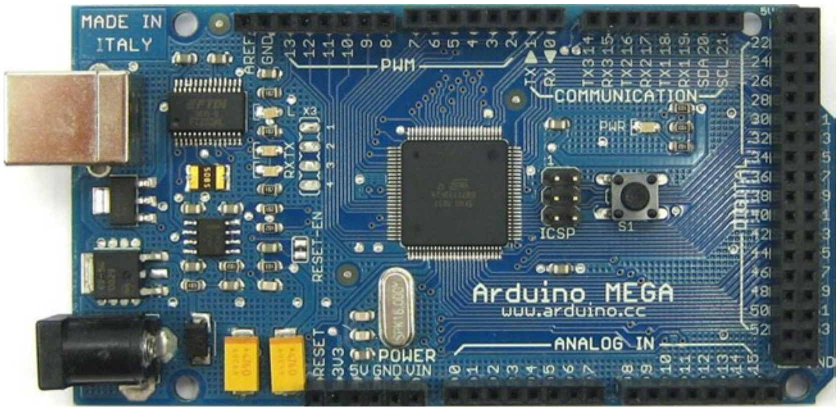


Figure 1-1. *An Arduino Mega (image by David Mellis)*

To put the above definition 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. The Arduino is what is known as a physical or embedded computing platform. For example, a simple use of an Arduino would be to turn a light on for a set period of time, let's say for 30 seconds, after a button has been pressed. In this example, the Arduino would have a lamp connected to it as well as a button. The Arduino would sit patiently waiting for the button to be pressed. When you press the button, 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 continue to wait for another button press. You could use this setup to control a lamp in a cupboard, for example.

You could extend this concept so that the device detects when the cupboard door has been opened or some other event has occurred, and automatically turns the lamp on, turning it off after a set period. You could go even further and connect a passive infrared (PIR) sensor to detect movement and 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. For example, it could be used to send a set of data received from sensors to a website to 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 or WiFi modules, or just about anything that outputs data or can be controlled. A look around the Internet will bring up a wealth of projects in which an Arduino has been used to read data from or control an amazing array of devices.

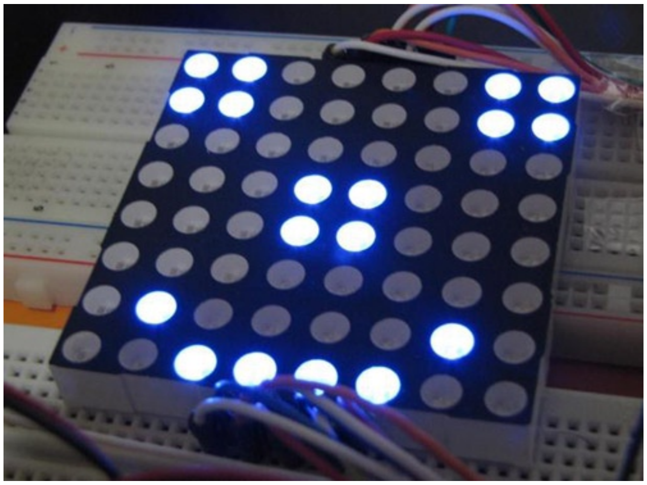


Figure 1-2. *A dot-matrix display controlled by an Arduino (image courtesy of Bruno Soares)*

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 5V voltage regulator. (Some Arduinos may use a switching regulator, and some, like the Due, are not 5 volt). Depending on what type of Arduino you have, it may also have a USB socket to enable it to be connected to a PC or Mac to upload or retrieve data. The board exposes the microcontroller's I/O (input/output) pins to enable you to connect those pins to other circuits or to sensors, etc.

To program the Arduino (make it do what you want it to), you also use the Arduino IDE, which is a piece of free software that enables you to program in the language that the Arduino understands. In the case of the Arduino, the language is based on C/C++ and can even be extended through C++ libraries. The IDE enables 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 those 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., are all open for anyone to take freely and do with what they like. 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 makes to this is that you cannot use the word *Arduino*, as this is reserved for the official board. Hence, the clone boards all have names such as Freeduino, Roboduino, etc.

The Arduino can also be extended with the use of “shields,” which are circuit boards containing other devices (for example, 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 (the places on your Arduino where you can output or input data) to the top of their 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, as you can make the exact same circuitry using a breadboard, some Stripboard or Veroboard (boards made up of strips of copper in a grid for home-soldered projects), or by making your own PCB. Most of the projects in this book are made using circuits on a breadboard.

As the designs are open source, a clone board, such as the Freeduino, can be 100 percent compatible with the Arduino and therefore any software, hardware, shields, etc. Some clones are compatible in most respects but may have intentional differences to support special features. Also, the Due (which is genuine Arduino) does have some issues such as its 3 volt operation, which may not work with all shields.

There are many different variants of the Arduino available. The most common one is Uno, released in 2010 (currently on Revision 3) and this is the board you will most likely see being used in the vast majority of Arduino projects across the Internet. You can also get the Due Leonardo, Duemilanove, Mega 2560, Mega ADK, Fio, Arduino Ethernet, Mini, Nano, Lilypad, and Bluetooth Arduinos. The latest additions to the product line are the Arduino Leonardo and the Arduino Due, which is the Arduino team’s first incursion into using ARM processors instead of AVR architecture processors The Due has a 32-bit processor instead of the usual 8-bit processor in the other Arduino variants, runs at 84MHz, and has 512KB of flash memory.

Probably the most versatile Arduino, and hence the reason it is so popular, is the Uno (prior to the Uno, the Duemilanove was the most popular). This is because they use a standard 28 pin chip attached to an IC (integrated circuit) socket. The beauty of this system is that if you make something neat with an Arduino and then want to turn it into something permanent, instead of using a relatively expensive Arduino board, you can simply use the Arduino to develop your device and program the chip, then pop the chip out of the board and place it into your own circuit board in your custom device. You would then 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. 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, and most of the Arduino parts suppliers will provide these.

The newer Arduino Uno has the advantage over the previous Arduino, the Duemilanove, in that it has a programmable USB chip on board which enables you to flash the chip in such a way that when you plug the device into your PC it will show up as any USB device you like, such as a keyboard, mouse, or joystick. This enables you to use the Arduino as an interface for creating your own USB devices. This is, however, an advanced feature and not for the faint hearted.

If you do a search on the Internet for *Arduino*, you will be amazed at the huge amount of websites dedicated to the Arduino or in which someone has used an Arduino to create a cool project. 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 about 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.

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

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

Setting Up Your Arduino

This section will explain how to set up your Arduino and the IDE for the first time. The instructions for both Windows and Macs are given. If you use Linux, then refer to the Getting Started instructions on the Arduino website at `http://playground.arduino.cc/learning/linux`. I will also presume you are using an Arduino Uno (see Figure 1-4), Duemilanove, Nano, Diecimila or Mega 2560 (or their equivalent clone) and are installing on either Windows 7 or a recent version of OSX (Lion or Mountain Lion). If you have a different type of board, then refer to the corresponding page in the Getting Started guide of the Arduino website.

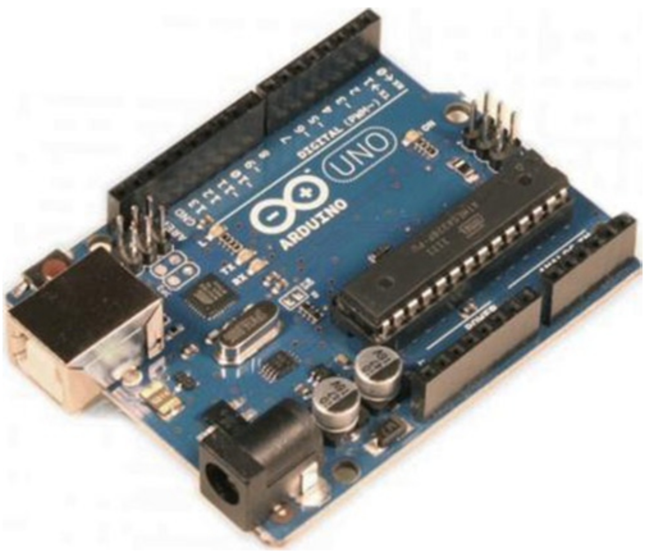


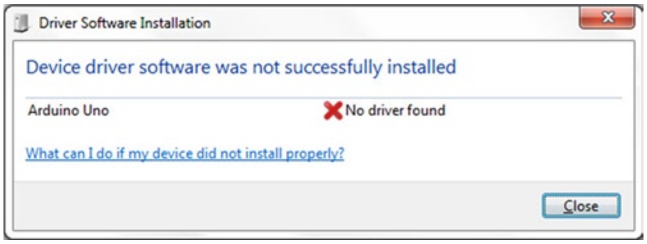
***Figure 1-4.** An Arduino Uno (Image courtesy of Earthshine Electronics)*

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.

Next, you need to 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 operating system.

If you have a Mac, once the Zip file has downloaded, unzip it and then you will see the Arduino icon. Drag it across to the Applications folder and drop it in there to install the program. You simply double-click the icon to start it. For Windows, download the ZIP file and, once complete, unzip it. Then put the unzipped folder in a place that suits you, keeping the directory structure in place.

Now you need to connect your Arduino board before installing the drivers and software. Connect the USB cable to the Arduino and plug the other end into a USB socket on your computer. You will see the green Power LED (marked PWR) light up on your board to show you it has power. If you are on a Mac, then there are no drivers to install. If you are on Windows, then it will now attempt to install the drivers for the Arduino. This auto attempt will fail and you will get a message that the "Device driver software was not successfully installed" (Figure 1-5); do not worry about this.


Driver Software Installation

Device driver software was not successfully installed

Arduino Uno ❌ No driver found

What can I do if my device did not install properly?

Figure 1-5. *The automatic attempt by Windows to install the drivers will fail. This is normal*

Click on the Start Menu and then click on Control Panel. Navigate to System and Security, click on System, and then open the Device Manager. On the list of hardware underneath Other Devices, you should see something similar to Figure 1-6 in which you have "Arduino Uno" with a yellow hazard icon over it.

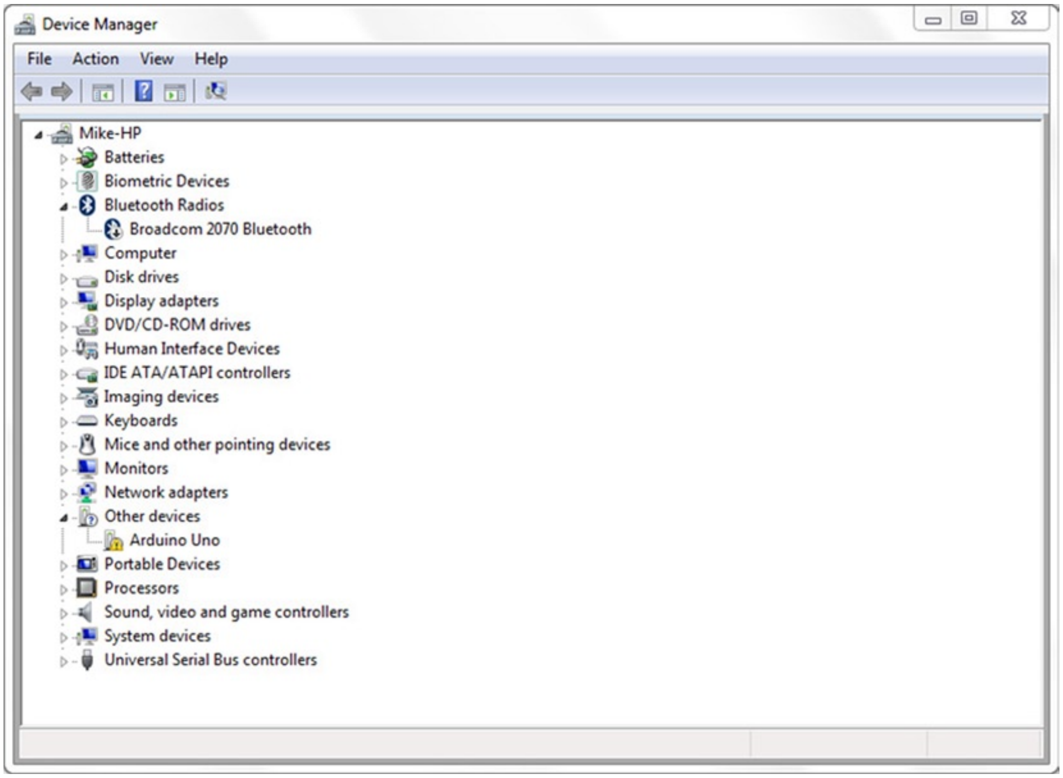
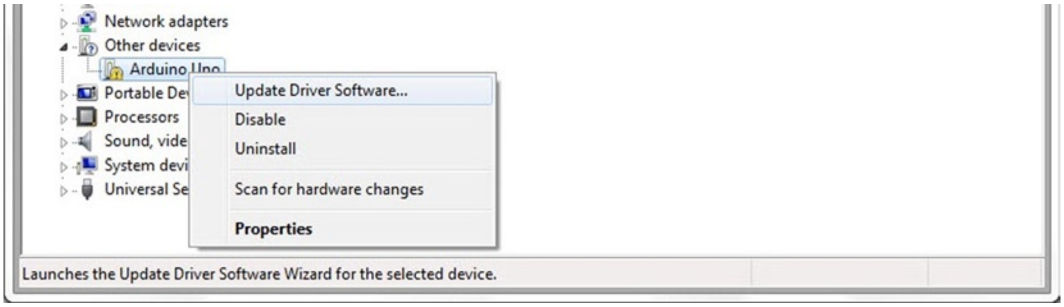


Figure 1-6. *The Windows Device Manager*

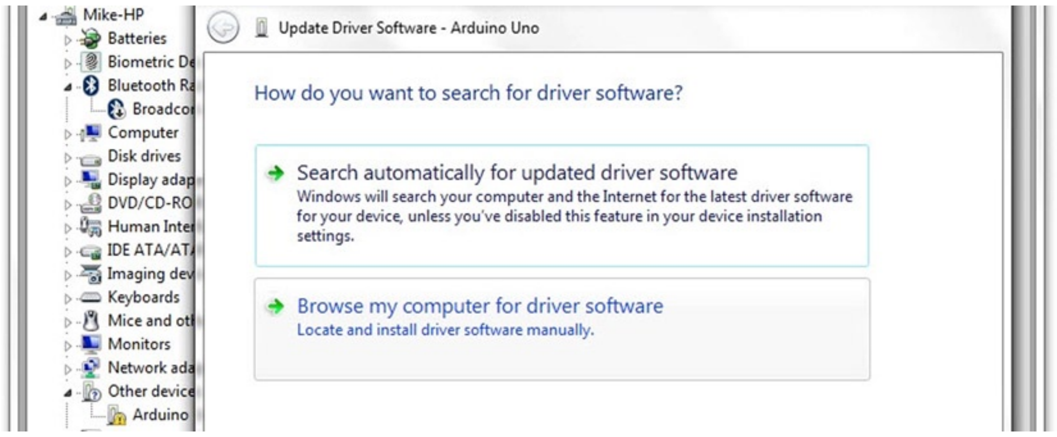
Right click on the Arduino Uno icon in the list and choose “Update Driver Software” (Figure 1-7).



Launches the Update Driver Software Wizard for the selected device.

Figure 1-7. *Right click and choose ”Update Driver Software”*

Now choose “Browse my computer for driver software”.



How do you want to search for driver software?

Search automatically for updated driver software
Windows will search your computer and the Internet for the latest driver software for your device, unless you've disabled this feature in your device installation settings.

Browse my computer for driver software
Locate and install driver software manually.

Figure 1-8. *Click on “Browse my computer for driver software”*

Next, browse to the driver folder of the Arduino installation, and then click the Next button. Windows will now finish the driver installation. If you get a message that says “Windows can’t verify the publisher of this driver software” then click the “Install this driver software anyway.” If you have a Mac, then there are no drivers to install.

Now that the drivers are installed, you are ready to open up the Arduino IDE. For Windows, double-click the **arduino.exe** file inside the unzipped Arduino folder. For a Mac, click the Arduino icon in the Applications folder. The IDE will now open up and present you with a blank sketch as in Figure 1-9.

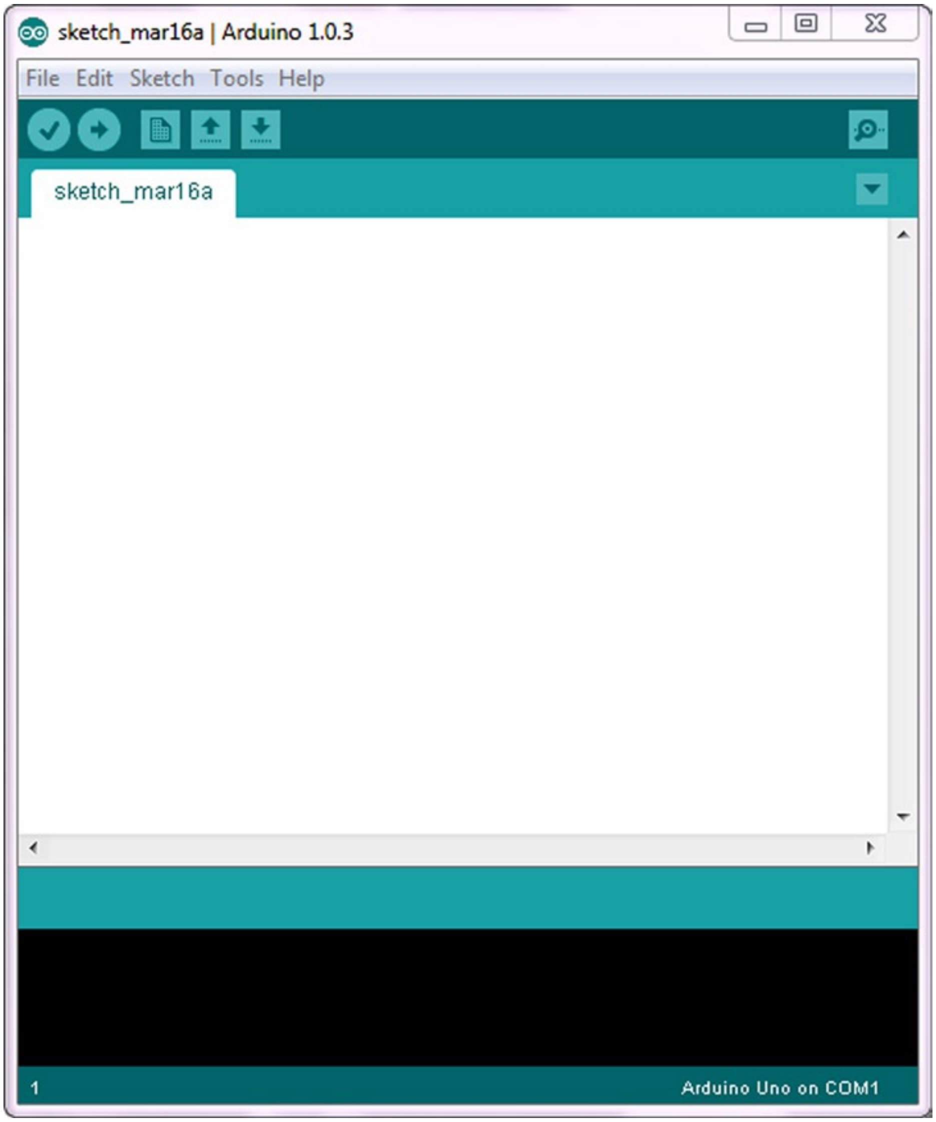


Figure 1-9. *The Arduino IDE*

Next, open up an example sketch to test out the IDE and the Arduino. Click File, then Examples, then 01.Basics, and finally, Blink (see Figure 1-10).

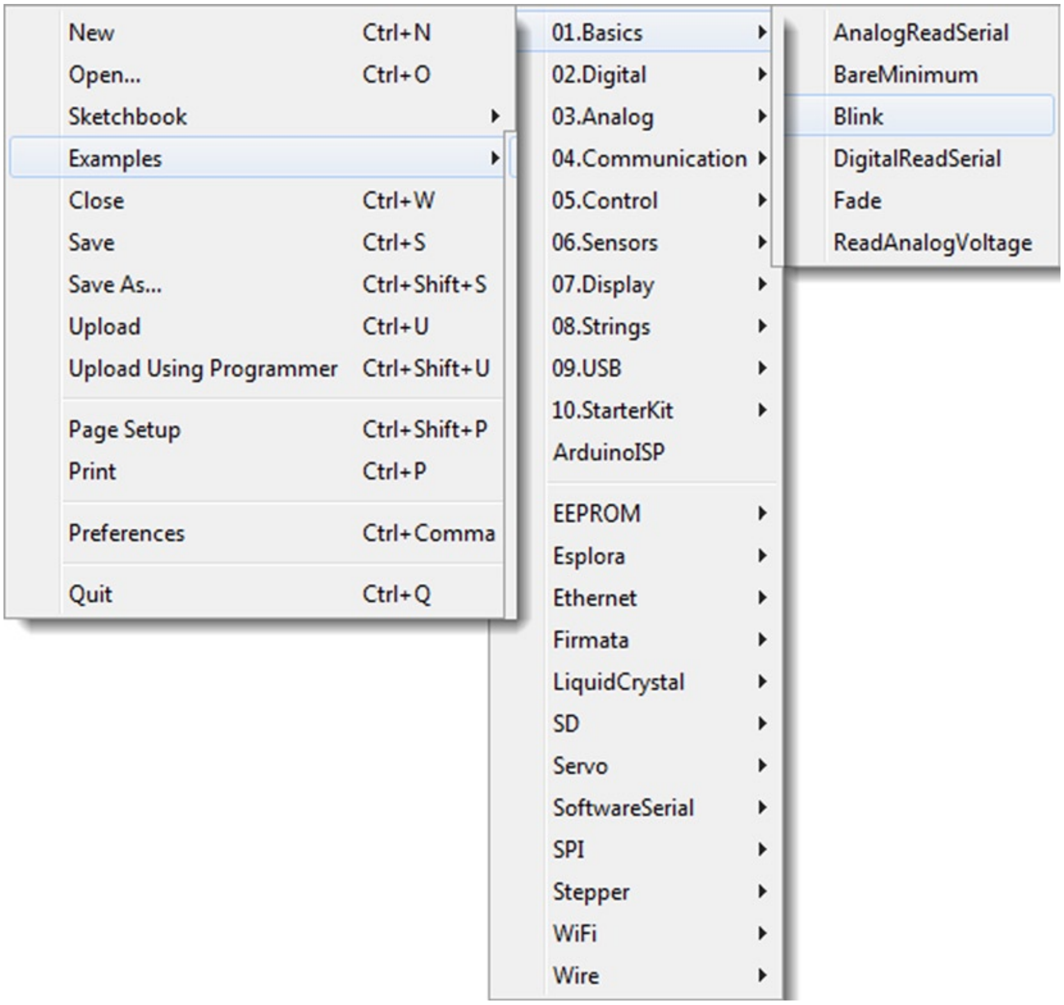
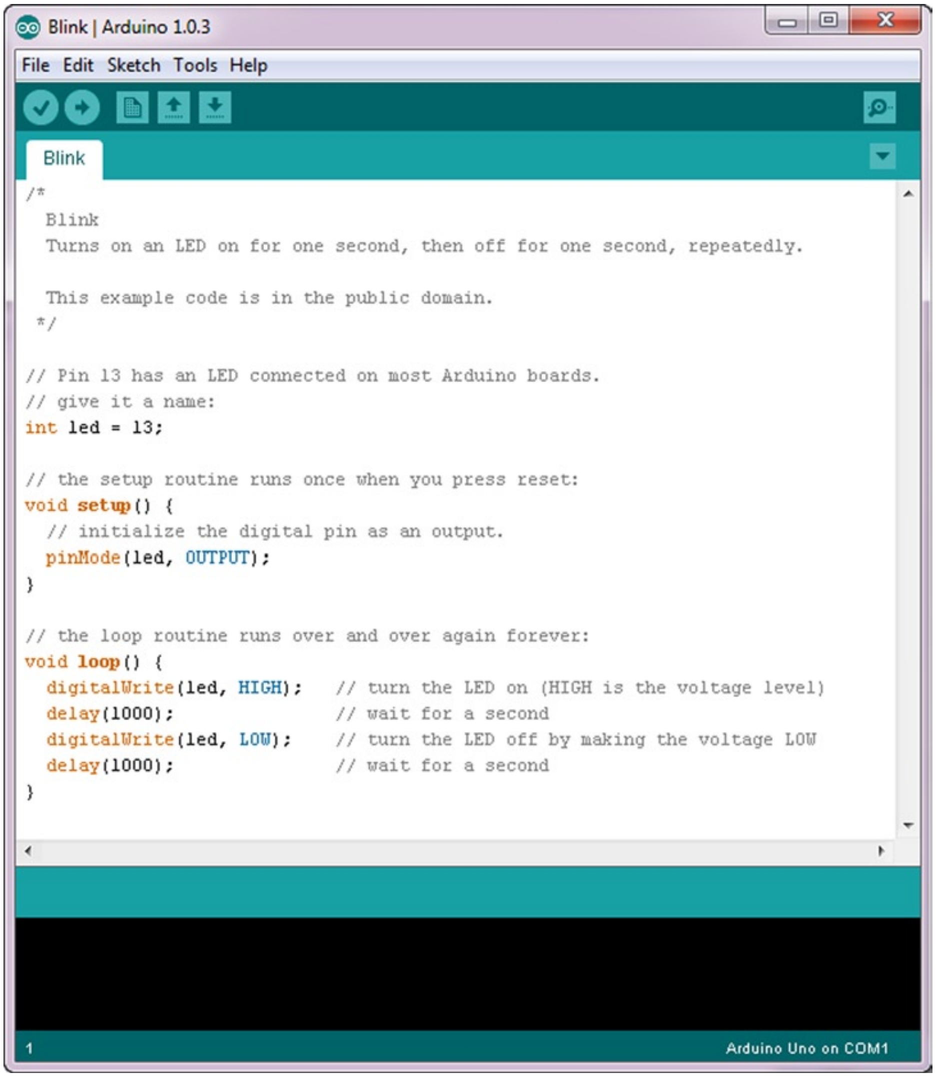


Figure 1-10. *The Arduino File Menu. Choose the Blink sketch*

This will load the Blink example sketch into the IDE and will look something like Figure 1-11.



```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  This example code is in the public domain.
 */

// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

Figure 1-11. *The IDE with the Blink sketch loaded*

Next, you will need to select your board from the list (see Figure 1-12) in **Tools ➤ Board**. For an **Arduino Uno**, select this from the top of the list. If you have an older Arduino Duemilanove or clone with an Atmega328 chip, you will need to select **Arduino Duemilanove or Nano w/ Atmega328.** If you have an even older board with an Atmega168 chip, select **Arduino Diecimila, Duemilanove, or Nano w/ ATmega168**, or you may even have a **Leonardo**, **Mega** or a **DUE**. Choose whichever board matches yours.

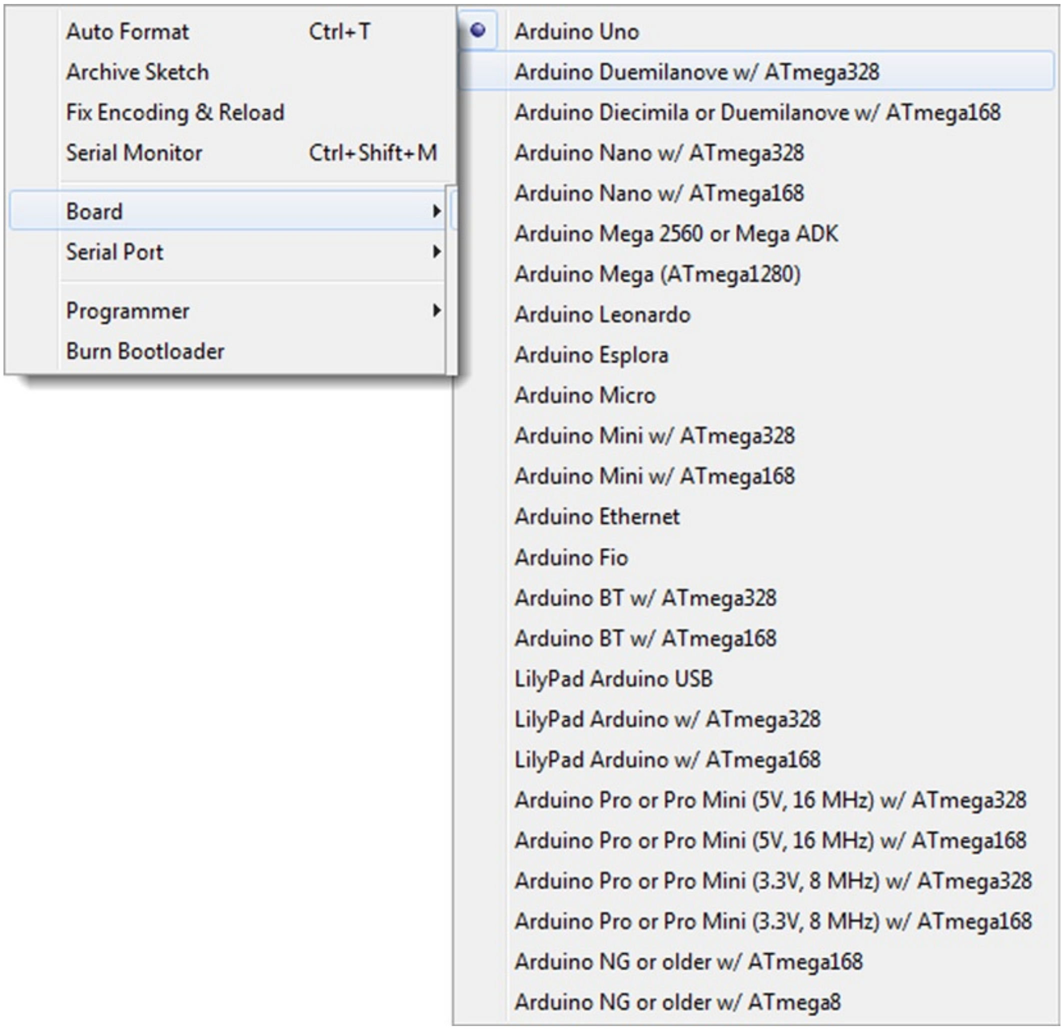

Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Programmer
Burn Bootloader
● Arduino Uno
Arduino Duemilanove w/ ATmega328
Arduino Diecimila or Duemilanove w/ ATmega168
Arduino Nano w/ ATmega328
Arduino Nano w/ ATmega168
Arduino Mega 2560 or Mega ADK
Arduino Mega (ATmega1280)
Arduino Leonardo
Arduino Esplora
Arduino Micro
Arduino Mini w/ ATmega328
Arduino Mini w/ ATmega168
Arduino Ethernet
Arduino Fio
Arduino BT w/ ATmega328
Arduino BT w/ ATmega168
LilyPad Arduino USB
LilyPad Arduino w/ ATmega328
LilyPad Arduino w/ ATmega168
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega328
Arduino Pro or Pro Mini (5V, 16 MHz) w/ ATmega168
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega328
Arduino Pro or Pro Mini (3.3V, 8 MHz) w/ ATmega168
Arduino NG or older w/ ATmega168
Arduino NG or older w/ ATmega8

Figure 1-12. *Select your board type*

Select the serial device of the Arduino board from **Tools ➤ Serial Port** (see Figure 1-13). If you are not sure what your port is, disconnect the Arduino and check the ports available, then reconnect the Arduino and see which port has now appeared (you may need to close and reopen the menu to get it to show).

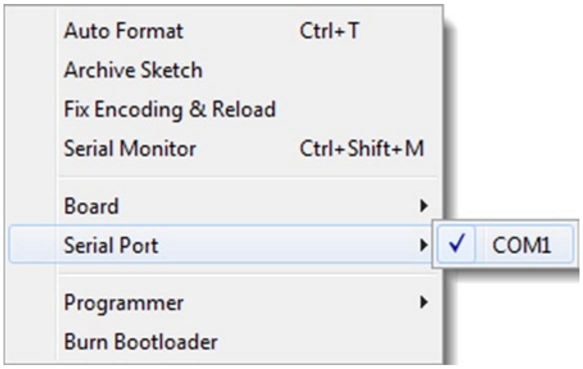


Figure 1-13. *Select the port*

Upload Your First Sketch

Now that you have installed the drivers and the IDE and have the correct board and ports selected, you can upload an example Blink sketch to the Arduino to test everything is working properly before moving on to the first project. Once you have loaded the Blink sketch into the Arduino IDE, you can upload it to the Arduino by simply clicking the Upload button (the second button from the left that is a right-facing arrow) 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 IDE will say "Compiling sketch . . .", which will then change to "Uploading . . . ." Next, 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.

After a few seconds, you should see the Pin 13 LED (the tiny LED above the TX and RX LEDs) start to flash on and off at one second intervals. If it does, then 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 the LED 13, which is a tiny orange LED soldered to the board and also connected to digital pin 13 from the microcontroller (see Figure 1-14).

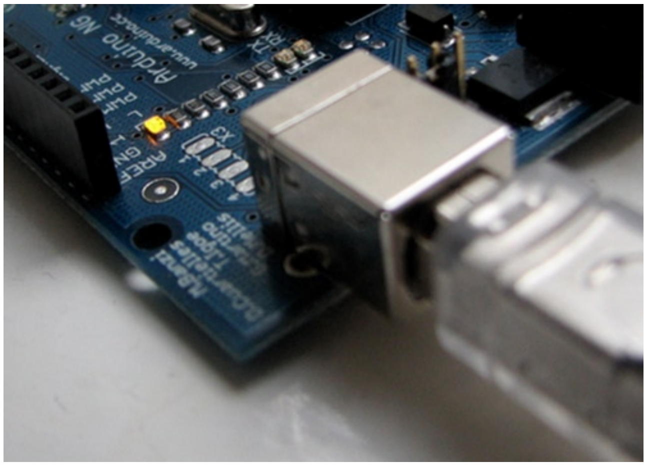


Figure 1-14. *LED 13 blinking*

Before we move onto Project 1, let's take a look at the Arduino IDE and I'll explain what each of the parts of the program do.

The Arduino IDE

The Arduino IDE (Integrated Development Environment) is what you will use to write the code for your Arduino, verify it, and upload it to your board. The current IDE version 1.x was released in November 2011. Previously, the Beta version numbers ran from 0001 to 0023 and version 1.0 was the first release candidate of the software. In version 1.0, the file extensions for the sketches changed from .pde to .ino to avoid conflicts with the Processing software (Processing is a project that the original IDE was based on). There were also some major changes to the Arduino language. If you want to port older Arduino code to the new IDE, you should read up on the Arduino website in the reference section about how the commands work if you get any errors with the older code.

When you open up the Arduino IDE, it will look very similar to the Windows version in the image below (Figure 1-15). If you are using OSX or Linux, there may be some slight differences, but the IDE is pretty much the same no matter what OS you use.

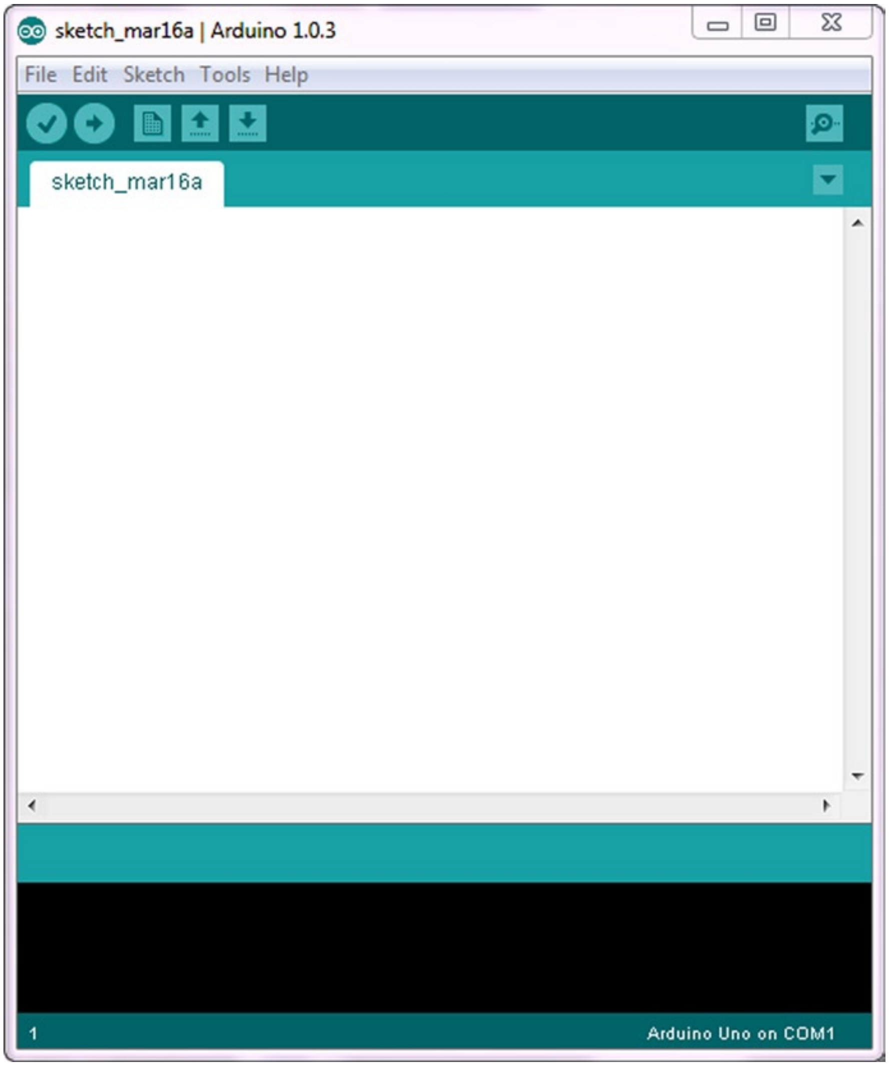


Figure 1-15. *What the IDE looks like when the application opens*

The IDE is split into four parts: the File Menu across the top of the program (or at the top of your screen in OSX), the Toolbar below this, the code or Sketch Window in the center, and the message window in the bottom. The Toolbar consists of six buttons, and underneath the Toolbar is a tab, or set of tabs, with the filename of the sketch within the tab. There is also one further button on the far right hand side which brings up the Serial Monitor window.

Along the top is the file menu with drop down menus headed under **Arduino**, **File**, **Edit**, **Sketch**, **Tools**, and **Help**. The buttons in the Toolbar (see Figure 1-16) provide convenient access to the most commonly used functions within this file menu.

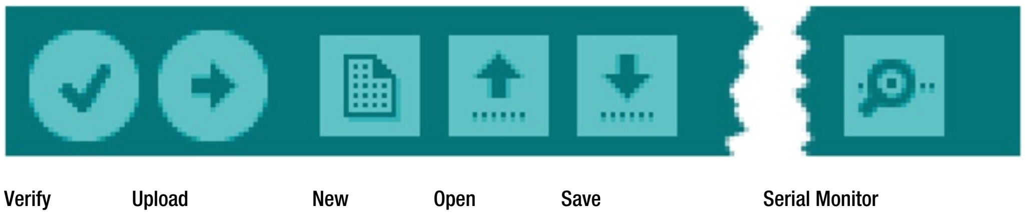


Figure 1-16. *The Toolbar*

The Toolbar buttons are listed in Figure 1-16. The functions of each of the buttons are as follows:-

Table 1-1. *The Toolbar Button Functions*

Verify	Checks the code for errors
Upload	Uploads the current sketch to the Arduino
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
Serial Monitor	Displays serial data being sent from the Arduino

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

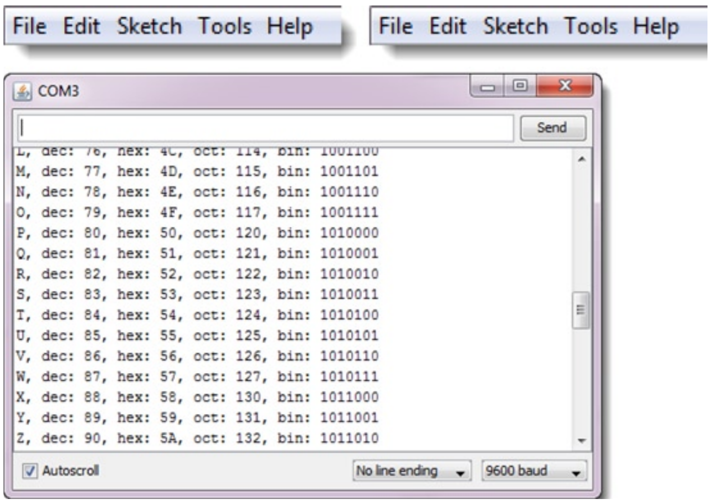
The **Upload** button will upload the code within the current sketch window to your Arduino. You need to make sure that you have the correct board and port selected (in the Tools menu) before uploading. It is essential that you save your sketch before you upload it to your board in case a strange error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

The **New** button will create a completely new and blank sketch ready for you to enter your code into. The IDE will ask you to enter a name and a location for your sketch (try to use the default location if possible) and will then give you a blank sketch ready to be coded. The tab at the top of the sketch will now contain the name you have given to your new sketch.

The **Open** button will present 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 once connected. The example sketches are invaluable for beginners to use as a foundation for your own sketch. Open the appropriate sketch for the device you are connecting and then modify the code to your own needs.

The **Save** button will save 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 **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. If you click the Serial Monitor button you will be presented with an image like the one in Figure 1-17.



```
L, dec: 76, hex: 4C, oct: 114, bin: 1001100
M, dec: 77, hex: 4D, oct: 115, bin: 1001101
N, dec: 78, hex: 4E, oct: 116, bin: 1001110
O, dec: 79, hex: 4F, oct: 117, bin: 1001111
P, dec: 80, hex: 50, oct: 120, bin: 1010000
Q, dec: 81, hex: 51, oct: 121, bin: 1010001
R, dec: 82, hex: 52, oct: 122, bin: 1010010
S, dec: 83, hex: 53, oct: 123, bin: 1010011
T, dec: 84, hex: 54, oct: 124, bin: 1010100
U, dec: 85, hex: 55, oct: 125, bin: 1010101
V, dec: 86, hex: 56, oct: 126, bin: 1010110
W, dec: 87, hex: 57, oct: 127, bin: 1010111
X, dec: 88, hex: 58, oct: 130, bin: 1011000
Y, dec: 89, hex: 59, oct: 131, bin: 1011001
Z, dec: 90, hex: 5A, oct: 132, bin: 1011010
```

Autoscroll [✓] No line ending 9600 baud

Figure 1-17. *The Serial Monitor in use*

On the bottom right hand 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 – bits and bytes will be explained later on).

At the top is a blank text box for you to enter text to send back to the Arduino and a **Send** button to send the text within that field. Note that the Serial Monitor will not receive any serial data unless you have set up the code inside your sketch to send serial data from the Arduino. Similarly, the Arduino will not receive any data sent unless you have coded it to do so.

There is a tick box on the bottom left where you can choose if you want the data in the serial monitor window to autoscroll or not.

The box to the left of the baud rate menu will affect the data sent from the serial monitor back to the Arduino. The default setting is "no line ending," meaning when you enter data into the text box on the serial monitor and press "send," the data will be sent as is. If you click the drop down menu, there are three other options for Newline, Carriage return. and Both NL+ Cr. By selecting one of these, the serial monitor will append an ascii code for a Newline, Carriage Return, or both on the end of any data entered into the serial monitor window when you click send. Bear this in mind when processing data sent from the serial monitor back to the Arduino.

Finally, the main area is where your serial data will be displayed. In the image above, the Arduino is running the `ASCIITable` sketch that can be found in the Communications examples. This program outputs ASCII characters, from the Arduino via serial (the USB cable) to the PC where the Serial Monitor then displays them.

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.

We will make use of the Serial Monitor later on in our projects when we read data from sensors and get the Arduino to send that data to the Serial Monitor, in human readable form for us to see.

The Message Window at the bottom of the IDE is where you will see error messages that the IDE will display to you when trying to connect to your board, upload code, or verify code.

Below the Message Window at the bottom left you will see a number. This is the current line that the cursor, within the code window, is at. 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 1-18).

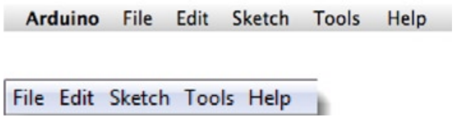
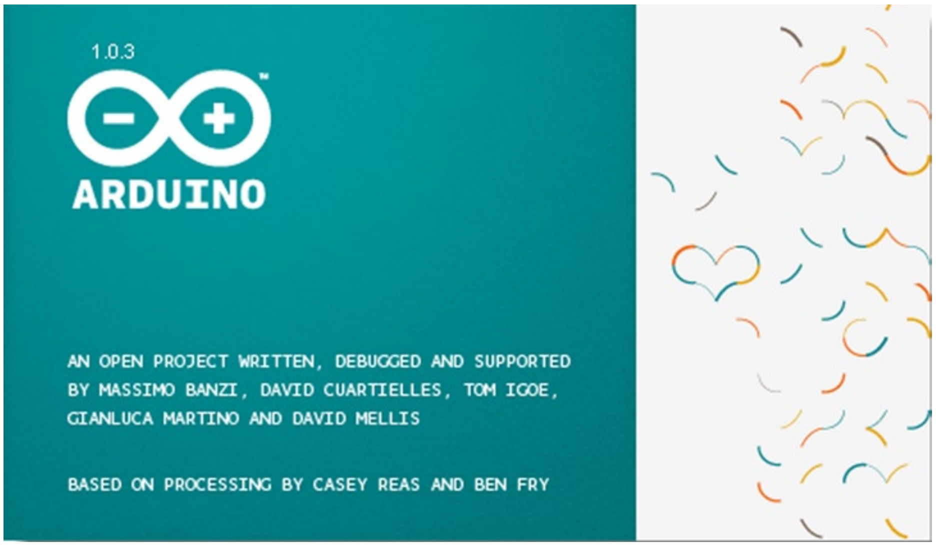

Arduino File Edit Sketch Tools Help
File Edit Sketch Tools Help

Figure 1-18. *The IDE menus (Top: OSX, Bottom: Windows)*

The first menu (on OSX) is the **Arduino menu** (see Figure 1-19). Within this is the **About Arduino** option, which when pressed will show you the current version number, a list of the people involved in making this amazing device, and some further information. On Windows PCs, the About Arduino item is on the Help menu.


1.0.3

ARDUINO

AN OPEN PROJECT WRITTEN, DEBUGGED AND SUPPORTED BY MASSIMO BANZI, DAVID CUARTIELLES, TOM IGOE, GIANLUCA MARTINO AND DAVID MELLIS

BASED ON PROCESSING BY CASEY REAS AND BEN FRY

Figure 1-19. *The About Arduino menu*

The next menu is the **File** menu. (see Figure 1-20). Here, you get access to options to create a new sketch, take a look at sketches stored in your Sketchbook, example files, options to save your Sketch (or Save As, if you want to give it a different name). You also have the option to upload your sketch to the Arduino, upload using a programmer (we will not be using this feature) as well as the print options for printing out your code.

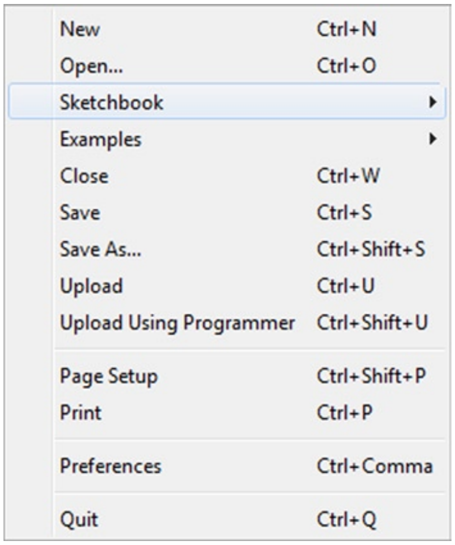


Figure 1-20. *The File Menu*

Near the bottom is the **Preferences** option. This will bring up the Preferences window where you can change various IDE options, such as where your default Sketchbook is stored, etc. Finally, there is the **Quit** option, which will quit the program.

Next is the **Edit** menu (see Figure 1-21) Here, you get options to enable you to cut, copy and paste sections of code. Select all of your code as well as find certain words or phrases within the code. Comment your code (adding comments to explain how it works), as well as increasing or decreasing indents. Also included are the useful Undo and Redo options, which come in handy when you make a mistake.

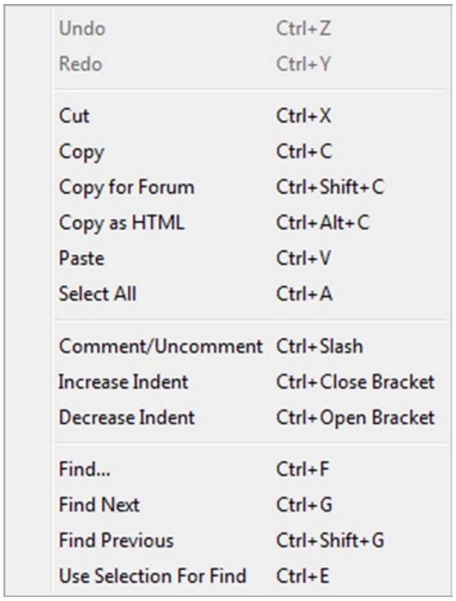

Undo Ctrl+Z
Redo Ctrl+Y
Cut Ctrl+X
Copy Ctrl+C
Copy for Forum Ctrl+Shift+C
Copy as HTML Ctrl+Alt+C
Paste Ctrl+V
Select All Ctrl+A
Comment/Uncomment Ctrl+Slash
Increase Indent Ctrl+Close Bracket
Decrease Indent Ctrl+Open Bracket
Find... Ctrl+F
Find Next Ctrl+G
Find Previous Ctrl+Shift+G
Use Selection For Find Ctrl+E

Figure 1-21. *The Edit Menu*

Our next menu is the **Sketch** menu (see Figure 1-22) which gives us access to the Verify/Compile functions and some other useful functions you will use later on. These include the Import Library option, which when clicked will bring up a list of the available libraries, stored within your libraries folder.

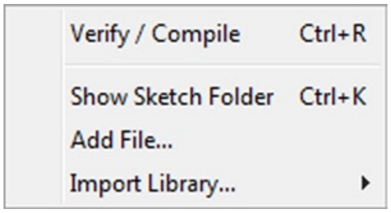


Figure 1-22. *The Sketch Menu*

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 reinventing the wheel by reusing code already made by someone else for various pieces of common hardware you may encounter whilst using the Arduino.

For example, one of the libraries you will find is Stepper, which is a set of functions you can use within your code to control a stepper motor. Somebody else has kindly already created all of the necessary functions necessary to control a stepper motor, and by including the Stepper library into our sketch, we can use those functions to control the motor as we wish. By storing commonly used code in a library, you can reuse that code over and over in different projects and also hide the complicated parts of the code from the user. We will go into greater detail concerning the use of libraries later on.

Finally, within the Sketch menu is the Show Sketch Folder option, which will open up the folder were your sketch is stored. Also, there is the Add File option, which will enable you to add another source file to your sketch. This functionality allows you to split larger sketches into smaller files and then add them to the main Sketch.

The next menu in the IDE is the **Tools** menu (see Figure 1-23). Within this are the options to select the board and serial port we are using, as we did when setting up the Arduino for the first time. Also we have the Auto Format function that formats your code to make it look nicer.

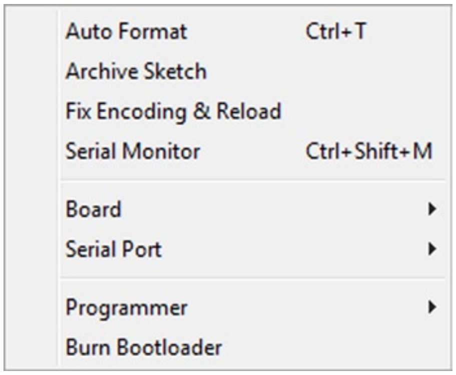


Figure 1-23. *Tools Menu*

The Copy for Forum option will copy 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 will enable you to compress your sketch into a ZIP file and will ask you where you want to store it. The Fix Encoding & Reload option is to convert code created in older versions of the IDE into the newer format.

The programmer button will enable you to choose a programmer, in case you are using an external device to upload code to your Arduino or wish to burn code to a chip in your own project. We will simply be using the USB cable we purchased with our Arduino.

Finally, the Burn Bootloader option can be used to burn the Arduino Bootloader (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 have replaced the chip in your Arduino or have bought blank chips to use in your own embedded project. Unless you plan on burning lots of chips, it is usually cheaper and easier to just buy an ATmega chip (see Figure 1-24) with the Arduino Bootloader already pre-programmed. Many online stores stock pre-programmed chips and these can be purchased pretty cheaply. The chip used in the Arduino Uno is an Atmel ATmega328.

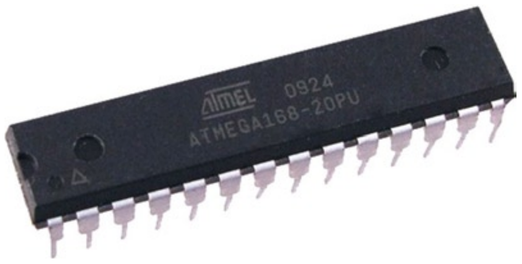


Figure 1-24. *An Atmel ATmega chip. The heart of your Arduino. (image courtesy of Earthshine Electronics)*

The final menu is the **Help** menu were you can find help menus for finding out 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 we work through the projects. As you become more proficient at using an Arduino and programming in C (the programming language we use to code on the Arduino) you may find the Arduino IDE is too basic and wish to use something with better functionality. Indeed, many expert Arduino programmers do not use the IDE at all and instead use professional IDE programs (some of which are free) such as Eclipse, ArduIDE, GNU/Emacs, AVR-GCC, AVR Studio, and even Apple's XCode.

So, 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.

Summary

In this chapter you have learnt what an Arduino is, a little bit about the different Arduino variants, what you can do with it, and what the basic components are that make up the Arduino board. Then you learnt how to install and set up the software and drivers for the Arduino, how to select the correct serial port, and upload a test sketch to your Arduino to make sure everything is working correctly.

Next we moved onto the IDE: how to use it and what the purpose of each of the buttons and menus is, including the serial monitor window. These are the basic concepts required to understand how to set up the software to work with the Arduino hardware. In the next chapter, we will put those concepts into practice by using the IDE to write our code and upload it to our Arduino board.

CHAPTER 2

Light 'Em Up

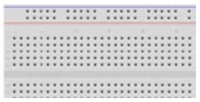
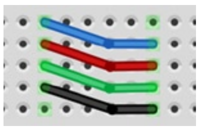
You are now going to work your way through the first four projects. These projects all use LED lights in various ways. You will learn about controlling outputs from the Arduino as well as simple inputs such as button presses. On the hardware side, you will learn about LEDs, buttons, and resistors, including 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 very first project, we are going to repeat the LED blink sketch that we used during our testing stage in Chapter 1. Except this time, we are going to connect an LED to one of the digital pins rather than use LED13 which is soldered to the board. We will also learn exactly how the hardware and the software for this project works as we go, learning a bit about electronics and coding in the Arduino language (which is a variant of C) at the same time.

Table 2-1 below shows the parts required for our very first project.

Table 2-1. *Parts Required for Project 1*

Breadboard	
5mm LED	
100Ω Resistor*	
Jumper Wires	

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

Parts Required

The best kind of breadboard to get for most of the projects in this book is an 840 tie-point breadboard. These are fairly standard-sized breadboards that usually measure approximately 16.5cm by 5.5cm and have 840 holes (or tie points) on the board. Usually, these have little dovetails on the side allowing you to connect several of them together to make larger breadboards. This is useful for larger 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 as this will enable you to calculate the resistor value needed. We will work out this value later in the project.

The jumper wires 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.

Connect It Up

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

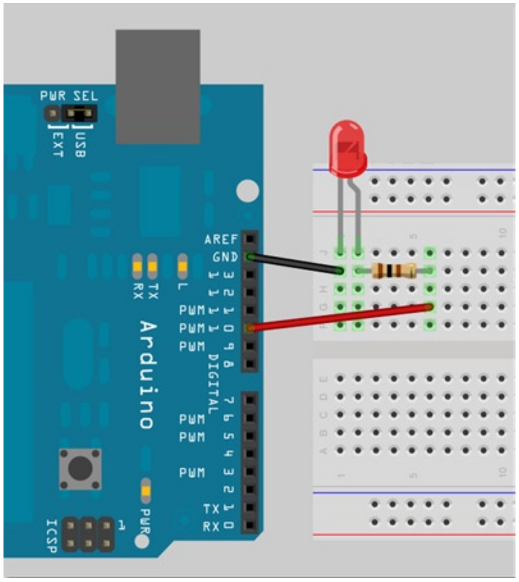


Figure 2-1. *The circuit for Project 1 — LED Flasher*

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 the picture. Be careful when inserting components into the breadboard. Your breadboard may be brand new and the grips in the holes will be stiff to begin with. Failure to insert components carefully could result in damage.

Make sure that your LED is connected the right way. Make sure the anode (positive) leg of the LED (usually the leg with the longer lead) is connected to digital pin 10. Make sure the anode of the LED (usually the lead with the longer leg) is connected to the resistor, and the cathode (usually the short leg) to ground. LEDs only light up when the anode is at a more positive voltage than the cathode. If you connect it backwards, it won't light, but won't damage the LED either in this circuit.

When you are sure that everything is connected up 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);
}
```

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

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

Project 1 — LED Flasher – Code Overview

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

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

This is simply a comment in your code and is ignored by the compiler (the part of the IDE that turns your code into instructions the Arduino can understand before uploading it). Everything following // (double slashes) on a line is ignored by the compiler. This allows you to add notes to yourself and others who may read the code explaining how the code works.

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

You can also put comments into a block by using the **/*** and ***/** delimiters, for example:

```
/* 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;
```

This is what is known as a variable. A variable is a place to store data. Imagine a variable as a small box where you can keep things. A variable is called a variable because you can change its contents. Later on, we will carry out mathematical calculations on variables to make our program do more advanced things. In this case, you are setting up a variable of type int or integer. An integer is a number within the range of -32,768 to 32,767. Next, you have assigned that integer the name of ledPin and have given it a value of 10. We didn't have to call it ledPin; we could have called it anything we wanted to. But, as we want our variable name to be descriptive, we call it ledPin to show that the use of this variable is to set which pin on the Arduino we are going to use to connect our LED. In this case, we are using digital pin 10. At the end of this statement is a semicolon. This is a symbol to tell the compiler that this statement is now complete.

Although we can call our variables anything we want, every variable name in C must start with a letter; the rest of the name can consist of letters, numbers, and underscore characters. C recognizes upper and lower case characters as being different. Finally, you cannot use any of C's keywords such as *main*, *while*, *switch*, etc. as variable names. Keywords are constants, variables, and function names that are defined as part of the Arduino language.

Don't use a variable name that is the same as a keyword. All keywords within the sketch will appear in red. So, you have set up an area in memory to store a number of type integer and have stored in that area the number 10.

Next we have our setup() function

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

An Arduino sketch must have a setup() and loop() function, otherwise it will not work. The setup() function 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 block of code assembled into one convenient block. For example, if we created our own function to carry out a whole series of complicated mathematics that had many lines of code, we could run that code as many times as we liked simply by calling the function name instead of writing out the code again each time. Later on, we will go into functions in more detail when we start to create our own. In the case of our program, the setup() function only has one statement to carry out. The function starts with

```
void setup()
```

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

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

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

All of the code within the function is contained within the curly braces. A { symbol starts the block of code and a } symbol ends the block. Anything in between those two symbols is code that belongs to the function. We will go into greater detail about functions later on in Project 4 in this chapter, so don't worry about them for now. All you need to know is that in this program, we have two functions, and the first function is called setup; its purpose is to setup anything necessary for our program to work before the main program loop runs.

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

Our setup function only has one statement and that is pinMode. Here we are telling the Arduino that we want to set the mode of one of our digital pins to be output mode, rather than input. Within the parenthesis, we put the pin number and the mode (OUTPUT or INPUT). Our pin number is ledPin, which has been previously set to the value 10 in our program. Therefore, this statement is simply telling the Arduino that the digital pin 10 is to be set to OUTPUT mode. As the setup() function runs only once, we now move onto the main function loop.

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

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

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

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

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

```
delay(1000);
```

This statement simply tells the Arduino to wait for 1,000 milliseconds (there are 1,000 milliseconds in a second) before carrying out the next statement which is

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

This will turn off the power going to digital pin 10, and therefore turn the LED off. There is then another delay statement for another 1,000 milliseconds and then the function ends. However, as this is our main loop() function, the function will now start again at the beginning.

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

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

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

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

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

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

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

Or, to make the LED flash on and off very fast

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

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

Project 1 — LED Flasher – Hardware Overview

The hardware used in Project 1 was:

Breadboard	
5mm LED	
100Ω Resistor*	
Jumper Wires	

** or whatever value you worked out that was appropriate for your LED*

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

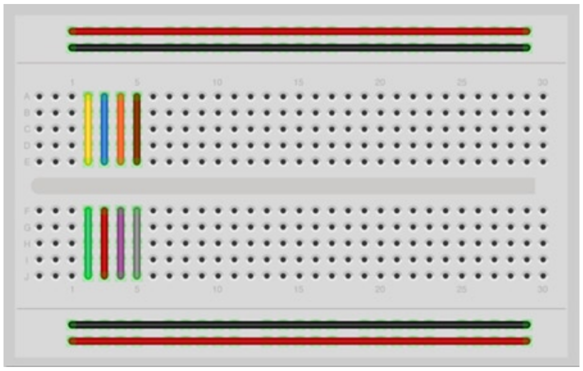


Figure 2-2. *How the metal strips in a breadboard are laid out*

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

The strips in the center run at 90 degrees to the power and ground rails in short lengths, and there is a gap in the middle to allow you to put integrated circuits across the gap and have each pin of the chip go to a different set of connected holes (see Figure 2-3).

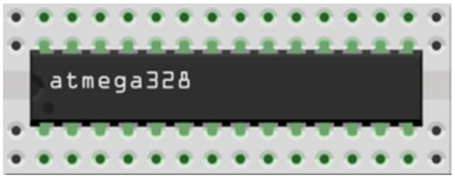


Figure 2-3. *An integrated circuit (or chip) plugged across the gap in a breadboard*

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

Resistance is measured in units called ohms. The symbol for ohms is the Greek omega symbol Ω. In this case, digital pin 10 is outputting 5 volts DC at 40mA (milliamps), according to the Atmega datasheet, and our LEDs require a voltage of 2v and a current of 35mA, according to their datasheet. We therefore need to put in a resistor that will reduce the 5V to 2V, and the current from 40mA to 35mA if we want to display the LED at its maximum brightness. If we want the LED to be dimmer, we 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 which resistor we 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. So, for our example LED, we have an LED with an LED voltage of 2 volts and a current of 35mA (milliamps) connected to a digital pin from an Arduino which gives out 5 volts; therefore, 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 ohms. 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 the resistor and/or the components connected with it may be damaged.

So, how do we find a 100Ω resistor? A resistor is too small to be written upon that could be readable by most people, so instead, resistors use a color code. Around the resistor, you will typically find four colored bands, and by using the color code in Table 2-2 you can find out the value of a resistor or which color codes a particular resistance will be.

Table 2-2. *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%

We need a 100Ω resistor, so if you look at the color table, you will see that we need 1 in the first band, which is brown, followed by a 0 in the next band, which is black, and we then need to multiply this by 10^1 (in other words add 1 zero) which is brown in the 3rd band. The final band indicates the tolerance of the resistor. If your resistor had a gold band, it would therefore have a tolerance of ±5%, which means the actual value of the resistor can vary between 95Ω and 105Ω. Therefore, if you had an LED that required two volts and 35mA, you would need a resistor with a brown, black, brown band combination.

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

If you needed a 10K (or 10 kilo-ohm) resistor (see Figure 2-4), you would need a brown, black, orange combination (1, 0, +3 zeros). If you needed a 570K resistor, the colors would be green, violet, and yellow, and so on.

In the same way, if you found a resistor and wanted to know which value it is, you would do the same in reverse. So, if you found this resistor and wanted to find out which value it was 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. Choose the correct resistance value for the LED you have purchased to complete this project.

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

An LED is a diode that also emits light. LEDs come in different colors and brightnesses, and can also emit light in the ultraviolet and infrared parts of the spectrum (as in the LEDs in your TV remote control).

If you look carefully at an LED, you will notice two things. One is that the legs are of different lengths, and also, that on one side of the LED it is flattened rather than cylindrical (see Figure 2-5). These are indicators to show you 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. However, LEDs also come

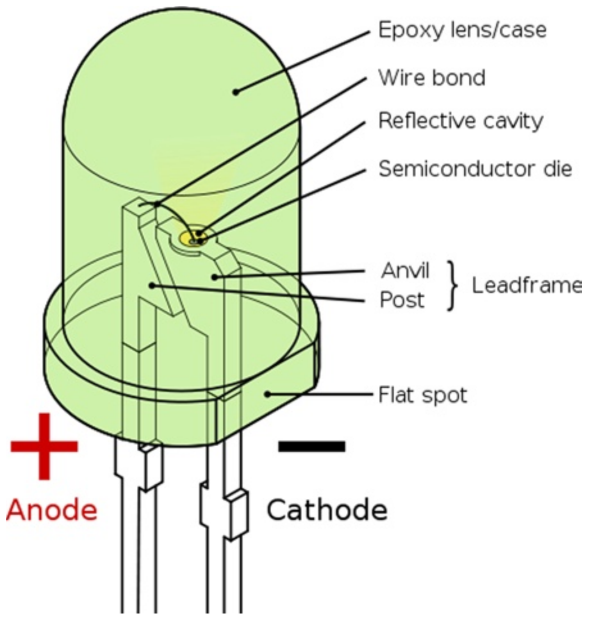


Figure 2-5. *The parts of an LED (image courtesy of Inductiveload from Wikimedia Commons)*

in rectangular shapes, surface mount style, or other forms. Always refer to the datasheet for your LED to check the correct anode/cathode orientation.

If you connect the LED the wrong way, it will not be damaged (unless you put very high currents through it). It is essential that you always put a resistor in series with the LED to ensure that the correct current gets to the LED. You can permanently damage the LED if you fail to do this. As well as single color LEDs, you can also obtain bicolor and tricolor LEDs. These will have several legs coming out of them with one common leg (that is, a common anode or common cathode).

An RGB LED has a red, green, and blue (hence RGB) LED in one package. The LED has four legs: one will be a common anode or cathode, common to all three LEDs and the other three will then go to the anode or cathode of the individual red, green and blue LEDs. 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 LED's.

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

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

For this project, we are going to leave the exact same circuit set up as in Project 1 (so no need for a hardware overview), but will use some different code to make the LED display a message in Morse code. In this case, we are going to get the LED to 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 our digital system as we can turn an LED on and off in the necessary pattern to spell out a word or a series of characters. In this case we will be signaling S.O.S., which in the Morse code alphabet is three dits (short flashes), followed by three dahs (long flashes), followed by three dits again.

We can therefore now code our sketch to flash the LED on and off in this pattern, signaling SOS.

Enter the code

Listing 2-2. Code for Project 2

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

// run once, when the sketch starts
void setup()
{
        // sets the digital 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 between 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 between 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 your code is error free and then upload it to your Arduino. If all goes well, you will now see the LED flash the Morse Code SOS signal, pause 5 seconds, then repeat.

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

So, let's take a look at this code and work out how it works.

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

Thus, the first part of the code is identical to the last project where we initialize a variable and then set pin 10 to be an output. In the main code loop, we can see the same kind of statements to turn the LEDs on and off for a set period of time, but this time, 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);
}
```

We can see that the LED is turned on for 150ms and then off for 100ms and we can see that those statements are within a set of curly braces and are therefore in a separate code block. A block is a set of one or more statements enclosed within curly braces. But, when we run the sketch, we 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 its code block execute three times. There are three expressions we can supply to the for loop. These are initialization, condition, increment. The **initialization** expression is evaluated first, and only once. Each time through the loop, the **condition** is tested; if it's true, the statement block is executed, and then the **increment** expression (if present) is executed. Then control goes back to the condition test and the process repeats until the condition is false when tested; the loop then ends.

So, first we need to initialize a variable to be the start number of the loop. In this case we set up variable *x* and set it to zero.

```
int x=0;
```

We 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 initialization expression is always executed, but if the condition is false going into the for loop, the loop body and increment expression are not executed. The < symbol is what is known as a comparison operator. These operators 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 our code, we are comparing *x* with the value of 3 to see if it is smaller than 3. If *x* is smaller than 3, then the code in the block will be executed. Otherwise, the loop will exit.

The final statement is

```
x++
```

This is a statement to increase the value of *x* by 1. We 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 semicolon after this final Page: 1expression in the for loop statement.

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

1 + 1 = 2

3 – 2 = 1

2 * 4 = 8

8 / 2 = 4

So, our for loop initializes the value of *x* to 0, then checks that the condition is met, which is that *x* is smaller than 3; if it is, it runs the code within the body of the for loop. It then increments *x* by the value of the increment expression. As long as the condition of the loop is met, the loop will keep on repeating.

So, now we know how the for loop works, we can see in our code that there are three for loops, one that loops three times and displays the dits. The next one repeats three times and displays the dahs. 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. A variable defined within a function, for(), or block is accessible within that function, for(), or block. Variables defined outside of any function are globally visible within the remainder of the file in which they are defined. They are not visible to separately compiled parts of the program unless declared there with the extern command. If you try to access *x* outside the for loop, you will get an error.

In between each for loop there is a small delay to make a tiny visible pause between the letters of S.O.S. Finally, the code waits for five seconds before the main program loop starts again.

Now let's move onto using multiple LEDs.

Project 3 – Traffic Lights

We are now going to create a set of traffic lights based on the system that will change from green to red, via amber, and back again, after a set length of time using the four-state system. We will use the parts listed in Table 2-3. This project could be used on a model railway to make a set of working traffic lights or for a child's toy town.

Table 2-3. *Parts Required for Project 3*

Breadboard	
Red Diffused LED	
Yellow Diffused LED	
Green Diffused LED	
3 x 150Ω Resistor*	
Jumper Wires	

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

Parts Required

In the parts list of Table 2-3, you will find all you need for this project. We will be using three LEDs and therefore, three current-limiting resistors.

Connect It Up

Connect your circuit as in Figure 2-6. This time, we have connected three LEDs with the anode of each one going to digital pins 8, 9 and 10, via a 150Ω current-limiting resistor (or whichever value you require) for each.

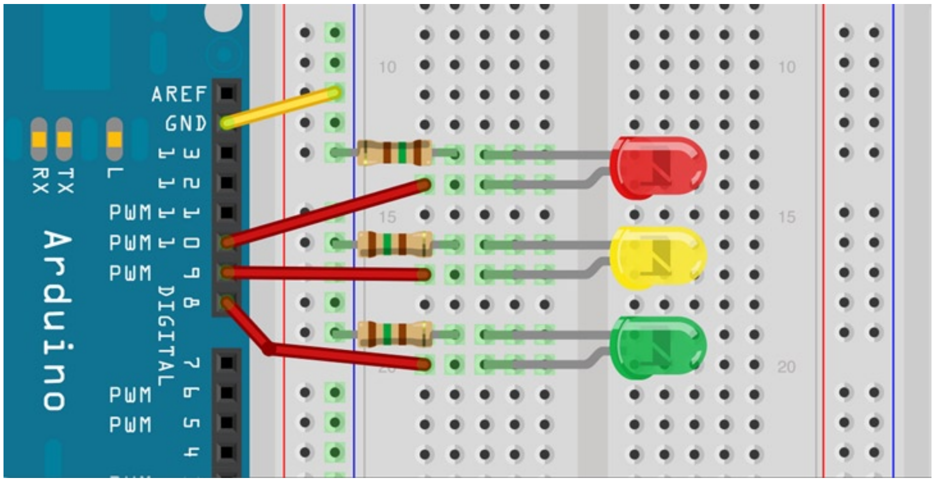


Figure 2-6. *The circuit for Project 3*

We have taken a jumper wire from the 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 current-limiting resistor (this time, connected to the cathode). For this simple circuit, it doesn't matter if the resistor is connected between the digital pin and the anode, or between the cathode and ground, as long as it is in *series* with the LED.

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 a traffic light system, as seen in Figure 2-7. If you have followed projects 1 and 2, then 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

}
```

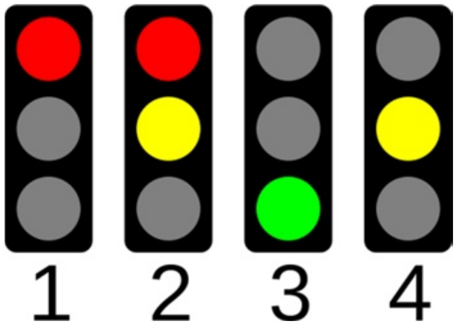


Figure 2-7. *The four states of a traffic light system (image by Alex43223 from WikiMedia)*

In the next project, we are going to build on project 3 by including a set of pedestrian lights and adding a push button to make the lights interactive. In doing so, you will learn about detecting an input.

Project 4 – Interactive Traffic Lights

This time, we are going to extend the previous project to include a set of pedestrian lights and a pedestrian push button used for requests 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.

For the first time, we will be able to interact with the Arduino and cause it to do something when we change the state of a button that the Arduino is watching (i.e., press the button to change the state from open to closed). In this project, we will also learn how to create our own functions in code.

From now on, when listing the parts required, we will no longer list the breadboard and jumper wires. Just assume that you will always need both of those.

Parts Required

The parts list in Table 2-4 is almost identical to that in Project 3. However, we have added an extra two LEDs and their respective current-limiting resistors to represent the pedestrian crossing lights. We have also included a switch so that you can control the lights and learn about reading an input pin.

Table 2-4. *Parts Required for Project 4*

2 x Red Diffused LEDs	
Yellow Diffused LED	
2 x Green Diffused LEDs	
10KΩ Resistor	
5 x Current-Limiting Resistors	
Pushbutton	

The five resistors for the LEDs are there to limit the current (and are thus called current limiting) going to the LED to prevent damage. Choose the appropriate value resistor for your project (see Chapter 1).

You will need a 10KΩ resistor for the pushbutton. This is what is known as a "pull-down resistor." We will cover pull-down and pull-up resistors later in the chapter. The pushbutton in Table 2-4 is sometimes referred to by suppliers as a "tactile switch," and is ideal for breadboard use.

Connect It Up

Connect your circuit as 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.

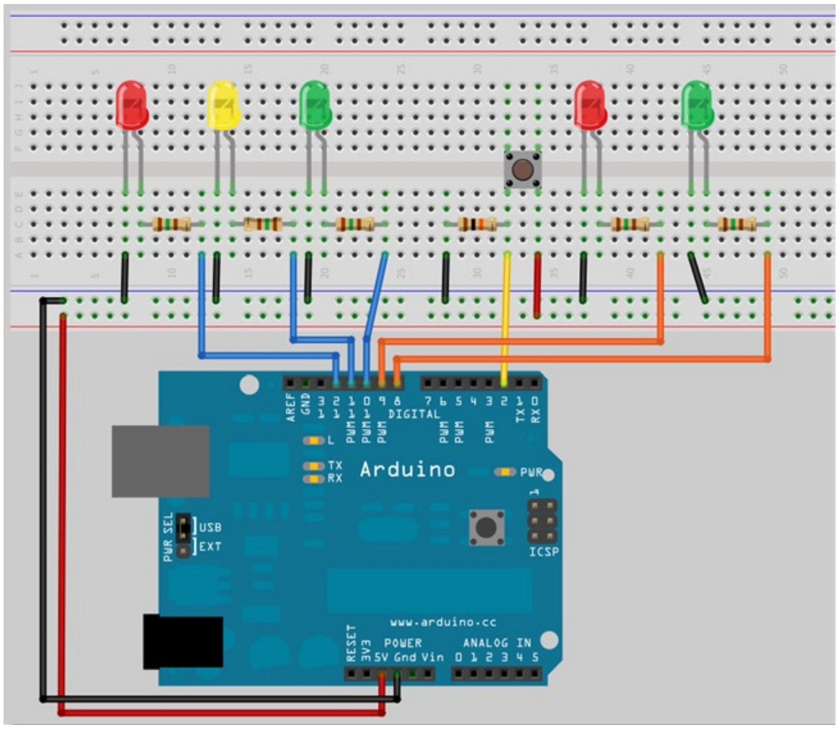


Figure 2-8. *Traffic light system with pedestrian crossing and request button*

Enter the Code

Enter the code in Listing 2-4 and verify and upload it.

When you run the program, you will see that the car traffic light starts on green to allow cars to pass and the pedestrian light is on red.

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

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 allowed to cross
unsigned long changeTime = 0; // time last pedestrian cycle completed
```

```
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
}
```

The code in this project is similar to the previous project. However, there are a few new statements and concepts that have been introduced, so let's take a look at those.

Project 4 – Interactive Traffic Lights - Code Overview

Most of the code in this project you will understand and recognize from previous projects. However, let us take a look at a few new keywords and concepts that have been introduced in this sketch.

```
unsigned long changeTime = 0;
```

Here we have a new data type for a variable. Previously, we have created integer data types, which can store a number between the range –32,768 and 32,767. This time, we have created a data type of long, which can store a number from –2,147,483,648 to 2,147,483,647. However, we have specified an unsigned long, which means the variable cannot store negative numbers, which gives us a range from 0 to 4,294,967,295. If we were to use an integer to store the length of time since the last change of lights, we 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, we don't want our program misbehaving due to our variable "overflowing" when it tries to store a number too high for the variable data type. That is why we use an unsigned long data type as we now get a huge length of time in between button presses. The number is initialized with 0.

4,294,967,295 * 1ms = 4,294,967 seconds

4,294,967 seconds = 71,582 minutes

71,582 minutes = 1,193 hours

1,193 hours = 49 days

As it is inevitable that a pedestrian crossing will get its button pressed at least once in 49 days, we shouldn't have a problem with this data type.

You may well ask why we don't just have one data type that can store huge numbers all the time and be done with it. Well, the reason we don't do that is because variables take up space in memory and 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 that much at all, but on a small microcontroller like the Atmega328 that the Arduino uses, it is essential that we use only the smallest variable data type necessary for our purpose.

There are various data types that we can use as our sketches and these are:-

Table 2-5. *Data Types*

Data type	Size	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

(*continued*)

Table 2-5. (*continued*)

Data type	Size	Number Range
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 on the Arduino, as you can see on the chart above. Some variables use only one byte of memory and others use four or more (don't worry about what a byte is for now as we will discuss this later). You cannot copy data from one data type to another; for example, if *x* was an int and *y* was a string, then *x* = *y* would not work as the two data types are different.

The Atmega168 has 1kB(1024 bytes) and the Atmega328 has 2kB(2048 bytes) of SRAM. This is not a lot, and 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. From the list above, we can clearly see that our use of the int data type is wasteful as it uses up two bytes and can store a number up to 32,767. As we have used int to store the number of our digital pin, which will only go as high as 13 on our Arduino (and up to 54 on the Arduino Mega), we have used up more memory than was necessary. We could have saved memory by using the byte data type, which can store a number between 0 and 255, which is more than enough to store the number of an I/O pin.

Next we have

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

This tells the Arduino that we want to use digital pin 2 (button = 2) as in INPUT. We are going to use pin 2 to check for button presses, so its mode needs to be set to input.

In the main program loop, we check the state of digital pin 2 with this statement:-

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

This initializes an integer called "state" (yes, it's wasteful and we should use a boolean), and then sets the value of state to be the value of the digital pin 2. The digitalRead statement reads the state of the digital pin within the parenthesis and returns it to the integer we have assigned it to. We 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, and if so, to execute the code within its code block. For example, if we wanted to turn an LED on if a variable called *x* rose above the value of 500, we could write

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

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

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

What we 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 as we have already set it to be the value read in from digital pin 2. We are also checking that the value of millis()-changeTime is greater than 5,000 (using the logical AND operator &&). 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. Our 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 five 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 we mean by that, 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, for example:

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

This if statement will run its code only if *x* is 5 and also if *y* is 10.

|| means true if either operand is true, e.g.:

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

This will run if *x* is 5 or if *y* is 10.

The ! or NOT statement means true if the operand is false, for example:

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

Will run if *x* is false, i.e., equals zero.

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 we draw a simple truth table (Table 2-6) for this statement, we can see how it works.

***Table 2-6.** 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 statement within the `if` command is

```
changeLights();
```

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 we have not passed any data to the function, nor have we had the function return any data. We will go into more detail later on about passing parameters and returning data from functions in Project 10, Chapter 3.

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.

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

The code within the function simply changes the vehicle lights to red, via amber, and then turns on the green pedestrian light. After a period of time set by the variable crossTime, the light flashes a few times to warn the pedestrian that his or her time is about to run out, then the pedestrian light goes red and the vehicle light goes from red to green, via amber, and returns 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 were last changed is greater than five seconds, it calls the changeLights() function again.

In this program, there was no benefit to putting the code into its own function apart from making the code look cleaner and to explain the concept of functions to you. It is only when a function is passed parameters and/or returns data that their true benefits come to light and we will take a look at that later on when we use functions again.Code size reduction is also a true benefit, which can come about by encapsulating code that is used in more than one place in the code – for example, the generation of dits in the SOS generation code. This benefit comes without the need for passed parameters or return values.

Next, in Project 5, we are going to use a lot more LEDs as we make a *Knight Rider*-style LED chase effect.

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 also. This is what is known as a pull-down resistor and is essential to ensure the button works properly. We will now 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. These are represented by the binary numbers 1 and 0. For Arduino inputs and outputs, the off (or zero) state is a voltage near to zero volts at the output and 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).

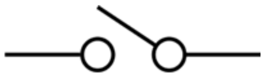


Figure 2-9. *The electronic symbol for a switch*

With a voltage going through the switch and the switch 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 and a voltage can be measured at the output. The open state can be thought of as a zero 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 five volts, then it is important that when the circuit outputs a 1 that the voltage is as close to five volts as possible. Similarly, when the output is a zero (or off), then 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, then that part of the circuit may be considered to be "floating," that is, it is neither in a high or low state. 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," then it may be interpreted as either a zero or a one, which is not desirable. Therefore, forcing it towards a desired state when it is inclined to float is desirable. The floating state is susceptible to electrical noise, and noise in a digital circuit may be interpreted as random 1s and 0s.

Pull-Down Resistors

In Figure 2-10, we have a schematic (a schematic is a conceptual rather than physical diagram of how the circuits are interconnected) in which a pull-down resistor being used. If the button is pressed, then the electricity takes the path of least resistance and moves between the five volts and the input pin, as 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 and would float in-between zero and five volts. In this circuit, the input will always be pulled to ground, or zero volts, when the button is not pressed and will be pulled towards five volts when the button is pressed. We have therefore made sure that in either state the pin is either reading zero or five volts and not floating in-between those two values. Now look at Figure 2-11.

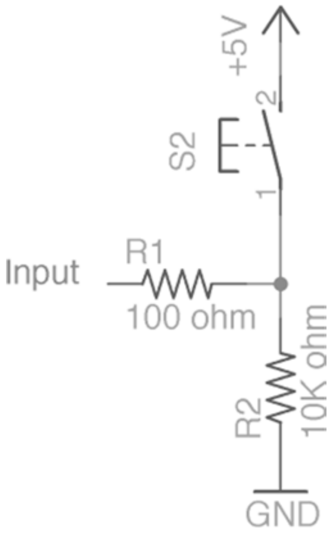


Figure 2-10. *A pull-down resistor circuit*

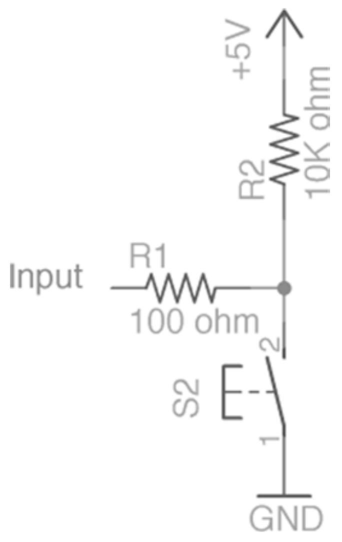


Figure 2-11. *A pull-up resistor circuit*

Pull-Up Resistors

In this circuit, we have swapped the pull-down resistor and the switch. The resistor now serves as a pull-up resistor. Now you can see that when the button is not pressed, the input pin is pulled towards the five volts and so 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 five volts and ground it would be a short circuit which would damage your circuit or power supply. But with the resistor it is no longer a short circuit as the resistor limits the amount of current that can flow to a small amount. 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 at an input pin is always either high or low, depending on your application. In Project 4 we used a pull-down resistor to ensure a button press could be registered correctly by the Arduino. Let's take a look at the pull-down resistor in that circuit again (see Figure 2-12).

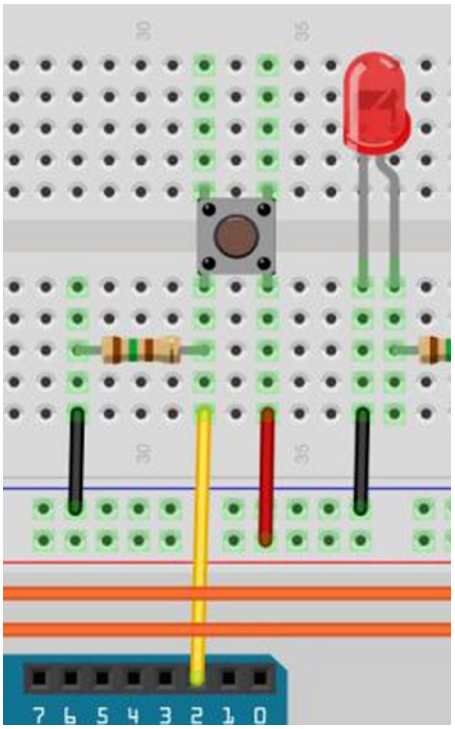


Figure 2-12. *A pull-down resistor from Project 4*

In this circuit, we have 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 one or a high state. By detecting if the input is high or low, we can detect if the button is pressed or not. If the resistor were not present, then 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 is not good as we could get false button presses registered. Hence, the pull-down resistor is vital in ensuring that the pin is pulled to ground or LOW when the button is not pressed.

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 we will be using in Chapter 6 has a Master Reset pin. This pin will reset the chip when it is pulled low. Therefore, it is essential that this pin is kept high at all times, unless we specifically want to do a reset. Therefore, we can hold this pin high using a pull-up resistor at all times to ensure that it does not reset. When we want to reset, we pull it low using a digital output set to LOW; at all other times, it will remain high. Many other ICs have pins that must be kept high for most of the time and only pulled low for various functions to be activated, such as a reset. A pull-up resistor will ensure that this pin stays at the HIGH state at all times and cannot be accidently pulled low. If the pull-up resistor was not in the circuit, then the pin could float and may randomly alternate between high and low. This could cause an unwanted reset or other function to be activated unwittingly.

The Arduino's Internal Pull-Up Resistors

Conveniently, the Arduino happens to have pull-up resistors connected to the digital pins (the analog pins have pull-up resistors also). These have a value of 20K ohms and need to be activated in 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. For example:

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

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

Now that you understand the use of pull-up and pull-down resistors, which will be used throughout the book, let us move onto Project 5 and make some lighting effects with our LEDs.

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 pin's mode between INPUT and OUTPUT
- The **`millis()`** function and how to use it
- Boolean operators and how to use them to make logical decisions

CHAPTER 3

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

We are now going to use a string of LEDs (10 in total) to make an LED chase effect (see Table 3-1), similar to that used on the car KITT in the *Knightrider* TV series or the Cylons in *Battlestar Galactica;* on the way, we'll introduce the concept of arrays.

Parts Required

Table 3-1. *Parts Required for Project 5*

10 x 5mm RED LEDs	
10 x Current-Limiting Resistors	

Connect It Up

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

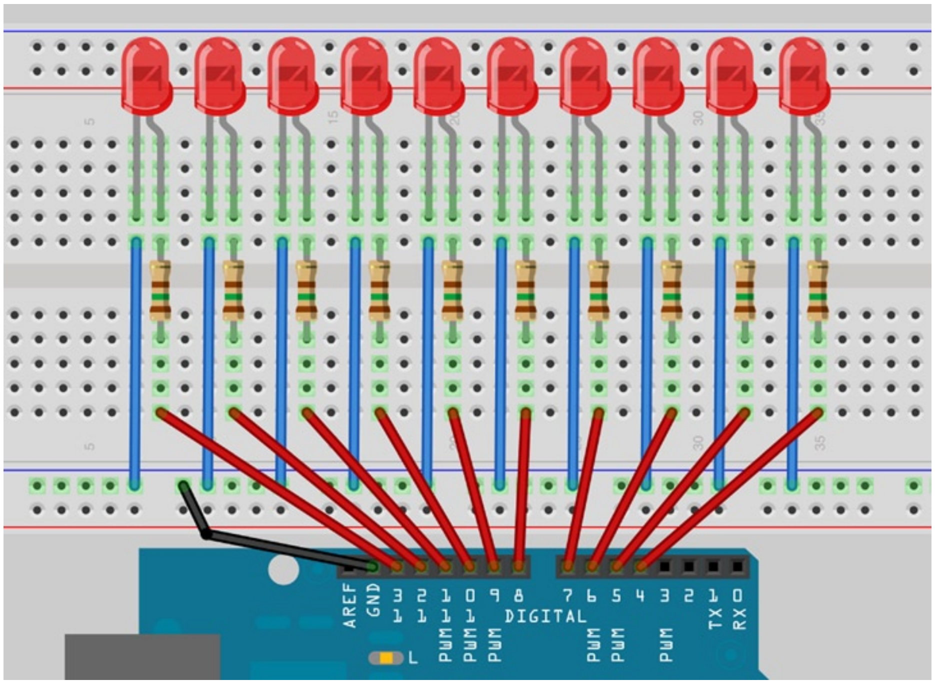


Figure 3-1. *The circuit for Project 5 – LED Chase Effect*

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;}
}
```

Now press the Verify button at the top of the IDE to make sure there are no errors in your code. If this is successful, you can now click the Upload button to upload the code to your Arduino. If you have done everything correctly, you should now see the LEDs appear to move along the line, then bounce back to the start. We have not introduced any different hardware in this project, so there is no need to take a look at that. However, we have introduced a new concept in the code of this project in the form of arrays. Let us take a look at the code for Project 5 and see how it works.

Project 5 – LED Chase Effect – Code Overview

Our very first line in this sketch is

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

which declares the ledPin variable to be an array (collection) of elements of data type *byte*. All the elements share the same name (in this case, ledPin), but can be selected individually by an index number, much like apartments in an apartment building. The [] after the variable name in the declaration tells the compiler this is an array variable rather than a simple (non-indexed) variable. We have then initialized the array with 10 values, which are the digital pin numbers 4 through to 13. To access an element of the array, we follow the array name with the index number of that element between square brackets. The index number between the [] doesn't have to be a constant—it can be a variable or expression. Arrays are zero indexed, which simply means that the index of the first element is zero, rather than one. So in our 10-element array, the index numbers are 0 to 9. In this case, the third element (ledPin[2]) has the value of 6, and the seventh element (ledPin[6]) has a value of 10.

You have to tell the compiler the size of the array if you do not initialize it with data first in the declaration. In our sketch we did not explicitly choose a size, as the compiler is able to count the values we have assigned to the array to work out that the size is 10 elements. If we had declared the array, but not initialized it with values at the same time, we would need to declare a size, for example we could have done this:

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

and then loaded data into the elements later on. To retrieve a value from the array, we 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, we have started off by declaring and initializing an array with 10 values that are the digital pin numbers used for the outputs to our 10 LEDs.

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

The function we created 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, then the number will either increase (+1) or decrease by one (currentLED +(–1)).

We then have an if statement to see if we have reached the end of the row of LEDs, and if so, we then 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. You have to stop the program and manually change the value of ledDelay, then upload the amended code to see any changes.

However, 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 by introducing a way to interact with the program and adjust the speed using a potentiometer.

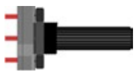
Project 6 – Interactive LED Chase Effect

Leave your circuit board as it was in Project 5. We are 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

In Table 3-2 you will see that the only part we add to this project that differs from the last is a potentiometer, or variable resistor. This will allow you to control the resistance read by the Arduino from the potentiometer to control the speed of the lights. Use the circuit from Project 5 and add the potentiometer in Table 3-2.

Table 3-2. *Parts Required for Project 6*

4.7KΩ Rotary Potentiometer	

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now add a potentiometer to the circuit so it is connected as 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.

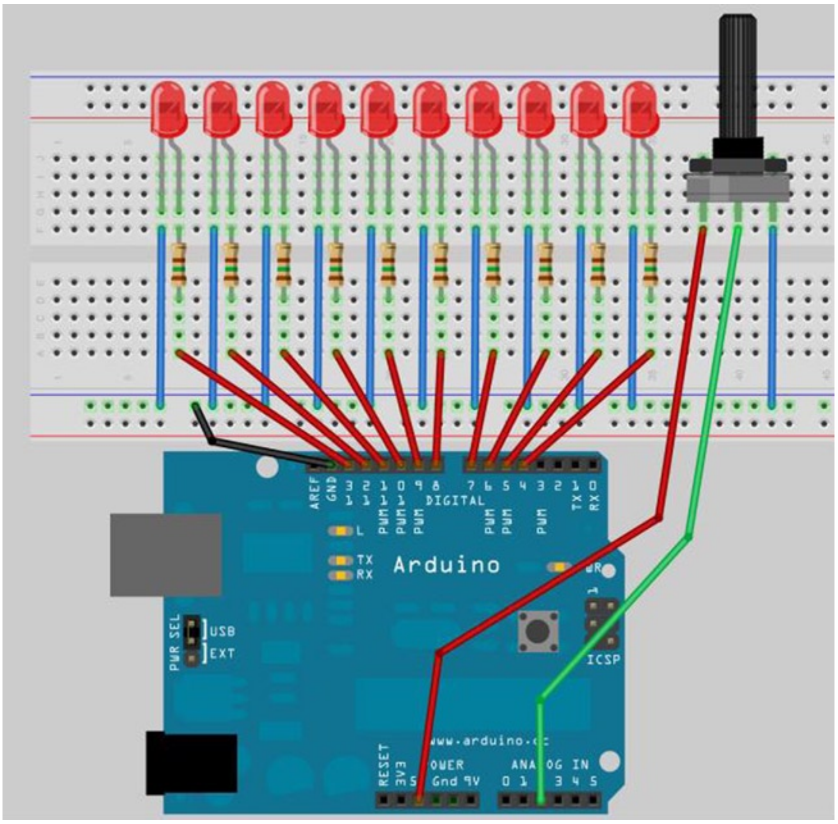


Figure 3-2. *The circuit for Project 6 – Interactive LED Chase Effect*

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;}
}
```

This time when you verify and upload your code, you should now 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's code. We have simply added a potentiometer to our hardware, and the code has additions to enable us to read the values from the potentiometer and use them to adjust the speed of the LED chase effect.

We first declare a variable for the potentiometer pin number

```
int potPin = 2;
```

as our potentiometer is connected to analog pin 2. To read the value from an analog pin, we use the analogRead command. The Arduino has 6 analog input/outputs with a 10-bit analog-to-digital convertor (we will discuss bits later on). 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 / 1,024 units or 0.0049 volts (4.9mV) per unit.

We need to set our delay using the potentiometer, so we will simply use the direct values read in from the pin to adjust the delay between 0 and 1,023 milliseconds (or just over 1 second). We do this by directly reading the value of the potentiometer pin into ledDelay. Notice that we do not need to set an analog pin to be an input or output as we do with a digital pin.

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

This is done during our main loop, and therefore, it is constantly being read and adjusted. By turning the knob, you can adjust the delay value between 0 and 1,023 milliseconds (or just over a second) and therefore have full control over the speed of the effect.

Okay, 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 was the 4K7 (4700Ω) potentiometer (see Figure 3-3).

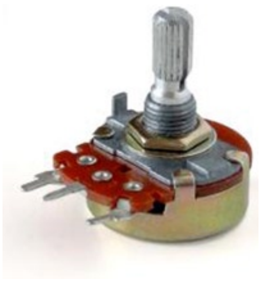


Figure 3-3. *A rotary potentiometer (image courtesy of Iain Fergusson)*

You have already come across a resistor and know how it works. The potentiometer is simply a fixed resistor with an adjustable wiper that can be used to divide the total resistance into two parts that add up to the total (fixed) value. In this project, we use a 4K7 (0r 4.7K Ω, K = 1,000) or 4,700Ω potentiometer which means its range is from 0 to 4,700 Ohms.

The potentiometer has three legs. By connecting up just two legs, the potentiometer becomes a variable resistor. By connecting all three legs and applying a voltage across it, the potentiometer becomes a voltage divider. This is how we have used it in our circuit. One side is connected to ground, the other to 5V, and the center pin to our analog pin. By adjusting the knob, a voltage between 0 and 5V will be available from the center pin; we can read the value of that voltage on Analog Pin 2 and use its value 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.

EXERCISE

Now try out the following two exercises. You have all the necessary knowledge 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 to both move toward each other, then appear to bounce off each other, and, finally, move back to the end.
- Exercise 2: Make a bouncing ball effect by turning the LEDs so that they are vertical. Then make an LED start at the bottom, then "bounce" up to the top LED, then back to the bottom; then next time, have it "bounce" only up to the 9th LED, then back down, then up to t^{he} 8th, etc., to simulate a bouncing ball losing momentum after each bounce.

Project 7 – Pulsating Lamp

We are now going to delve further into a more advanced method of controlling LEDs. So far we have simply turned the LED on or off. How about being able to adjust its brightness, too? Can we do that with an Arduino? Yes, we can.

Time to go back to basics.

Parts Required

Table 3-3 lists the parts required for Project 7. We simply use an LED and resistor, as we did in Project 1.

Table 3-3. *Parts required for Project 7*

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 3-4.

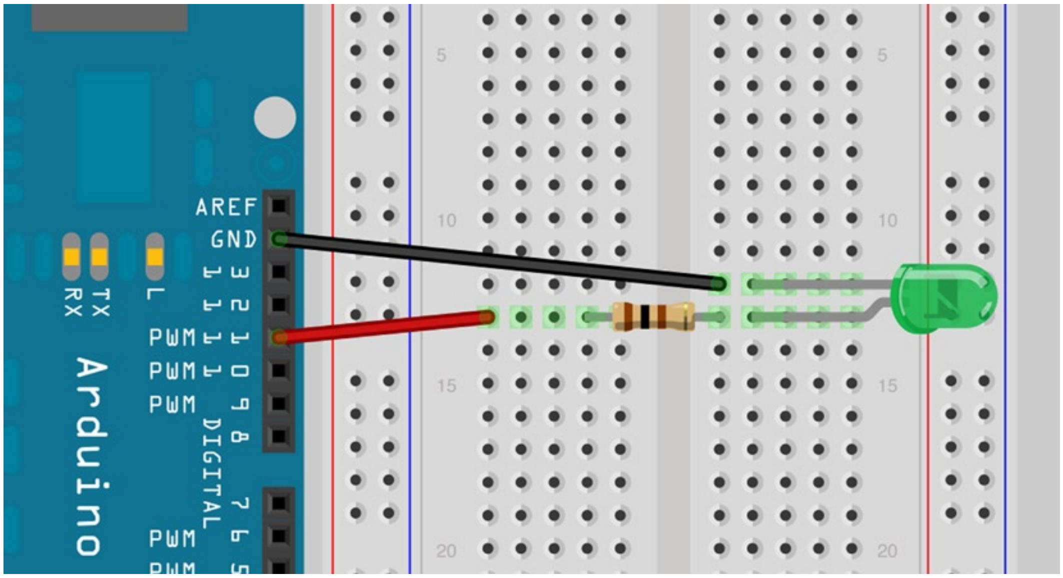


Figure 3-4. *The circuit for Project 7 – Pulsating Lamp*

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 now see your LED pulsate on and off steadily. Instead of a simple on/off state, we are now adjusting its brightness. Let's find out how this works.

Project 7 – Pulsating Lamp – Code Overview

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

We first set up 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 Pin 11.

The concept here is that we 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 fading up to full brightness and back down again.

We use the sin() function, which is a mathematical function for the sine of an angle. We need to give the function the angle, expressed in radians. We have a for loop that goes from 0 to 179; we don't want to go past halfway as this will take us into negative values, and the brightness value we need to put out to Pin 11 needs to be from 0 to 255 only.

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. We then transfer the result to ledVal, multiplying it by 255 to give us our value. The result from the sin() function will be a number between –1 and 1, so we need to multiply that by 255 to give us our maximum brightness. We “cast” the floating point value of sinVal into an integer by the use of int() in the statement

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

Then we send that value out to digital pin 11 using the statement

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

Casting means to convert a value in one data type to a different data type (in this case, floating point to integer, by throwing away the portion after the decimal point). But, how can we send an analog value to a digital pin? Well, if we take a look at our Arduino and look at the digital pins, you can see that 6 of those pins (3, 5, 6, 9, 10 & 11) have PWM written next to them. Those pins differ from the remaining digital pins in that they are able to send out a PWM signal, which stands for pulse width modulation.

PWM is a technique for getting analog results from digital means. On these pins, the Arduino sends out a rectangle 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 pin 11 using analogWrite(), the ON period would be zero, or it would have a 0% duty cycle. If you were to send a value of 64 (25% of the maximum of 255), the pin would be ON for 25% of the time and OFF for 75% of the time. The value of 191 would have a duty cycle of 75% and a value of 255 would have a duty cycle of 100%. The pulses run at a speed of approximately 500Hz, or 2 milliseconds each.

So, from this we can see in our sketch that the LED is being turned on and off very fast. If the duty cycle were 50% (a value of 127), then the LED would pulse on and off at 500Hz and would display at half the maximum brightness. It is basically an illusion that we can use to our advantage. It allows us to use the digital pins to output a simulated analog value to our LEDs.

Note that even though only 6 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 on, we will revisit PWM to utilize it to create audible tones using a piezo sounder.

Project 8 – RGB Mood Lamp

In the last project, we saw that we could adjust the brightness of an LED using the PWM capabilities of the Atmega chip. We will now take advantage of this capability by using a red, green, and blue LED and by mixing the colors to create any color we wish. From that, we will create a mood lamp similar to the kind you often see for sale in shops nowadays.

Parts Required

This time we are going to use three LEDs, one red, one green, and one blue. (Tables appear in color in the eBook version of this book and in grayscale in the print version.)

Table 3-4. *Parts Required for Project 7*

Red Diffused 5mm LED	
Green Diffused 5mm LED	
Blue Diffused 5mm LED	
3 x Current-Limiting Resistors	

Connect It Up

Connect up your three LEDs as in Figure 3-5. Get a piece of paper about A5 size (6" by 8"), roll it into a cylinder, then tape it so it remains that way. Then place the cylinder over the top of the three LEDs. This will diffuse the light from the three LEDs and merge the colors into one.

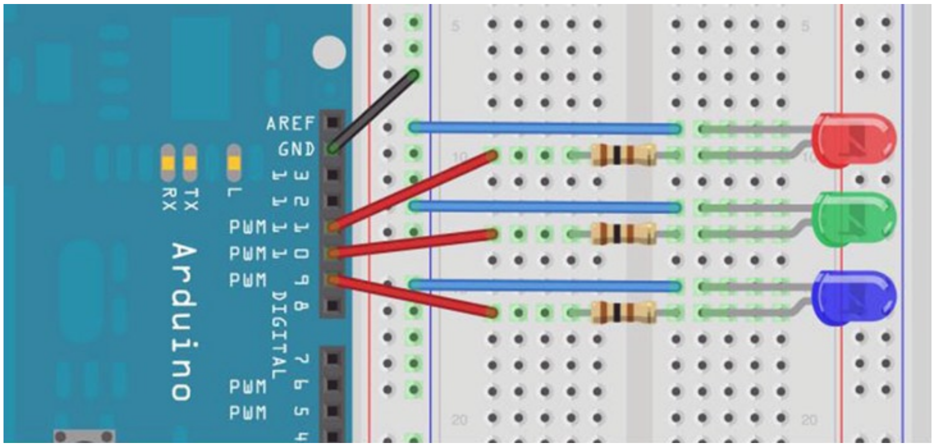


Figure 3-5. *The circuit for Project 8 – RGB Mood Lamp*

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()
{
        Serial.begin(9600);
        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, the map can generate different colors by adjusting the brightness of each of the three LEDs in such a way as to give us a different RGB value. Alternatively, you could have used an RGB LED. This is a single 5mm LED, with four legs (some have more). One leg is either a common anode (positive) or common cathode (negative) and the other three 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 would give us pure red. A value of 0, 255, 0 would give pure green, and 0, 0, 255, pure blue. By mixing these, we can get any color we like. This is the additive color model (see Table 3-5). If you were just to turn the LEDs ON or OFF (i.e., not have different brightnesses), you would still get different colors, as in Table 3-5.

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
0	0	0	Black

By adjusting the brightnesses using PWM, we can get every other color in between, too. By placing the LEDs close together and by mixing their values, the light spectra of the three colors added together make a single color (see Figure 3-6). By diffusing the light with our paper cylinder, we ensure that the colors are mixed nicely. The LEDs can be placed into any object that will diffuse the light, or you can 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).

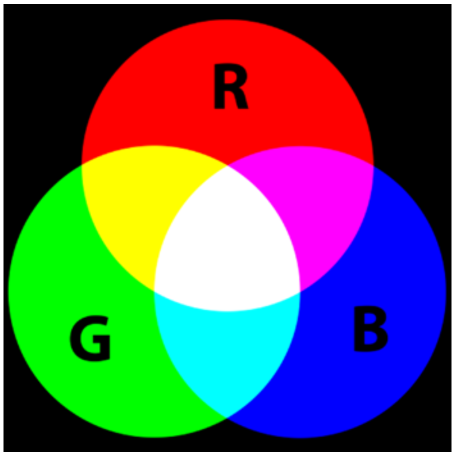


Figure 3-6. *Mixing R, G and B to get different colors*

The total range of colors we can get using PWM with a range of 0 to 255 is 16,777, 216 colors (256 x 256 x 256) which is much more than we would ever need.

In the code, we start off by declaring some floating point arrays and also some integer variables that will store our RGB values as well as an increment value.

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

int red, green, blue;
```

In the setup function we have

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

The randomSeed command is used for creating random (actually pseudo-random) numbers. Computer chips are not able to produce truly random numbers, so they use a mathematical function that generates a very long sequence of pseudo-random values before repeating. By setting a "seed," you can tell the computer where in the sequence to start returning random numbers. In this case, the value we give to the randomSeed is a value read from analog pin 0. As we don't have anything connected to analog pin 0, all we will read is a random number created by analog noise.

Once we have set a "seed" for our random number, we can create one using the random() function. We then have two sets of RGB values stored in a three element array. RGB1 is the RGB value we 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 we want the lamp to transition to

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

In this case, we have set the RGB values to a random number set by random(256), which will give us a number between 0 and 255 inclusive (as the number will always range from zero upward).

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

In the main program loop, we first take a look at the start and end RGB values and work out which 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). We do this with

```
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.

We then 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];
  }
```

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

After we 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. We now need to decide upon another set of three random values ready for the next time. We 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. The reason we do this is to try and force primary colors from time to time to ensure that we don't always just get pastel shades. We have 300 chances out of 556 of 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* as in constrain(x, a, b) where *x* is the number we 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*, then it sets it to *a*; if it is higher than *b*, it sets it to *b*. In our case, we make sure that the number is between 0 and 255, which is the range of our PWM output.

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

Forcing a bias towards one or more of the other two channels ensures that we get more vibrant and less pastel shades of color and also ensures that, from time to time, one or more channels are turned off completely, giving a more interesting change of lights (or moods).

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, using PWM again, to recreate the effect of a flickering flame. If you were to place these LEDs inside a model house on a model railway layout, for example, you could create a special effect of the house being on fire, or you could place it into a fake fireplace in your house to give a fire effect. This is a simple example of how LEDs can be used to create special effects for movies, stage plays, model dioramas, etc.

Parts Required

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

Table 3-6. *Parts Required for Project 7*

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

Connect It Up

Power down your Arduino, then connect up your three LEDs as in Figure 3-7. This is essentially the same circuit as in project 8, but using one red and two yellow LEDs instead of 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 the surface you wish the effect to be projected against.

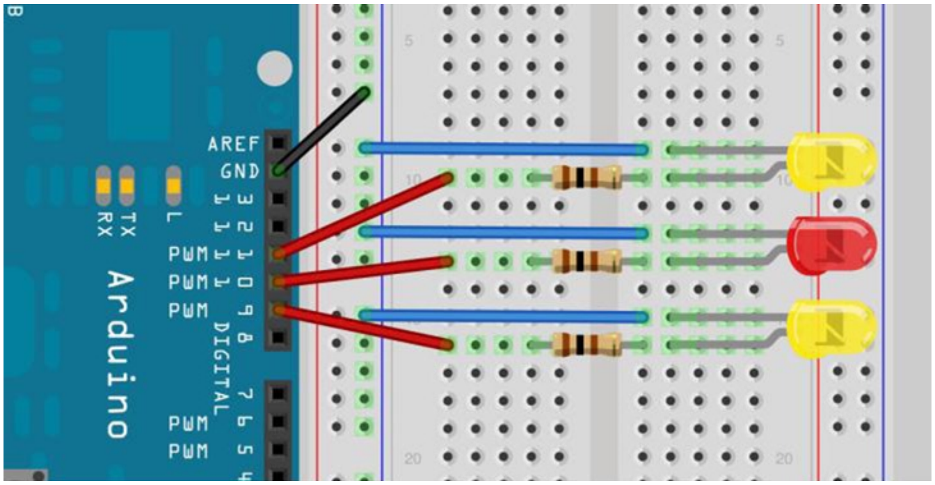


Figure 3-7. *The circuit for Project 9 – LED Fire Effect*

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));
}
```

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

If you have done everything correctly, you should now see the LEDs flickering in a random manner to simulate a flame or fire effect.

Now let's take a look at the code and find out how it works.

Project 9 – LED Fire Effect – Code Overview

So let's take a look at the code for this project. First we declare and initialize some integer variables that will hold the values for the digital pins we are going to connect our LEDs to.

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

We then set them up to be outputs.

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

The main program loop then sends out a random value between 0 and 120, and then adds 135 to it to get full LED brightness, to the PWM pins 9, 10 and 11.

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

Finally, we have a random delay between zero and 100ms.

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

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

As the hardware is simple, we will jump right into Project 10.

EXERCISE

Now 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 on an emergency vehicle.

Project 10 – Serial Controlled Mood Lamp

For Project 10 we will revisit the circuit from Project 8, RGB Mood Lamp, but will now delve into the world of serial communications and control our lamp by sending commands from the PC to the Arduino using the serial monitor in the Arduino IDE. This project also introduces how we manipulate text strings. So set up the hardware as 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);
        while(Serial.available())
                Serial.read();
        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, " ,");          // Note that this is a space before the comma in " , "
        while (parameter != NULL) {
                setLED(parameter);
                parameter = strtok (NULL, " ,");  // space before the comma in " , "
}

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

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.

Now 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 can now enter the R, G, and B values for each of the three LEDs manually; the LEDs will change to the color you have input. For example, if you enter R255, the Red LED will display at full brightness. If you enter R255, G255, then both the red and green LEDs will display at full brightness.

Now enter R127, G100, B255 and you will get a nice purplish color. If you type r0, g0, b0, all the LEDs will turn off.

The input text is designed to accept both a lower-case or upper-case R, G, and B and then a value from 0 to 255. Any values over 255 will be dropped down to 255 maximum. You can enter a comma or a space in between parameters and you can enter 1, 2, or 3 LED values at any one time. For example,

```
r255 b100
```

```
r127 b127 g127
```

```
G255, B0
```

```
B127, R0, G255
```

and so forth.

Project 10 – Serial-Controlled Mood Lamp – Code Overview

This project introduces a whole bunch of new concepts, including serial communication, pointers, and string manipulation. So, hold on to your hats; this will take a lot of explaining.

First we set up an array of char (characters) to hold our text string. We have made it 18 characters long, which is longer than the maximum of 16 we will allow to ensure we don't get "buffer overflow" errors.

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

We 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 our setup function, we set the three digital pins to be outputs. But, before that we have the Serial.begin command.

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

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

Next,

```
while(Serial.available())
        Serial.read();
```

will flush out any characters that happen to be in the serial line so that it is empty and ready for input/output. It does this by checking whether there is any serial data available (in our case, junk data) and if so, reading it. The data read clears out any data held in the serial buffer.

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

In the main loop we have an if statement. The condition it is checking for is

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

The Serial.available command checks to see if any characters have been received on the serial line. If any characters have been received, then the condition is met and the code within the if statements code block is now executed. The Serial.available() command returns the number of characters waiting to be read.

```
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. We will use index to keep track of our position within the character array.

We then set a delay of 100. The purpose of this is to give time for the full command to be received before we carry on and process the data. If we don't do this, it is possible that the code will start to process the text string before we have received all of the data. The serial communications line is very slow compared to the speed at which the rest of the code is executing. When you send a string of characters, the Serial.available function will immediately have a value higher than zero and the "if" statement will start to execute. If we didn't have the delay(100) statement in there, the Arduino could start to execute the code within the if statement before all of the text string had been received and the serial data processed may only be the first few characters of the line of text entered.

After we have waited 100 ms for the full text string to be received, we then declare and initialize the numChar integer to be the number of characters within the text string. For example, If we sent this text in the serial monitor:

```
R255, G255, B255
```

Then the value of numChar would be 17. It is 17, and not 16, as at the end of each line of text there is an invisible character called a NULL character. This is a "nothing" symbol and simply tells the Arduino that the end of the line of text has been reached.

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

After this comes a while command. This is something we haven't come across before, so let me explain. We 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

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

In our code the while loop is

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

The condition it is checking is simply numChar, so, in other words, it is checking that the value stored in the integer numChar is not zero. numChar has—after it. This is what is known as a post-decrement. This simply means that the numChar is decremented AFTER it is tested for the while(). If we had used –numChar, the value in numChar would be decremented (have one subtracted from it) before it was evaluated. In our case, the while loop checks the value of numChar and then subtracts one 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 we 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();
```

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 we have entered into the serial monitor's text window.

The Serial.read() command reads incoming serial data, one byte at a time. So now that our character array has been filled with the characters we 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 we have

```
splitString(buffer);
```

This is a call to the function we have created 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();
}
```

We don't plan to return any data from the function, so its data type has been set to void. We pass the function one parameter and that is a char data type that we have called data. Instead of passing all the elements within the array to the function, C passes a pointer to (in other words, the location in memory of) the first element of the array. (This is known as pass by reference rather than pass by value.) So our declaration of the function needs to accept a pointer argument. We tell the compiler that it is a pointer variable by adding an asterisk * to the front of the variable name.

Pointers in a Nutshell

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

> The type in the pointer declaration tells what type of data the pointer points to.
>
> char *mytext; declares mytext to be a pointer that can point to char (character) data.
>
> int *nextnumber; declares nextnumber to be a pointer that can point to integers.

Pointers have to be initialized before they can be used. We can set the pointer to the location of another variable, array, or array element by using the & (address of) operator to get the address of the variable or element we want to point to.

We can assign one pointer variable to another pointer variable of the same type. They will then both point to the same thing. We can get the value stored at the memory location the pointer currently points to by using the * (dereference)operator. For example:

```
char mychar;                        // declares a simple variable that can hold a character.
char buff[] = {'a','b','c','d'};    // declares an array of characters
char *mytext;                       // declares a variable that can point to character data
mytext = &buff[1];                  //  initializes mytext to point to the location of
                                                                  buff[1] in memory;
mychar = *mytext;                   //  retrieves the character that mytext points to
                                                             (stored in buff[1]) and
                                    //  copies that character (in this case, 'b') to mychar.
```

Incrementing or decrementing a pointer causes it to point to the memory location following or preceding the location it pointed to. So if it points to an array element, incrementing the pointer causes it to point to the next array element.

```
mytext++;              //  mytext now points to buff[2];
```

We can change the contents of the memory the pointer points to by dereferencing the pointer on the left side of an assignment statement.

> *mytext = 'g'; stores a 'g' in the location mytext points to. (buff[2], after the lines above).

A special value (NULL) is used to represent a non-valid (empty) pointer value. If the value of a pointer is NULL, attempts to fetch or store a value at what it points to are undefined (in other words, may cause your program to stop working). Search functions will frequently return a pointer value of NULL to indicate they couldn't find what you asked them to search for.

Since pointers frequently point to arrays, the language lets you subscript a pointer just like an array.

mytext[1] refers to the value stored in the first location after the location mytext currently points to. If mytext points to buff[2], mytext[1] refers to the contents of buff[2+1] (buff[3]), which holds 'd' in this case. mytext[0] is the same as *mytext.

When we called splitString, we sent it the contents of "buffer" (actually a pointer to it as we saw above).

```
splitString(buffer);
```

So we have called the function and passed it (by reference) the entire contents of the buffer character array.

The first command is

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

and this is our way of sending data back from the Arduino to the PC. In this case, the print command sends whatever is within the parenthesis to the PC, via the USB cable, where we can read it in the serial monitor window. We have sent the words "Data entered: ". Text must be enclosed within quotes "". The next line is similar

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

and again we have sent data back to the PC; this time, we send the character pointer variable called data. The character pointer variable we have called "data" points to the contents of the "buffer" character array that we passed to the function. So, if our text string entered was

```
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.

This time, the print command has ln on the end to make it println. This simply means "print" and then advance to the next line.

When we print using the print command, the cursor (the point at where the next symbol will appear) remains at the end of whatever we have printed. When we use the println command, a carriage return and linefeed follow our text, causing the cursor to drop down to the next line after our text is printed.

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

If we look at our 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," or in other words, the contents of the array called "buffer," and then issue a linefeed, or drop the cursor down to the next line. If we issue another print or println statement after this, whatever is printed in the serial monitor window will appear on the next line underneath the last.

We then create a new char pointer variable called parameter

```
Char* parameter;
```

and as we are going to use 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 as 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.

We then use a strtok command, which is a very useful command which enables us to manipulate text strings. Strtok gets its name from String and Token as its purpose is to split a string using delimiters. In our case, the delimiter it is looking for is a space or a comma. It is used to split text strings into smaller strings called tokens.

We pass the "data" array to the strtok command as the first argument and the delimiters (enclosed within quotes) as the second argument. Hence

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

And it splits the string at that point. So we are using it to set “parameter” to be the part of the string up to a space or a comma.

So, if our text string was

```
R127 G56 B98
```

Then after this statement the value of “parameter” will be

```
R127
```

as the strtok command would have split the string up to the first occurrence of a space of a comma.

After we have set the variable “parameter” to the part of the text string we want to strip out (i.e., the part up to the first space or comma), we then enter a while loop whose condition is that parameter is not empty (i.e., we haven’t reached the end of the string) using

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

Within the loop we call our second function

```
setLED(parameter);
```

We will look at this later on. Then it sets the variable “parameter” to the next part of the string up to the next space or comma. We do this by passing to strtok a NULL parameter

```
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 part 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, which is then 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';
}
while(Serial.available())
 Serial.read();
```

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

```
G125 B55
```

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

```
G125
B55
```

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

So let's take a look at 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);
        }
}
```

We can see that this function contains three very similar if statements. We will therefore take a look at just one of them as the other two are almost identical.

```
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). We use the logical OR command whose symbol is || to check if the letter is an *r* OR an *R*, as it is fine to enter the r in either upper or lower case.

If it is an *r* or an *R*, then the if statement knows we wish to change the brightness of the Red LED and so the code within executes. First we 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. These are the string we are passing it, a pointer to a variable where strtol( ) can store the character after the number string (which we don't use as we have already stripped the string using the strtok command and hence pass a NULL pointer) and then the "base," which in our case is base 10. The string we are passing it will contain decimal digits (as opposed to binary, octal, or hexadecimal digits which would be base 2, 8, and 16 respectively). So, in other words, we declare an integer and set it to the value of the text string after the letter *R* (or the numeric portion of it).

Next, we use the constrain command to make sure that Ans goes from 0 to 255 and no more. We 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 thing, but for the green and the blue LEDs.

We have covered a lot of ground and a lot of new concepts in this project. To make sure you understand exactly what is going on in this code, I am going to set the project code side by side with pseudo-code (the computer language translated into a language humans can understand). See Table 3-7.

Table 3-7. *An Explanation for the Code in Project 10 Using Pseudo-Code*

The C Programming Language	Pseudo-Code
// Project 10 - Serial controlled RGB Lamp	A comment with the project number and name
char buffer[18];	Declare a character array of 18 letters
int red, green, blue;	Declare 3 integers called red, green and blue
int RedPin = 11;	An integer for which pin to use for red LED
int GreenPin = 10;	" " Green
int BluePin = 9;	" " Blue
void setup()	The setup function
{	
Serial.begin(9600);	Set serial comms to run at 9600 chars per second
Serial.flush();	Flush the serial line
pinMode(RedPin, OUTPUT);	Set the red LED pin to be an output pin
pinMode(GreenPin, OUTPUT);	Same for green
pinMode(BluePin, OUTPUT);	And blue
}	
void loop()	The main program loop
{	
if (Serial.available() > 0) {	If data is sent down the serial line...
int index=0;	Declare integer called index and set to 0
delay(100); // let the buffer fill up	Wait 100 millseconds
int numChar = Serial.available();	Set numChar to the length of the incoming data from serial

(*continued*)

Table 3-7. (*continued*)

The C Programming Language	Pseudo-Code
if (numChar>15) {	If numchar is greater than 15 characters...
numChar=15;	Make it 15 and no more
}	
while (numChar--) {	While numChar is not zero (subtract 1 from it)
buffer[index++] = Serial.read();	Set element[index] to value read in (add 1 to index)
}	
splitString(buffer);	Call splitString function and send it data in buffer
}	
}	
void splitString(char* data) {	The splitString function references buffer data
Serial.print("Data entered: ");	Print "Data entered: "
Serial.println(data);	Print value of data and then drop down a line
char* parameter;	Declare char data type parameter
parameter = strtok (data, " ,");	Set it to text up to the first space or comma
while (parameter != NULL) {	While contents of parameter are not empty..
setLED(parameter);	Call the setLED function
parameter = strtok (NULL, " ,");	Set parameter to next part of text string
}	
// Clear the text and serial buffers	Another comment
for (int x=0; x<16; x++) {	We will do the next line 16 times
buffer[x]='\0';	Set each element of buffer to NULL (empty)
}	
Serial.flush();	Flush the serial comms
}	
void setLED(char* data) {	A function called setLED is passed buffer
if ((data[0] == 'r') || (data[0] == 'R')) {	If first letter is *r* or *R*...
int Ans = strtol(data+1, NULL, 10);	Set integer Ans to number in next part of text
Ans = constrain(Ans,0,255);	Make sure it is between 0 and 255
analogWrite(RedPin, Ans);	Write that value out to the red pin

(*continued*)

Table 3-7. (*continued*)

The C Programming Language	Pseudo-Code
Serial.print("Red is set to: ");	Print out "Red is set to: "
Serial.println(Ans);	And then the value of Ans
}	
if ((data[0] == 'g') || (data[0] == 'G')) {	If first letter is *g* or *G*...
int Ans = strtol(data+1, NULL, 10);	Set integer Ans to number in next part of text
Ans = constrain(Ans,0,255);	Make sure it is between 0 and 255
analogWrite(GreenPin, Ans);	Write that value out to the green pin
Serial.print("Green is set to: ");	Print out "Green is set to: "
Serial.println(Ans);	And then the value of Ans
}	
if ((data[0] == 'b') || (data[0] == 'B')) {	If first letter is *b* or *B*...
int Ans = strtol(data+1, NULL, 10);	Set integer Ans to number in next part of text
Ans = constrain(Ans,0,255);	Make sure it is between 0 and 255
analogWrite(BluePin, Ans);	Write that value out to the blue pin
Serial.print("Blue is set to: ");	Print out "Blue is set to: "
Serial.println(Ans);	And then the value of Ans
}	
}	

Hopefully, you can use this "pseudo-code" to make sure you understand exactly what is going on in this project's code.

We are now going to leave LEDs behind for a little while and look at how to make sounds from your Arduino using a piezo sounder.

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
- 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

CHAPTER 4

Simple Sounders and Sensors

We are now going to get our Arduino to make some simple sounds using a piezo sounder. Being able to produce sounds using the Arduino is a great way to add alarms, warning beeps, and alert notifications to the device we are creating. Since version 0018 of the Arduino IDE, a new command has been added that allowed tones to be created easily. We will also find out how to use the piezo as a sensor and learn how to read data from it. Finally, we will look at a light sensor. You will learn how to make simple sounds and pick up the basics of reading analog sensors along the way.

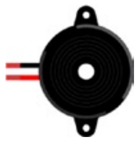
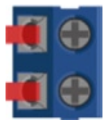
We are going to learn how to use the `tone()` command in Project 11 by making a simple alarm, similar to a car alarm.

Project 11 – Piezo Sounder Alarm

By connecting a piezo sounder to a digital output pin, we are going to create a wailing alarm sound. We are going to use the same principle that we used in Project 7 to make a pulsating lamp by creating a sine wave. But this time, we replace the LED with a piezo sounder or piezo disc. The full list of parts required can be found in Table 4-1. When choosing your piezo sounder, make sure that you do *not* purchase an active buzzer, that is, one with a built-in oscillator. You need a passive piezo sounder that uses AC voltage, not DC.

Parts Required

Table 4-1. *Parts Required for Project 11*

Piezo Sounder (or piezo disc)	
2-Way Screw Terminal	

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 connect it to the Arduino as in Figure 4-1. Next, connect your Arduino back to the USB cable and power it up.

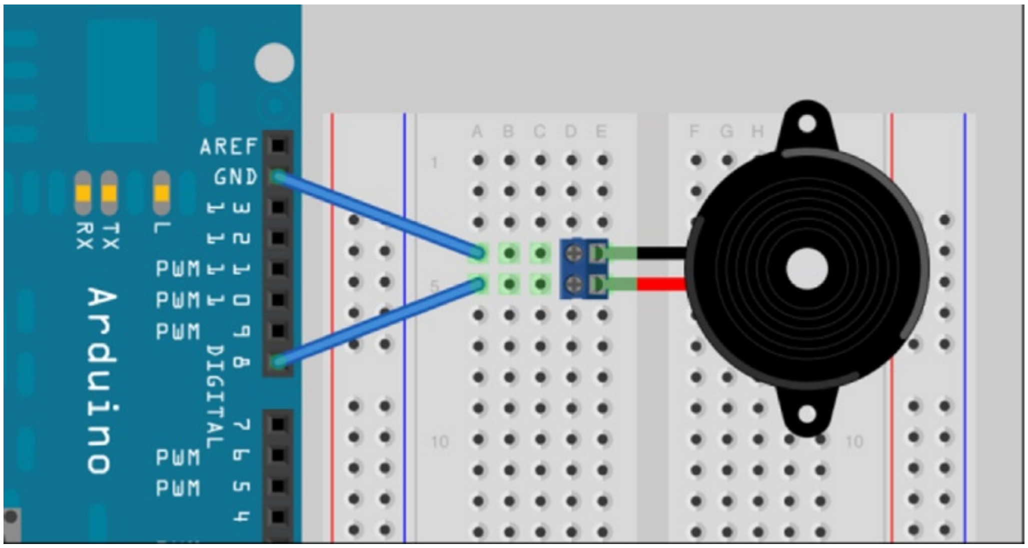


Figure 4-1. *The circuit for Project 11 – Piezo Sounder Alarm*

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);
        }
}
```

Once you have uploaded the code, there will be a slight delay, and then your piezo will start to emit sounds. If everything is working as planned, you will 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, so let's see how it works.

Project 11 – Piezo Sounder Alarm – Code Overview

First we set up two variables

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

The `sinVal` float variable will hold the sine value that will cause the tone to rise and fall in the same way that the lamp in Project 7 pulsated. The `toneVal` variable will be used to take the value in `sinVal` and convert it to a frequency we require.

In the `setup` function, we now set digital pin 8 to an output.

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

Then we move onto the main loop. As in Project 7, we set a `for` loop to run from 0 to 179 to ensure that the sine value does not go into the negative.

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

We convert the value of *x* into radians, 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));
```

We take 2,000 and add the `sinVal` multiplied by 1,000. This gives us a good range of frequencies for the rising and falling tone to go between as the sine wave rises and falls.

Next, we use the `tone()` command that was introduced in IDE 0018 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, we run a delay of 2 milliseconds in-between the frequency changes to ensure the sine wave rises and falls at the speed we require.

```
delay(2);
```

You may be wondering why we did not put the 2 milliseconds into the duration parameter of the tone() command, like this:

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

This is because our 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 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 on when you learn how to connect sensors to your Arduino. Then, you could activate an alarm when a sensor threshold has been reached, for example, if someone gets too close to an ultrasonic detector or if a temperature gets too high.

If you change the values of 2,000 and 1,000 in the toneVal calculation and the length of the delay, you can generate different alarm sounds. Experiment a little, and see what sounds you can make.

Project 11 – Piezo Sounder Alarm – Hardware Overview

We have used two new components in this project, a screw terminal and a piezo sounder. We have used the screw terminal as the wires from your piezo sounder or disc will be too thin and soft to insert into the breadboard. The screw terminal will have pins on it to enable you to push it into a breadboard.

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

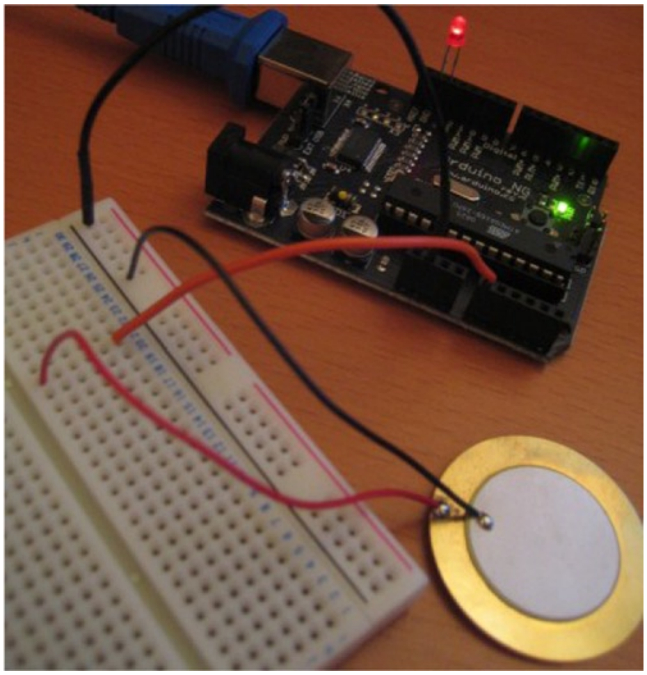


Figure 4-2. *A piezo disc and Arduino (Image courtesy of Patrick H. Lauke / splintered.co.uk)*

Piezoelectric materials, notably crystals and certain ceramics, have the ability to produce electricity when mechanical stress is applied to them. The effect finds useful applications in the production and detection of sound, generation of high voltages, electronic frequency generation, microbalances, and ultrafine focusing of optical assemblies.

The effect is also reversible, in that 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, and hence, cause a "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 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. We will be utilizing this feature of a piezo disc in Project 13 when we make a knock sensor. Before we get to that, we'll do one more project with a piezo as an output.

Project 12 – Piezo-Sounder Melody Player

Rather than using the piezo to make annoying alarm sounds, how about using it to play a melody? We are going to get our Arduino to play 'Oh My Darling Clementine'. Leave the circuit exactly as it was in Project 11. We are just going to change 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_F3, NOTE_F3, NOTE_F3,  NOTE_C3,
  NOTE_A3, NOTE_A3, NOTE_A3,  NOTE_F3,
  NOTE_F3, NOTE_A3, NOTE_C4,  NOTE_C4,  NOTE_AS3, NOTE_A3, NOTE_G3,
  NOTE_G3, NOTE_A3, NOTE_AS3, NOTE_AS3,
  NOTE_A3, NOTE_G3, NOTE_A3,  NOTE_F3,
  NOTE_F3, NOTE_A3, NOTE_G3,  NOTE_C3,  NOTE_E3,  NOTE_G3, NOTE_F3
};

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

```
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);
}
```

Once you have uploaded the code, there will be a slight delay and then your piezo will start to play a tune. You may recognize it as 'Oh My Darling Clementine'. A few new concepts are involved in this project so let's take a look.

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 something new to us, but is very simple and very useful, too. `#define` simply defines a token (name) and its value. For example, consider the following:

```
#define PI 3.14159265358979323846264338327950288419716939937510
```

This will allow you to substitute `PI` in any calculation instead of having to type out pi to 50 decimal places.
Here are another few examples:

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

This code means you can put a `TRUE` or `FALSE` into your code instead of a 0 or a 1. This makes logical statements a lot clearer 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 three main advantages to doing this instead of simply using the numbers.

- First, the code now 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.
- Second, if you change your display at a later date to a larger resolution, let's say, a 16 x 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 in which the display resolution appears. 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.

- Third, with the #defines, a third party reading the code no longer needs to know the specifics of the display attached to recognize which portions of the code are dealing with the display width and height.

In the case of Project 12, we create a whole set of `define` directives in which the tokens are the notes C3 through to B4 and the values are the frequencies required to create that note. The first note of our melody is F3 and its corresponding frequency is 173 hertz. This is middle F on the musical scale. Not all of the notes defined are used in our 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 in length, or a half, a quarter, an eighth, or a sixteenth of a bar. 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), and therefore we multiply then length of the bar, in our case, 1,500 milliseconds, by 0.25 to get the length of a quarter note.

```
1500 x QUARTER = 375 milliseconds
```

Next we define an integer array called `tune[]` and fill it with the notes for 'Oh My Darling Clementine'.

```
int tune[] = {
  NOTE_F3, NOTE_F3, NOTE_F3,  NOTE_C3,
  NOTE_A3, NOTE_A3, NOTE_A3,  NOTE_F3,
  NOTE_F3, NOTE_A3, NOTE_C4,  NOTE_C4, NOTE_AS3, NOTE_A3, NOTE_G3,
  NOTE_G3, NOTE_A3, NOTE_AS3, NOTE_AS3,
  NOTE_A3, NOTE_G3, NOTE_A3,  NOTE_F3,
  NOTE_F3, NOTE_A3, NOTE_G3,  NOTE_C3, NOTE_E3,  NOTE_G3, NOTE_F3
};
```

After this, we create another array, this time a float, that will hold the duration of the each note as it is played.

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

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

```
int length;
```

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

In our `setup` routine, we first set digital pin 8 to an output

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

then initialize the integer `length` with the number of notes in the array

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

and this is done using the `sizeof()` function.

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, we do not need to concern ourselves with bits and bytes. We will go into them later in the book.

So, our 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])
```

In our case, this is equivalent to

```
26 / 2 = 13
```

If you replace the tune in the project with one of your own, then 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 in which the length of the data types differ from those on the Arduino.

In the main loop, we 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 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 1,500 milliseconds multiplied by the note length (e.g., 0.25 for a quarter note, 0.125 for an eighth note, etc.).

Before the next note is played, we cease the tone generated on pin 8

```
noTone(8);
```

This is to ensure that when two notes that are the same 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, we run a delay of 5 seconds before repeating the melody.

```
delay(5000);
```

To create the notes for this tune, I used public domain sheet music for 'Oh My Darling Clementine' on the internet and typed the notes into the `tune[]` array, followed by the note lengths in the `duration[]` array. Notice 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, then change the value of 1,500 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, which could damage it.

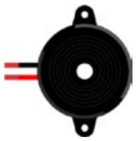
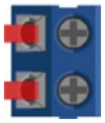
We are now going to use the piezo disc for another purpose and that is its ability to produce a current when the disc is squeezed or knocked. Utilizing this feature, we are going to make a knock sensor in Project 13.

Project 13 – Piezo Knock Sensor

A piezo disc works when an electric field (voltage) is applied across ceramic material in the disc, causing it to change shape and therefore make a sound (a click). The disc also works in reverse in that when the disc is knocked or squeezed, the force on the material causes an electric charge (voltage) to be generated. We can read that current using the Arduino and we are going to do that now by making a knock sensor.

Parts Required

Table 4-2. *Parts Required for Project 13*

Piezo Sounder (or piezo disc)	
2-Way Screw Terminal	
5mm LED (any color)	
1MΩ Resistor	
150Ω Current-Limiting 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 that you have the circuit in Figure 4-3. A piezo disc works better for this project than a piezo sounder. The resistor on the piezo is there to discharge any piezo capacitance built up in the piezo while it is being used.

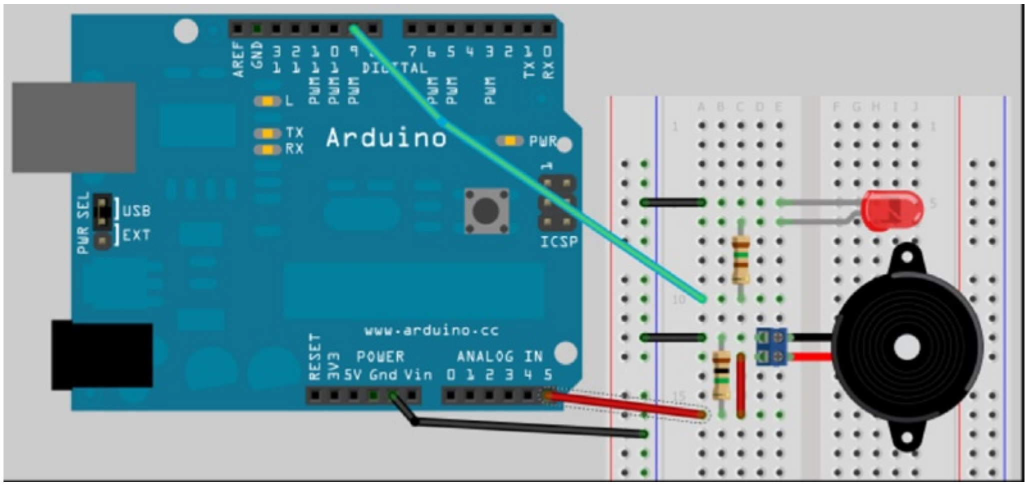


Figure 4-3. *The circuit for Project 13 – Piezo Knock Sensor*

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 show you 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. The threshold value in the code was set for the 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

We haven't learnt any new code commands in this project, but we will go over how it works anyway.

First, we set up the necessary variables for our 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 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 our main loop, we first read the analog value from 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 we have set, i.e., to determine that it really is a knock or squeeze (the piezo is very sensitive as you can see if you set the threshold to a very low value). If it is, then it sets `ledValue` to 255, which is the maximum voltage out of PWM pin 9.

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

We then write that value to PWM pin 9. As `ledValue` is a float, we "cast" it to an integer, as the `analogWrite` function can only accept an integer and not a floating value. (*Casting* simply means "to convert to a different data type.")

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

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

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

We want the LED to dim gently, and hence, we use a float to store the brightness value of the LED instead of an integer. That way, we can reduce its value by a small amount (in our case 0.05), meaning it will take a little while as the main loop repeats for the value of `ledValue` to reach zero, when the LED will be at its dimmest and off. If you want the LED to dim slower or faster, then increase or decrease this value.

Finally, we don't want `ledValue` to go below zero as PWM pin 9 can only output a value from 0 to 255, so we check if it is smaller, or equal to zero, and if it is smaller, we 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.

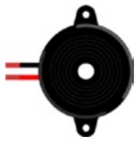
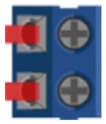
Now let's introduce a new sensor, the light-dependent resistor, or LDR.

Project 14 – Light Sensor

We are now going to look at 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 will have a very high resistance. As photons (light) land on the detector, the resistance will decrease. The more light there is, the lower the resistance will be. By reading the value from the sensor, we can detect if it is light, dark, or anywhere in-between. In this project, we will use an LDR to detect light and a piezo sounder to give audible feedback of the amount of the light detected.

Parts Required

Table 4-3. *Parts required for Project 14*

Piezo Sounder (or Piezo Disc)	
2-Way Screw Terminal	
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 in Figure 4-4. Check all of your connections before reconnecting the power to the Arduino.

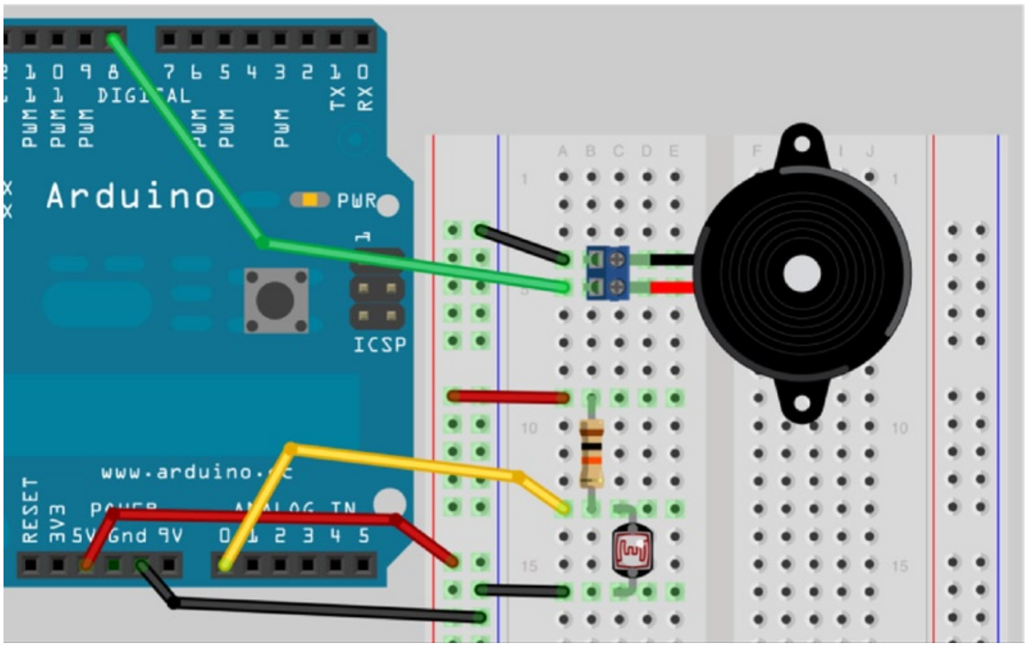


Figure 4-4. *The circuit for Project 14 – Light Sensor*

The LDR can be inserted any way around; it does not have polarity. I found a 10kΩ resistor worked well with my LDR. You may need to try different resistor settings until you find one suitable for the LDR you have. 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 below.

Listing 4-4. Code for Project 14

```
// 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 will start to make short beeps. The gap between the beeps will be long if the LDR is in the shade and will be short when bright light is shone on it. This will give a Geiger counter-type effect, but with our detector detecting photon particles instead of ionizing radiation. 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, but move your LDR around to point it at dark and light areas.

The code for Project 14 is very simple and you should be able to work out how it works yourself without any help. However, we will take a look at 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 photocell or photoresistor. LDRs . come in various shapes and sizes (see Figure 4-5) and in different ranges of resistance.

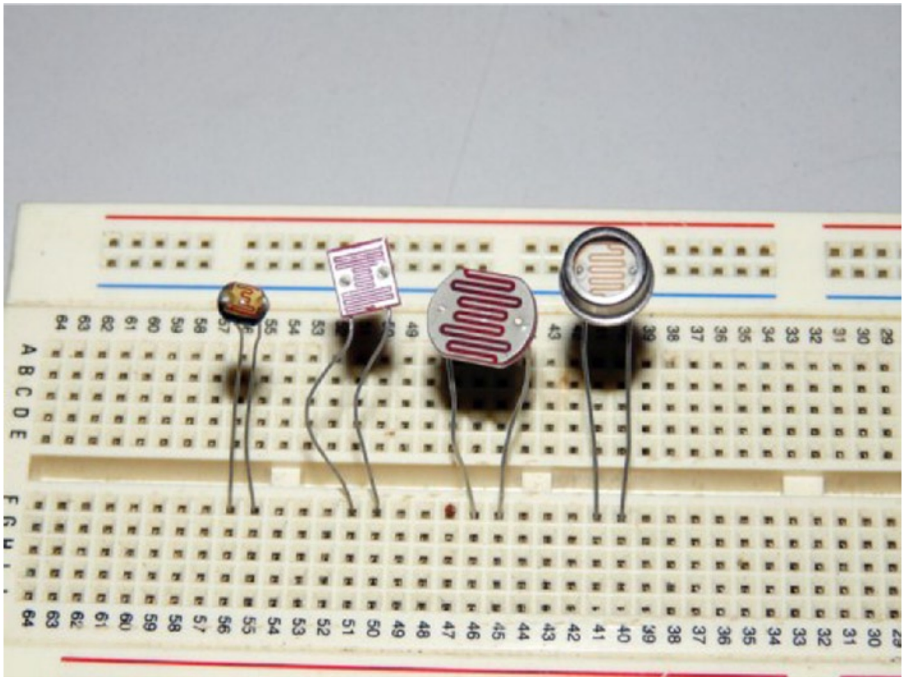


Figure 4-5. *Different kinds of LDRs (image by cultured_society2nd)*

Each of the legs on the LDR go to an electrode on either side. Between the 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 causes it to lose 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 thing in your circuit is a voltage divider (also known as a potential divider). This is where the resistor comes in. By using two resistors in series and taking the voltage across just one of them, you can reduce the voltage going into the circuit. In our case, we have a resistor of a fixed value (10kΩ or so) and a variable resistor in the form of a LDR. As it happens, the potentiometer used in Project 6 was also a voltage divider. Let us take a look at a standard voltage divider circuit using resistors and see how it works. In Figure 4-6 we have a voltage divider using two resistors.

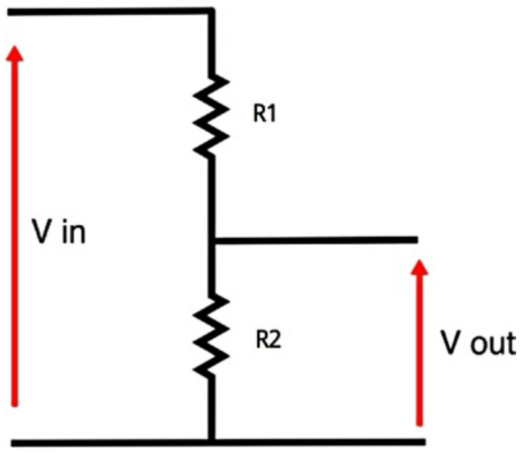


Figure 4-6. *A voltage divider*

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 is when measured across R2 is

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

So, if in the circuit in Figure 4-6 we had 100Ω resistors (or 0.1kΩ) for both R1 and R2, and 5V going into Vin, our formula would be

$$\frac{100}{100 + 100} \times 5 = 2.5 \text{ volts}$$

Let’s do it again with 470Ω resistors

$$\frac{470}{470 + 470} \times 5 = 2.5 \text{ volts}$$

We get 2.5 volts again. This demonstrates that the value of the resistors is not important, but the ratio between them is. However, the value of the resistors will affect the amount of current drawn by the divider. The divider using 100 ohm resisters would use 4.7 times as much current as the one using 470 ohm resistors. Let's try a 1kΩ and a 500Ω resistor.

$$\frac{500}{500 + 1000} \times 5 = 1.66 \text{ volts}$$

With the bottom resistor half the value of the top one, we 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.

$$\frac{2000}{2000 + 1000} \times 5 = 3.33 \text{ volts}$$

which is two thirds of the voltage going in. So, let's apply this to the LDR. We shall presume that the LDR has a range of around 100kΩ when in the dark and 10kΩ in bright light. Table 4-4 shows what voltages we will get out of our circuit as the resistance changes.

Table 4-4. *Vout Values for Example LDR with 5V as Vin*

Light Level	R1	R2 (LDR)	Vout
Darkest	10kΩ	100kΩ	4.55v
25%	10kΩ	73kΩ	4.40v
50%	10kΩ	45kΩ	4.09v
75%	10kΩ	28kΩ	3.68v
Lightest	10kΩ	10kΩ	2.5v

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

Summary

In Chapter 4, 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

CHAPTER 5

Driving a DC Motor

It is now time to move onto controlling a DC motor. If you ever plan on building a robot or any device that requires the use of a motor, then the skills you are about to learn will be essential. Driving a motor requires currents higher than the Arduino can safely provide from its outputs, so we will need to make use of transistors and diodes to ensure that we have enough current for the motor and diodes for protection of the Arduino.

The hardware overview will explain how these work. For our first project, we will control a motor using a very simple method and will then go on to use the very popular L293D Motor Driver chip. Later in the book we will also learn how to use these to control a stepper motor.

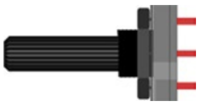
Project 15 – Simple Motor Control

We are first going to simply control the speed of a DC motor, in one direction, using a power transistor, diode, and external power supply to power the motor and a potentiometer to control the speed. Any suitable NPN power transistor designed for high current loads can replace the TIP120 transistor. However, I would highly recommend you use a power Darlington-type transistor. Make sure you choose a transistor that can handle the voltage and current your motor will draw. It may be necessary to fit a heat sink to the transistor if it is pulling more than about an amp.

The external power supply can be a set of batteries or a "wall wart"-style external DC power supply. The power source must have enough voltage and current to drive the motor. The voltage must not exceed that required by the motor. For my testing purposes I used a DC power supply that provided 5V at 500mA. This was enough for the 5V DC motor I was using. If you use a power supply with voltage higher than the motor can take, you may damage it permanently. Table 5-1 lists the parts required for the next project.

Parts Required

Table 5-1. *Parts Required for Project 15*

DC Motor	
10kΩ Potentiometer	
TIP120 Transistor*	
1N4001 Diode*	
Jack Plug	
External Power Supply	
1K Resistor	

**or suitable equivalent*

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now, take the required parts and connect them up as in Figure 5-1. It is essential for this project that you check and double check all of your connections are as they should be before supplying power to the circuit, as failure to do so may result in damage to your components or even your Arduino. The diode, in particular, is essential to protect the Arduino from back EMF, which we will explain later.

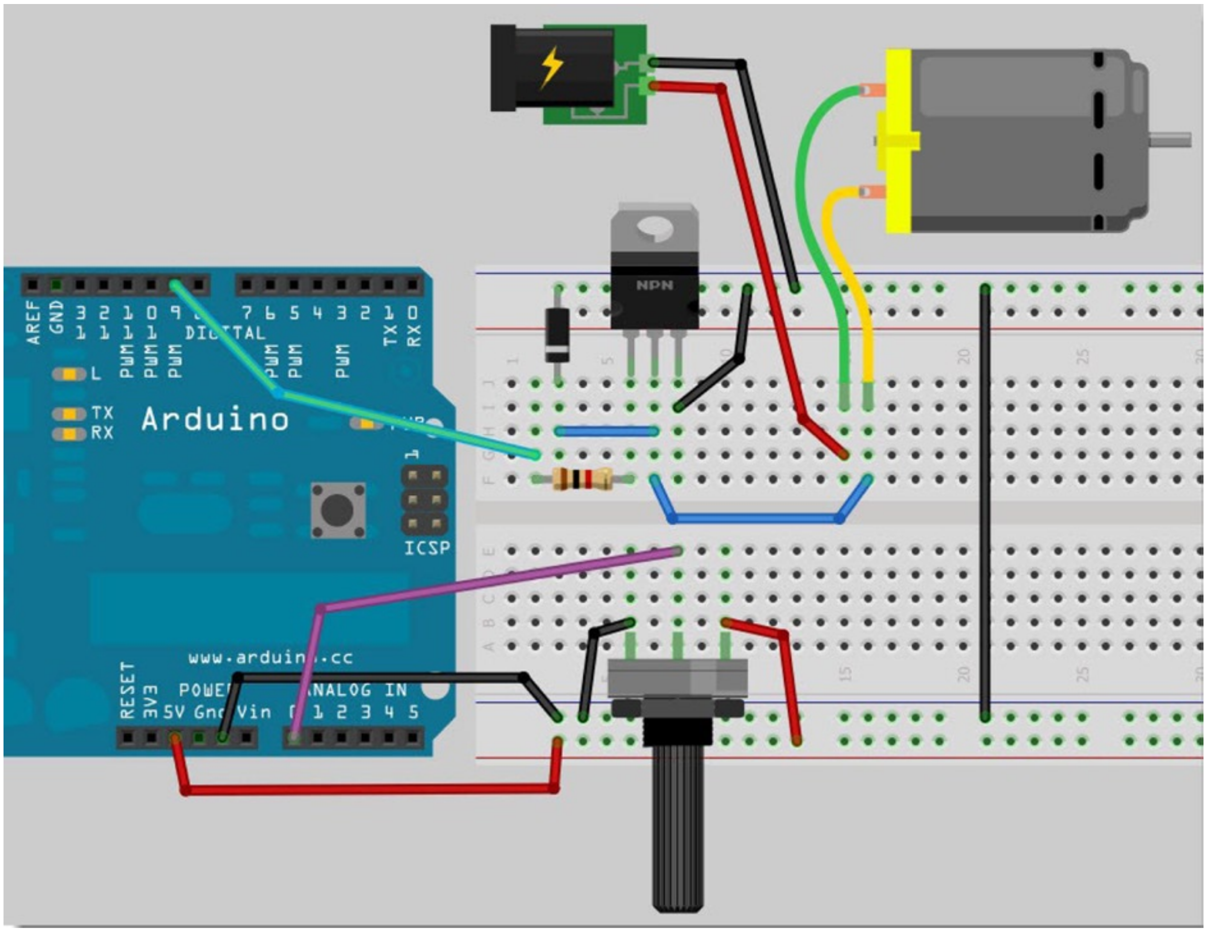


Figure 5-1. *The circuit for Project 15 – Simple Motor Control*

Enter the Code

Open up your Arduino IDE and type in the code from listing 5-1:

Listing 5-1. Code for Project 15

```
// Project 15 - Simple Motor Control

int potPin = 0;           // Analog in 0 connected to the potentiometer
int transistorPin = 9;    // PWM Pin 9 connected to the base of the transistor
int potValue = 0;         // value returned from the potentiometer
```

```
void setup() {
  // set  the transistor pin as output:
  pinMode(transistorPin, OUTPUT);
}

void loop() {
  // read the potentiometer, convert it to 0 - 255:
  potValue = analogRead(potPin) / 4;
  // use that to control the transistor:
  analogWrite(transistorPin, potValue);
}
```

Before uploading your code, disconnect the external power supply to the motor and ensure the potentiometer is turned fully counterclockwise. Now upload the code to the Arduino. Once the code is uploaded, connect the external power supply. You can now turn the potentiometer to control the speed of the motor.

Project 15 – Simple Motor Control – Code Overview

First we declare three variables that will hold the value for the analog pin connected to the potentiometer, the PWM pin connected to the base of the transistor, and one to hold the value read back from the potentiometer from analog pin 0.

```
int potPin = 0;           // Analog in 0 connected to the potentiometer
int transistorPin = 9;    // PWM Pin 9 connected to the base of the transistor
int potValue = 0;         // value returned from the potentiometer
```

In the setup( ) function we set the pin mode of the transistor pin to output.

```
void setup() {
  // set  the transistor pin as output:
  pinMode(transistorPin, OUTPUT);
}
```

In the main loop potValue is set to the value read in from analog pin 0 (the potPin) and then divided by 4.

```
potValue = analogRead(potPin) / 4;
```

We need to divide the value read in by 4 as the analog value will range from 0 for 0 volts to 1,023 for 5 volts. The value we need to write out to the transistor pin can only range from 0 to 255, so we divide the value of analog pin 0 (max 1023) by 4 to give the maximum value of 255 for setting the digital pin 9 (Using analogWrite so we are using PWM).

The code then writes out to the transistor pin the value of the pot.

```
analogWrite(transistorPin, potValue);
```

In other words, when you rotate the potentiometer, different values ranging from 0 to 1,023 are read in and these are converted to the range 0 to 255, and then that value is written out (via PWM) to digital pin 11, which changes the speed of the DC motor. Turn the pot all the way to the left and the motor goes off; turn it to the right and it speeds up until it reaches maximum speed.

The code is very simple and we have learned nothing new. Let us now take a look at the hardware used in this project and see how it all works.

Project 15 – Simple Motor Control – Hardware Overview

The circuit is essentially split into two sections. Section 1 is our potentiometer, which is connected to 5V and ground with the center pin going into analog pin 0. As the potentiometer knob is rotated, the resistance divider changes to allow voltages from 0 to 5V to come out of the center pin, whose value is read using analog pin 0.

The second section is what controls the power to the motor. The digital pins on the Arduino give out a maximum of 40mA (milliamps). However, the datasheet says the highest output current that is guaranteed is only 20mA. You should never be operating at absolute maximum. A DC motor may require around 500mA to operate at full speed, and this is obviously too much for the Arduino. If we were to try to drive the motor directly from a pin on the Arduino serious and permanent damage could occur.

Therefore, we need to find a way to supply it with a higher current. We therefore take power from an external power supply, which will give us enough current to power the motor. We could use the 5V output from the Arduino, which can provide up to 800mA when connected to an external power supply, and less when powered by USB. However, Arduino boards are expensive and it is all too easy to damage them when connecting them up to high current, electrically noisy loads such as DC motors. We will therefore play it safe and use an external power supply. Also, your motor may require 9V or 12V or higher voltages, and this is beyond anything the Arduino can supply.

However, this project controls the speed of the motor, so we need a way to control the power to speed up or slow down the motor. This is where the TIP-120 transistor comes in.

Transistors

A transistor is a power amplifier that can also be used as a digital switch. In our circuit we use it as a switch. The electronic symbol for the type of transistor we are using in this project looks like Figure 5-2.

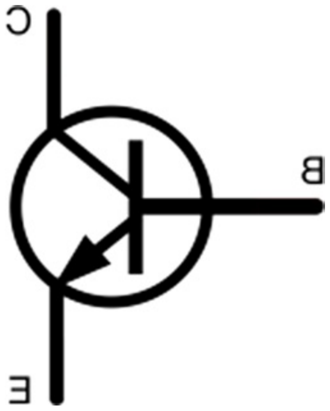


Figure 5-2. *The symbol for an NPN transistor*

The transistor has three legs: one is the base, one is the collector, and the third, the emitter. These are marked as C, B and E on the diagram. In our circuit we have up to 5 volts going via a resistor to the base via digital pin 9. The collector is connected to one terminal on the motor. The emitter is connected to ground. Whenever we apply a voltage to the base, via pin 9, this makes the transistor turn on, allowing current to flow through it between the emitter and collector, thus powering the motor that is connected in series with this circuit. By applying a small current at the base we can control a larger current between the emitter and collector.

Transistors are the key components in just about any piece of modern electronic equipment. Many people consider the transistor to be the greatest invention of the twentieth century. For bipolar transistors, current flowing through the base lead allows a (roughly) proportional, but much larger amount of current to flow from the collector to the emitter. The ratio between the base current and the corresponding collector current is known as the gain of the transistor. The gain of the power Darlington transistor we used in this project is about 1,000, meaning a small (1mA) current at the base can control an amp of current flow between the emitter and collector. As we are pulsing our signal, the transistor is turned on and off many times per second and it is this pulsed current that controls the speed of the motor.

Motors

A motor is an electromagnet and it has a magnetic field while power is supplied to it. When the power is removed, the magnetic field collapses and this collapsing field can produce a reverse voltage to go back up its wiring. This could seriously damage your Arduino, and that is why the diode has been placed the way it is on the circuit. The white stripe on the diode normally goes to ground. Power will flow from the positive side to the negative side. As we have it the wrong way around, no power will flow down it at all. However, if the motor were to produce a “back EMF” (electromotive force) and send current back down the wire, the diode would act as a valve and prevent it from doing so. The diode in our circuit is therefore put in place to protect your Arduino.

In a given circuit at a given time, a diode may be forward-biased (anode more positive than cathode) or reverse-biased (cathode more positive than anode). Simple diodes conduct well when forward-biased, and conduct very little when reverse-biased. The diode behaves differently in one situation than in the other, but its proper orientation depends on what we want it to accomplish in a given circuit. In this case, we want to provide a safe path (i.e., not through the Arduino or transistor) for the electricity generated by the collapsing field in the motor. The voltage across the motor from the collapsing field is the opposite polarity from the voltage that created the field. Placing a diode with the cathode (white stripe) to the power supply side of the motor and the anode (other lead) to the transistor side of the motor will accomplish this. When we have the motor turned on, the diode is reverse-biased, so it passes a negligible leakage current. When the transistor turns off, the induced voltage from the motor forward biases the diode, which gives the induced current a safe discharge path.

If you were to connect a DC motor directly to a multimeter with no other components connected to it, and then spin the shaft of the motor, you will see that the motor generates a current. This is exactly how wind turbines work. When we have the motor powered up and spinning, and we then remove the power, the motor will keep on spinning under its own inertia until it comes to a stop. In that short time while the motor is spinning without power applied, it is generating a current. This is called “back EMF,”as I mentioned above. The diode acts as a valve and ensures that the electricity does not flow back to the circuit and damage other components.

Diodes

Diodes are one-way valves. In exactly the same way that a non-return valve in a water pipe will allow water to flow in one direction, but not in the other, the diode allows a current to flow in one direction, but not the other. We have already come across diodes when we used LEDs. An LED is a light emitting diode and has a polarity. You connect the positive terminal of the power, usually to the long lead on the LED to make the LED turn on. If you turn the LED around, the LED will not only fail to illuminate but will also prevent electricity from flowing across its terminals. The 1N4001 diode has a white band around it next to the cathode (negative) lead. Other diodes may mark the cathode differently, such as a black band on glass-body or metal-case diodes. Imagine the band as a barrier. Electricity flows through the diode from the terminal that has no barrier. When you reverse the voltage and try to make it flow through the side that has the band, the current will be stopped.

Diodes are essential in protecting circuits from a reverse voltage, which occurs if you connect a power supply the wrong way around or if a voltage is reversed, such as the back EMF in our circuit. Therefore, always try to use them in your circuits whenever there is a danger of the power being reversed, either by user error or via phenomena such as back EMF.

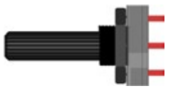
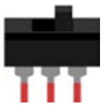
Project 16 – Using an L293D Motor Driver IC

In our previous project, we used a transistor to control the motor. In this project, we are going to use a very popular motor driver IC called an L293D. The advantage of using this chip is that we can control two motors at the same time, as well as control their direction. The chip can also be used to control a stepper motor, as we will find out in Project 28. There is also a pin-for-pin compatible chip available known as the SN754410, which can also be used and has a higher current rating. The IC has its own internal diodes, so we do not need one for this project.

Parts Required

Table 5-2 shows the parts required for Project 16.

***Table 5-2.** Parts Required for Project 16*

DC Motor	
L293D or SN754410 Motor Driver IC	
10KΩ Potentiometer	
Slide Switch	
10KΩ Resistor	
Heatsink	
2 x 0.1uF capacitors	

Connect It Up

First, make sure your Arduino is powered off by unplugging it from the USB cable. Now take the required parts and connect them up as in Figure 5-3. Again, check the circuit thoroughly before powering it up. The L293D can get hot when in use, and therefore a heatsink is recommended. Glue the heatsink to the top of the chip using a special thermal epoxy glue, thermally conductive double-sided tape or use a clip-on style heat sink. The larger the heatsink, the better. Be warned that the temperature can get hot enough to melt the plastic on a breadboard or any wires touching it. Do not touch the heatsink as you may burn yourself and ensure you do not leave the circuit powered up and unattended in case it overheats. It may be prudent to use stripboard instead of a breadboard for this project to save damaging your breadboard due to heat.

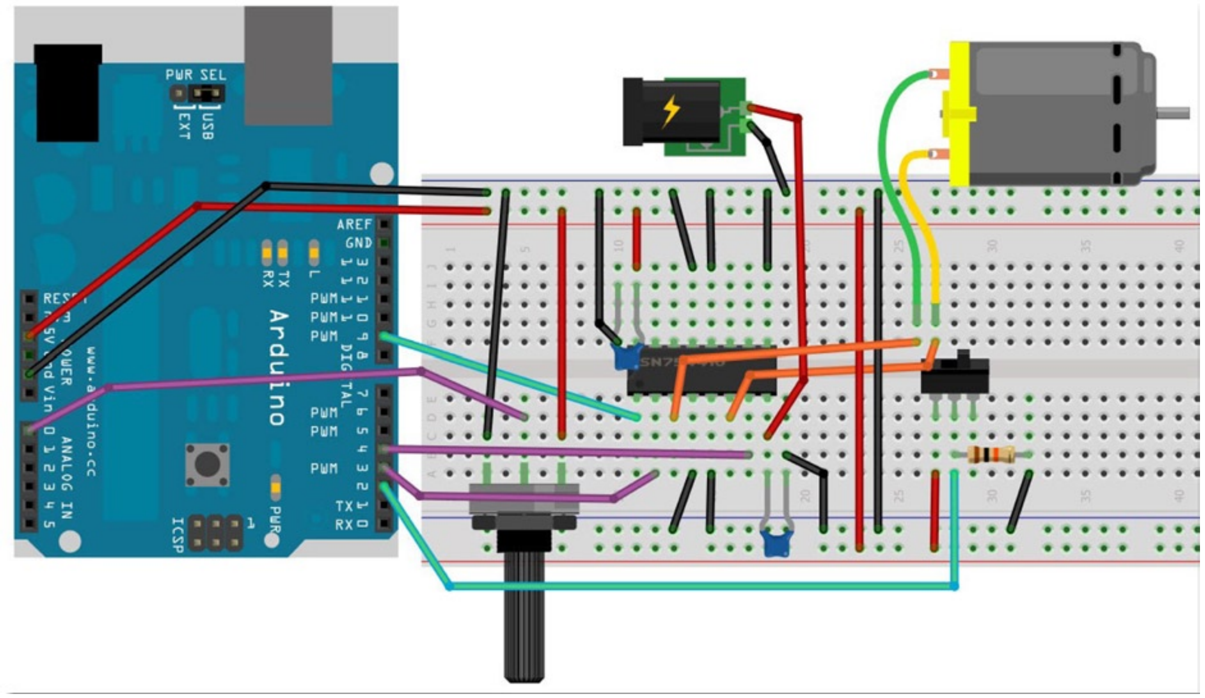


Figure 5-3. *The circuit for Project 16*

Enter the Code

Once you are satisfied that your circuit is connected up correctly, upload the code in Listing 5-2. Do not connect the external power supply at this stage.

Listing 5-2. Code for Project 16

```
// Project 16 - Using an L293D Motor Driver IC
#define switchPin 2 // switch input
#define motorPin1 3 // L293D Input 1
#define motorPin2 4 // L293D Input 2
#define speedPin 9  // L293D enable pin 1
#define potPin 0    // Potentiometer on analog Pin 0
int Mspeed = 0;     // a variable to hold the current speed value

void setup() {
//set switch pin as INPUT
pinMode(switchPin, INPUT);

// set remaining pins as outputs
pinMode(motorPin1, OUTPUT);
pinMode(motorPin2, OUTPUT);
pinMode(speedPin, OUTPUT);
}
```

```
void loop() {
  Mspeed = analogRead(potPin)/4;   // read the speed value from the potentiometer
  analogWrite (speedPin, Mspeed);  // write speed to Enable 1 pin
  if (digitalRead(switchPin)) {    // If the switch is HIGH, rotate motor clockwise
    digitalWrite(motorPin1, LOW);  // set Input 1 of the L293D low
    digitalWrite(motorPin2, HIGH); // set Input 2 of the L293D high
  }
  else {  // if the switch is LOW, rotate motor anti-clockwise
    digitalWrite(motorPin1, HIGH); // set Input 1 of the L293D low
    digitalWrite(motorPin2, LOW);  // set Input 2 of the L293D high
  }
}
```

Once the code has finished uploading, set the potentiometer at its midpoint and plug in the external power supply. The motor will now rotate and you can adjust its speed by turning the potentiometer. To change direction of the motor, first set the speed to minimum, then flick the switch. The motor will now rotate in the opposite direction. Again be careful of that chip, as it will get very hot once powered up.

Project 16 – Using an L293D Motor Driver IC – Code Overview

The code for this project is very simple. First we define the pins we are going to put to use on the Arduino.

```
#define switchPin 2 // switch input
#define motorPin1 3 // L293D Input 1
#define motorPin2 4 // L293D Input 2
#define speedPin 9  // L293D enable pin 1
#define potPin 0    // Potentiometer on analog Pin 0
```

Then set an integer to hold the speed value read from the potentiometer.

```
int Mspeed = 0;  // a variable to hold the current speed value
```

In the setup function we then set the appropriate pins to either inputs or outputs.

```
pinMode(switchPin, INPUT);

pinMode(motorPin1, OUTPUT);
pinMode(motorPin2, OUTPUT);
pinMode(speedPin, OUTPUT);
```

In the main loop we first read in the value from the potentiometer connected to analog pin 0 and store it in Mspeed.

```
Mspeed = analogRead(potPin)/4; // read the speed value from the potentiometer
```

Then we set the PWM value on PWM pin 9 to the appropriate speed.

```
analogWrite (speedPin, Mspeed); // write speed to Enable 1 pin
```

Then we have an if statement to decide if the value read in from the switch pin is either HIGH or LOW. If it is HIGH, then output 1 on the L293D is set to LOW and output 2 is set to HIGH. This will be the same as output 2 having a positive voltage and output 1 being ground, causing the motor to rotate in one direction.

```
if (digitalRead(switchPin)) {      // If the switch is HIGH, rotate motor clockwise
    digitalWrite(motorPin1, LOW);  // set Input 1 of the L293D low
    digitalWrite(motorPin2, HIGH); // set Input 2 of the L293D high
  }
```

If the switch pin is LOW, then output 1 is set to HIGH and output 2 is set to LOW, reversing the direction of the motor.

```
else {                            // if the switch is LOW, rotate motor anti-clockwise
  digitalWrite(motorPin1, HIGH); // set Input 1 of the L293D low
  digitalWrite(motorPin2, LOW);  // set Input 2 of the L293D high
}
```

The loop will now repeat, checking for a new speed value or a new direction chosen and setting the appropriate speed and direction pins. As you can see, using the motor driver IC is not at all as daunting as you might have thought at first. In fact, it has made your life a lot easier. Trying to recreate the above circuit and code without it would have been far more complex. Never be intimidated by ICs as they are usually a lot simpler than they first appear. A slow and careful read of their datasheets will reveal their secrets. Let's see how the new components introduced in this project work.

Project 16 – Using an L293D Motor Driver IC – Hardware Overview

The new component we have come across in Project 16 is the motor driver IC. This will be either the L293D or the SN754410, depending on what you have chosen (there are other chips available and a little research on the Internet will find other pin-compatible motor drivers). The L293D is recommended as the SN754410 can run hot and require a heat sink.

The L293D is what is known as a dual H-bridge. An H-bridge is a useful and simple electronic concept. You could make your own H-bridge using switches. See Figure 5-4.

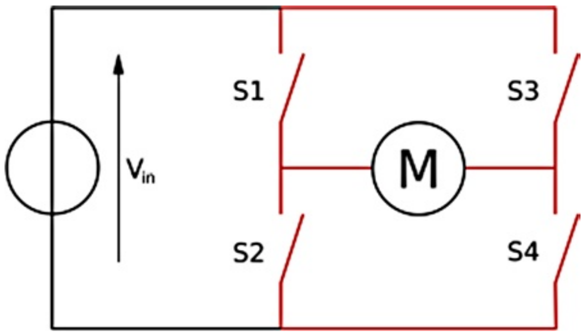


Figure 5-4. *A H-bridge made of switches (image by Cyril Buttay)*

In Figure 5-4, a motor is connected to four switches. The configuration is called an *H-bridge* because it resembles a letter H, with the load bridge as the center. Now take a look at Figure 5-5.

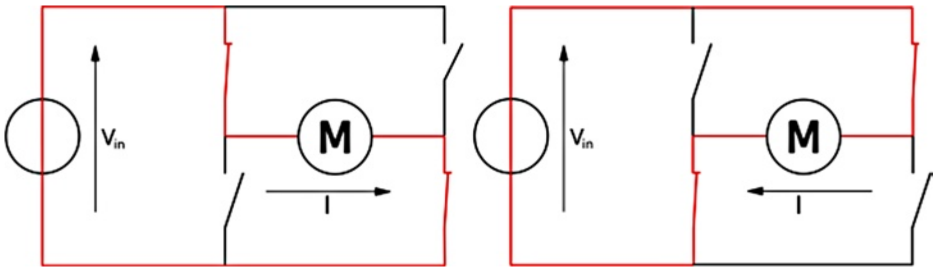


***Figure 5-5.** Changing motor direction on the H-bridge (image by Cyril Buttay)*

On the left hand side, we have the top left and the bottom right switches closed. When closed, the current will flow across the motor from left to right and the motor will rotate. If we then open those switches and close the top right and bottom left switches instead, the current will flow across the motor in the opposite direction and hence cause it to rotate in the opposite direction. This is how an H-bridge works.

The motor driver IC is made up of 2 H-bridges, and instead of switches, it uses transistors. Just as the transistor in project 15 was used as a switch, the transistors inside the H-bridge switch on and off in the same configuration as in Figure 5-5 to make your motor rotate in either direction. The chip is a dual H-bridge as it has two H-bridges inside it; thus, you are able to control two motors and their directions at the same time.

If you wanted to, you could make your own H-bridge out of transistors and diodes and it would do the same job as the L293D. The L293D has the advantage of being tiny and having all those transistors and diodes packaged up inside a small space.

The L293D has two power supply pins in addition to the four ground pins. Pin 16 (Vcc1) is the supply voltage for the logic circuitry of the chip, and should be fed from the Arduino's 5V supply. Pin 8 (Vcc2) is the supply voltage for the power circuitry of the chip (which actually drives the motor). The grounds of both supplies and all four ground pins of the chip should be connected together. The two H-bridges on the chip act independently, sharing only the two power supplies. Pin 1 serves as an enable for the H-bridge controlled by pins 2 and 7. Pin 9 serves as an enable for the H-bridge controlled by pins 10 and 15. If an enable input is connected to a logic low (or ground), all four switches of the corresponding H-bridge will be turned off (open). If an enable is a logic high (> 2.4 V) the pair of inputs for the corresponding H-bridge will control which of the 4 switches of H bridge are turned on, as follows:

H-Bridge 1-2 (enable controlled by pin 1)

pin 2	pin 7	Output pin 3	Output pin 6	Description
L	L	Connects to Ground	Connects to Ground	Motor off (brake)
L	H	Connects to Ground	Connects to Vcc2	Motor on (one direction)
H	L	Connects to Vcc2	Connects to Ground	Motor on (opposite direction)
H	H	Connects to Vcc2	Connects to Vcc2	Motor off (brake)

H-bridge 3-4 (enable controlled by pin9)

pin 10	pin 15	Output pin 11	Output pin 14	Description
L	L	Connects to Ground	Connects to Ground	Motor off (brake)
L	H	Connects to Ground	Connects to Vcc2	Motor on (one direction)
H	L	Connects to Vcc2	Connects to Ground	Motor on (opposite direction)
H	H	Connects to Vcc2	Connects to Vcc2	Motor off (brake)

The minimum supply voltage is 4.5 volts (applies to both supplies, Vcc1 and Vcc2). The maximum supply voltage for Vcc1 is 5.5 volts on the 754410 (higher on the L293D). The maximum supply voltage for Vcc2 (motor supply) is 36 volts. If Vcc2 is below 4.5 volts, strange things may happen on outputs, and chips may get hot.

Transistors are not perfect switches. When the transistors in the L293D are turned on, there will be a voltage drop across them, so the across the motor will usually be 1.5 to 2.5 volts less than Vcc2. (That 1.5 to 2.5 volts is what causes the chip to get warm.) The higher the motor current, the more power will be lost in the transistors. This is one of those facts of life. When dealing with more powerful motors, better driver chips and heat sinks are needed.

EXERCISE

Now try out the following exercise

- Exercise 1: Instead of just one motor, try connecting up two, with two direction switches and potentiometers to control the direction and speed. Figure 5-6 shows the pinouts of the chip.

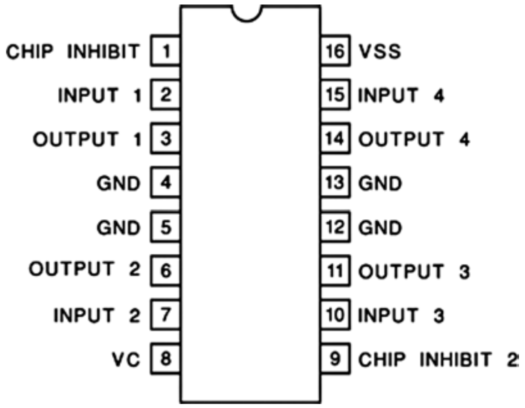


Figure 5-6. *The pinouts for the L293D (or SN754410)*

Summary

Chapter 5 showed you how to use motors in your projects. It also introduced you to the important concepts concerning transistors and diodes; you will use both of these components a lot in your projects, especially if you are controlling devices that require larger voltages or currents than the Arduino can supply.

You now also know how to construct an H-bridge and how to use it to change the direction of a motor. You have also used the popular L293D motor driver IC; you will work with this chip later on in the book when you look at a different kind of motor. The skills required to drive a motor are vital if you are going to build a robot or a radio-controlled vehicle of any kind using an Arduino.

Subjects and concepts covered in Chapter 5

- Using a transistor to power high power devices
- Controlling a motor's speed using PWM
- The current supplied by an Arduino digital pin
- Using an external power supply is essential for high power devices

- The concept of transistors and how they work
- Motors and how they work
- Using a diode to avoid back EMF
- How a diode works
- How motors can be used to generate power
- How vital diodes can be in protecting a circuit
- How to power a motor with an L293D motor driver IC
- Why heatsinks are essential for dissipating heat from ICs
- The concept of a H-bridge and how it can be used to change motor direction

CHAPTER 6

Binary Counters and Shift Register I/O

We are now going to go back to controlling LEDs. This time, however, we will not be driving them directly from the Arduino. Instead we are going to use a fantastic little chip known as a shift register. These ICs (integrated circuits) will enable us to control eight separate LEDs using just three pins from the Arduino and, as you will see in Project 18, a total of 16 LEDs, again using just three pins from the Arduino.

To demonstrate the way the data is output from these chips, we are going to create two binary counters, first using a single shift register and then moving onto two chips cascaded (you will learn what cascading is in Project 18). Chapter 6 is going to delve into some pretty advanced material, so you might want to take a deep breath before we dive in.

Project 17 – Shift Register 8-Bit Binary Counter

In Project 17, we are going to use additional ICs (integrated circuits) in the form of shift registers, to enable us to drive LEDs to display a binary counter (we will explain what binary is shortly). In this project, we will drive eight LEDs independently using just three output pins from the Arduino.

Parts Required

The parts required for Project 17 are shown in Table 6-1.

Table 6-1. *Parts Required for Project 17*

1 x 74HC595 Shift Register IC	
8 x 560Ω Resistors*	
8 x 5mm LEDs	
0.1uF Capacitor	

**or suitable equivalent*

Connect It Up

Examine the diagram carefully. Connect the 5V to the top rail of your breadboard and the ground to the bottom. The chip has a small dimple on one end; this dimple goes to the left. Pin 1 is below the dimple, pin 8 at bottom right, pin 9 at top right, and pin 16 at top left.

You now need wires to go from the 5V supply to pins 10 and 16, as well as wires from ground to pins 8 and 13.

A wire goes from digital pin 8 to pin 12 on the IC. Another one goes from digital pin 12 to pin 11 on the IC, and finally, one from digital pin 11 to pin 14 on the IC.

The 8 LED's have a 560Ω resistor between the cathode and ground. The anode of LED 1 goes to pin 15. The anode of LEDs 2 to 8 goes to pins 1 to 7 on the IC.

You will need a bypass capacitor (otherwise known as a decoupling capacitor) to go between the power supply pin and ground. Make sure its voltage rating is higher than the supply voltage being used. The leads should be kept short, with the capacitor as close to the chip as possible. The purpose of this capacitor is to reduce the effect of electrical noise on the circuit.

Once you have connected everything up as in Figure 6-1, check that your wiring is correct and the IC and LEDs are the right way around. Then enter the code.

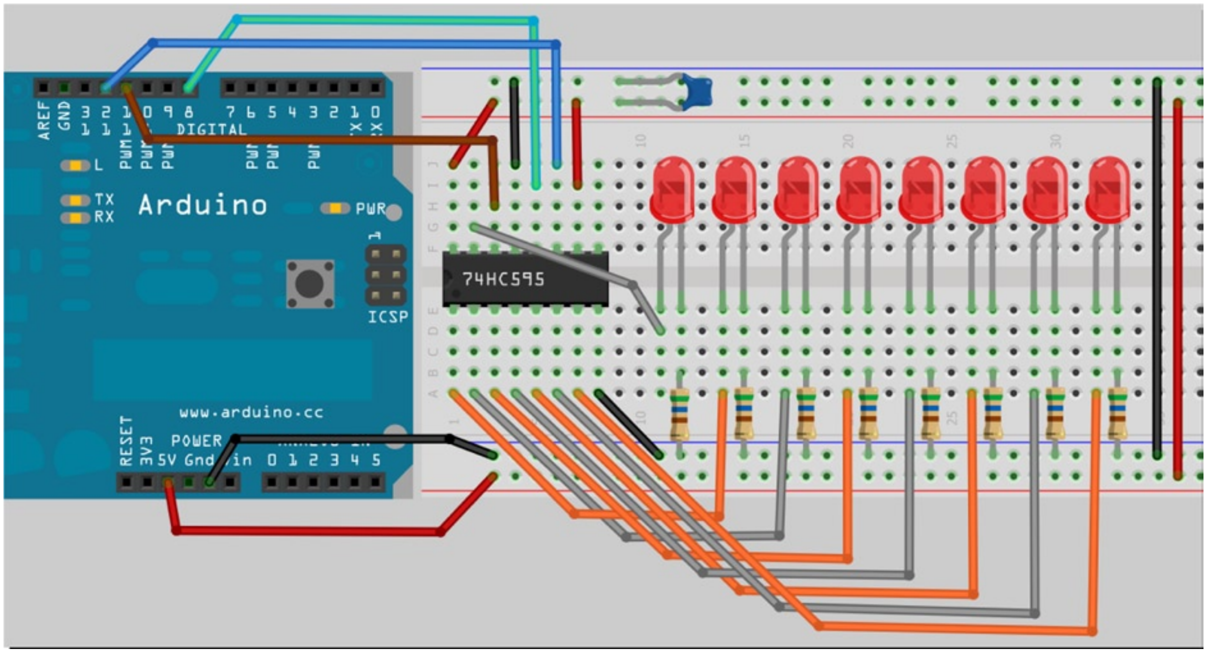


Figure 6-1. *The circuit for Project 17 – Shift Register 8-Bit Binary Counter*

Enter the Code

Enter the following code in Listing 6-1 and upload it to your Arduino. Once the code runs, you will see the LEDs turn on and off individually as they count in binary every second from 0 to 255, then start again.

Listing 6-1. Code for Project 17

```
// Project 17

int latchPin = 8;  //Pin connected to Pin 12 of 74HC595 (Latch)
int clockPin = 12; //Pin connected to Pin 11 of 74HC595 (Clock)
int dataPin = 11;  //Pin connected to Pin 14 of 74HC595 (Data)
```

```
void setup() {
	//set pins to output
	pinMode(latchPin, OUTPUT);
	pinMode(clockPin, OUTPUT);
	pinMode(dataPin, OUTPUT);
}

void loop() {
	//count from 0 to 255
	for (int i = 0; i < 256; i++) {
		shiftDataOut(i);
		//set latchPin low then high to send data out
		digitalWrite(latchPin, LOW);
		digitalWrite(latchPin, HIGH);
		delay(1000);
	}
}

void shiftDataOut(byte dataOut) {
	// Shift out 8 bits LSB first, clocking each with a rising edge of the clock line
	boolean pinState;

	for (int i=0; i<=7; i++)  {              // for each bit in dataOut send out a bit
		digitalWrite(clockPin, LOW);     //set clockPin to LOW prior to sending bit

		// if the value of DataOut and (logical AND) a bitmask
		// are true, set pinState to 1 (HIGH)
		if ( dataOut & (1<<i) ) {
			pinState = HIGH;
		}
		else {
			pinState = LOW;
		}

		//sets dataPin to HIGH or LOW depending on pinState
		digitalWrite(dataPin, pinState); //send bit out before rising edge of clock
		digitalWrite(clockPin, HIGH);
	}
	digitalWrite(clockPin, LOW); //stop shifting out data
}
```

The Binary Number System

Before we take a look at the code and the hardware for Project 17, let's take a look at the binary number system, as it is essential to understand binary to successfully program a microcontroller.

Human beings use a base 10, or decimal number system, because we have 10 fingers on our hands. Computers do not have fingers, and so the best way for a computer to count is by a state of either ON or OFF (1 or 0). A logic device, such as a computer, can detect if a voltage is there (1) or if it is not (0), and so uses a binary, or base 2 number system, since this number system can easily be represented in an electronic circuit with a high or low voltage state.

In our number system, base 10, we have 10 digits ranging from 0 to 9. When we count to the next digit after 9, the digit resets back to zero, but a 1 is incremented to the tens column to its left. Once the tens column reaches 9, incrementing this by 1 will reset it to zero, but we add 1 to the hundreds column to its left, and so on.

```
000,001,002,003,004,005,006,007,008,009
010,011,012,013,014,015,016,017,018,019
020,021,023 .........
```

In binary, the exact same thing happens, except the highest digit is 1. So, adding 1 to 1 results in the digit resetting to zero and 1 being added to the column to the left.

```
000, 001
010, 011
100, 101...
```

An 8-bit number (or a byte) is represented as in Table 6-2

Table 6-2. *An 8-bit Binary Number*

2^7 128	2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1
0	1	0	0	1	0	1	1

The number above in binary is 01001011; in decimal, this is 75.
This is worked out as follows:

1 x 1 = 1

1 x 2 = 2

1 x 8 = 8

1 x 64 = 64

Add all that together and you get 75.
Here are some other examples (See Table 6-3.)

Table 6-3. *Binary number examples*

Dec	2^7 128	2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1
75	0	1	0	0	1	0	1	1
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
12	0	0	0	0	1	1	0	0
27	0	0	0	1	1	0	1	1
100	0	1	1	0	0	1	0	0
127	0	1	1	1	1	1	1	1
255	1	1	1	1	1	1	1	1

So now that you understand binary, we will take a look at the hardware, before looking at the code.

■ **Tip** You can use Google to convert between a decimal and a binary number, and vice versa.

For example, to convert 171 decimal to binary, type ***171 in binary*** into the Google search box which returns ***171 = 0b10101011***

The 0b prefix shows the number is a binary number and not a decimal one. So the answer is 10101011.

To convert a binary number to decimal, do the reverse, e.g., enter ***0b11001100 in decimal*** into the search box; this returns ***0b11001100 = 204***

Project 17 – Shift Register 8-Bit Binary Counter - Hardware Overview

We are going to do things in a different order for this project, and take a look at the hardware before we look at the code.

We are using a shift register, specifically, the 74HC595 type of shift register. This type of shift register is an 8-bit serial-in, serial or parallel-out shift register with output latches. This means that you can send data to the shift register in series and send it out in parallel. In series means 1 bit at a time. Parallel means lots of bits (in this case, 8) at a time. So you give the shift register data (in the form of 1s and 0s) one bit at a time. Each bit is shunted along as the next bit is entered. When a ninth bit is entered, the first bit entered will be shunted out the end of the shift register and lost (unless we do something with it—see the next project). When you raise the latch signal from LOW to HIGH, the current contents of the shift register are copied to the output latches. (Copying data to the output latches does not affect the contents of the shift register.)

This type of shift register is usually used for serial to parallel data conversion. Other shift registers are parallel in, serial out, and used for parallel to serial data conversion. Some shift registers can do both. In our case, as the data that is output is 1s and 0s (or 0V and 5V), we can use it to turn a bank of 8 LEDs on and off.

The shift register for this project requires only three inputs from the Arduino. The outputs of the Arduino and the inputs of the 595 are shown in Table 6-4). We are not using the features of pins 10 (master reset) and 13 (output enable input). However, these pins still need to be set to high (pin 10) or low (pin 13) for the project to work.

Table 6-4. *Pins Used*

Arduino Pin	595 Pin	Description
8	12	Storage Register Clock Input
11	14	Serial Data Input
12	11	Shift Register Clock Input

We are going to refer to pin 11 as the clock pin, pin 14 as the data pin, and pin 12 as the latch pin.

Imagine that the latch is like a camera that takes a snapshot of the contents of the shift register when the latch pin goes from LOW to HIGH and outputs that snapshot to the 8 pins (QA-QH or Q0 to Q7 depending on your datasheet—see Figure 6-2). The clock is simply a pulse of 0s and 1s and the data pin is where we send data from the Arduino to the 595.

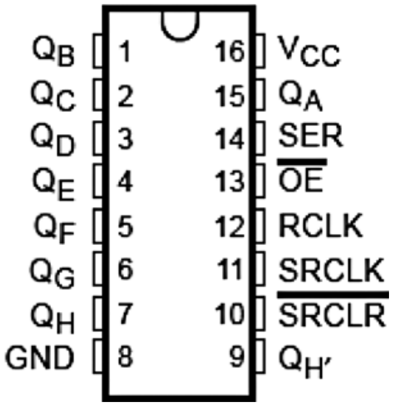


Figure 6-2. *Pin diagram of a 595 chip*

To use the shift register, the latch pin and clock pin must be set to LOW. The latch pin will remain at LOW until all 8 bits have been set. This allows data to be entered into the storage register (a place inside the IC for storing a 1 or a 0). We then present either a HIGH or LOW signal at the data pin and then set the clock pin to HIGH. By setting the clock pin to HIGH, we copy the data presented at the data pin into the storage register. Once this is done, we set the clock to LOW again, then present the next bit of data at the data pin. Once we have done this 8 times, we have sent a full 8 bit number into the 595. The latch pin is now raised, which copies the data from the storage register into the shift register and outputs it from Q0 to Q7 (pin 15, 1 to 7).

The sequence of events is described in Table 6-5.

Table 6-5. *Sequence of Events*

Pin	State	Description
Data	HIGH	First bit of data (1)
Clock	HIGH	Clock goes HIGH. Data stored.
Clock	LOW	Ready for next bit. Prevent any new data being stored.
Data	HIGH	2nd bit of data (1)
Clock	HIGH	2nd bit stored
…	…	…
Data	LOW	8th bit of data (0)
Clock	HIGH	Store the data
Clock	LOW	Prevent any new data being stored
Latch	LOW	Latch lowered, then raised to copy data from
Latch	HIGH	the shift register to the output pins

I have connected a logic analyzer (a device that lets you see the 1s and 0s coming out of a digital device) to my 595 while this program is running and the image in Figure 6-3 shows the output of the Arduino to the 595. You can see from this diagram that the binary number 00110111 (reading from right to left) or decimal 55, has been sent to the IC.

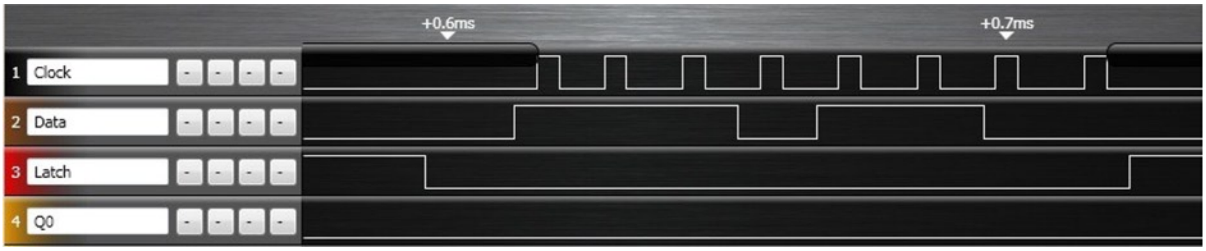


Figure 6-3. *The output from the 595 shown in a logic analyzer*

So to summarize the use of a single shift register in this project, we have 8 LEDs attached to the 8 outputs of the register. Data is sent to the data pin, one bit at a time, the cLock pin is set to HIGH to store that data, then back down to LOW, ready for the next bit. After all 8 bits have been entered, the latch is set to LOW and then to HIGH, which copies data from the shift register to the 8 output pins.

If you want to read more about shift registers, take a look at the part number on the IC (e.g. 74HC595N or SN74HC595N, etc.) and enter that into Google. You can then find the specific datasheet for the IC and read more about it.

I'm a huge fan of the 595 chip. It is very versatile and can increase the number of digital output pins that the Arduino has. The standard Arduino has 19 digital outputs (the 6 analog pins can also be used as digital pins numbered 14 to 19). Using 8-bit shift registers, you can expand that to 49 (6 x 595s plus one spare pin left over). They also operate very fast, typically at 100MHz, meaning you can send data out at approximately 100 million times per second (if you wanted to and if the Arduino was capable of doing so). This means you can also send PWM signals via software to the ICs and enable brightness control of the LEDs too.

As the outputs are simply logic HIGH or LOW of output voltages, they can also be used to switch other low-powered devices on and off (or even high-powered devices with the use of transistors or relays), or to send data to devices (e.g. an old dot-matrix printer or other serial device).

All of the 595 shift registers from any manufacturer are just about identical to each other. There are also larger shift registers with 16 outputs or higher. Some ICs advertised as LED driver chips are, when you examine the datasheet, simply larger shift registers (e.g., the M5450 and M5451 from STMicroelectronics).

Project 17 – Shift Register 8-Bit Binary Counter – Code Overview

The code for Project 17 looks pretty daunting at first look. But when you break it down into its component parts, you'll see it's not as complex as it looks.

First, three variables are initialized for the three pins we are going to use.

```
int latchPin = 8;
int clockPin = 12;
int dataPin = 11;
```

Then, in setup, the pins are all set to outputs.

```
pinMode(latchPin, OUTPUT);
pinMode(clockPin, OUTPUT);
pinMode(dataPin, OUTPUT);
```

The main loop simply runs a for loop counting from 0 to 255. On each iteration of the loop, the latchPin is set to LOW to enable data entry, then the function called shiftDataOut is called, passing the value of i in the for loop to the function. Then the latchPin is set to HIGH, transferring data from the shift register to the output latches and output pins. Finally there is a delay of half a second before the next iteration of the loop commences.

```
void loop() {
  //count from 0 to 255
  for (int i = 0; i < 256; i++) {
    //set latchPin low to allow data flow
    digitalWrite(latchPin, LOW);
    shiftDataOut(i);
    //set latchPin to high to lock and send data
    digitalWrite(latchPin, HIGH);
    delay(500);
  }
}
```

The shiftDataOut function receives as a parameter a byte (8-bit number), which will be our number between 0 and 255. We have chosen a byte for this usage as it is exactly 8 bits in length and we need to send only 8 bits out to the shift register.

```
void shiftDataOut(byte dataOut) {
```

Then a boolean variable called pinState is initialized. This will store the state we wish the relevant pin to be in when the data is sent out (1 or 0).

```
boolean pinState;
```

After this, we are ready to send the 8 bits in series to the 595 one bit at a time.

A for loop that iterates 8 times is set up.

```
for (int i=0; i<=7; i++)  {
```

The clock pin is set low prior to sending a data bit.

```
digitalWrite(clockPin, LOW);
```

Now an if/else statement determines if the pinState variable should be set to a 1 or a 0.

```
if ( dataOut & (1<<i) ) {
      pinState = HIGH;
}
else {
      pinState = LOW;
}
```

The condition for the if statement is:

```
dataOut & (1<<i).
```

This is an example of what is called a "bitmask," and we are now using bitwise operators. These are logical operators similar to the boolean operators we used in previous projects. However, the bitwise operators act on numbers at the bit level.

In this case we are using the bitwise and (&) operator to carry out a logical operation on two numbers. The first number is dataOut and the second is the result of (1<<i). Before we go any further, let's take a look at the bitwise operators.

Bitwise Operators

The bitwise operators perform calculations at the bit level on variables. There are 6 common bitwise operators and these are:

```
&       bitwise and
|       bitwise or
^       bitwise xor
~       bitwise not
<<      bitshift left
>>      bitshift right
```

Bitwise operators can only be used between integers. Each operator performs a calculation based on a set of logic rules. Let us take a close look at the bitwise AND (&) operator.

Bitwise AND (&)

The bitwise AND operator acts according to this rule:-

If both inputs are 1, the resulting outputs are 1, otherwise the output is 0.

Another way of looking at this is:

```
0 0 1 1       Operand1
0 1 0 1       Operand2
_________
0 0 0 1       (Operand1 & Operand2)
```

A type int is a 16-bit value, so using & between two int expressions causes 16 simultaneous AND operations to occur, as in a section of code like this:

```
int x = 77;   //binary: 0000000001001101
int y = 121;  //binary: 0000000001111001
int z = x & y;//result: 0000000001001001
```

In this case 77 & 121 = 73

Let's look at the remaining operators.

Bitwise OR (|)

If either or both of the inputs is 1, the result is 1, otherwise it is 0.

```
0 0 1 1       Operand1
0 1 0 1       Operand2
_____________
0 1 1 1       (Operand1 | Operand2)
```

Bitwise XOR (^)

If only 1 of the inputs is 1, then the output is 1. If both inputs are 1, then the output 0.

```
0 0 1 1       Operand1
0 1 0 1       Operand2
__________
0 1 1 0       (Operand1 ^ Operand2)
```

Bitwise NOT (~)

The bitwise NOT operator is applied to a single operand to its right.

The output becomes the opposite of the input. Zeros get converted to ones, and ones to zeros.

```
0 0 1 1       Operand1
__________
1 1 0 0       ~Operand1
```

Bitshift Left (<<), Bitshift Right (>>)

The bitshift operators move all of the bits in the integer to the left or right. with the number of bits specified by the right operand.

```
variable << number_of_bits

E.g.

byte x = 9 ;     // binary: 00001001
byte y = x << 3; // binary: 01001000 (or 72 dec)
```

Any bits shifted off the end of the row are lost forever. You can use the left bitshift to multiply a number by powers of 2 and the right bitshift to divide by powers of 2 (work it out).

Now that we have taken a look at the bitshift operators, let's return to our code.

Project 17 – Code Overview (continued)

The condition of the if/else statement was

```
dataOut & (1 << i)
```

And we now know this is a bitwise AND (&) operation. The right-hand operand inside the parenthesis is a left bitshift operation. This is a "bitmask." The 74HC595 will only accept data one bit at a time. We therefore need to convert the 8-bit number in dataOut into a single bit number representing each of the 8 bits in turn. The bitmask allows us to ensure that the pinState variable is set to either a 1 or a 0 depending on what the result of the bitmask calculation is. The right hand operand is the number 1 bit shifted i number of times. As the for loop makes the value of i go from 0 to 7, we can see that 1 bitshifted i times, each time through the loop, will result in these binary numbers (see Table 6-6).

Table 6-6. *The Results of 1<<i*

Value of I	Result of (1<<i) in Binary
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000

So you can see that the 1 moves from right to left as a result of this operation.

Now, the & operator's rules state that

If both inputs are 1, the resulting outputs are 1, otherwise the output is 0.

So, the condition of

```
dataOut & (1<<i)
```

will result in a 1 if the corresponding bit in the same place as the bitmask is a 1, otherwise it will be a zero. For example, if the value of dataOut was decimal 139 or 10001011 binary, then each iteration through the loop will result in the values in Table 6-7 (see Table 6-7).

Table 6-7. *The Results of b10001011<<i*

Value of I	Result of b10001011(1<<i) in Binary
0	00000001
1	00000010
2	00000000
3	00001000
4	00000000
5	00000000
6	00000000
7	10000000

So, every time there is a 1 in the I position (reading from right to left) the value comes out as a non-zero value (or TRUE) and every time there is a 0 in the I position, the value comes out at 0 (or FALSE).

The `if` condition will therefore carry out its code in the block if the value is higher than 0 (or in other words if the bit in that position is a 1) or "else" (if the bit in that position is a 0) it will carry out the code in the else block.

So looking at the if/else statement once more

```
  if ( dataOut & (1<<i) ) {
     pinState = HIGH;
   }
   else {
     pinState = LOW;
   }
```

And cross referencing this with the truth table in Table 6-7, we can see that for every bit in the value of dataOut that has the value of 1 that pinState will be set to HIGH and for every value of 0 it will be set to LOW.

The next piece of code writes either a HIGH or LOW state to the data pin and then sets the clock pin to HIGH to write that bit into the storage register.

```
digitalWrite(dataPin, pinState);
digitalWrite(clockPin, HIGH);
```

Finally the Clock Pin is set to low to ensure no further bit writes.

```
digitalWrite(clockPin, LOW);
```

So, in simple terms, this section of code looks at each of the 8 bits of the value in dataOut one by one and sets the data pin to HIGH or LOW accordingly, then writes that value into the storage register.

This is simply sending the 8-bit number out to the 595 one bit at a time. Then the main loop sets the latch pin to HIGH to send out those 8 bits simultaneously to pins 15 and 1 to 7 (QA to QH) of the shift register, thus making our 8 LEDs show a visual representation of the binary number stored in the shift register.

Your brain may hurt after Project 17, so take a rest, stretch your legs, and take another deep breath before you dive into Project 18, in which we will now use two shift registers daisy-chained together.

Project 18 – Dual 8-Bit Binary Counters

In Project 18, we will use the parts listed in Table 6-8 to daisy chain (or cascade) another 74HC595 IC onto the one used in Project 17 to create a dual binary counter.

Parts Required

Table 6-8. *Parts Required for Project 18*

1 x 74HC595 Shift Register IC	
16 x Current Limiting Resistors	
8 x Red LEDs	
8 x Green LEDs	
0.1uF Capacitor	

Connect It Up

The first 595 is wired the same way as in Project 17. The second 595 has +5V and ground wires going to the same pins as on the first 595. Then, add a wire from pin 9 on IC 1 to pin 14 on IC 2. Add another from pin 11 on IC 1 to pin 11 on IC 2 and pin 12 on IC 1 to pin 12 on IC 2. Connect the capacitor between the supply and ground as before.

The same outputs as on the first 595 going to the first set of LEDs go from the second IC to the second set of LEDs. Examine the diagrams carefully in Figure 6-4 and 6-5.

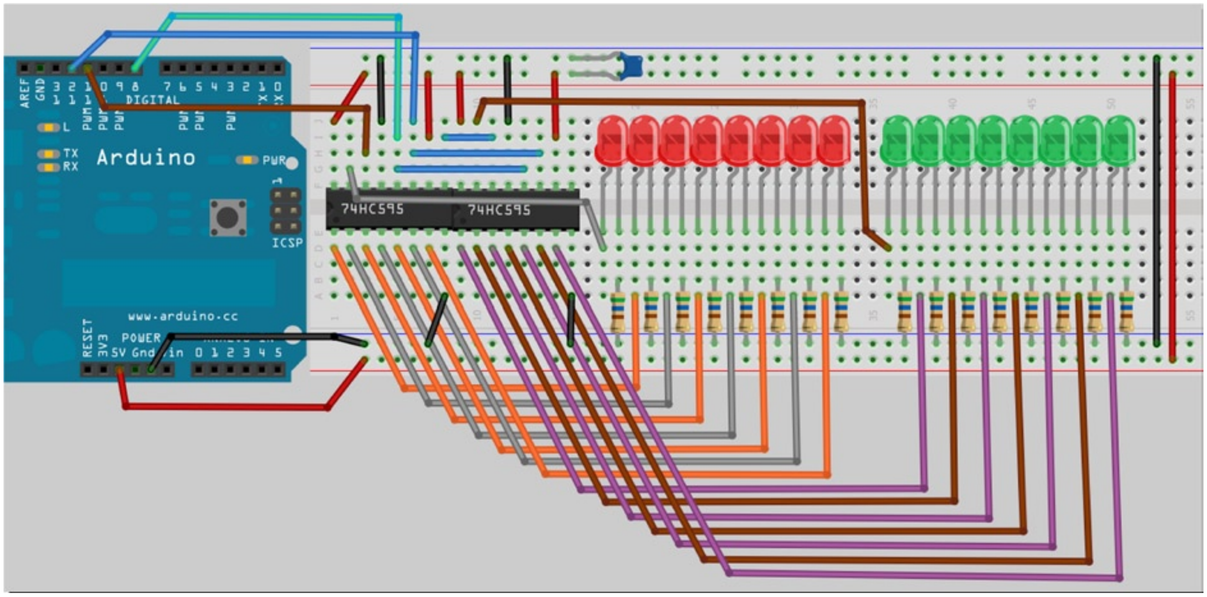


Figure 6-4. *The circuit for Project 18*

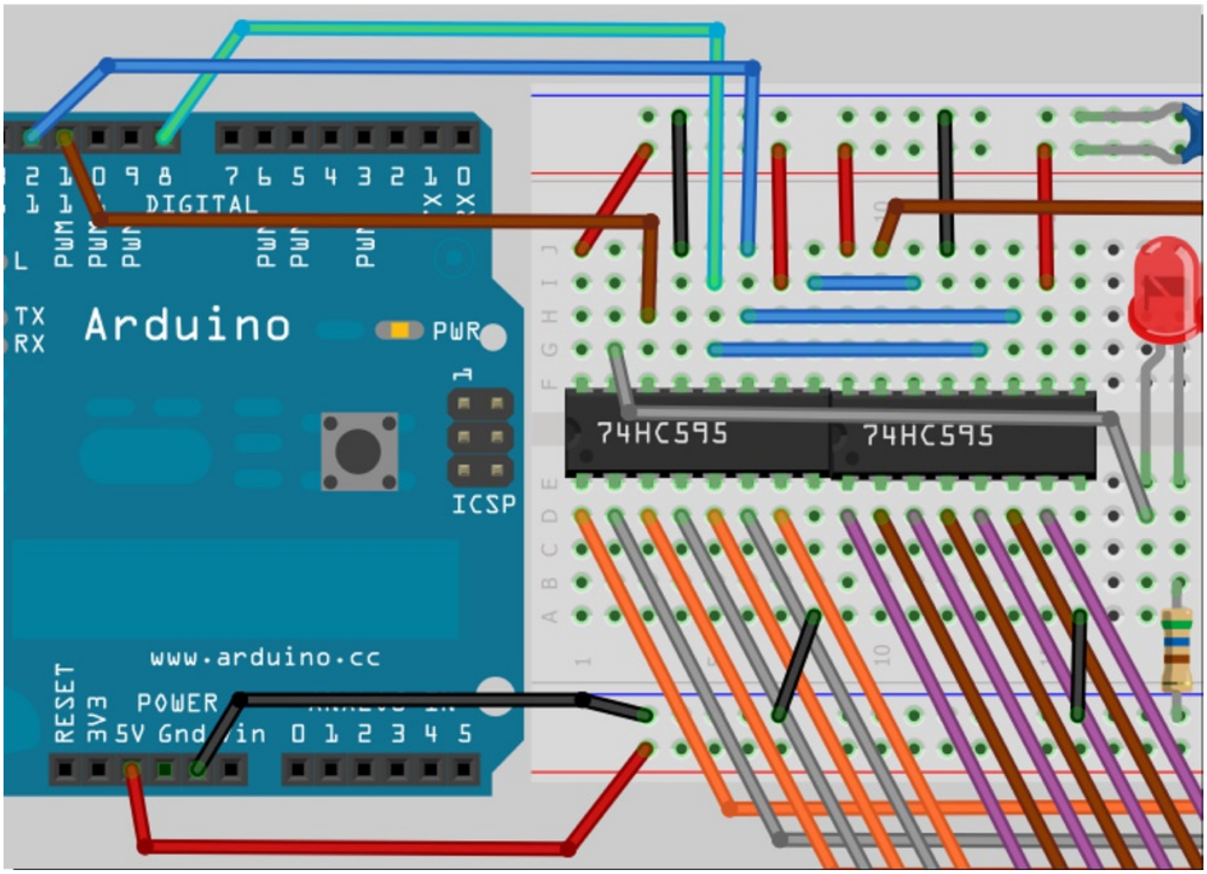


Figure 6-5. *The circuit for Project 18. Close up of the wiring of the ICs*

Enter the Code

Enter the following code in Listing 6-2 and upload it to your Arduino. When you run this code, you will see the green set of LEDs count up (in binary) from 0 to 255 and the red LEDs count down from 255 to 0 at the same time.

Listing 6-2. Code for Project 18

```
// Project 18

int latchPin = 8;  //Pin connected to Pin 12 of 74HC595 (Latch)
int clockPin = 12; //Pin connected to Pin 11 of 74HC595 (Clock)
int dataPin = 11;  //Pin connected to Pin 14 of 74HC595 (Data)

void setup() {
        //set pins to output
        pinMode(latchPin, OUTPUT);
        pinMode(clockPin, OUTPUT);
        pinMode(dataPin, OUTPUT);
}
```

```
void loop() {
        for (int i = 0; i < 255; i++) {                        //count from 0 to 255
                digitalWrite(latchPin, LOW);                   //set latchPin low to allow data flow
                shiftOut(dataPin, clockPin, LSBFIRST, i);      // shift out first 8 bits
                  shiftOut(dataPin, clockPin, LSBFIRST, 255-i); // shiftOut second 8 bits//set
                  latchPin to high to lock and send data
                digitalWrite(latchPin, HIGH);
                delay(250 );
        }
}
```

Project 18 Code & Hardware Overview

The code for Project 18 is very similar to that of Project 17. We have simply added a second instruction to shift out another 8 bits to the second shift register. However, you will notice immediately that despite this being a more complex project than Project 17, we have less code. How is that? Well, you will notice that the shiftDataOut function has been replaced with shiftOut() and that this function is colored red in the Arduino IDE. This is because the shiftOut command is an integral part of the Arduino language, as shifting out 8 bits of data to shift registers or LED controller ICs is used so commonly in the Arduino world that the ability to do so was introduced in the language.

The syntax for the shiftOut function is as follows:

shiftOut(`dataPin`, `clockPin`, bitOrder, `value`)

The dataPin and clickPin were set at the start of the program and you set their pinMode to OUTPUT in the setup() loop. The bitOrder decides if the bits are shifted out starting with the most significant bit (the bit at the far left) or the least significant bit first (the bit at the far right of the 8). You use the words MSBFIRST for most significant bit first or LSBFIRST for the least significant bit first. We want to send out the least significant bit first, so we will be using LSBFIRST. Finally, the last value is the digit we are sending out. This is a byte as it is 8 bits maximum (0–255).

In Project 17, we made our own shiftDataOut function that does exactly what shiftOut does. The reason I made you do this extra work was so that you could see exactly what was going on and how the shift register worked. This is essential knowledge for using shift registers. However, from now on we will use the shiftOut function.

In the main loop, the shiftOut routine sends 8 bits to the 595. In the main loop, we have put two sets of calls to shiftOut, one sending the value of I and the other sending 255-i. We call shiftOut twice before we set the latch to HIGH. This will send two sets of 8 bits, or 16 bits in total, to the 595 chips before the latch is set HIGH to copy the contents of the shift register to the output pins, which in turn make the LEDs go on or off.

The second 595 is wired exactly the same way as the first one. The clock and latch pins are tied to the same pins of the first 595. However, we have a wire going from pin 9 on IC 1 to pin 14 on IC 2. Pin 9 is the data output pin and pin 14 is the data input pin.

The data is input to pin 14 on the first IC from the Arduino. The second 595 chip is “daisy chained” to the first chip by pin 9 on IC 1, which is outputting data, into pin 14 on the second IC, which is the data input.

What happens is, as you enter a 9th bit and above, the data in IC 1 gets shunted out of its data pin and into the data pin of the second IC. So, once all 16 bits have been sent down the data line from the Arduino, the first 8 bits sent would have been shunted out of the first chip and into the second. The second 595 chip will contain the FIRST 8 bits sent out and the first 595 chip will contain the SECOND 8 bits, or bits 9 to 16.

An almost unlimited number of 595 chips can be daisy chained in this manner.

EXERCISE

Exercise. Using the same circuit for Project 18 and all 16 LEDs, recreate the Knight Rider (or Cylon) light effect, making the LEDs bounce back and forth across all 16 LEDs.

Summary

In Chapter 6 we covered a lot of ground on the use of an external IC to give us extra output pins on our Arduino. Although we could have done these projects without the external IC, the shift registers have made life a lot easier for us. The code for these projects, if we were to not use the shift registers, would be a lot more complex. The use of an IC designed to take serial data and output it in a parallel fashion is ideal for controlling lines of LEDs in this way and has decreased the complexity, not increased it.

Never be daunted by using external ICs. A slow and methodical read of the datasheets will reveal how they work. Datasheets at first glance look complicated, but most of the information in them is irrelevant to you, and once you strip out just the bits relevant to your project you will be able to understand how the chip works.

In Chapter 7 we are going to continue to use shift registers, but this time we are going to use them to control LED dot-matrix displays, which contain at least 64 LEDs per unit. We can control a large number of LEDs at the same time using a great technique known as multiplexing, which we will learn about in the next chapter.

Subjects and Concepts covered in Chapter 6

- The binary number system and how to convert to and from decimal
- How to use a shift register to input serial data and output parallel data
- Using an external IC to decrease the complexity of a project
- Sending a parameter to a function call
- Using boolean variables
- The concept and use of bitwise operators
- Using bitwise operators to create a bitmask
- How to cascade (or daisy chain) two or more shift registers
- The use of the `shiftOut()` function.

CHAPTER 7

LED Displays

So far you have dealt with individual 5mm LEDs. LEDs can also be obtained in a package known as a dot matrix display, the most popular being a matrix of 8×8 LEDs or 64 LEDs in total. You can also obtain bi-color dot matrix displays (e.g. red and green) or even RGB dot matrix displays, which can display any color and contains a total of 192 LEDs in a single display package. In this chapter, you are going to deal with a standard single color 8×8 dot matrix display and learn how to display images and text. You will start off with a simple demonstration of creating an animated image on an 8×8 display and then move onto more complex display projects. Along the way, you will learn the very important concept of multiplexing.

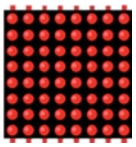
Project 19 – LED Dot Matrix Display – Basic Animation

In this project, you shall again use two sets of shift registers. These will be connected to the rows and columns of the dot matrix display. You will then show a simple object, or sprite, on the display and animate it. The main aim of this project is to show you how a dot matrix display works and introduce the concept of multiplexing because this is an invaluable skill to have.

Parts Required

You will need two shift registers (74HC595) and eight current-limiting resistors. You also need to obtain a common cathode (C-) dot matrix display and the datasheet for the display so you know which pin connects to which row or column. You will need the parts listed in Table 7-1 for this project.

Table 7-1. *Parts Required for Project 19*

2 × 74HC595 Shift Register IC	
8 × Current-Limiting Resistors (510Ω)	
8×8 DotMatrix Display (C-)	
Bypass (Decoupling) Capacitor	

The current limiting resistors value will depend on the LED you are using. Check out this tutorial for a guide as to which one to select: https://www.sparkfun.com/tutorials/219

Connect It Up

Examine the diagram carefully. It is important you do not connect the Arduino to the USB cable or power until the circuit is complete; you risk damaging the shift registers or the dot matrix display otherwise. This is a complicated wiring exercise so be careful. Make sure you connect things slowly and methodically.

The wiring diagram in Figure 7-1 is relevant to the specific dot matrix unit that I used in creating this project, a mini 8×8 red dot matrix display unit. However, your display may (and quite likely will) have different pins than the one I used. You *must* read the datasheet of the unit you have bought to ensure that the correct output pins on the shift registers go to the correct pins on the dot matrix. For a good tutorial in PDF format on how to read a datasheet, go to www.egr.msu.edu/classes/ece480/goodman/read_datasheet.pdf and download the PDF file.

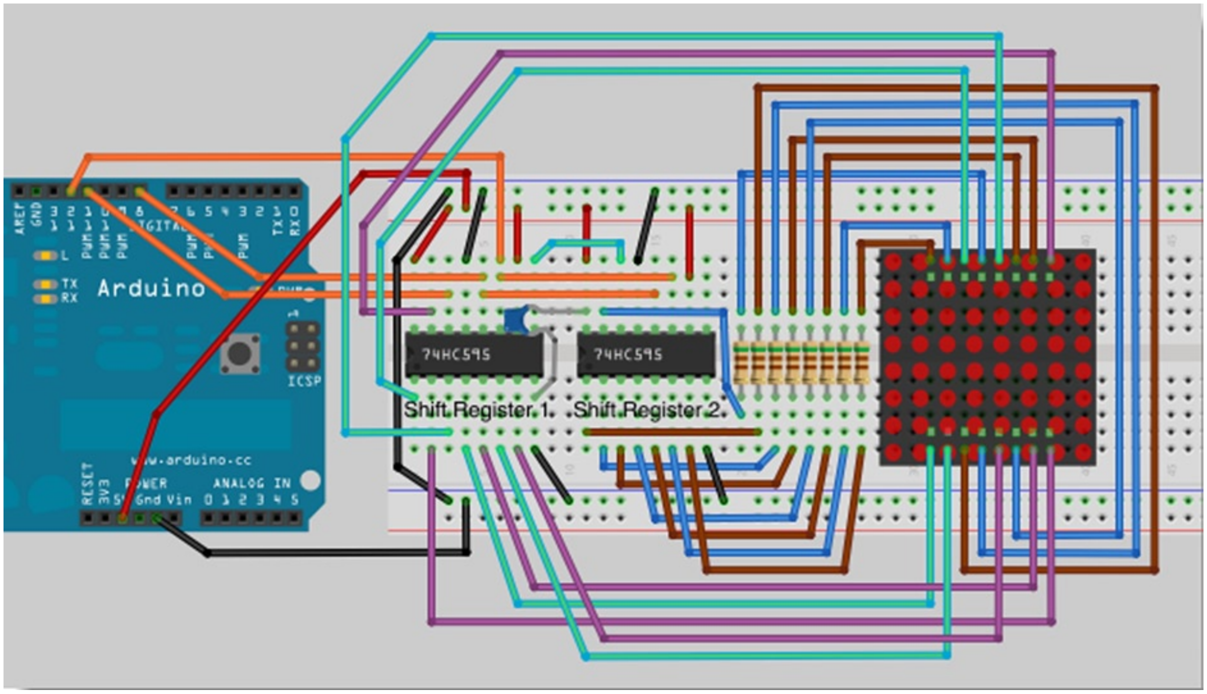


Figure 7-1. *The circuit for Project 19 – LED Dot Matrix – Basic Animation*

To make this easier, Table 7-2 shows which pins from the shift registers need to go to which pins on the dot matrix display. The matrix pins correlate to my specific display (as in Figure 7-2) so adjust the circuit accordingly so that it is correct for the type of display you have obtained. Read the datasheet first as the pins will no doubt differ from mine.

Table 7-2. *Pins Required for the Dot-Matrix Display with Circuit for Reference*

Row	Shift Reg 1	Matrix Pin
Row 1	Pin 15	9
Row 2	Pin 1	14
Row 3	Pin 2	8
Row 4	Pin 3	12
Row 5	Pin 4	1
Row 6	Pin 5	7
Row 7	Pin 6	2
Row 8	Pin 7	5
Column	**Shift Reg 2**	**Matrix Pin**
Column 1	Pin 15	13
Column 2	Pin 1	3
Column 3	Pin 2	4
Column 4	Pin 3	10
Column 5	Pin 4	6
Column 6	Pin 5	11
Column 7	Pin 6	15
Column 8	Pin 7	16

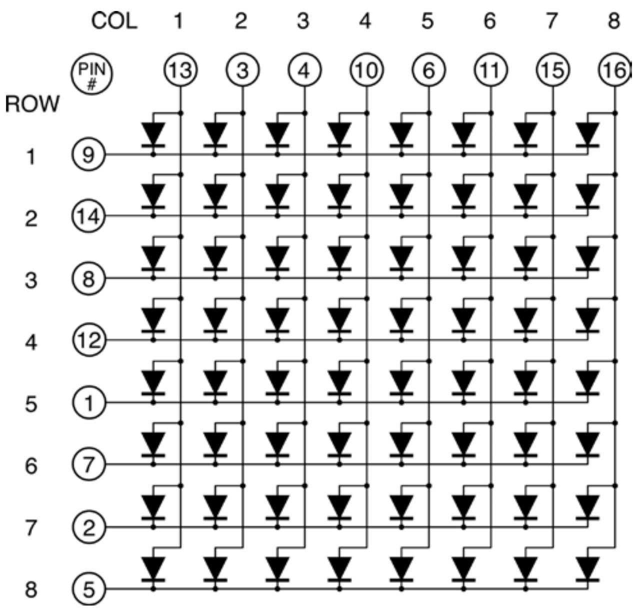


Figure 7-2. *A typical schematic for an 8×8 LED dot matrix display*

The schematic for the dot-matrix display used to create this project is in Figure 7-2. As you can see, the rows and columns (anodes and cathodes) are not ordered logically. Using Table 7-2 and the schematic in Figure 7-2, you can see that pin 15 on shift register 1 needs to go to row 1 on the display and hence goes to pin 9 on the display. Pin 1 on the shift register needs to go to row 2 and hence goes to pin 14 on the display, and so on. Current limiting resistors are placed on the connections between the shift register and the display. You need to carefully choose values for these to suit the display you are using. Refer to the Sparkfun tutorial at `https://www.sparkfun.com/tutorials/219` for the formula for calculating the right value.

You will need to read the datasheet of the display you have and go through a similar exercise to ascertain which pins from the shift register go to which pins on the LED display.

Enter the Code

Once you have confirmed that your wiring is correct, enter the code in Listing 7-1 and upload it to your Arduino. You will also need to download the TimerOne library. This can be downloaded from the Arduino website at `www.arduino.cc/playground/Code/Timer1`. Once you have downloaded the library, unzip it and put the entire TimerOne folder into the hardware/libraries folder inside the Arduino installation (Arduino/Contents/Resources/Java/libraries on a Mac). Make sure the folder is simply called TimerOne, remove any other characters after the e. This is an example of an external library. The Arduino IDE comes preloaded with many libraries, such as Ethernet, LiquidCrystal, Servo, etc. The TimerOne library is an external library, which you simply need to download and install in the libraries folder for it to work (you will need to restart your IDE before it will be recognized).

Listing 7-1. Code for Project 19

```
// Project 19
#include <TimerOne.h> // Install the library first or the code won't work

int latchPin = 8;     //Pin connected to Pin 12 of 74HC595 (Latch)
int clockPin = 12;    //Pin connected to Pin 11 of 74HC595 (Clock)
int dataPin = 11;     //Pin connected to Pin 14 of 74HC595 (Data)

byte led[8];          // 8 element unsigned integer array to hold the sprite

void setup() {
  // set the 3 digital pins to outputs
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);

  // Load the binary representation of the image into the array
  led[0] = B11111111;
  led[1] = B10000001;
  led[2] = B10111101;
  led[3] = B10100101;
  led[4] = B10100101;
  led[5] = B10111101;
  led[6] = B10000001;
  led[7] = B11111111;

  // set a timer of length 10000 microseconds (1/100th of a second)
  // and attach the screenUpdate function to the interrupt timer
  Timer1.initialize(10000);
  Timer1.attachInterrupt(screenUpdate);
}
```

```
// invert each row of the binary image and wait 1/4 second
void loop() {

  for (int i=0; i<8; i++) {
    led[i]= ~led[i];
  }
  delay (250);
}

// Display the image
void screenUpdate() {
 byte row = B10000000; // row 1
 for (byte k = 0; k < 8; k++) {

      shiftOut(dataPin, clockPin, LSBFIRST, led[k] ); // LED array
      shiftOut(dataPin, clockPin, LSBFIRST, ~row);    // row binary number (active low)

      // latch low to high to output data
      digitalWrite(latchPin, LOW);
      digitalWrite(latchPin, HIGH);

      // bitshift right
      row = row >> 1;
 }

// Turn all rows off until next interrupt
      shiftOut(dataPin, clockPin, LSBFIRST, 0);
      shiftOut(dataPin, clockPin, LSBFIRST, ~0);

      // latch low to high to output data
      digitalWrite(latchPin, LOW);
      digitalWrite(latchPin, HIGH);
}
```

A library is simply a collection of code that someone else has written to give you functionality that would otherwise require you to write from scratch. This is code re-use and it helps to speed up your development process. After all, there is nothing to be gained from reinventing the wheel. If someone has already created some code to carry out a task and that code is out in the public domain, then make use of it.

Once the code is run, you will see concentric squares on the display. Approximately every quarter of a second the display will invert to give a basic animated effect to the image.

Remember, you will need the Timer1 library for this project to work.

Project 19 – LED Dot-Matrix – Basic Animation – Hardware Overview

For this project, you'll take a look at the hardware before you look at how the code works. It will make the code easier to understand.

You learned how to use the 74HC595 in the previous projects. The only addition to the circuit this time is an 8×8 LED dot-matrix unit.

Dot-matrix units typically come in either a 5×7 or 8×8 matrix of LEDs. The LEDs are wired in the matrix such that either the anode or cathode of each LED is common in each row. In other words, in a common anode LED dot-matrix unit, each row of LEDs would have all of their anodes in that row wired together. The cathodes of the LEDs would all be wired together in each column. The reason for this will become apparent soon.

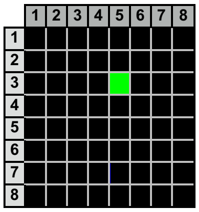
A typical single color 8×8 dot-matrix unit will have 16 pins, 8 for each row and 8 for each column. You can also obtain bi-color units (e.g. red and green) as well as full color RGB (red, green, and blue) units—the ones used in large video walls. Bi or Tri (RGB) color units have two or three LEDs in each pixel of the array. These are very small and next to each other.

By turning on different combinations of red, green, or blue in each pixel and by varying their brightnesses, any color can be obtained.

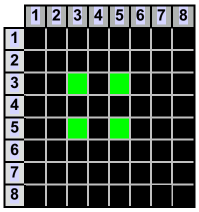
The reason the LEDs are wired together by row and by column is to minimize the number of pins required. If this were not the case, a single color 8×8 dot-matrix unit would need 65 pins, one for each LED and a common anode or cathode connector. By wiring the rows and columns together, only 16 pins are required.

However, this now poses a problem if you want a particular LED to light in a certain position. If, for example, you had a row anode/column cathode unit and wanted to light the LED at X, Y position 5, 3 (5th column, 3rd row), then you would apply a positive voltage to the 3rd Row pin and ground the 5th column pin.

The LED in the 5th column and 3rd row would now light.



Now let's imagine that you want to also light the LED at column 3, row 5. So you apply a positive voltage to the 5th row and ground the 3rd column pin. The LED at column 3, row 5 now illuminates. But wait…the LEDs at column 3, row 3 and column 5, row 5 have also lit up.



This is because you are applying power to row 3 and 5 and grounding columns 3 and 5. You can't turn off the unwanted LEDs without turning off the ones you want on. It would appear that there is no way you can light just the two required LEDs with the rows and columns wired together as they are. The only way this would work would be to have a separate pinout for each LED, meaning the number of pins would jump from 16 to 65. A 65-pin dot-matrix unit would be very hard to wire up and control because you'd need a microcontroller with at least 64 digital outputs.

Is there a way to get around this problem? Yes there is, and it is called *multiplexing* (or *muxing*).

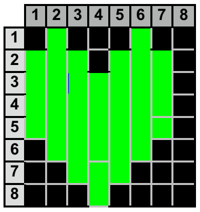
Multiplexing

Multiplexing is the technique of switching one row of the display on at a time. By grounding the columns containing the LEDs you want lit in the row, then applying positive voltage to only that row, (or the other way around for column anode/row cathode displays), the chosen LEDs in that row will illuminate. That row is then turned off and the next row is turned on, again with the appropriate columns chosen and the LEDs in the second row will now illuminate. Repeat with each row until you get to the bottom and then start again at the top.

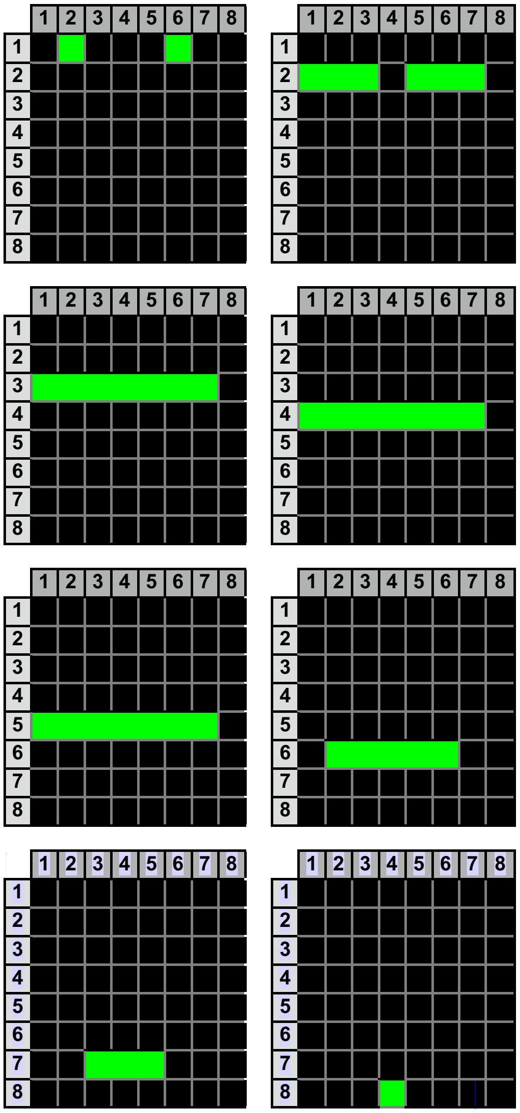
If this is done fast enough (at more than 100Hz, or 100 times per second) then the phenomenon of *persistence of vision* (where an afterimage remains on the retina for approximately 1/25th of a second) will mean that the display will appear to be steady, even though each row is turned on and off in sequence.

By using this technique, you get around the problem of displaying individual LEDs without the other LEDs in the same column or row also being lit.

For example, you want to display the following image on your display:



Then each row would be lit in turn like so:



By scanning down the rows and illuminating the respective LEDs in each column of that row and doing this very fast (more than 100Hz i.e. 100 times per second) the human eye will perceive the image as steady and the image of the heart will be recognizable in the LED pattern.

You are using this multiplexing technique in the Project's code. That's how you're to display the heart animation without also displaying extraneous LEDs.

Project 19 – LED Dot-Matrix – Basic Animation – Code Overview

The code for this project uses a feature of the ATmega chip called a Hardware Timer. This is essentially a timer on the chip that can be used to trigger an event. These events are called interrupts, because they cause the processor to interrupt the code it was processing and go process a function (interrupt service routine (ISR)) to handle the event. When it is finished processing the ISR, it resumes processing the program back where it was when the interrupt happened. This is much like a person interrupting what they are doing to answer a phone call, then going back to what they were doing when the phone call is over. In your case you are setting the timer up to generate an event every 10000 microseconds, which is every 100th of a second.

You make use of a library that enables easy use of interrupts, the TimerOne library. TimerOne makes creating an ISR very easy. You simply tell the function what the interval is (in this case, 10000 microseconds) and the name of the function you wish to activate every time the interrupt is fired (in this case, it's the `screenUpdate()` function). Note that interrupt service routines need to be kept short (less than the time between interrupts), otherwise the processor will never get back to the main code. (Or worse, overflow the stack.)

TimerOne is an external library, so you need to include it in your code. This is easily done using the include command:

```
#include <TimerOne.h>
```

Next, the pins used to interface with the shift registers are declared:

```
int latchPin = 8;                   //Pin connected to Pin 12 of 74HC595 (Latch)
int clockPin = 12;                  //Pin connected to Pin 11 of 74HC595 (Clock)
int dataPin = 11;                   //Pin connected to Pin 14 of 74HC595 (Data)
```

Next, create an array of type byte that has eight elements. The led[8] array will be used to store the image you are going to display on the dot matrix display:

```
byte led[8];                        // 8 element byte array to store the sprite
```

In the setup routine, you set the latch, clock, and data pins as outputs:

```
void setup() {
        pinMode(latchPin, OUTPUT);  // set the 3 digital pins to outputs
        pinMode(clockPin, OUTPUT);
        pinMode(dataPin, OUTPUT);
```

Once the pins have been set to outputs, the led array is loaded with the 8-bit binary images that will be displayed in each row of the 8×8 dot matrix display:

```
led[0] = B11111111;                 // enter the binary representation of the image
led[1] = B10000001;                 // into the array
led[2] = B10111101;
led[3] = B10100101;
led[4] = B10100101;
led[5] = B10111101;
```

```
led[6] = B10000001;
led[7] = B11111111;
```

By looking at the array above, you can make out the image that will be displayed, which is a box within a box. The 1s indicate where an LED will be lit and the 0s where it will be off. You can, of course, adjust the 1s and 0s yourself to make any 8×8 sprite you wish.

After this, the `Timer1` object is used. First, the Timer needs to be initialized with the frequency it will be activated at. In this case, you set the period to 10000 microseconds, or 1/100th of a second. Once the interrupt has been initialized, you need to attach to the interrupt a function that will be executed every time the time period is reached. This is the `screenUpdate()` function which will fire every 1/100th of a second:

```
// set a timer of length 10000 microseconds (1/100th of a second)
Timer1.initialize(10000);
// attach the screenUpdate function to the interrupt timer
Timer1.attachInterrupt(screenUpdate);
```

In the main loop, a `for` loop cycles through each of the eight elements of the led array and inverts the contents using the ~ or NOT bitwise operator. This simply turns the binary image into a negative of itself by turning all 1s to 0s and all 0s to 1s. Then it waits 250 milliseconds before repeating.

```
for (int i=0; i<8; i++) {
led[i]= ~led[i]; // invert each row of the binary image
}
delay(250);
```

You now have the `screenUpdate()` function. This is the function that the interrupt is activating every 100th of a second. This whole routine is very important because it is responsible for ensuring the LEDs in the dot matrix array are lit correctly and displaying the image you wish to convey. It's a very simple but effective function.

```
void screenUpdate() {                                  // function to display image
// Display the image
byte row = B10000000;                                  // row 1
 for (byte k = 0; k < 8; k++) {

      shiftOut(dataPin, clockPin, LSBFIRST, led[k] ); // LED array (inverted)
      shiftOut(dataPin, clockPin, LSBFIRST, row);     // row binary number

      digitalWrite(latchPin, LOW);                     // latch low to high to output data
      digitalWrite(latchPin, HIGH);
      row = row >> 1;                                  // bitshift right
 }
// Clear the matrix
row = B10000000;                                       // row 1
 for (byte k = 0; k < 8; k++) {

      shiftOut(dataPin, clockPin, LSBFIRST, 255);     // LED array (inverted)
      shiftOut(dataPin, clockPin, LSBFIRST, row);     // row binary number

      digitalWrite(latchPin, LOW);                     // latch low to high to output data
      digitalWrite(latchPin, HIGH);
      row = row >> 1;                                  // bitshift right
 }
}
```

A byte called `row` is declared and initialized with the value B10000000:

```
byte row = B10000000;                         // row 1
```

You now cycle through the led array and send that data out to the shift registers (which is processed with the bitwise NOT ~ to make sure the columns you want to display are turned off, or grounded), followed by the row:

```
for (byte k = 0; k < 8; k++) {

      shiftOut(dataPin, clockPin, LSBFIRST, led[k] ); // LED array (inverted)
      shiftOut(dataPin, clockPin, LSBFIRST, row);     // row binary number

      digitalWrite(latchPin, LOW);                    // latch low to high to output data
      digitalWrite(latchPin, HIGH);
      row = row >> 1;                                 // bitshift right
 }
```

Once you have shifted out that current row's 8 bits, the value in the row is bit shifted right one place so that the next row is displayed (i.e. the row with the 1 in it gets displayed only). You learned about the `bitshift` command in Chapter 6.

```
row = row >> 1;                               // bitshift right
```

Next you repeat the above but this time ground all the rows so the display is cleared. This is essential to ensure that the display is not left on too long while it waits for the next ISR to be executed, as this will cause your last row to be brighter than the rest.

```
row = B10000000;                              // row 1
 for (byte k = 0; k < 8; k++) {

      shiftOut(dataPin, clockPin, LSBFIRST, 255);     // LED array (inverted)
      shiftOut(dataPin, clockPin, LSBFIRST, row);     // row binary number

      digitalWrite(latchPin, LOW);                    // latch low to high to output data
      digitalWrite(latchPin, HIGH);
      row = row >> 1;                                 // bitshift right
 }
```

Remember from the hardware overview that the multiplexing routine is only displaying one row at a time, turning it off and then displaying the next row. This flicker is done at 100Hz, which is too fast for the human eye to see.

So, the basic concept here is that you have an interrupt routine that executes every 100th of a second. In that routine, you simply take a look at the contents of a screen buffer array (in this case, `led[]`) and display it on the dot-matrix unit one row at a time, but do this so fast that it all seems to be lit at once.

The main loop of the program is simply changing the contents of the screen buffer array and letting the ISR do the rest.

The animation in this project is very simple, but by manipulating the 1s and 0s in the buffer you can make anything you like, from shapes to scrolling text, appear on the dot-matrix unit. Let's try a variation on this in the next project; you'll create an animated scrolling sprite.

Project 20 – LED Dot-Matrix Display – Scrolling Sprite

You’re going to use the same circuit, but with a slight variation in the code to create a multi-frame animation that also scrolls from right to left. In doing so, you will be introduced to the concept of multi-dimensional arrays. You’ll also learn a little trick to get bitwise rotation (or circular shift). To start, you’ll use the exact same circuit as in Project 19.

Enter the Code

Enter and upload the code in Listing 7-2.

Listing 7-2. Code for Project 20

```
// Project 20
#include <TimerOne.h>

int latchPin = 8;  //Pin connected to Pin 12 of 74HC595 (Latch)
int clockPin = 12; //Pin connected to Pin 11 of 74HC595 (Clock)
int dataPin = 11;  //Pin connected to Pin 14 of 74HC595 (Data)
byte frame = 0;    // variable to store the current frame being displayed

byte led[8][8] = { {0, 56, 92, 158, 158, 130, 68, 56},                // 8 frames of an animation
                      {0, 56, 124, 186, 146, 130, 68, 56},
                      {0, 56, 116, 242, 242, 130, 68, 56},
                      {0, 56, 68, 226, 242, 226, 68, 56},
                      {0, 56, 68, 130, 242, 242, 116, 56},
                      {0, 56, 68, 130, 146, 186, 124, 56},
                      {0, 56, 68, 130, 158, 158, 92, 56},
                      {0, 56, 68, 142, 158, 142, 68, 56} };

void setup() {
        pinMode(latchPin, OUTPUT);                  // set the 3 digital pins to outputs
        pinMode(clockPin, OUTPUT);
        pinMode(dataPin, OUTPUT);

        Timer1.initialize(10000);                   // set a timer of length 10000 microseconds
        Timer1.attachInterrupt(screenUpdate);       // attach the screenUpdate function
}

void loop() {
        for (int i=0; i<8; i++) {                   // loop through all 8 frames of the animation
                for (int j=0; j<8; j++) {           // loop through the 8 rows per frame
                        led[i][j]= led[i][j] << 1 | led[i][j] >> 7;    // bitwise rotation
                }
        }
        frame++;                                    // go to the next frame in the animation
        if (frame>7) { frame =0;}                   // make sure we go back to frame 0 once past 7
        delay(100);                                 // wait a bit between frames
}
```

```
void screenUpdate() {                                // function to display image
   byte row = B10000000;                             // row 1
   for (byte k = 0; k < 8; k++) {

      shiftOut(dataPin, clockPin, LSBFIRST, led[frame][k] );           // LED array
      shiftOut(dataPin, clockPin, LSBFIRST, ~row); // row select (active low)

      // create a low to high transition on latchPin to transfer output to display
      digitalWrite(latchPin, LOW);
      digitalWrite(latchPin, HIGH);
      row = row >> 1;                                // bitshift right
   }
   // turn all rows off until next timer interrupt so last row isn't on longer than others
   shiftOut(dataPin, clockPin, LSBFIRST, 0 );      // column doesn't matter w/ all rows off
   shiftOut(dataPin, clockPin, LSBFIRST, ~0);      // select no row

   // create a low to high transition on latchPin to transfer output to display
   digitalWrite(latchPin, LOW);
   digitalWrite(latchPin, HIGH);
}
```

When you run Project 20, you will see an animated wheel rolling along. The hardware hasn't changed so I don't need to discuss that. Let's see how this code works.

Project 20 – LED Dot-Matrix – Scrolling Sprite – Code Overview

Again, you load the TimerOne library and set the three pins that control the shift registers:

```
#include <TimerOne.h>

int latchPin = 8;  //Pin connected to Pin 12 of 74HC595 (Latch)
int clockPin = 12; //Pin connected to Pin 11 of 74HC595 (Clock)
int dataPin = 11;  //Pin connected to Pin 14 of 74HC595 (Data)
```

Then you declare a variable of type byte and initialize it to zero. This will store the number of the currently displayed frame of the eight-frame animation:

```
byte frame = 0;    // variable to store the current frame being displayed
```

Next, you set up a two dimensional array of type byte:

```
byte led[8][8] = { {0, 56, 92, 158, 158, 130, 68, 56},                // 8 frames of an animation
                   {0, 56, 124, 186, 146, 130, 68, 56},
                   {0, 56, 116, 242, 242, 130, 68, 56},
                   {0, 56, 68, 226, 242, 226, 68, 56},
                   {0, 56, 68, 130, 242, 242, 116, 56},
                   {0, 56, 68, 130, 146, 186, 124, 56},
                   {0, 56, 68, 130, 158, 158, 92, 56},
                   {0, 56, 68, 142, 158, 142, 68, 56} };
```

Arrays were introduced in Chapter 3. An array is a collection of variables that are accessed using an index number. This array differs in that it has two sets of numbers for the elements. In Chapter 3 you declared a one-dimensional array like this:

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

Here you have to create a two-dimensional array with two sets of index numbers. In this case, your array is 8 X 8, or 64 elements in total. A two-dimensional array is pretty much the same as a two-dimensional table in that you can access a single cell by referencing the row and column number accordingly. Table 7-3 shows you how to access the elements in your array.

Table 7-3. *The Elements in Your Array*

	0	1	2	3	4	5	6	7
0	0	56	92	158	158	130	68	56
1	0	56	124	186	146	130	68	56
2	0	56	116	242	242	130	68	56
3	0	56	68	226	242	226	68	56
4	0	56	68	130	242	242	116	56
5	0	56	68	130	146	186	124	56
6	0	56	68	130	158	158	92	56
7	0	56	68	142	158	142	68	56

The rows represent the first number of the array index, i.e. `byte led[7][..]` and the columns represent the second number of the array index, i.e. `byte led[..][7]`. To access the number 158 in the 6th row and 4th column, you would use `byte led[5][3]`. Remember the indexes start at 0.

Notice that when you declared the array, you took the opportunity to initialize it with data at the same time. To do this with a two-dimensional array, you put the entire data within curly brackets and each set of data from the second index into its own curly bracket with a comma after it, like so:

```
byte led[8][8] = { {0, 56, 92, 158, 158, 130, 68, 56},
                   {0, 56, 124, 186, 146, 130, 68, 56},
                   {0, 56, 116, 242, 242, 130, 68, 56}, // etc, etc.
```

The two dimensional array will store the eight frames of your animation. The first index of the array will reference the frame of the animation and the second index which of the 8 rows of 8- bit numbers make up the pattern of LEDs to turn on and off. To save space in the code, the numbers have been converted from binary to decimal. If you were to see the binary numbers, you would make out the following animation in Figure 7-3.

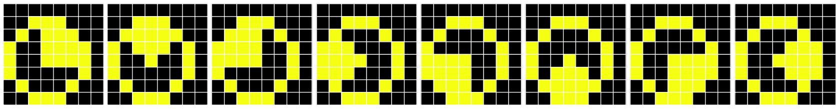


Figure 7-3. *The rolling wheel animation*

Of course, you can change this animation to anything you want and increase or decrease the number of frames, too. Draw out your animation on graph paper and then convert the rows to 8-bit binary numbers and put them into your array.

In the setup function, you set the three pins to output again, create a timer object with a length of 10000 microseconds, and attach the `screenUpdate()` function to the interrupt:

```
void setup() {
        pinMode(latchPin, OUTPUT);                          // set the 3 digital pins to outputs
        pinMode(clockPin, OUTPUT);
        pinMode(dataPin, OUTPUT);

        Timer1.initialize(10000);                           // set a timer of length 10000 microseconds
        Timer1.attachInterrupt(screenUpdate);               // attach the screenUpdate function
}
```

In the main loop, as in Project 19, you loop through the eight rows of the sprite. However, this loop is inside another loop that repeats eight times and this loop controls which frame you wish to display:

```
void loop() {
        for (int i=0; i<8; i++) {           // loop through all 8 frames of the animation
                for (int j=0; j<8; j++) {   // loop through the 8 rows per frame
```

Next, you get every element in the array, one by one, and bitshift the value left by one. However, using a neat little logic trick, you ensure that whatever bit is shifted off the left hand side rolls around back to the right hand side. This is done with the following command:

```
led[i][j]= led[i][j] << 1 | led[i][j] >> 7; // bitwise rotation
```

What is happening here is that the current element of the array, chosen by the integers i and j, is shifted one place to the left. However, you then take that result and logic OR the number with the value of `led[i][j]` that has been bit shifted seven places to the right. Let's take a look at how that works.

Let's say the current value of led[i][j] is 156. This is the binary number 10011100. If this number gets bit shifted left by one, you end up with 00111000. You now take the same number, 156, and bit shift it right seven times. You now have 00000001. In other words, you have shifted the far left binary digit from the left hand side to the right hand side. You now carry out a logical OR bitwise operation on the two numbers. Remember that the bitwise OR calculation will produce a one if there is a one in either digit, like so:

```
00111000  |
00000001  =
________
00111001
```

So, you have shifted the number left one place, and OR'ed that with the same number shifted right seven places. As you can see above, the result is the same as shifting the number left by one and shifting any digit that has shifted off the left back to the right hand side. This is known as a *bitwise rotation* or a *circular shift*, and this technique is frequently used in digital cryptography. You can carry out a bitwise rotation on any length unsigned integer data type using the calculation

```
i << n | i >> (a - n);
```

where n is the number of digits you wish to rotate the number by and a is the length, in bits, of your original digit.

Next, you increase the frame value by one, check that it isn't greater than seven, and if so, set it back to zero again. This will cycle through each of the eight frames of the animation one by one until you reach the end of the frames and then repeat. Finally, there is a delay of 100 milliseconds.

```
frame++;                              // go to the next frame in the animation
if (frame>7) { frame =0;}             // make sure we go back to frame 0 once past 7
delay(100);                           // wait a bit between frames
```

You then run the `screenUpdate()` and `shiftOut` functions as you did in the previous shift register-based projects. In the next project, you'll be using an LED dot-matrix again, but this time you won't be using shift registers. Instead, you'll use the popular MAX7219 chip.

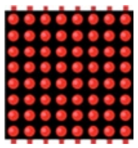
Project 21 – LED Dot-Matrix Display – Scrolling Message

There are many different ways to drive LEDs. Using shift registers is one way and they have their advantages. However, there are lots of ICs available that are specifically designed to drive LED displays and make life a lot easier for you. One of the most popular LED Driver ICs in the Arduino community is the MAX7219 serial interfaced, 8-digit LED Display Driver chips made by Maxim. These chips are designed to control 7-segment numeric LED displays of up to 8 digits, bar graph displays, or 8×8 LED dot-matrix displays, which is what you will be using them for. The Arduino IDE comes with a library called Matrix plus some example code that was specifically written for the MAX7219 chips. Using this library will make using these chips a breeze. However, in this project you are not going to use any external libraries. Instead, you are going to do things the hard way and write every piece of code yourself. That way, you will learn exactly how the MAX7219 chip works, and you can transfer these skills to utilizing any other LED driver chip you wish.

Parts Required

You will need a MAX7219 LED Driver IC. Alternatively, you can use an Austria Microsystems AS1107, which is pretty much identical to the MAX7219 and will work with no changes to your code or circuit. The 8x8 dot-matrix display needs to be oriented as column anode as we are outputting the characters one row at a time on the segment lines, which are designed to drive the anodes of the LEDs. See Table 7-4 for the full list of parts.

Table 7-4. *Parts Required for Project 21*

Bypass (Decoupling) Capacitor	
MAX7219 (or AS1107)	
Current-setting Resistor (39KΩ)	
8×8 Dot-matrix Display (C-)	

Connect It Up

Examine the diagram carefully in Figure 7-4. Make sure your Arduino is powered off while connecting the wires. The wiring from the MAX7219 to the dot-matrix display in Figure 7-4 is set up for the display unit I used for creating this project. The pinout of your display may well be different. So instead of relying on the connections shown in Figure 7-4, wire up the pins coming out of the MAX7219 to the appropriate column and row pins on your display (see Table 7-5 for the pinouts). Reading horizontally will show which two devices are connected and to what pins. On the display, the columns are the cathodes and the rows are the anodes. On my display, I found Row 1 was at the bottom and Row 8 the top. You may need to reverse the order on your own display if you find your letters are upside down or back to front. Connect the 5V from the Arduino to the positive rail of the breadboard and the ground to the ground rail.

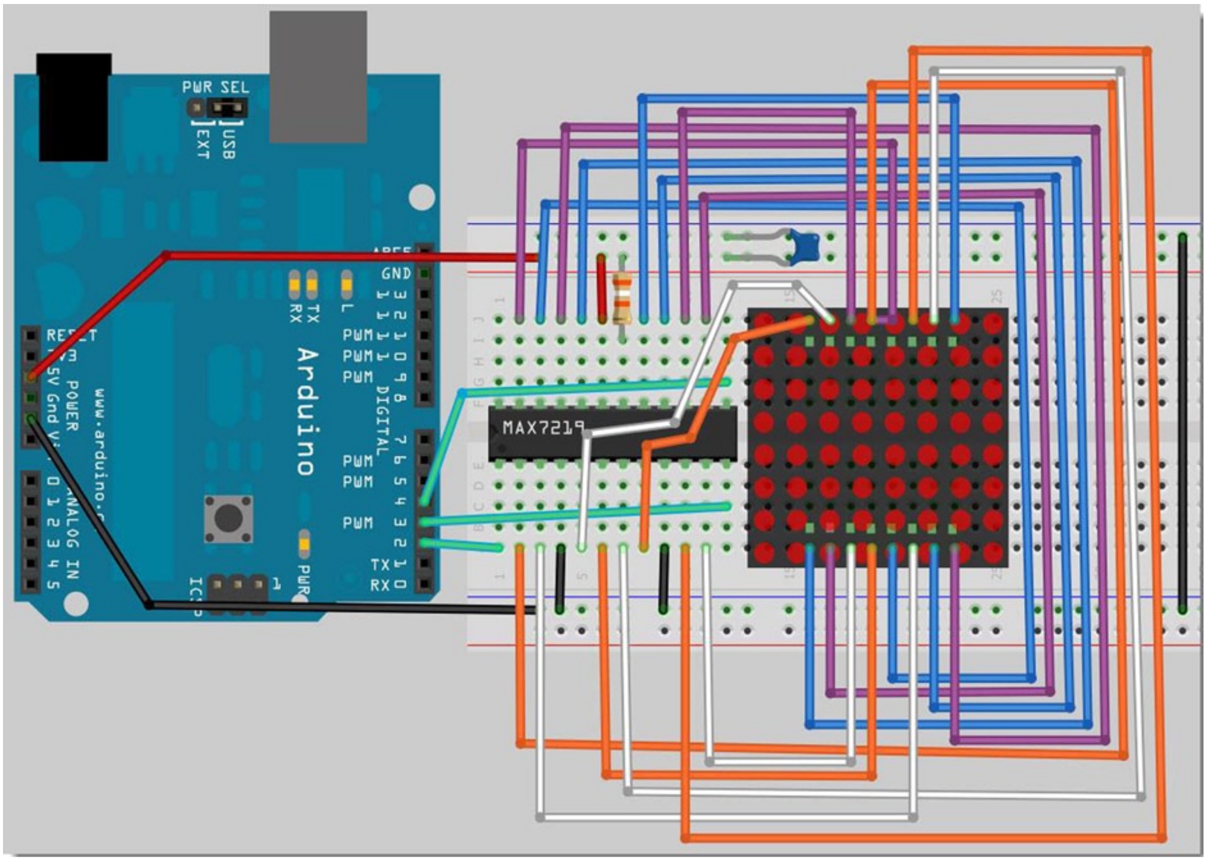


Figure 7-4. *The circuit for Project 21*

Table 7-5. *Pinouts between the Arduino, IC, and the Dot-matrix Display*

Arduino	MAX7219	Display	Other
Digital 2	1 (DIN)		
Digital 3	12 (LOAD)		
Digital 4	13 (CLK)		
	4, 9		Gnd
	19		+5v
	18 (ISET)		Resistor to +5v
	2 (DIG 0)	Column 1	
	11 (DIG 1)	Column 2	
	6 (DIG 2)	Column 3	
	7 (DIG 3)	Column 4	
	3 (DIG 4)	Column 5	
	10 (DIG 5)	Column 6	
	5 (DIG 6)	Column 7	
	8 (DIG 7)	Column 8	
	22 (SEG DP)	Row 1	
	14 (SEG A)	Row 2	
	16 (SEG B)	Row 3	
	20 (SEG C)	Row 4	
	23 (SEG D)	Row 5	
	21 (SEG E)	Row 6	
	15 (SEG F)	Row 7	
	17 (SEG G)	Row 8	

Check your connections before powering up the Arduino.

Enter the Code

Enter and upload the code in Listing 7-3.

Listing 7-3. Code for Project 21

```
#include <avr/pgmspace.h>
#include <TimerOne.h>

int DataPin = 2;   // Pin 1 on MAX
int LoadPin = 3;   // Pin 12 on MAX
int ClockPin = 4;  // Pin 13 on MAX
byte buffer[8];
```

```
#define SCAN_LIMIT_REG 0x0B
#define DECODE_MODE_REG 0x09
#define SHUTDOWN_REG 0x0C
#define INTENSITY_REG 0x0A

static byte font[][8] PROGMEM = {
// The printable ASCII characters only (32-126)
{B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000},
{B00000100, B00000100, B00000100, B00000100, B00000100, B00000100, B00000000, B00000100},
{B00001010, B00001010, B00001010, B00000000, B00000000, B00000000, B00000000, B00000000},
{B00000000, B00001010, B00011111, B00001010, B00011111, B00001010, B00011111, B00001010},
{B00000111, B00001100, B00010100, B00001100, B00000110, B00000101, B00000110, B00011100},
{B00011001, B00011010, B00000010, B00000100, B00000100, B00001000, B00001011, B00010011},
{B00000110, B00001010, B00010010, B00010100, B00001001, B00010110, B00010110, B00001001},
{B00000100, B00000100, B00000100, B00000000, B00000000, B00000000, B00000000, B00000000},
{B00000010, B00000100, B00001000, B00001000, B00001000, B00001000, B00000100, B00000010},
{B00001000, B00000100, B00000010, B00000010, B00000010, B00000010, B00000100, B00001000},
{B00010101, B00001110, B00011111, B00001110, B00010101, B00000000, B00000000, B00000000},
{B00000000, B00000000, B00000100, B00000100, B00011111, B00000100, B00000100, B00000000},
{B00000000, B00000000, B00000000, B00000000, B00000000, B00000110, B00000100, B00001000},
{B00000000, B00000000, B00000000, B00000000, B00001110, B00000000, B00000000, B00000000},
{B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000100},
{B00000001, B00000010, B00000010, B00000100, B00000100, B00001000, B00001000, B00010000},
{B00001110, B00010001, B00010011, B00010001, B00010101, B00010001, B00011001, B00001110},
{B00000100, B00001100, B00010100, B00000100, B00000100, B00000100, B00000100, B00011111},
{B00001110, B00010001, B00010001, B00000010, B00000100, B00001000, B00010000, B00011111},
{B00001110, B00010001, B00000001, B00001110, B00000001, B00000001, B00010001, B00001110},
{B00010000, B00010000, B00010100, B00010100, B00011111, B00000100, B00000100, B00000100},
{B00011111, B00010000, B00010000, B00011110, B00000001, B00000001, B00000001, B00011110},
{B00000111, B00001000, B00010000, B00011110, B00010001, B00010001, B00010001, B00001110},
{B00011111, B00000001, B00000001, B00000001, B00000010, B00000100, B00001000, B00010000},
{B00001110, B00010001, B00010001, B00001110, B00010001, B00010001, B00010001, B00001110},
{B00001110, B00010001, B00010001, B00001111, B00000001, B00000001, B00000001, B00000001},
{B00000000, B00000100, B00000100, B00000000, B00000000, B00000100, B00000100, B00000000},
{B00000000, B00000100, B00000100, B00000000, B00000000, B00000100, B00000100, B00001000},
{B00000001, B00000010, B00000100, B00001000, B00001000, B00000100, B00000010, B00000001},
{B00000000, B00000000, B00000000, B00011110, B00000000, B00011110, B00000000, B00000000},
{B00010000, B00001000, B00000100, B00000010, B00000010, B00000100, B00001000, B00010000},
{B00001110, B00010001, B00010001, B00000010, B00000100, B00000100, B00000000, B00000100},
{B00001110, B00010001, B00010001, B00010101, B00010101, B00010001, B00010001, B00011110},
{B00001110, B00010001, B00010001, B00010001, B00011111, B00010001, B00010001, B00010001},
{B00011110, B00010001, B00010001, B00011110, B00010001, B00010001, B00010001, B00011110},
{B00000111, B00001000, B00010000, B00010000, B00010000, B00010000, B00001000, B00000111},
{B00011100, B00010010, B00010001, B00010001, B00010001, B00010001, B00010010, B00011100},
{B00011111, B00010000, B00010000, B00011110, B00010000, B00010000, B00010000, B00011111},
{B00011111, B00010000, B00010000, B00011110, B00010000, B00010000, B00010000, B00010000},
{B00001110, B00010001, B00010000, B00010000, B00010111, B00010001, B00010001, B00001110},
{B00010001, B00010001, B00010001, B00011111, B00010001, B00010001, B00010001, B00010001},
{B00011111, B00000100, B00000100, B00000100, B00000100, B00000100, B00000100, B00011111},
{B00011111, B00000100, B00000100, B00000100, B00000100, B00000100, B00010100, B00001000},
{B00010001, B00010010, B00010100, B00011000, B00010100, B00010010, B00010001, B00010001},
{B00010000, B00010000, B00010000, B00010000, B00010000, B00010000, B00010000, B00011111},
{B00010001, B00011011, B00011111, B00010101, B00010001, B00010001, B00010001, B00010001},
{B00010001, B00011001, B00011001, B00010101, B00010101, B00010011, B00010011, B00010001},
{B00001110, B00010001, B00010001, B00010001, B00010001, B00010001, B00010001, B00001110},
{B00011110, B00010001, B00010001, B00011110, B00010000, B00010000, B00010000, B00010000},
{B00001110, B00010001, B00010001, B00010001, B00010001, B00010101, B00010011, B00001111},
{B00011110, B00010001, B00010001, B00011110, B00010100, B00010010, B00010001, B00010001},
{B00001110, B00010001, B00010000, B00001000, B00000110, B00000001, B00010001, B00001110},
{B00011111, B00000100, B00000100, B00000100, B00000100, B00000100, B00000100, B00000100},
{B00010001, B00010001, B00010001, B00010001, B00010001, B00010001, B00010001, B00001110},
{B00010001, B00010001, B00010001, B00010001, B00010001, B00010001, B00001010, B00000100},
{B00010001, B00010001, B00010001, B00010001, B00010001, B00010101, B00010101, B00001010},
{B00010001, B00010001, B00001010, B00000100, B00000100, B00001010, B00010001, B00010001},
{B00010001, B00010001, B00001010, B00000100, B00000100, B00000100, B00000100, B00000100},
{B00011111, B00000001, B00000010, B00000100, B00001000, B00010000, B00010000, B00011111},
{B00001110, B00001000, B00001000, B00001000, B00001000, B00001000, B00001000, B00001110},
{B00010000, B00001000, B00001000, B00000100, B00000100, B00000010, B00000010, B00000001},
{B00001110, B00000010, B00000010, B00000010, B00000010, B00000010, B00000010, B00001110},
{B00000100, B00001010, B00010001, B00000000, B00000000, B00000000, B00000000, B00000000},
{B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00011111},
{B00001000, B00000100, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000},
{B00000000, B00000000, B00000000, B00001110, B00010010, B00010010, B00010010, B00001111},
{B00000000, B00010000, B00010000, B00010000, B00011100, B00010010, B00010010, B00011100},
{B00000000, B00000000, B00000000, B00001110, B00010000, B00010000, B00010000, B00001110},
{B00000000, B00000001, B00000001, B00000001, B00000111, B00001001, B00001001, B00000111},
{B00000000, B00000000, B00000000, B00011100, B00010010, B00011110, B00010000, B00001110},
{B00000000, B00000011, B00000100, B00000100, B00000110, B00000100, B00000100, B00000100},
{B00000000, B00001110, B00001010, B00001010, B00001110, B00000010, B00000010, B00001100},
{B00000000, B00010000, B00010000, B00010000, B00011100, B00010010, B00010010, B00010010},
{B00000000, B00000000, B00000100, B00000000, B00000100, B00000100, B00000100, B00000100},
{B00000000, B00000010, B00000000, B00000010, B00000010, B00000010, B00000010, B00001100},
{B00000000, B00010000, B00010000, B00010100, B00011000, B00011000, B00010100, B00010000},
{B00000000, B00010000, B00010000, B00010000, B00010000, B00010000, B00010000, B00001100},
{B00000000, B00000000, B00000000, B00001010, B00010101, B00010001, B00010001, B00010001},
{B00000000, B00000000, B00000000, B00010100, B00011010, B00010010, B00010010, B00010010},
{B00000000, B00000000, B00000000, B00001100, B00010010, B00010010, B00010010, B00001100},
{B00000000, B00011100, B00010010, B00010010, B00011100, B00010000, B00010000, B00010000},
{B00000000, B00001110, B00010010, B00010010, B00001110, B00000010, B00000010, B00000001},
{B00000000, B00000000, B00000000, B00001010, B00001100, B00001000, B00001000, B00001000},
{B00000000, B00000000, B00001110, B00010000, B00001000, B00000100, B00000010, B00011110},
{B00000000, B00010000, B00010000, B00011100, B00010000, B00010000, B00010000, B00001100},
{B00000000, B00000000, B00000000, B00010010, B00010010, B00010010, B00010010, B00001100},
{B00000000, B00000000, B00000000, B00010001, B00010001, B00010001, B00001010, B00000100},
{B00000000, B00000000, B00000000, B00010001, B00010001, B00010001, B00010101, B00001010},
{B00000000, B00000000, B00000000, B00010001, B00001010, B00000100, B00001010, B00010001},
{B00000000, B00000000, B00010001, B00001010, B00000100, B00001000, B00001000, B00010000},
{B00000000, B00000000, B00000000, B00011111, B00000010, B00000100, B00001000, B00011111},
{B00000010, B00000100, B00000100, B00000100, B00001000, B00000100, B00000100, B00000010},
{B00000100, B00000100, B00000100, B00000100, B00000100, B00000100, B00000100, B00000100},
{B00001000, B00000100, B00000100, B00000100, B00000010, B00000100, B00000100, B00001000},
{B00000000, B00000000, B00000000, B00001010, B00011110, B00010100, B00000000, B00000000}
};
```

```
void clearDisplay() {
  for (byte x=0; x<8; x++) {
  buffer[x] = B00000000;
  }
  screenUpdate();
}

void initMAX7219() {
      pinMode(DataPin, OUTPUT);
      pinMode(LoadPin, OUTPUT);
      pinMode(ClockPin, OUTPUT);
      clearDisplay();
      writeData(SCAN_LIMIT_REG, B00000111);  // scan limit set to 0:7
      writeData(DECODE_MODE_REG, B00000000); // decode mode off
      writeData(SHUTDOWN_REG, B00000001);    // Set shutdown register to normal operation
      intensity(15);                         // Values 0 to 15 only (4 bit)
}

void intensity(int intensity) {
        writeData(INTENSITY_REG, intensity); //B0001010 is the Intensity Register
}

void writeData(byte msb, byte lsb) {
   digitalWrite(LoadPin, LOW);               // set loadpin ready to receive data
   shiftOut(DataPin, ClockPin, MSBFIRST, (msb));
   shiftOut(DataPin, ClockPin, MSBFIRST, (lsb));
   digitalWrite(LoadPin, HIGH);              // latch the data
}

void scroll(char myString[], int rate) {

byte firstChrRow, secondChrRow;
byte ledOutput;
byte chrIndex = 0;                           // Initialise the string position index
byte Char1, Char2;
byte scrollBit = 0;
byte strLength = 0;

unsigned long time;
unsigned long counter;

    while (myString[strLength]) {            // Increment count till we reach the end of the string
    strLength++;}
     counter = millis();
  while (chrIndex < (strLength)) {
   time = millis();
  if (time > (counter + rate)) {
     Char1 = constrain(myString[chrIndex],32,126);
     Char2 = constrain(myString[chrIndex+1],32,126);
     for (byte y= 0; y<8; y++) {
         firstChrRow = pgm_read_byte(&font[Char1 - 32][y]);
        secondChrRow = (pgm_read_byte(&font[Char2 - 32][y])) << 1;
```

```
          ledOutput = (firstChrRow << scrollBit)
                      | (secondChrRow >> (8 - scrollBit) );
            buffer[y] = ledOutput;
          }
        scrollBit++;
        if (scrollBit > 6) {
        scrollBit = 0;
        chrIndex++;
        }
              counter = millis();
       }
     }
   }

   void screenUpdate() {
    for (byte row = 0; row < 8; row++) {
     writeData(row+1, buffer[row]);
    }
   }

   void setup() {
       initMAX7219();
       Timer1.initialize(10000);                // initialize timer1 and set interrupt period
       Timer1.attachInterrupt(screenUpdate);
       Serial.begin(9600);
   }

   void loop() {
     clearDisplay();
     scroll("   BEGINNING ARDUINO   ", 45);
     scroll("   Chapter 7 - LED Displays   ", 45);
     scroll("   HELLO WORLD!!!  :)   ", 45);
   }
```

When you upload the code, you will see a message scroll across the display. You can of course change the text in the loop function to your own.

Project 21 – LED Dot-Matrix – Scrolling Message – Hardware Overview

Again, to make it easier to understand the code you will need to know how the MAX7219 chip works first so let's look at the hardware before the code.

The MAX7219 operates very similarly to the shift registers in that you have to enter data in a serial fashion, bit by bit. A total of 16 bits must be loaded into the device at a time. The chip is easy to use, and it uses just three pins from the Arduino. Digital Pin 2 goes to Pin 1 on the MAX, which is the Data In. Digital Pin 3 goes to Pin 12 on the MAX, which is LOAD and finally, Digital Pin 4 goes to Pin 13 on the MAX, which is the clock. See Figure 7-5 for the pinouts of the MAX7219.

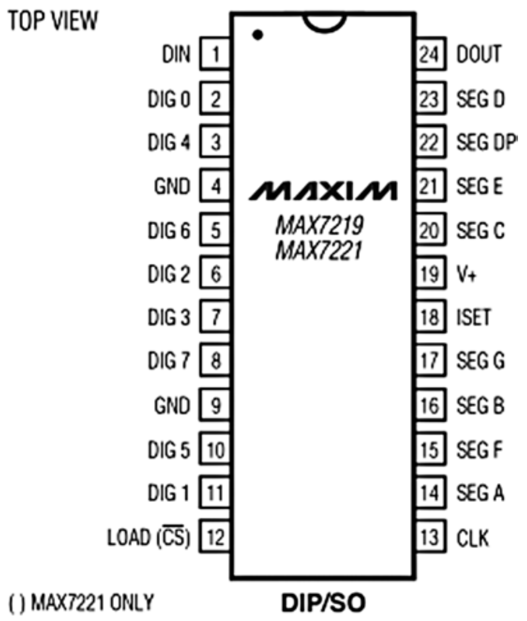


Figure 7-5. *Pin diagram for the MAX7219*

The LOAD pin is pulled low and the first bit of the data is set as either HIGH or LOW at the DIN pin. The CLK pin is set to oscillate between LOW and HIGH by the Arduino. On the rising edge of the clock pulse, the bit at the DIN pin is shifted into the internal register. The clock pulse then falls to LOW and the next data bit is set at the DIN pin before the process repeats. After all 16 bits of data have been pushed into the register as the clock falls and rises 16 times, the LOAD pin is finally set to HIGH and this latches the data into the register. Figure 7-6 is the timing diagram from the MAX7219 datasheet and shows how the three pins need to be manipulated to send data bits D0 to D15 into the device. The DOUT pin, which is pin 24, is not used in this project. But, if you had more than one MAX7219 chip daisy chained together, the DOUT of the first chip is connected to the DIN of the second and so on. Data is clocked out of the DOUT pin on the falling edge of the clock cycle.

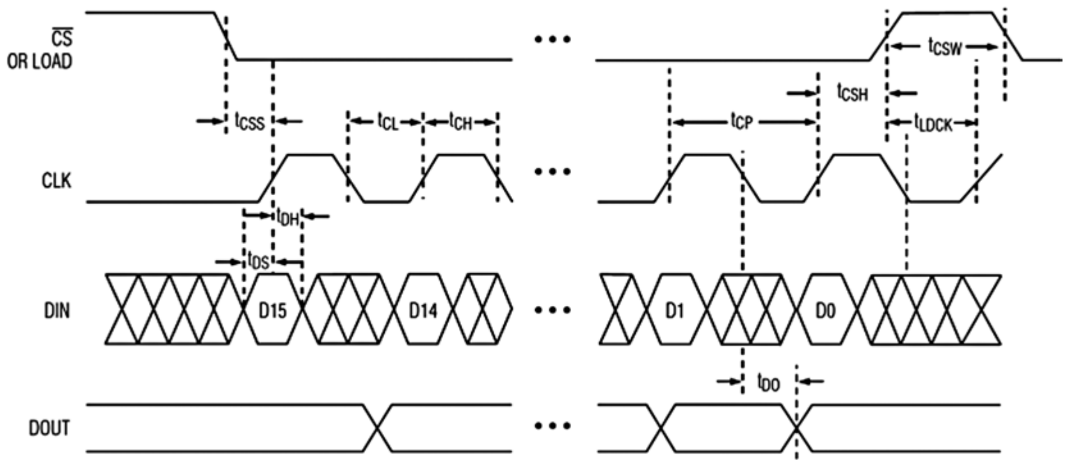


Figure 7-6. *Timing diagram for the MAX7219*

You need to recreate this timing sequence in your code to be able to send the appropriate codes to the chip. The chip can source a current of up to 100mA, which is more than enough for most dot matrix displays.
If you wish to read the datasheet for the MAX7219, you can download it from Maxim at
http://datasheets.maxim-ic.com/en/ds/MAX7219-MAX7221.pdf

The device accepts data in 16 bit packets (or commands). D15 or the msb (most significant bit) is sent first so the order goes from D15 down to D0. The first 4 bits are "don't care" bits, i.e. they are not used by the IC so they can be anything. The next 4 bits make up the register address and then the final 8 bits make up the data. Table 7-6 shows the serial data format and Table 7-7 shows the Register Address Map.

Table 7-6. Serial Data Format (16 bits) of the MAX7219

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X		ADDRESS			MSB			DATA			LSB	
											X				
											X				
											X				
											X				
											X				
											X				
											X				

Table 7-7. Register Address Map of the MAX7219

REGISTER	ADDRESS					HEX CODE
	D15-D12	D11	D10	D9	D8	
No-Op	X	0	0	0	0	0xX0
Digit 0	X	0	0	0	1	0xX1
Digit 1	X	0	0	1	0	0xX2
Digit 2	X	0	0	1	1	0xX3
Digit 3	X	0	1	0	0	0xX4
Digit 4	X	0	1	0	1	0xX5
Digit 5	X	0	1	1	0	0xX6
Digit 6	X	0	1	1	1	0xX7
Digit 7	X	1	0	0	0	0xX8
Decode Mode	X	1	0	0	1	0xX9
Intensity	X	1	0	1	0	0xXA
Scan Limit	X	1	0	1	1	0xXB
Shutdown	X	1	1	0	0	0xXC
Display Test	X	1	1	1	1	0xXF

For example, as you can see from the register address map in Table 7-7, the address for the intensity register is 1010 binary. The intensity register sets the brightness of the display with values from the dimmest at 0 to the brightest at 15 (B000 to B1111). To set the intensity to 15 (maximum), you would send the following 16 bits with the most significant bit (the bit on the far left) being sent first and the least significant bit (the bit at the far right) being sent last:

```
0000101000001111
```

The second 8 bits is the data being sent to the intensity register. The first 4 bits of the intensity register are again "don't care" so you send B0000. The next 4 bits determine the intensity. In this case, you want the maximum intensity, which is a value of B1111. By sending out these 16 bits to the device, you set the display intensity to maximum. The entire 16 bit value you want to send is B0000101000001111, as it is sent msb (most significant bit) first and lsb (least significant bit) last.

Another address you will be using is the scan limit. Remember that the MAX7219 is designed to work with 7-segment LED displays (see Figure 7-7).

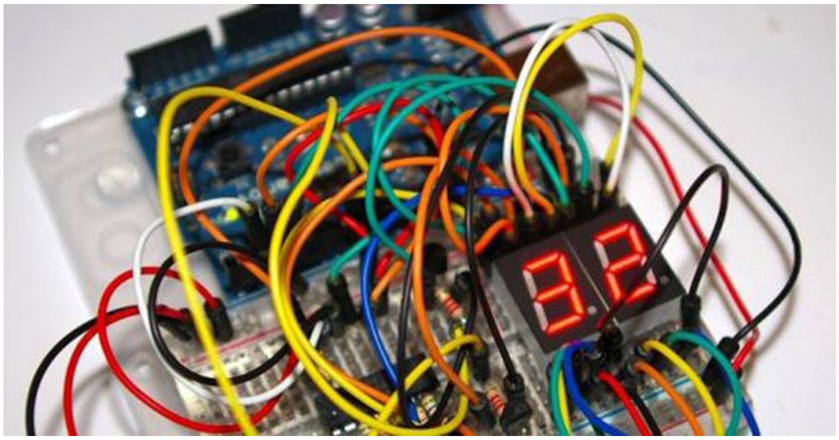


Figure 7-7. *7-segment LED display (image by Tony Jewell)*

The scan limit decides how many of the 8 digits are to be lit. In your case, you are not using 7-segment displays, but 8×8 dot-matrix displays. We are using all 8 digit lines (0..7) driving our display, so we set the scan limit register to B00000111 (drive digits 0..7).

The decode mode register is only relevant if you are using 7-segment displays, so it will be set to B00000000 to turn decode off.

Finally, you will set the shutdown register to B00000001 to ensure it is in normal operation and not shutdown mode. If you set the shutdown register to B00000000, then the current sources are all pulled to ground, which then blanks the display.

For further information about the MAX7219 IC, read the datasheet. Just read the parts of the datasheet that are relevant to your project and you will see that it is a lot easier to understand than it appears at first.

Now that you (hopefully) understand how the MAX7219 works, let's take a look at the code and see how to make it display scrolling text.

Project 21 – LED Dot-Matrix – Scrolling Message – Code Overview

The first thing you do at the start of the sketch is to load in the two libraries that you will be utilizing in the code:

```
#include <avr/pgmspace.h>
#include <TimerOne.h>
```

The first library is the `pgmspace` or Program Space utilities. The functions in this library allow your program to access data stored in program space or flash memory. The Arduino with the ATmega328 chip has 32KB of flash memory (2KB of this is used by the bootloader, so 30KB is available). The Arduino Mega has 128KB of flash memory, 4KB of which is used by the bootloader. The program space is exactly that, the space that your program will be stored in. You can utilize the free unused space in the flash memory by using the Program Space utilities. This is where you will store the extremely large 2D array that will hold the font for your characters.

The second library is the `TimerOne` library that was first used in Project 19. Next, the three Digital Pins that will interface with the MAX7219 are declared:

```
int DataPin = 2;  // Pin 1 on MAX
int LoadPin = 3;  // Pin 12 on MAX
int ClockPin = 4; // Pin 13 on MAX
```

Then you create an array of type buffer with 8 elements:

```
byte buffer[8];
```

This array will store the pattern of bits that will decide what LEDs are on or off when the display is active.

Next we define 4 addresses within the MAX7219 chip that we will use later in the code.

```
#define SCAN_LIMIT_REG 0x0B
#define DECODE_MODE_REG 0x09
#define SHUTDOWN_REG 0x0C
#define INTENSITY_REG 0x0A
```

Next comes a large 2D array of type byte:

```
static byte font[][8] PROGMEM = {
// The printable ASCII characters only (32-126)
{B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000, B00000000},
{B00000100, B00000100, B00000100, B00000100, B00000100, B00000100, B00000000, B00000100},
..etc
```

This array is storing the pattern of bits that make up the font you will use to show text on the display. It is a two-dimensional array of type `static byte`. You have also added after the array declaration the command `PROGMEM`. This is an attribute from the Program Space utilities and it tells the compiler to store this array in the flash memory instead of the SRAM (Static Random Access memory).

SRAM is the memory space on the ATmega chip that is normally used to store the variables and character strings used in your sketch. They are copied from program space and into SRAM when used. However, the array used to store the font is made up of 96 characters made up of eight bytes each. The array is 95 x 8 elements, which is 760 elements in total, and each element is a byte (8 bits). The font therefore takes up 760 bytes in total. The ATmega328 has only 2KB, or approximately 2000 bytes of memory space for variables. Once you add to that the other variables and strings of text used in the program, you run the risk of running out of memory fast.

The Arduino has no way of warning you that it is running out of memory. Instead it just crashes. To prevent this from happening, you are storing this array in the flash memory instead of SRAM, as it has much more space for you to play with. The sketch is around 2800 bytes and the array is just under 800 bytes, so you have used about 3.6KB out of the 30KB flash memory available to you.

Then you start to create the various functions you will require for the program. The first one will simply clear the display. Whatever bits are stored in the buffer array will be displayed on the matrix. The `clearDisplay()` function simply cycles through all eight elements of the array and sets their values to zero so that no LEDs are lit and the display is blank. It then calls the `screenUpdate()` function that displays the pattern stored in the buffer[] array on the matrix. In this case, as the buffer contains nothing but zeros, nothing will be displayed.

```
void clearDisplay() {
        for (byte x=0; x<8; x++) {
                buffer[x] = B00000000;
        }
        screenUpdate();
}
```

The next function, `initMAX7219()`, has the job of setting up the MAX7219 chip ready for use. First, the three pins are set to OUTPUT:

```
void initMAX7219() {
        pinMode(DataPin, OUTPUT);
        pinMode(LoadPin, OUTPUT);
        pinMode(ClockPin, OUTPUT);
```

The display is then cleared:

```
clearDisplay();
```

The scan limit is set to 7, decode mode is turned off, and the shutdown register is set to normal operation:

```
writeData(SCAN_LIMIT_REG, B00000111);        // scan limit set to 0:7
writeData(DECODE_MODE_REG, B00000000);       // decode mode off
writeData(SHUTDOWN_REG, B00000001);          // Set shutdown register to normal operation
```

Then the intensity is set to maximum by calling the `intensity()` function:

```
intensity(15); // Values 0 to 15 only (4 bit)
```

Next comes the `intensity()` function itself, which simply takes the value passed to it and writes it to the intensity register by calling the `writeData` function:

```
void intensity(int intensity) {
        writeData(INTENSITY_REG, intensity); //B0001010 is the Intensity Register
}
```

The next function does most of the hard work. Its job is to write the data out to the MAX7219 one bit at a time using the Arduino's shiftOut function. The function requires two parameters, both bytes, and these make up the Most Significant Byte (not bit) and Least Significant Byte of the 16-bit number:

```
void writeData(byte MSB, byte LSB) {
```

Next, the loadPin is set to LOW. This unlatches the data in the IC's register ready to receive new data:

```
digitalWrite(LoadPin, LOW);                  // set loadpin ready to receive data
```

Next we use the shiftOut() function to shift out two bytes to the MAX7219. The two bytes are msb (Most Significent Byte) and lsb (Least Significent Byte). The msb is the row of the matrix and the lsb is the dot pattern for the relevant character.

```
shiftOut(DataPin, ClockPin, MSBFIRST, (msb));
shiftOut(DataPin, ClockPin, MSBFIRST, (lsb));
```

Then the loadPin is set to low to copy the contents of the shift registers over to the output pins.

```
digitalWrite(LoadPin, HIGH);                 // latch the data
```

Next is the `scroll()` function, which is what displays the appropriate characters of the text string on the display. The function accepts two parameters, the first being the text string you wish to display and the second is the rate in which you wish the scrolling to occur in milliseconds between refreshes:

```
void scroll(char myString[], int rate) {
```

Then two variables of type byte are set up. These will store one of the eight rows of bit patterns that make up the particular character being displayed:

```
byte firstChrRow, secondChrRow;
```

Another byte is declared and called `ledOutput`. This will store the result of a calculation on the first bit pattern and second bit pattern of the characters, and it will decide which LEDs are on or off (this will be explained shortly):

```
byte ledOutput;
```

Another variable of type byte is declared and called `chrIndex` and initialized to zero. chrIndex will store the current position in the text string being displayed, starting at zero and incrementing up to the length of the string:

```
byte chrIndex = 0;                           // Initializse the string position pointer
```

Another two bytes are declared. These will hold the current character and the next one in the string:

```
byte Char1, Char2;                           // the two characters that will be displayed
```

These differ from `firstChrRow` and `secondChrRow` in that they store the ASCII (American Standard Code for Information Interchange) value of the character to be displayed and the next one in the string. `firstChrRow` and `secondChrRow` store the pattern of bits that make up the letters to be displayed.

All letters, numbers, symbols, etc. that can be displayed on a computer screen or sent via serial have an ASCII code. This is simply an index number to state which character it is in the ASCII table. Characters 0 to 31 are control codes, and you will not be using those as they cannot be displayed on your dot matrix display. You will use ASCII characters 32 to 196 which are the 95 printable characters. These start at number 32, which is a space, and go up to 126, which is the tilde (~) symbol. The printable ASCII characters are listed in Table 7-8.

Table 7-8. *The Printable ASCII Characters*

```
 !"#$%&'()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~
```

Another byte is declared and initialized to zero. This will store how many bits the character pattern of the current set of letters need to be shifted to give the impression of scrolling from right to left:

```
byte scrollBit = 0;
```

Another byte will hold the length of the string of characters. This is initialized to zero:

```
byte strLength = 0;
```

Then two variables of type unsigned long are declared and these will store the current time in milliseconds since the Arduino chip was booted up or reset and another one to store the same value, but this time after a `while` routine has run. Together these will ensure that the bits are shifted only after a specified time, in milliseconds, so it scrolls at a readable speed:

```
unsigned long time;
unsigned long counter;
```

You now need to find out how many characters are in the string. There are several ways of doing this, but in your case you simply set up a `while` loop that checks if there is data in the current array index, which is `strLength` (initialized to zero), and if so, increments the `strLength` variable by one. The loop then repeats until the condition of `myString[strLength]` is false, i.e. there are no more characters in the string, and then `strLength`, which has been incremented by one on each iteration, will now hold the length of the string:

```
while (myString[strLength]) {strLength++;}
```

Next, you set the value of counter to the value of `millis()`. You came across `millis()` in Project 4. It stores the value, in milliseconds, since the Arduino was turned on or reset:

```
counter = millis();
```

A `while` loop will now run on the condition that the current character position is smaller than the string length minus one:

```
while (chrPointer < (strLength-1)) {
```

The variable `time` is set to the current value of `millis()`:

```
time = millis();
```

An `if` statement then checks if the current time is greater than the last time stored plus the value in movement_interval, i.e. 45 milliseconds, and if so, runs the code block within it:

```
if (time > (counter + rate)) {
```

`Char1` is loaded with the ASCII character value of the character at `chrPointer` in the `myString` array and `Char2` with the one after that:

```
Char1 = constrain(myString[chrIndex],32,126);
Char2 = constrain(myString[chrIndex+1],32,126);
```

Here we have used the Arduino's constrain() function to ensure that only visible ASCII characters are printed. Any non-printable characters such as the null terminal or line feed are outside of the printable range and we don't wish to try and print those. Therefore the constrain function takes three parameters, the first is the number we wish to constrain, the next is the lower range and the third is the upper range. Any number lower than the lower range will be constrained to that number and any number higher than the upper range is constrained to the upper range number. i.e. Any character with a code less than 32 is made into 32, which is a space, and any character with a code higher than 126 is changed to 126, which is a ~.

A `for` loop now iterates through each of the eight rows:

```
for (byte y= 0; y<8; y++) {
```

You now read the font array and put the bit pattern in the current row of eight into `firstChrRow` and the second into `secondChrRow`. Remember that the font array is storing the bit patterns that make up the characters in the ASCII table, but only the printable ones from 32 to 126. The first index of the array is an index made up of the ASCII code of the character -32 (as we don't use the first 32 characters in the ASCII table) and the second element of the array stores the eight rows of bit patterns that make up that character. For example, the letters A and Z are ASCII characters 65 and 90, respectively. You deduct 32 from these numbers to give you your array index.

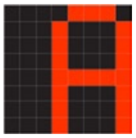
So the bit pattern for the letter A, which is ASCII code 65, is stored in array element 33 (65-32) and the second dimension of the array at that index stores the eight bit patterns that make up the letter. The letter Z is ASCII code 90, which is index number 58 in the array. The data in font[33][0...7], for the letter A, is

```
{B00001110, B00010001, B00010001, B00010001, B00011111, B00010001, B00010001, B00010001},
```

If you put that data on top of each other so you can see it clearer, you have

```
B00001110
B00010001
B00010001
B00010001
B00011111
B00010001
B00010001
B00010001
```

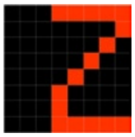
and if you look closely, you will see the following pattern, which makes up the letter A:



For the letter Z, the data in the array is

```
B00011111
B00000001
B00000010
B00000100
B00001000
B00010000
B00010000
B00011111
```

which corresponds with the LED bit pattern of



To read the bit pattern, you need to access the font, which is stored in program space and not SRAM, as is usual. To do this, you need to make use of one of the utilities from the pgmspace library, `pgm_read_byte`.

```
firstChrRow = pgm_read_byte(&font[Char1 - 32][y]);
secondChrRow = (pgm_read_byte(&font[Char2 - 32][y])) << 1;
```

When you access program space, you are obtaining data held in the flash memory. .To do so, you need to know the address in memory where the data is stored. (Each storage location in memory has a unique address number.)

To do that, you use the & symbol in front of a variable. When you do that, you do not read the data in that variable, you read the address where the data is stored instead. The `pgm_read_byte` command needs to know the flash memory address of the data you want to retrieve, so you put a & symbol in front of `font[Char1 - 32][y]` to make `pgm_read_byte(&font[Char1 - 32][y])`. This simply means that you read the byte in program space stored at the address of `font[Char1 - 32][y]`.

The value of `secondChrRow` is bitshifted left one simply to make the gap between letters smaller, thereby making them more readable on the display. This is because there are no bits used to the left of all characters for three spaces. You could bitshift it left by two to bring them closer but it starts to become hard to read if you do.

The next line loads the bit pattern for the relevant row into `ledOutput`:

```
ledOutput = (firstChrRow << scrollBit) | (secondChrRow >> (8 - scrollBit) );
```

As you want the letters to scroll from right to left, you bitshift left the first letter by `scrollBit` amount of times and the second letter by 8 – `scrollBit` amount of times. You then logical OR the results together to merge them into the 8-bit pattern required to display. For example, if the letters you were displaying were A and Z, then the patterns for both would be

```
B00001110  B00011111
B00010001  B00000001
B00010001  B00000010
B00010001  B00000100
B00011111  B00001000
B00010001  B00010000
B00010001  B00010000
B00010001  B00011111
```

So, the calculation above on the top row, when `scrollBit` is set to 5, i.e. the letters have scrolled five pixels to the left, would be

```
B110000000  B000000111
```

which is the top row of the A bitshifted left 5 times and the top row of the Z bitshifted right 3 times (8-5). You can see that the left-hand pattern is what you get when the letter A is scrolled left by five pixels and the right-hand pattern is what you get if the letter Z was scrolled in from the right-hand side by 5 pixels. The logical OR, which is the | symbol, has the effect of merging these two patterns together to create

```
B11000111
```

which is what you would get if the letter A and Z were next to each other and scrolled left five pixels.

The next line loads that pattern of bits into the appropriate row of the screen buffer:

```
buffer[y] = ledOutput;
```

The scrollBit is increased by one:

```
scrollBit++;
```

Then an `if` statement checks if the `scrollBit` value has reached 7. If so, it sets it back to zero and increases the `chrPointer` by one so the next time the function is called, it will display the next two sets of characters:

```
if (scrollBit > 6) {
        scrollBit = 0;
        chrPointer++;
}
```

Finally, the value of counter is updated to the latest `millis()` value:

```
counter = millis();
```

The `screenUpdate()` function simply takes the eight rows of bit patterns you have loaded into the eight-element buffer array and writes it to the chip, which in turn displays it on the matrix:

```
void screenUpdate() {
        for (byte row = 0; row < 8; row++) {
                writeData(row+1, buffer[row]);
        }
}
```

After setting up these six functions, you finally reach the `setup()` and `loop()` functions of the program. In `setup()`, the chip is initialized by calling `initMax7219()`, a timer is created and set to a refresh period of 10000 microseconds, and the `screenUpdate()` function attached. As before, this ensures the `screenUpdate()` function is activated every 10000 microseconds no matter what else is going on.

```
void setup() {
        initMAX7219();
        Timer1.initialize(10000);         // initialize timer1 and set interrupt period
        Timer1.attachInterrupt(screenUpdate);
}
```

Finally, the main loop of the program simply has four lines. The first clears the display and then the next three call the scroll routine to display the three lines of text and set them scrolling across the display.

```
void loop() {
        clearDisplay();
        scroll("   BEGINNING ARDUINO   ", 45);
        scroll("   Chapter 7 - LED Displays   ", 45);
        scroll("   HELLO WORLD!!!  :)   ", 45);
}
```

You can, of course, change the text in the code so that the display says anything you wish. Project 21 was pretty complex as far as the code goes. As I said at the start, all of this hard work could have been avoided if you had simply used some of the pre-existing LED matrix libraries that are available in the public domain. But in doing so, you would not have learned how the MAX7219 chip works. By doing things the hard way, you should now have a good understanding of the MAX7219 chip and how to control it. These skills can be used to operate just about any other external IC as the principles are pretty much the same.

In the next project, you will make use of these libraries so you can see how they make life easier for you. Let's take your display and have a bit of fun with it.

Project 22 – LED Dot Matrix Display – Pong Game

Project 21 was hard going and a lot to take in. So, for Project 22 you are going to create a simple game with simple code using the dot matrix display and a potentiometer. This time you are going to use one of the many available libraries for controlling LED dot matrix displays to see how much easier it can make your life when coding.

Parts Required

The parts required are the same as Project 21 with the addition of a 10KΩ Linear Potentiometer (Table 7-8).

Table 7-8. *Additional parts for Project 22*

Same as Project 21 plus....	
10KΩ Linear Potentiometer	

Connect It Up

Leave the circuit the same as in Project 21 and add a linear potentiometer.

Potentiometers come in either linear or logarithmic versions. A linear potentiometer has its resistance proportional to the distance between the contact (wiper) and one end terminal inside. A logarithmic potentiometer is made of a material that varies in resistance from end of the pot to the other. This results in a device where the output voltage is a logarithmic function of the slider position.

A linear potentiometer is used here because we want the same proportional amount of motion at each end of the potentiometer's travel. Logarithmic potentiometers are used in audio applications because some sound adjustments need a non-linear logarithmic control applied. Then the control can be marked in steps that have the same apparent size. If you use the wrong type here, your pong bat will move faster on one side of the screen than the other.

The left and right pins go to Ground and +5V, respectively, and the center pin goes to Analog Pin 5.

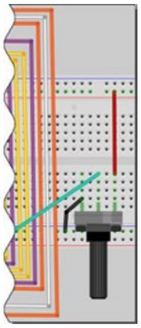


Figure 7-8. *Add a potentiometer to the Project 21 circuit. The middle pin of the potentiometer goes to Analog Pin 5*

Upload the Code

Prior to uploading the code you will need to download, unpack and install the LED library. Go to http://arduino.cc/playground/uploads/Main/LedControl.zip and download the file, unzip it and then place the LedControl folder into your libraries folder as before (See http://playground.arduino.cc//Main/LedControl for further info.). Next, upload the code from Listing 7-4. When the program is run, a ball will start from a random location on the left and head towards the right. Using the potentiometer, control the paddle to bounce the ball back towards the wall. As time goes by, the speed of the ball will increase faster and faster until you will not be able to keep up with it.

Listing 7-4. Code for Project 22//Project 22

```
#include "LedControl.h"
LedControl myMatrix = LedControl(2, 4, 3, 1); // create an instance of a Matrix

int column = 0, row = random(8);
int directionX = 1, directionY = 1;
int paddle1 = 5, paddle1Val;
int movement_interval = 300;
int counter = 0;

void setup()
{
  myMatrix.shutdown(0,false);
  /* Set the brightness to a medium values */
  myMatrix.setIntensity(0,8);
  randomSeed(analogRead(0));
  oops();
}

void loop()
{
   paddle1Val = analogRead(paddle1);
   paddle1Val = map(paddle1Val, 0, 1024, 0,6);
   if (column == 6 && (paddle1Val == row ||
      paddle1Val+1 == row || paddle1Val+2 == row)) {directionX = -1;}
  if (column == 0) {directionX = 1;}
  if (row == 7) {directionY = -1;}
  if (row == 0) {directionY = 1;}
  if (column == 7) { oops();}
  column += directionX;
  row += directionY;
 displayDashAndDot();
 counter++;
}

void oops() {
  for (int x=0; x<3; x++) {
    myMatrix.clearDisplay(0);
    delay(250);
    for (int y=0; y<8; y++) {
      myMatrix.setRow(0, y, 255);
    }
    delay(150);
  }
  counter=0;
  movement_interval=300;
  column=0;
  row = random(8);
  displayDashAndDot();
}

void displayDashAndDot() {
  myMatrix.clearDisplay(0);
  myMatrix.setLed(0, column, row, HIGH);
  myMatrix.setLed(0, 7, paddle1Val, HIGH);
  myMatrix.setLed(0, 7, paddle1Val+1, HIGH);
  myMatrix.setLed(0, 7, paddle1Val+2, HIGH);
  if (!(counter % 10)) {movement_interval -= 5;}
  delay(movement_interval);
}
```

When the ball bounces past the paddle, the screen will flash and the game will restart. See how long you can go for before the game resets.

■ **Note** Remember, the project will not work without loading the libraries as described above first.

Project 22 – LED Dot Matrix – Pong Game

The code for Project 22 is really simple. After all, you are taking a break from the hard work done in Project 21.
First we will include an external library. This is the LedControl library that you placed into your libraries folder as before. The #include command simply tells the IDE to include the code inside the LedControl library into your code.

```
#include "LedControl.h"
```

You then create an instance of an `LedControl` object like so

```
LedControl myMatrix = LedControl(2, 4, 3, 1); // create an instance of a Matrix
```

This creates an `LedControl` object called myMatrix. The `LedControl` object requires four parameters. The first three are the pin numbers for the MAX7219; the order is Data In, Clock, and Load. The final number is for the number of the chip (in case you are controlling more than one MAX7219 and display).

Then you decide which column and row the ball will start in. The row is decided using a random number.

```
int column = 0, row = random(8)+              // decide where the ball will start
```

Now two integers are declared to decide the direction the ball will travel in. If the number is positive, it will head from left to right and bottom to top, respectively, and if negative, it will be in reverse.

```
int directionX = 1, directionY = 1;           // make sure it heads from left to right first
```

You decide which pin is being used for the paddle (the potentiometer) and declare an integer to hold the value read from the analog pin:

```
int paddle1 = 5, paddle1Val;                  // Pot pin and value
```

The interval between ball movements is declared in milliseconds:

```
int movement_interval = 300;
```

Then you declare and initialize a counter to zero:

```
int counter = 0;
```

The `setup()` function enables the display by ensuring that the power-saving mode is set to false. The intensity is set to medium and then the game is initialized using the oops() function, which we will look at later. Before you start, the `randomSeed` is set with a random value read from an unused analog pin.

```
void setup()
{
        myMatrix.shutdown(0, false); // enable display
        myMatrix.setIntensity(0, 8); //  Set the brightness to medium
        randomSeed(analogRead(0));
       oops();
}
```

In the main loop, you start by reading the analog value from the paddle:

```
paddle1Val = analogRead(paddle1);
```

Then those values are mapped to between 1 and 6:

```
paddle1Val = map(paddle1Val, 0, 1024, 0, 6);
```

The map command requires five parameters. The first is the number to map. Next are the low and high values of the number and the low and high values you wish to map it to. In your case, you are taking the value in `paddle1Val`, which is the voltage read from Analog Pin 5. This value ranges from 0 at 0 volts to 1023 at 5 volts. You want those numbers mapped to read only between 0 and 5 as this is the row the paddle will be displayed on when drawn on the display.

You now need to decide if the ball has hit a wall or a paddle, and if so, bounce back (the exception being if it goes past the paddle0. The first `if` statement checks that the ball has hit the paddle. It does this by deciding if the column the ball is in is column 6 and has hit the paddle:

```
if (column == 6 && (paddle1Val == row || paddle1Val+1 == row || paddle1Val+2 == row))
{directionX = -1;}
```

There are two conditions that have to be met for the ball direction to change. The first is that the column is 6, the second is that the ball is on the same row that either of the 3 dots that make up the paddle is in. This is done by nesting a set of `or` (logical OR ||) commands inside brackets. The result of this calculation is checked first and then the result added to the three && statements in the first set of brackets.

The next three sets of `if` statements check if the ball has hit the top, bottom, or left side walls, and if so, reverses the ball direction:

```
if (column == 0) {directionX = 1;}
if (row == 7) {directionY = -1;}
if (row ==) {directionY = 1;}
```

Finally, if the ball is in column 7, it has obviously not hit the paddle but has gone past it. If that is the case, call the `oops()` function to flash the display and reset the values:

```
if (column == 7) { oops();}
```

The column and row coordinates are increased by the values in directionX and directionY:

```
column += directionX;
row += directionY;
```

Next, the function that displays the bat and paddle is called.

```
displayDashAndDot();
```

Let us take a look at this function. The ball is drawn at the column and row location. This is done with the `setLed` command of the LedControl library:

```
myMatrix.setLed(0, column, row, HIGH);
```

The `setLed` command requires four parameters. The first number is the address of the display, then the x and y (or column and row) coordinates and, finally, a HIGH or LOW for on and off. These are used again to draw the three dots that make up the paddle at column 7 and row `paddle1Val` (plus one above and one above that).

```
myMatrix.setLed(0, 7, paddle1Val, HIGH);
myMatrix.setLed(0, 7, paddle1Val+1, HIGH);
myMatrix.setLed(0, 7, paddle1Val+2, HIGH);
```

You then check if the modulo of `counter % 10` is NOT (logical NOT !) true, and if so, decrease the movement_interval by five. Modulo is the remainder when you divide one integer by another. In your case, you divide counter by 10 and check if the remainder is a whole number or not. This basically ensures that the movement_interval is only increased after a set time.

```
if (!(counter % 10)) {movement_interval -= 5;}
```

A delay in milliseconds of the value of movement_interval is activated and then the counter value is increased by one:

```
delay(movement_interval);
counter++;
}
```

Finally, the `oops()` function causes a `for` loop within a `for` loop to clear the display and then fills in all rows repeatedly with a 250 millisecond delay in between. This makes all LEDs flash on and off to indicate the ball has gone out of play and the game is about to reset. Then all of the values of counter, movement_interval, and column are set to their starting positions and a new random value chosen for row.

```
void oops() {
  for (int x=0; x<3; x++) {
  myMatrix.clearDisplay(0);
  delay(250);
              for (int y=0; y<8; y++) {
                      myMatrix.setRow(0, y, 255);
              }
              delay(250);
        }
        counter=0;       // reset all the values
        movement_interval=300;
        column=0;
        row = random(8) // choose a new starting location
}
```

The `setRow` command works by passing the address of the display, the row value, and then the binary pattern of which LEDs to turn on or off. In this case, you want them all on, which is binary 11111111 and decimal 255.

The purpose of Project 22 was to show how much easier it is to control an LED Driver chip if you use a ready-made library of code designed for the chip. In Project 21 you did it the hard way, coding everything from scratch; in Project 22, the hard work was all done for you behind the scenes. There are other Matrix libraries available and the Playground section of the Arduino website will give you further information, or you can search for information on controlling LED Matrices on the Arduino Forum where you will find plenty of useful information. You can use whichever library suits your needs best.

In the next chapter, you will look at a different type of dot matrix display, the LCD.

EXERCISE

Take the concepts from Projects 21 and 22 and combine them. Make a Pong Game but have the code keep a tab of the score (in milliseconds since the game started). When the ball goes out of play, use the scrolling text function to show the score of the game just played (milliseconds the player survived) and the highest score to date.

Summary

Chapter 7 introduced you to some pretty complex subjects, including the use of external ICs. You are not even halfway through the projects yet and you already know how to control a dot matrix display using both shift registers and a dedicated LED driver IC. Also, you learned how to code things the hard way and then how to code the easy way by incorporating a ready-made library designed for your LED Driver IC.

You have also learned the sometimes baffling concept of multiplexing, a skill that will come in very handy for many other things as well as dot matrix displays.

Subjects and concepts covered in Chapter 7:

- How a dot matrix display is wired up
- How to install an external library
- The concept of multiplexing (or muxing)
- How to use multiplexing to turn on 64 LEDs individually using just 16 output pins
- The basic concept of timers
- How to use the TimerOne library to activate code no matter what else is going on
- How to include external libraries in your code using the #include directive
- How to use binary numbers to store LED images
- How to invert a binary number using a bitwise NOT ~
- How to take advantage of persistence of vision to trick the eye
- How to store animation frames in multidimensional arrays
- How to declare and initialize a multidimensional array
- Accessing data in a specific element in a two-dimensional array
- How to do a bitwise rotation (aka circular shift)
- How to control LED dot matrix displays using shift registers
- How to control LED dot matrix displays using MAX7219 ICs
- How to time pulses correctly to load data in and out of external ICs
- How to store a character font in a two-dimensional array
- How to read timing diagrams from datasheets
- How to use the registers in the MAX7219
- How to bypass the SRAM limits and store data in program space
- How to reverse the order of bits
- How to make text and other symbols scroll across a dot matrix display
- The concept of the ASCII character table
- Choosing a character from its ASCII code
- How to find out the length of a text string
- How to write to and read from program space
- How to obtain the address in memory of a variable using the & symbol
- How to use the LedControl.h library to control individual LEDs and rows of LEDs
- How to use the logical operators
- How to make life easier and code development faster using code libraries

CHAPTER 8

Liquid Crystal Displays

Now we are going to move onto another popular method of displaying text and symbols and that is the LCD (liquid crystal display). These are the type of displays typically used in calculators or alarm clocks. Many Arduino projects use these, so it is essential that you know how to use them, too. LCD displays require driver chips to control them and these are built into the display. The most popular type of driver chip is the Hitachi HD44780 (or compatible); these are used to drive most of the common LCDS.

Creating projects based around LCD displays is easy, thanks to a whole array of LCD code libraries that are available. The Arduino IDE comes with a nice library called `LiquidCrystal.h`, which has a great list of features. We will therefore be using this one in our projects.

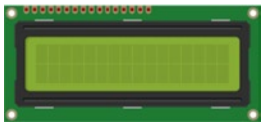
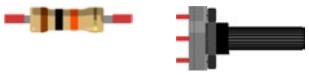
Project 23 – Basic LCD Control

To start with, we will create a demonstration project that will show off most of the functions available in the `LiquidCrystal.h` library. We will be using a backlit 16x2 LCD Display.

Parts Required

You will need to obtain an LCD display that uses the HD44780 driver. There are many available, and they come in all kinds of colors. As an amateur astronomer, I particularly like the red-on-black displays as they preserve your night vision. These have red text on a black background. You can, of course, choose any available color text and background you wish. Your display must have a backlight and be able to display sixteen columns and two rows of characters. These are often referred to as 16x2 LCD displays.

Table 8-1. *Parts Required for Project 23*

16x2 Backlit LCD	
Current Limiting Resistor (Backlight)	
Contrast Setting Resistor or pot	

Connect It Up

The circuit for Project 23 is pretty simple. What you will need is the datasheet for the LCD you are using. The following pins (see Table 8-2) from the Arduino, +5V and Gnd, need to go to the LCD.

Table 8-2. *Pins to Use for the LCD*

Arduino	Other	Matrix
Digital 11		Enable
Digital 12		RS (Register Select)
Digital 5		DB4 (Data Pin 4)
Digital 4		DB5 (Data Pin 5)
Digital 3		DB6 (Data Pin 6)
Digital 3		DB7 (Data Pin 7)
	Gnd	Vss (GND)
	Gnd	R/~WRead/Write)
	+5v	Vdd
	Gnd via resistor	Vo (Contrast)
	+5V via resistor	A/Vee (Power for LED)
	Gnd	Gnd for LED

Data pins 0 to 3 are not used, as we are going to use what is known as 4-bit mode. For a typical LCD display the circuit in Figure 8-1 will be correct.

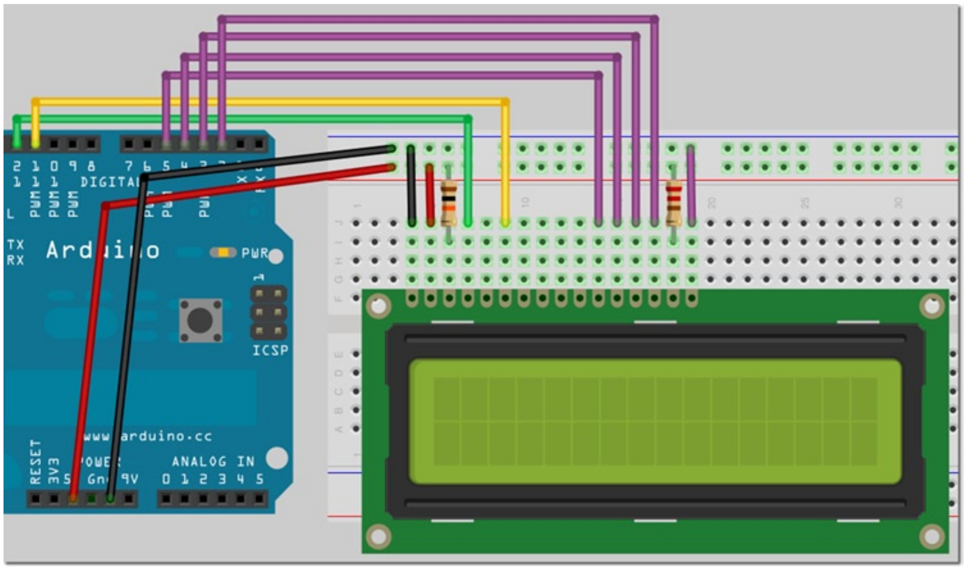


Figure 8-1. *The circuit for Project 23 – Basic LCD Control*

The contract adjustment pin on the LCD must be connected via a current-limiting resistor to adjust the contrast the desired level. A value of around 10KΩ should suffice. If you find it difficult to get the right value, then connect a potentiometer with value between about 4KΩ to 10KΩ with the left leg to +5V, the right leg to ground, and the center leg to the contrast adjustment pin (pin 3 on my LCD). This way, you can use the knob to adjust the contrast until you can see the display clearly.

The backlight on the LCD I used required 4.2V, so I added the appropriate current-limiting resistor between +5V and the LED power supply pin (pin 15 in my LCD). You could, of course, connect the LED power pin to a PWM pin (via a current-limiting resistor) on the Arduino and use a PWM output to control the brightness of the backlight. For simplicity's sake we will not use that method in this project. Once you are happy that your circuit matches mine with the correct pins going between the Arduino, +5V, and ground (according to the LCDs datasheet), then you can enter the code

Enter the Code

Check your wiring, then upload the code from listing 8-1.

Listing 8-1. Code for Project 23

```
// PROJECT 23
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // create an lcd object and assign the pins

void setup() {
       lcd.begin(16, 2);               // Set the display to 16 columns and 2 rows
}

void loop() {
       // run the 7 demo routines
       basicPrintDemo();
       displayOnOffDemo();
       setCursorDemo();
       scrollLeftDemo();
       scrollRightDemo();
       cursorDemo();
       createGlyphDemo();
}

void basicPrintDemo() {
       lcd.clear();                    // Clear the display
       lcd.print("Basic Print");       // print some text
       delay(2000);
}

void displayOnOffDemo() {
       lcd.clear();                    // Clear the display
       lcd.print("Display On/Off");    // print some text
       for(int x=0; x < 3; x++) {      // loop 3 times
               lcd.noDisplay();        // turn display off
               delay(1000);
```

```
                lcd.display();            // turn it back on again
                delay(1000);
        }
}

void setCursorDemo() {
        lcd.clear();                      // Clear the display
        lcd.print("SetCursor Demo");      // print some text
        delay(1000);
        lcd.clear();                      // Clear the display
        lcd.setCursor(5,0);               // cursor at column 5 row 0
        lcd.print("5,0");
        delay(2000);
        lcd.setCursor(10,1);              // cursor at column 10 row 1
        lcd.print("10,1");
        delay(2000);
        lcd.setCursor(3,1);               // cursor at column 3 row 1
        lcd.print("3,1");
        delay(2000);
}

void scrollLeftDemo() {
        lcd.clear();                      // Clear the display
        lcd.print("Scroll Left Demo");
        delay(1000);
        lcd.clear();                      // Clear the display
        lcd.setCursor(7,0);
        lcd.print("Beginning");
        lcd.setCursor(9,1);
        lcd.print("Arduino");
        delay(1000);
        for(int x=0; x<16; x++) {
                lcd.scrollDisplayLeft();  // scroll display left 16 times
                delay(250);
        }
}

void scrollRightDemo() {
        lcd.clear();                      // Clear the display
        lcd.print("Scroll Right");
        lcd.setCursor(0,1);
        lcd.print("Demo");
        delay(1000);
        lcd.clear();                      // Clear the display
        lcd.print("Beginning");
        lcd.setCursor(0,1);
        lcd.print("Arduino");
        delay(1000);
        for(int x=0; x<16; x++) {
                lcd.scrollDisplayRight(); // scroll display right 16 times
                delay(250);
        }
}
```

```
void cursorDemo() {
       lcd.clear();                // Clear the display
       lcd.cursor();               // Enable cursor visible
       lcd.print("Cursor On");
       delay(3000);
       lcd.clear();                // Clear the display
       lcd.noCursor();             // cursor invisible
       lcd.print("Cursor Off");
       delay(3000);
       lcd.clear();                // Clear the display
       lcd.cursor();               // cursor visible
       lcd.blink();                // cursor blinking
       lcd.print("Cursor Blink On");
       delay(3000);
       lcd.noCursor();             // cursor invisible
       lcd.noBlink();              // blink off
}

void createGlyphDemo() {
       lcd.clear();

       byte happy[8] = {           // create byte array with happy face
       B00000,
       B00000,
       B10001,
       B00000,
       B10001,
       B01110,
       B00000,
       B00000};

       byte sad[8] = {             // create byte array with sad face
       B00000,
       B00000,
       B10001,
       B00000,
       B01110,
       B10001,
       B00000,
       B00000};

       lcd.createChar(0, happy);   // create custom character 0
       lcd.createChar(1, sad);     // create custom character 1

       for(int x=0; x<5; x++) {    // loop animation 5 times
               lcd.setCursor(8,0);
               lcd.write((byte)0); // write custom char 0
               delay(1000);
               lcd.setCursor(8,0);
               lcd.write(1);       // write custom char 1
               delay(1000);
       }
}
```

Project 23 – Basic LCD Control – Code Overview

First we load in the library that we are going to use to control the LCD. The Arduino IDE comes with a library called `LiquidCrystal.h` which is nice and easy to understand and use. There are many other libraries and code examples available for different types of LCDs and you can find them all on the Arduino playground at `http://www.arduino.cc/playground/Code/LCD`

```
#include <LiquidCrystal.h>
```

Now we need to create the `LiquidCrystal` object and set the appropriate pins.

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // create an lcd object and assign the pins
```

So we have created a `LiquidCrystal` object and called it `lcd`. That is really a variable that works like a handle on the object so we can refer to it by name. The first two parameters set the pins for RS (Register Select) and Enable. The last four parameters are data pins D4 to D7. Since we are using 4-bit mode, we are only using four of the eight data pins on the display.

The difference between 4-bit and 8-bit modes is that in 8-bit mode we can send data one byte at a time, whereas in 4-bit mode the 8 bits have to be split up into to 4-bit numbers (known as nibbles). This makes the code larger and more complex. However, as we are using a ready-made library, we are not going to worry about that. However, if you were writing space or time critical code you would consider writing directly to the LCD in 8-bit mode. Using 4-bit mode has the advantage of saving 4 pins, which is useful if we want to connect other devices at the same time.

In the `setup()` function we initialize the display to the size required, in our case, 16 columns and 2 rows.

```
lcd.begin(16, 2); // Set the display to 16 columns and 2 rows
```

The main program loop cycles through seven different demo routines, one by one, before restarting. Each demo routine shows one set of related routines in the LiquidCrystal.h library.

```
void loop() {
       // run the 7 demo routines
       basicPrintDemo();
       displayOnOffDemo();
       setCursorDemo();
       scrollLeftDemo();
       scrollRightDemo();
       cursorDemo();
       createGlyphDemo();
}
```

The first demo is `basicPrintDemo()` and is designed to show use of the `print()` method. This demo simply clears the display using `lcd.clear()` then prints to the display using `lcd.print()`. Note that if you had initialized your LiquidCrystal object and called it, for example, LCD1602, then these methods would be `LCD1602.clear()` and `LCD1602.print()` accordingly. In other words, the method comes after the name of the object, with a dot between them.

The `print()` method will print whatever is inside the brackets at the current cursor location. The default starting cursor location is always column 0 and row 0, which is the top left corner. When you use `print()`, the text you provide will be added to the display and the cursor is moved to the end of the new text. Any additional text will be added there. After clearing the display the cursor will be set to the default, or home, position.

```
void basicPrintDemo() {
       lcd.clear();                 // Clear the display
       lcd.print("Basic Print");    // print some text
       delay(2000);
}
```

The second demo is designed to show off the `display()` and `noDisplay()` methods. These methods simply enable or disable the display. The code prints out "Display On/Off" and then runs a `for()` loop three times to turn the display off; wait one second, turn it back on, wait another second, then repeat. Whenever you turn the display off, whatever was printed on the screen before it went off will be preserved when the display is re-enabled.

```
void displayOnOffDemo() {
       lcd.clear();                 // Clear the display
       lcd.print("Display On/Off"); // print some text
       for(int x=0; x < 3; x++) {   // loop 3 times
               lcd.noDisplay();     // turn display off
               delay(1000);
               lcd.display();       // turn it back on again
               delay(1000);
       }
}
```

The next demo shows of the `setCursor()` method. All this method does is set the cursor to the column and row location set within the brackets. The demonstration sets the cursor to three locations and prints that location in text on the display. The `setCursor()` method is useful for controlling the layout of your text and ensuring your output goes to the appropriate part of the display screen.

```
void setCursorDemo() {
       lcd.clear();                 // Clear the display
       lcd.print("SetCursor Demo"); // print some text
       delay(1000);
       lcd.clear();                 // Clear the display
       lcd.setCursor(5,0);          // cursor at column 5 row 0
       lcd.print("5,0");
       delay(2000);
       lcd.setCursor(10,1);         // cursor at column 10 row 1
       lcd.print("10,1");
       delay(2000);
       lcd.setCursor(3,1);          // cursor at column 3 row 1
       lcd.print("3,1");
       delay(2000);
}
```

There are two methods in the library for scrolling text. One is the `scrollDisplayLeft()` and the other is `scrollDisplayRight()`. It is pretty obvious from the names which is which. Two demo routines now run to show off these methods. The first prints "Beginning Arduino" on the right side of the display and scrolls it left 16 times, which will make it scroll off the screen.

```
void scrollLeftDemo() {
       lcd.clear();                      // Clear the display
       lcd.print("Scroll Left Demo");
       delay(1000);
       lcd.clear();                      // Clear the display
       lcd.setCursor(7,0);
       lcd.print("Beginning");
       lcd.setCursor(9,1);
       lcd.print("Arduino");
       delay(1000);
       for(int x=0; x<16; x++) {
               lcd.scrollDisplayLeft();  // scroll display left 16 times
               delay(250);
       }
}
```

The next demo does the same, starting with the text on the left and scrolling it right 16 times till it scrolls off the screen.

```
void scrollRightDemo() {
       lcd.clear();                      // Clear the display
       lcd.print("Scroll Right");
       lcd.setCursor(0,1);
       lcd.print("Demo");
       delay(1000);
       lcd.clear();                      // Clear the display
       lcd.print("Beginning");
       lcd.setCursor(0,1);
       lcd.print("Arduino");
       delay(1000);
       for(int x=0; x<16; x++) {
               lcd.scrollDisplayRight(); // scroll display right 16 times
               delay(250);
       }
}
```

The cursor so far has been invisible. It is always there, but not yet seen. Whenever you clear the display, the cursor returns to the top left corner or column 0 and row 0. After printing some text, the cursor will sit just after the last character printed. The next demo clears the display, then turns the cursor on with `cursor()` and prints some text. The cursor will be visible, just after this text, as an underscore symbol (_).

```
void cursorDemo() {
       lcd.clear();                      // Clear the display
       lcd.cursor();                     // Enable cursor visible
       lcd.print("Cursor On");
       delay(3000);
```

The display is cleared again and this time the cursor is turned off, which is the default mode, using `noCursor()`. Now the cursor is hidden again.

```
lcd.clear();                        // Clear the display
lcd.noCursor();                     // cursor invisible
lcd.print("Cursor Off");
delay(3000);
```

Next, the cursor is enabled and blink mode is also enabled using `blink()`.

```
lcd.clear();                        // Clear the display
lcd.cursor();                       // cursor visible
lcd.blink();                        // cursor blinking
lcd.print("Cursor Blink On");
delay(3000);
```

This time the cursor will not only be visible, but will be blinking on and off. This mode is useful if you are waiting for some text input from a user. The blinking cursor will act as a prompt to enter some text.

Finally, the cursor is turned off and blink turned off also to put the cursor back into the default mode.

```
       lcd.noCursor();              // cursor invisible
       lcd.noBlink();               // blink off
}
```

The final demo called `createGlyphDemo()` creates a custom character. Most LCDs have the ability to program custom characters. The standard 16x2 LCD has space for 8 custom characters to be stored in memory. The characters are 5 pixels wide by 8 pixels high (a pixel is a picture element, i.e. the individual dots that make up a digital display). The display is cleared and then two arrays of type byte are initialized with the binary pattern of a happy and a sad face. The binary patterns are 5 bits wide.

```
void createGlyphDemo() {
       lcd.clear();

       byte happy[8] = {            // create byte array with happy face
       B00000,
       B00000,
       B10001,
       B00000,
       B10001,
       B01110,
       B00000,
       B00000};

       byte sad[8] = {              // create byte array with sad face
       B00000,
       B00000,
       B10001,
       B00000,
       B01110,
       B10001,
       B00000,
       B00000};
```

Then we create the two custom characters using the `createChar()` method. This requires two parameters: the first is the number of the custom character (0 to 7 in the case of the LCD I used that can store a maximum of 8) and the second is the name of the array in which the custom character's binary pattern is stored. This creates and stores that pattern in memory on the LCD.

```
lcd.createChar(0, happy);          // create custom character 0
lcd.createChar(1, sad);            // create custom character 1
```

A for loop will now loop through itself five times. On each iteration, the cursor is set to column 8 and row 0 and the first custom character is written to that location using the `write()` method. This writes the custom character, within the brackets (cast as byte) to the cursor location. The first character, which is a happy face, is written to the cursor location and after a delay of one second, the second character, which is a sad face, is then written to the same cursor location. This repeats five times to make a crude animation.

```
       for(int x=0; x<5; x++) {    // loop animation 5 times
               lcd.setCursor(8,0);
               lcd.write((byte)0); // write custom char 0
               delay(1000);
               lcd.setCursor(8,0);
               lcd.write(1);       // write custom char 1
               delay(1000);
       }
}
```

Project 23 covered most of the popular methods within the `LiquidCrystal.h` library. There are others to discover and you can read about them in the Arduino Reference library at http://www.arduino.cc/en/Reference/LiquidCrystal.

Project 23 – Basic LCD Control – Hardware Overview

The new component in this project was obviously the LCD. A liquid crystal display works by using the light modulating properties of liquid crystals. The display is made up of pixels, each one filled with liquid crystals. These pixels are arrayed either in front of a backlighting source or a reflector. The crystals are placed into layers sandwiched between polarizing filters. The two polarizing panels are aligned at 90 degrees to each other which blocks light. The first polarizing filter will polarize the light waves so that they all run in one orientation only. The second filter, being at 90 degrees to the first, will block the light. Imagine the filter is made up of very thin slits going in one direction or 90 degrees to the first. Light polarized in one direction will go through slits in the same orientation, but when reaching the second filter, which has its slits running the other way, will not pass through.

By running a voltage across the rows and columns of the layers, the crystals can be made to change orientation and line up with the electric field. This causes the light to twist through 90 degrees and allows it through the second filter. The grid of pixels the LCD is made up of are arranged into smaller grids that make up the characters. A typical 16 x 2 LCD will have 16 character grids in two rows. As we've noted, each character grid is 5 pixels wide by 8 pixels high. If you turn the contrast up very high on your display the 32 arrays of 5x8 pixels will become visible.

That is really all you need to know about how LCDs work. Let us now put our LCD to use and make a temperature display.

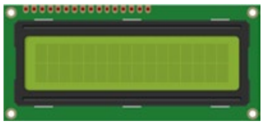
Project 24 – LCD Temperature Display

This project will be a simple demonstration of using an LCD to present useful information to the user, in this case, the temperature readout from an analog temperature sensor. We will also add a button to enable the temperature to be displayed in either centigrade or Fahrenheit, whichever you prefer. Also, the maximum and minimum temperature will be displayed on the second row.

Parts Required

The parts required are the same as for Project 23 with the addition of a button and an analog temperature sensor. Make sure that the temperature sensor only outputs positive voltages.

Table 8-3. *Parts Required for Project 24*

16x2 Backlit LCD	
Current-Limiting Resistor (Backlight)	
Contrast-Setting Resistor	
Pull-Down Resistor	
Pushbutton	
Analog Temp Sensor	

Connect It Up

Leave the exact same circuit that you set up for Project 23. Then add a pushbutton and temperature sensor as in Figure 8-2.

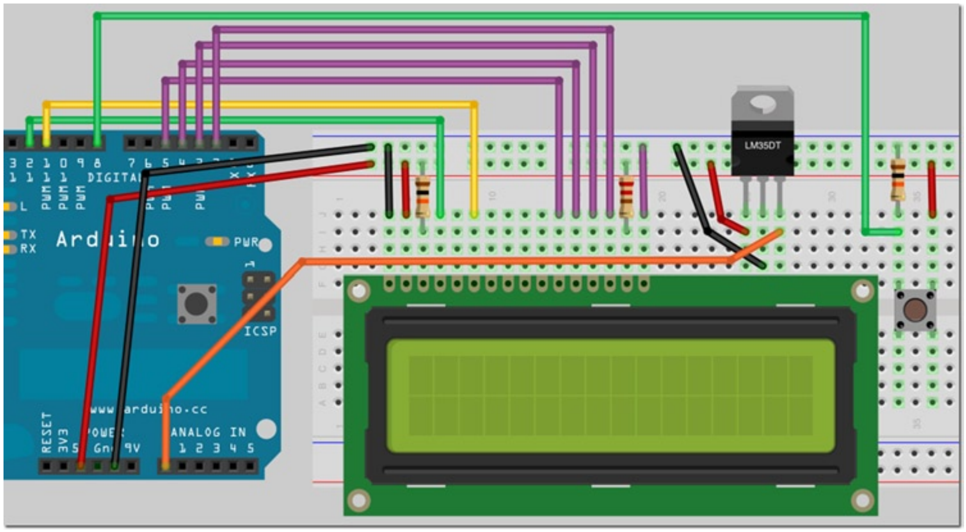


Figure 8-2. *The circuit for Project 24 – LCD Temperature Display*

I have used an LM35DT temperature sensor, which has a range from 0ºC to 100ºC. You can, of course, use any analog temperature sensor you wish. The LM35 ranges from –55ºC to +150ºC. You will need to adjust your code accordingly (more on this later).

Enter the Code

Check your wiring, then upload the code from listing 8-2.

Listing 8-2. Code for Project 24

```
// PROJECT 24
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);          // create an lcd object and assign the pins
int maxC=0, minC=100, maxF=0, minF=212;
int scale = 1;
int buttonPin=8;

void setup() {
       lcd.begin(16, 2);                        // Set the display to 16 columns and 2 rows
       analogReference(INTERNAL);
       // analogReference(INTERNAL1V1); If you have an Arduino Mega
       pinMode(buttonPin, INPUT);
lcd.clear();
}

void loop() {
       lcd.setCursor(0,0);                      // set cursor to home position
       int sensor = analogRead(0);              // read the temp from sensor
       int buttonState = digitalRead(buttonPin); // check for button press
       switch (buttonState) {                   // change scale state if pressed
               case HIGH:
                      scale=-scale;             // invert scale
                      lcd.clear();
       }

       switch (scale) {                         // decide if C or F scale
               case 1:
                      celsius(sensor);
                      break;
               case -1:
                      fahrenheit(sensor);
       }
       delay(250);
}

void celsius(int sensor) {
       lcd.setCursor(0,0);
       int temp = sensor * 0.1074188;           // convert to C
       lcd.print(temp);
       lcd.write(B11011111);                    // degree symbol
```

```
        lcd.print("C ");
        if (temp>maxC) {maxC=temp;}
        if (temp<minC) {minC=temp;}
        lcd.setCursor(0,1);
        lcd.print("H=");
        lcd.print(maxC);
        lcd.write(B11011111);
        lcd.print("C L=");
        lcd.print(minC);
        lcd.write(B11011111);
        lcd.print("C ");
}

void fahrenheit(int sensor) {
        lcd.setCursor(0,0);
        float temp = ((sensor * 0.1074188) * 1.8)+32; // convert to F
        lcd.print(int(temp));
        lcd.write(B11011111);                         // print degree symbol
        lcd.print("F ");
        if (temp>maxF) {maxF=temp;}
        if (temp<minF) {minF=temp;}
        lcd.setCursor(0,1);
        lcd.print("H=");
        lcd.print(maxF);
        lcd.write(B11011111);
        lcd.print("F L=");
        lcd.print(minF);
        lcd.write(B11011111);
        lcd.print("F ");
}
```

When you run the code,the current temperature will be displayed on the LCD on the top row. The bottom row will display the maximum and minimum temperatures recorded since the Arduino was turned on or the program reset. By pressing the button you can change the temperature scale between Celsius and Fahrenheit.

Project 24 – LCD Temperature Display

As before, the `LiquidCrystal` library is loaded into our sketch.

```
#include <LiquidCrystal.h>
```

A `LiquidCrystal` object called `lcd` is initialized and the appropriate pins set.

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // create an lcd object and assign the pins
```

Some integer variables to hold the maximum and minimum temperatures in degrees C and F are declared and initialized with impossible max and min values. These will be updated as soon as the program runs for the first time.

```
int maxC=0, minC=100, maxF=0, minF=212;
```

A variable called `scale` of type `int` is declared and initialized with `1`. The scale variable will decide if we are using Celsius or Fahrenheit as our temperature scale. By default it is set to `1`, which is Celsius. If you wish you can change this to `-1` for Fahrenheit.

```
int scale = 1;
```

An integer to store the pin being used for the button is declared and initialized.

```
int buttonPin=8;
```

In the `setup()` function we set the display to be 16 columns and 2 rows.

```
lcd.begin(16, 2); // Set the display to 16 columns and 2 rows
```

The reference for the analog pin is then set to `INTERNAL`.

```
analogReference(INTERNAL);
```

If you have an Arduino Mega this needs to be

```
analogReference(INTERNAL1V1);
```

This gives us a better range on the Arduino's ADC (analog to digital convertor). The output voltage of the LM35DT at 100ºC is 1V. If we were using the default reference of 5 volts, then at 50ºC, which is half of the sensor's range, the reading on the ADC would be 0.5V = (0.5/5)*1023 = 102, which is only about 10% of the ADC's range. When using the internal reference voltage of 1.1 volts, the value at the analog pin at 50ºC is now 0.5V = (0.5/1.1)*1023 = 465.

As you can see, this is almost half way through the entire range of values that the analog pin can read (0 to 1,023). It has therefore increased the resolution and accuracy of the reading and increased the sensitivity of the circuit. However, accuracy depends on the accuracy of the internal reference versus the accuracy of the 5 volt supply voltage. According to the data sheet for the Atmega328, the internal reference may be anywhere from 1.0 to 1.2 volts, which is up to 11% error. The 5V regulator on the Arduino Uno is 1% tolerance.

The button pin is now set to an input and the LCD display cleared.

```
pinMode(buttonPin, INPUT);
lcd.clear();
```

In the main loop, the program starts off by setting the cursor to its home position.

```
lcd.setCursor(0,0);                    // set cursor to home position
```

Then we read a value from the temperature sensor on analog pin 0.

```
int sensor = analogRead(0);            // read the temp from sensor
```

Then we read the state of the button and store the value in buttonState.

```
int buttonState = digitalRead(buttonPin); // check for button press
```

Now we need to decide if the button has been pressed or not and if so, to change the scale from Celsius to Fahrenheit, or vice versa.

This is done using a switch/case statement.

```
switch (buttonState) {      // change scale state if pressed
       case HIGH:
              scale=-scale; // invert scale
              lcd.clear();
}
```

This is a new concept that we have not come across before. The switch/case command controls the flow of the program by specifying which code should be executed if different conditions have been met. The `switch/case` statement compares the value of a variable with values in the case statements and if `true`, runs the code after that case statement.

For example, if we had a variable called var and we wanted things to happen if its value was either `1`, `2` or `3`, then we could decide what to do for either of those values by using

```
switch (var) {
       case 1:
               // run this code here if var is 1
               break;
       case 2:
               // run this code here if var is 2
               break;
       case 3:
               // run this code here if var is 3
               break;
       default:
               // if nothing else matches run this code here
       }
```

So, the `switch/case` statement will check the value of `var`. If it is `1`, then it will run the code within the `case 1` block up to the `break` command. The `break` command is used to exit out of the `switch/case` statement. Without it, the code would fall through to the next case block and carry on executing until a `break` command is reached or the end of the `switch/case` statement is reached. If none of the values checked for are reached, then the code within the default section will be executed. Note that the default section is optional and not necessary.

In our case we only check one case and that is if the buttonState is `HIGH`. If so, the value of scale is exchanged (from C to F or vice-versa) and the display is cleared.

Next is a short delay.

```
delay(250);
```

Then another switch/case statement to check if the value of scale is either a 1 for Celsius or a -1 for Fahrenheit and if so to run the appropriate functions.

```
       switch (scale) { // decide if C or F scale
               case 1:
                      celsius(sensor);
                      break;
               case -1:
                      fahrenheit(sensor);
       }
}
```

Next we have the two functions to display the temperatures on the LCD. One is if we are working in Celsius and the other is for Fahrenheit. The functions have a single parameter. We pass it an integer value, which will be the value read from the temperature sensor.

```
void celsius(int sensor) {
```

The cursor is set to the home position.

```
lcd.setCursor(0,0);
```

Then we take the sensor reading and convert it to degrees Celsius by multiplying by 0.1074188.

```
int temp = sensor * 0.1074188;                 // convert to C
```

This factor is reached by taking 1.1, which is the reference voltage and dividing it by the range of the ADC, which is 1,024 then dividing by the sensor sensitivity which is 0.01V/°C.

```
1.1/1024=0.0010742188
0.0010742188/0.01=0.10742188
```

We then print that converted value to the LCD, along with the character whose code value is `B11011111`, which is a degree symbol, followed by a C to indicate we are displaying the temperature in Celsius.

```
lcd.print(temp);
lcd.write(B11011111);                          // degree symbol
lcd.print("C  ");
```

To keep a running score of the maximum and minimum temperatures recorded since the Arduino was turned on, we use the Arduino's max() and min() functions. They both take two variables as their parameters and return either the maximum of the two numbers or the minimum. We store that result.

```
maxC = max(maxC,temp);
minC = min(minC, temp);
```

Then on the second row we print H (for HIGH) and the value of `maxC,` and then an L (for LOW) followed by the degree symbol and a letter C.

```
lcd.setCursor(0,1);
lcd.print("H=");
lcd.print(maxC);
lcd.write(B11011111);
lcd.print("C L=");
lcd.print(minC);
lcd.write(B11011111);
lcd.print("C  ");
```

The Fahrenheit function does exactly the same thing, except it converts the temperature in Celsius to Fahrenheit by multiplying it by 1.8 and adding 32.

```
float temp = ((sensor * 0.10742188) * 1.8)+32; // convert to F
```

Now that you know how to use an LCD to display useful information, you create your own projects to display anything you like on an LCD, to display sensor data or to create a simple user interface.

Summary

In Chapter 8 you have learned about the most commonly used functions from the LiquidCrystal.h library. This includes clearing the display, printing text to specific locations on the screen, making the cursor visible or invisible or even blink, and even how to make the text scroll to the left or the right. Project 24 gave you a simple application of these functions in a real world example of a temperature sensor. This gave you an oversight of how an LCD could be used in a real project to display data.

Subjects and concepts covered in Chapter 8:

- How to load the `LiquidCrystal.h` library
- How to wire an LCD up to an Arduino
- How to adjust the backlight brightness and display contrast using different resistor values
- That the backlight brightness can be controlled from a PWM pin
- How to declare and initialize a `LiquidCrystal` object
- How to set the correct number of columns and rows on the display
- How to clear the LCD display using `clear()`
- How to print to the cursor location using `print()`
- How to turn the display on and off using `display()` and `noDisplay()`
- How to set the cursor location using `setCursor(x, y)`
- How to scroll the display left using `scrollDisplayLeft()`
- How to scroll the display right using `scrollDisplayRight()`
- How to enable or disable the cursor `using cursor()` and `noCursor()`
- How to make a visible cursor blink `using blink()`
- How to create custom characters using `createChar()`
- How to write a single character to the cursor location using `write()`
- How an LCD display works
- How to read values from an analog temperature sensor
- How to increase ADC resolution using an internal voltage reference
- Decision making using the `switch/case` statement
- How to convert ADC values to temperature readings in both Celsius and Fahrenheit
- How to convert the code to read from different temperature sensors with different ranges

CHAPTER 9

Servos

In this chapter, we are going to look at servo motors or servomechanisms. A servo is a motor with a feedback system that helps to control the position of the motor. Servos typically rotate through 180 degrees, although you can also buy continuous rotation servos or even modify a standard one for continuous rotation. If you have ever owned a radio-controlled (RC) airplane, you have come across servos; they are used to control the flight surfaces. RC cars use them for the steering mechanism, and RC boats use them to control the rudder. Likewise, they are often used as the moving joints in small robot arms and for controlling movement in animatronics. Perhaps by the end of this chapter you'll be inspired to put some servos inside a teddy bear or other toy to make it move. Figures 9-1 and 9-2 show other ways to use servos.

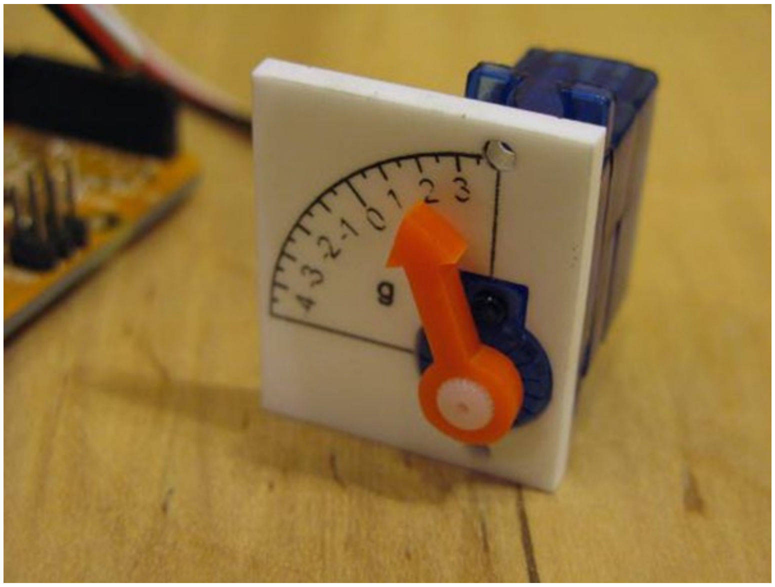


Figure 9-1. *A servo being used to control a meter (image by Tod E. Kurt)*

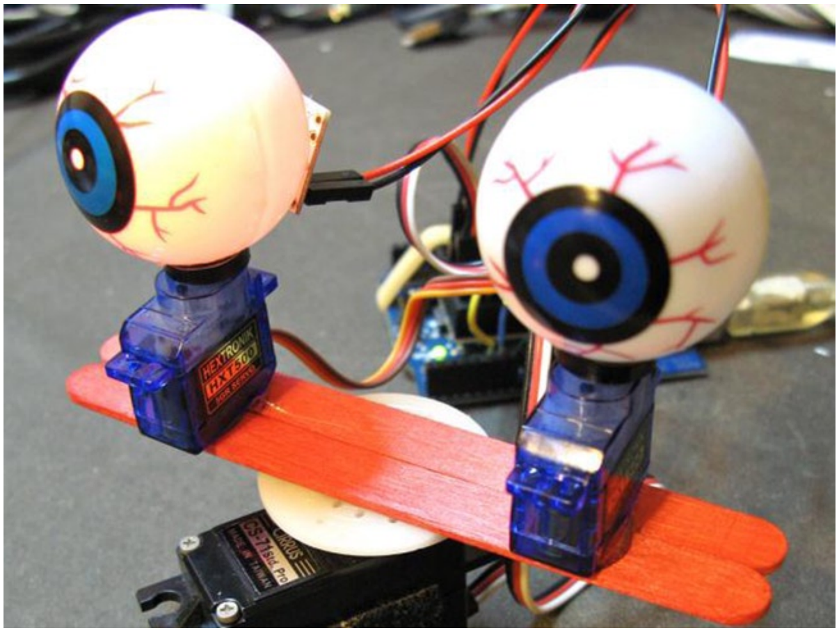


Figure 9-2. *Three servos controlling a head and eyeballs for a robot (image by Tod E. Kurt)*

Servos are really easy to control thanks to the servo library that comes with the Arduino IDE. The three projects in this chapter are all quite simple and small compared to some of the other projects in the book and yet are very effective. Let's start off with a really simple program to control one servo, then move onto two servos, and finish with two servos controlled by a joystick.

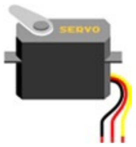
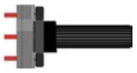
Project 25 – Servo Control

In this very simple project you will control a single servo using a potentiometer.

Parts Required

You will need to obtain a standard RC servo; any of the small or mid-sized servos will do. The servo must be powered by its own power supply. Do not power it from the Arduino's 5V supply as this will cause both noise and excessive heat, potentially disrupting the program or worse, damaging the Arduino. Use an external 5V supply or a battery pack. Make sure the ground of both the Arduino and the power supply are connected. Also, add a resistor in series between the Arduino's output and the control input of the servo to limit current if the Arduino is on when the servo's supply is off. A 220 ohm resistor will do here. Also, you'll need a potentiometer; pretty much any value rotary potentiometer will do. I used a 4.7K ohm one for testing. Note that you may also wish to connect your Arduino to an external DC power supply. See Table 9-1 for the list of parts for this project.

Table 9-1. *Parts Required for Project 25*

Standard RC Servo	
Rotary Potentiometer	
220 ohm resistor	

Connect It Up

The circuit for Project 25 is extremely simple. Connect it as shown in Figure 9-3.

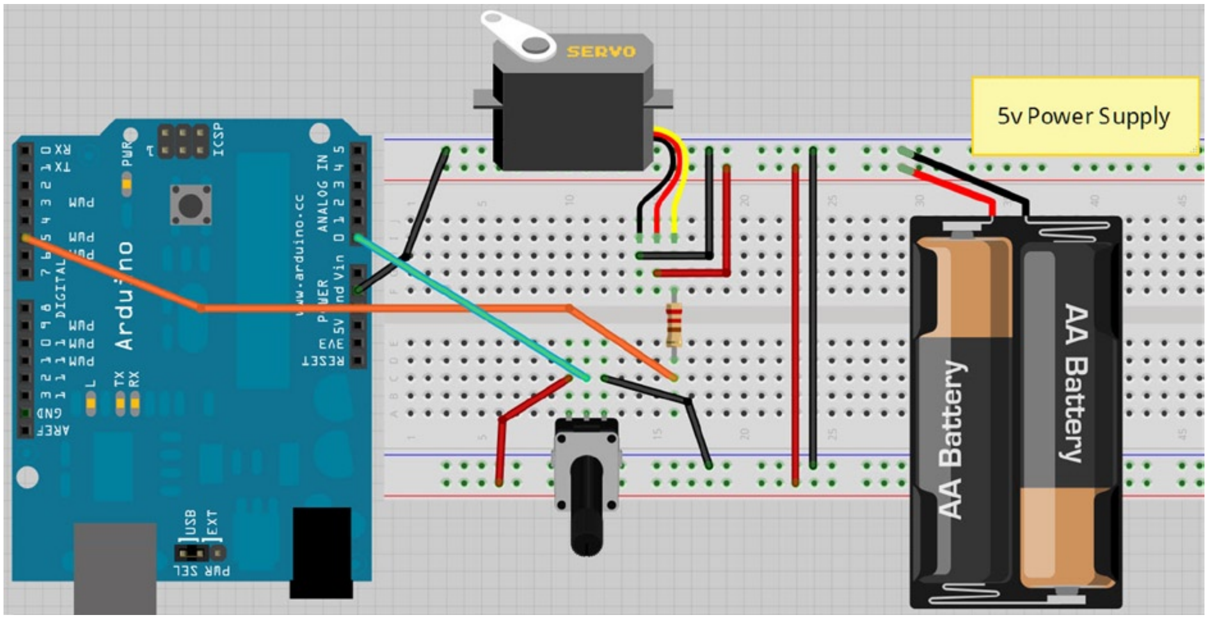


Figure 9-3. *The circuit for Project 25 – Servo Control*

Remember to power the servo from an external 5V supply and not from the Arduino. The servo has three wires coming from it. One will be red and will go to +5V. One will be black or brown and will go to ground. The third will be white, yellow, or orange and will be connected to digital pin 5 via a 220 ohm resistor.

The rotary potentiometer has the outer pins connected to +5V and ground and the middle pin to analog pin 0. Once everything is connected as it should be, enter the code below.

Enter the Code

And now for one of the shortest programs in the book, see Listing 9-1!

Listing 9-1. Code for Project 25

```
// Project 25
#include <Servo.h>

Servo servo1;  // Create a servo object

void setup()
{
        servo1.attach(5);  // Attaches the servo on Pin 5 to the servo object
}

void loop()
{
        int angle = analogRead(0); // Read the pot value
        angle=map(angle, 0, 1023, 0, 180); // Map the values from 0 to 180 degrees
        servo1.write(angle); // Write the angle to the servo
        delay(15); // Delay of 15ms to allow servo to reach position
}
```

Project 25 – Servo Control – Code Overview

First, the Servo.h library is included:

```
#include <Servo.h>
```

Then a servo object called `servo1` is declared:

```
Servo servo1;  // Create a servo object
```

In the setup loop, you attach the servo you have just created to pin 5:

```
servo1.attach(5);  // Attaches the servo on Pin 5 to the servo object
```

The attach command attaches a created servo object to a designated pin. The attach command can take either one parameter, as in your case, or three parameters. If three parameters are used, the first parameter is the pin, the second is the minimum (0 degree) angle in pulse width in microseconds (defaults to 544), and the third parameter is the maximum degree angle (180 degrees) in pulse width in microseconds (defaults to 2400). This will be explained in the hardware overview. For most purposes you can simply set the pin and ignore the optional second and third parameters.

You can connect up to 12 servos to an Arduino Duemilanove (or equivalent) and up to 48 on the Arduino Mega—perfect for robotic control applications!

Note that using this library disables the analogWrite (PWM) function on pins 9 and 10. On the Mega you can have up to 12 motors without interfering with the PWM functions. The use of between 12 and 23 motors will disable the PWM functionality on pins 11 and 12.

In the main loop, read the analog value from the potentiometer connected to analog pin 0:

```
int angle = analogRead(0); // Read the pot value
```

Then that value is mapped so the range is now between 0 and 180, which will correspond to the degree angle of the servo arm:

```
angle=map(angle, 0, 1023, 0, 180); // Map the values from 0 to 180 degrees
```

Then you take your servo object and write the appropriate angle, in degrees, to it (the angle must be between 0 and 180 degrees):

```
servo1.write(angle); // Write the angle to the servo
```

Finally, a delay of 15 ms is programmed to allow the servo time to move into position:

```
delay(15); // Delay of 15ms to allow servo to reach position
```

You can also `detach()` a servo from a pin, which will disable it and allow the pin to be used for something else. Also, you can `read()` the current angle from the servo (this is the last value passed to the `write()` command).

You can read more about the servo library on the Arduino website at `http://arduino.cc/en/Reference/Servo`.

Project 25 – Servo Control – Hardware Overview

A servo is a little box that contains a DC electric motor, a set of gears between the motor and an output shaft, a position-sensing mechanism, and the control circuit. The position-sensing mechanism feeds back the servo's position to the control circuitry, which uses the motor to adjust the servo arm to the position that the servo should be at.

Servos come in many sizes, speeds, strengths, and precisions. Some of them can be quite expensive. The more powerful or precise the servo is, the higher the price. Servos are most commonly used in radio-controlled aircraft, cars, and boats.

The servo's position is controlled by providing a set of pulses. This is PWM, which you have come across before. The width of the pulses is measured in milliseconds. The rate at which the pulses are sent isn't particularly important; it's the width of the pulse that matters to the control circuit. Typical pulse rates are between 400Hz and 50Hz.

On a standard servo the center position is reached by providing pulses at 1.5 millisecond width, the -45 degree position from centre of range by providing 1 millisecond pulses, and the +45 degree position by providing 2 millisecond pulses. You will need to read the datasheet for your servo to find the pulse widths required for the different angles. However, you are using the servo library for this project, so you don't need to worry: the library provides the required PWM signal to the servo. Whenever you send a different angle value to the servo object, the code in the library takes care of converting the angle to an appropriate pulse width signal to the servo.

Some servos provide continuous rotation. Alternatively, you can modify a standard servo relatively easily to provide continuous rotation.

A continuous rotation servo is controlled in the same way, by providing an angle between 0 and 180 degrees. However, a value of 0 will provide rotation at full speed in one direction, a value of 90 will be stationary, and a value of 180 will provide rotation at full speed in the opposite direction. Values in-between these will make the servo rotate in one direction or the other and at different speeds. Continuous rotation servos are great for building small robots (see Figure 9-4). They can be connected to wheels to provide precise speed and direction control of each wheel.

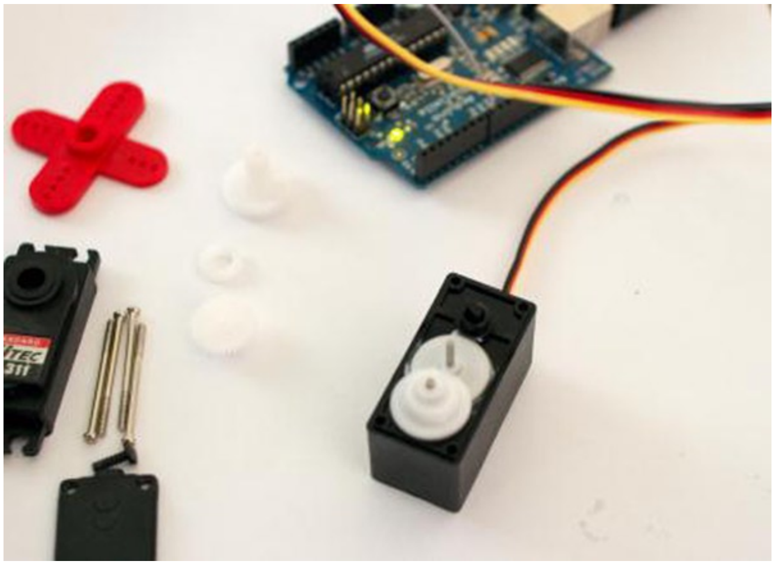


Figure 9-4. *Modifying a servo to provide continuous rotation (image by Adam Grieg)*

There is another kind of servo known as a linear actuator that rotates a shaft to a desired position, allowing you to push and pull items connected to the end of the shaft. These are used a lot in the TV program *MythBusters* by their resident robotics expert, Grant Imahara.

Note that if the servo "growls" at one end of the range or the other, the Arduino is trying to drive the servo beyond its range, which will result in high current. This can be prevented by specifying the second and third arguments when calling servo.attach( ), and increasing the minimum pulse width from the 544 microsecond default or decreasing the maximum pulse width from the 2400 microsecond default until the growling no longer occurs when the commanded position is not changing.

Project 26 – Dual Servo Control

You'll now create another simple project, but this time you'll control two servos using commands from the serial monitor. You learned about serial control in Project 10 when you were changing the colors on an RGB lamp with serial commands. So let's cannibalize the code from Project 10 to make this one.

Parts Required

This project requires two servos. You will not need the potentiometer. Parts are listed in Table 9-2.

Table 9-2. *Parts Required for Project 26*

Standard RC Servo × 2	
220 ohm resistor x 2	

Connect It Up

The circuit for Project 26 is, again, extremely simple. Connect it as shown in Figure 9-5. Basically, you remove the potentiometer from the last project and wire a second servo up to digital pin 6 via a 220 ohm resistor.

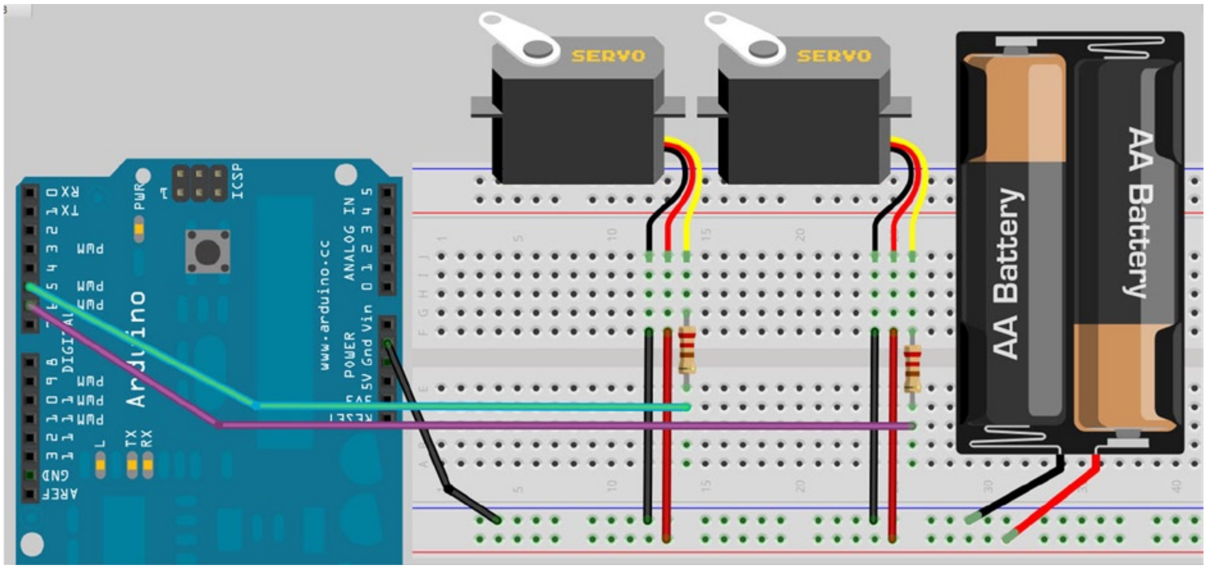


Figure 9-5. *The circuit for Project 26 – Dual Servo Control*

Enter the Code

Enter the code in Listing 9-2.

Listing 9-2. Code for Project 26

```
// Project 26
#include <Servo.h>

char buffer[11];
Servo servo1;  // Create a servo object
Servo servo2;  // Create a second servo object
```

```
void setup()
{
        servo1.attach(5);  // Attaches the servo on pin 5 to the servo1 object
        servo2.attach(6);  // Attaches the servo on pin 6 to the servo2 object
        Serial.begin(9600);
        while(Serial.available())
                Serial.read();
        servo1.write(90);  // Put servo1 at home position
        servo2.write(90);  // Put servo2 at home postion
        Serial.println("STARTING...");
}

void loop()
{
        if (Serial.available() > 0) { // Check if data has been entered
                int index=0;
                delay(100); // Let the buffer fill up
                int numChar = Serial.available(); // Find the string length
                if (numChar>10) {
                numChar=10;
                }
                while (numChar--) {
                        // Fill the buffer with the string
                        buffer[index++] = Serial.read();
                }
                buffer[index]='\0';
                splitString(buffer); // Run splitString function
        }
}

void splitString(char* data) {
        Serial.print("Data entered: ");
        Serial.println(data);
        char* parameter;
        parameter = strtok (data, " ,"); //String to token
        while (parameter != NULL) { // If we haven't reached the end of the string...
                setServo(parameter); // ...run the setServo function
                parameter = strtok (NULL, " ,");
        }
        while(Serial.available())
                Serial.read();
}

void setServo(char* data) {
        if ((data[0] == 'L') || (data[0] == 'l')) {
                int firstVal = strtol(data+1, NULL, 10); // String to long integer
                firstVal = constrain(firstVal,0,180); // Constrain values
                servo1.write(firstVal);
                Serial.print("Servo1 is set to: ");
                Serial.println(firstVal);
        }
        if ((data[0] == 'R') || (data[0] == 'r')) {
                int secondVal = strtol(data+1, NULL, 10); // String to long integer
                secondVal = constrain(secondVal,0,255);   // Constrain the values
                servo2.write(secondVal);
                Serial.print("Servo2 is set to: ");
                Serial.println(secondVal);
        }
}
```

To run the code, open up the serial monitor window. The Arduino will reset, and the servos will move to their central locations. You can now use the serial monitor to send commands to the Arduino.

The left servo is controlled by sending an L and then a number between 0 and 180 for the angle. The right servo is controlled by sending an R and the number. You can send individual commands to each servo or send both commands at the same time by separating the commands with a space or comma, like so:

```
L180
L45 R135
L180,R90
R77
R25 L175
```

This is a simple example of how you could send commands down a wire to an Arduino-controlled robot arm or an animatronic toy. Note that the serial commands don't have to come from the Arduino serial monitor. You can use any program that is capable of communicating over serial or write your own in Python or C++.

Project 26 – Dual Servo Control – Code Overview

The code for this project is basically unchanged from that of Project 10. I will therefore not go into each command in detail. Instead, I will give an overview if it is something already covered. Read up on Project 10 for a refresher on how the string manipulation commands work.

First the Servo.h library header file is included:

```
#include <Servo.h>
```

Then an array of type `char` is created to hold the text string you enter as a command into the serial monitor:

```
char buffer[11];
```

Two servo objects are created:

```
Servo servo1;  // Create a servo object
Servo servo2;  // Create a second servo object
```

In the setup routine, attach the servo objects to pins 5 and 6:

```
servo1.attach(5);  // Attaches the servo on pin 5 to the servo1 object
servo2.attach(6);  // Attaches the servo on pin 6 to the servo2 object
```

Then begin serial communications and then we run a while(Serial.available()) Serial.read(); which reads data in only once it is available.

```
Serial.begin(9600);
while(Serial.available())
Serial.read();
```

Both servos have a value of 90, which is the center point, written to them so that they start off in the central position:

```
servo1.write(90);  // Put servo1 at home position
servo2.write(90);  // Put servo2 at home position
```

Then the word "STARTING......" is displayed in the serial monitor window so you know the device is ready to receive commands:

```
Serial.println("STARTING...");
```

In the main loop, check if any data has been sent down the serial line

```
if (Serial.available() > 0) { // check if data has been entered
```

and if so, let the buffer fill up and obtain the length of the string, ensuring it does not overflow above the maximum of 10 characters. Once the buffer is full, you call the `splitString` routine sending the buffer array to the function:

```
int index=0;
delay(100); // Let the buffer fill up
int numChar = Serial.available(); // Find the string length
if (numChar>10) {
        numChar=10;
}
while (numChar--) {
        // Fill the buffer with the string
        buffer[index++] = Serial.read();
}
Buffer[index]="\o";
splitString(buffer); // Run splitString function
```

The `splitString` function receives the buffer array, splits it into separate commands if more than one is entered, and calls the `setServo` routine with the parameter stripped from the command string received over the serial line:

```
void splitString(char* data) {
        Serial.print("Data entered: ");
        Serial.println(data);
        char* parameter;
        parameter = strtok (data, " ,"); //String to token
        while (parameter != NULL) { // If we haven't reached the end of the string...
                setServo(parameter); // ...run the setServo function
                parameter = strtok (NULL, " ,");
        }
        while(Serial.available())
                Serial.read();
}
```

The setServo routine receives the smaller string sent from the `splitString` function and checks if an L or R is entered, and if so, moves either the left or right servo by the amount specified in the string:

```
void setServo(char* data) {
        if ((data[0] == 'L') || (data[0] == 'l')) {
                int firstVal = strtol(data+1, NULL, 10);  // String to long integer
                firstVal = constrain(firstVal,0,180);     // Constrain values
                servo1.write(firstVal);
                Serial.print("Servo1 is set to: ");
                Serial.println(firstVal);
        }
        if ((data[0] == 'R') || (data[0] == 'r')) {
                int secondVal = strtol(data+1, NULL, 10); // String to long integer
                secondVal = constrain(secondVal,0,255);   // Constrain the values
                servo2.write(secondVal);
                Serial.print("Servo2 is set to: ");
                Serial.println(secondVal);
```

I've glossed over these last two functions as they are almost identical to those in Project 10. If you cannot remember what was covered in Project 10, feel free to go back and reread it.

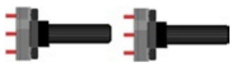
Project 27 – Joystick Servo Control

For another simple project, let's use a joystick to control the two servos. You'll arrange the servos in such a way that you get a pan-tilt head, such as is used for CCTV cameras or for camera or sensor mounts on robots.

Parts Required

Leave the circuit as it was for the last project, and add either two potentiometers or a two-axis potentiometer joystick. See Table 9-3 for the parts list.

***Table 9-3.** Parts Required for Project 27*

Standard RC Servo × 2	
2-axis potentiometer joystick (or two potentiometers)	
220 ohm resistor x 2	

Connect It Up

The circuit for Project 27 is the same as for Project 26, with the addition of the joystick. See Figure 9-6.

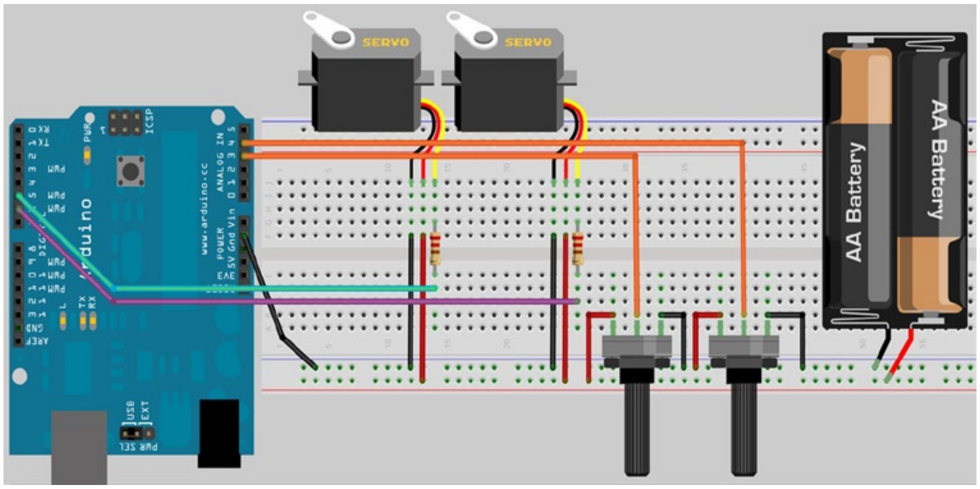


Figure 9-6. *The circuit for Project 27 – Joystick Servo Control*

A potentiometer joystick is simply that: a joystick made up of two potentiometers at right angles to each other. The axles of the pots are connected to a lever that is swung back and forth by the stick and returned to their center positions thanks to a set of springs.

Connection is therefore easy: the outer pins of the two pots go to +5V and ground and the center pins go to analog pins 3 and 4 (instead of analog pin 1 used in Project 25) via the resistors. If you don't have a joystick, two potentiometers arranged at 90 degrees to each other will suffice.

Connect the two servos so that one has its axle vertical and the other horizontal at 90 degrees to the first servo and attached to the first servo's armature sideways. See Figure 9-7 for how to connect the servos. Some hot glue will do for testing. Use stronger glue for a permanent fixing.

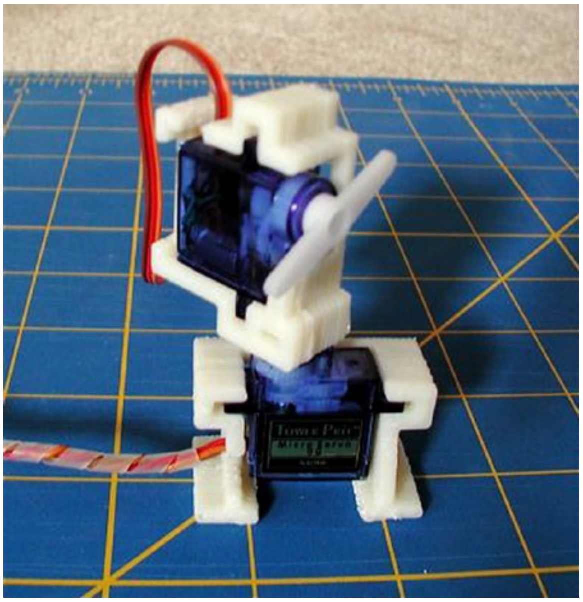


Figure 9-7. *Mount one servo on top of the other (image by David Stokes)*

Alternatively, get one of the ready-made pan and tilt servo sets you can buy for robotics. These can be picked up cheaply on eBay.

When the bottom servo moves, it causes the top servo to rotate, and when the top servo moves, its arm rocks back and forth. You could attach a webcam or an ultrasonic sensor to the arm, for example.

The joystick can be purchased from eBay or an electrical supplier. You could also find an old C64 or Atari joystick. However, there is a cheap alternative available called a PS2 controller: it contains two two-axis potentiometer joysticks as well as a set of vibration motors and other buttons. These can be purchased on eBay very cheaply and are easily taken apart to access the parts within (see Figure 9-8). If you don't want to take the controller apart, you could access the digital code coming from the cable of the PS2 controller. In fact, there are Arduino libraries to enable you to do just this. This will give you full access to all of the joysticks and buttons on the device at once.

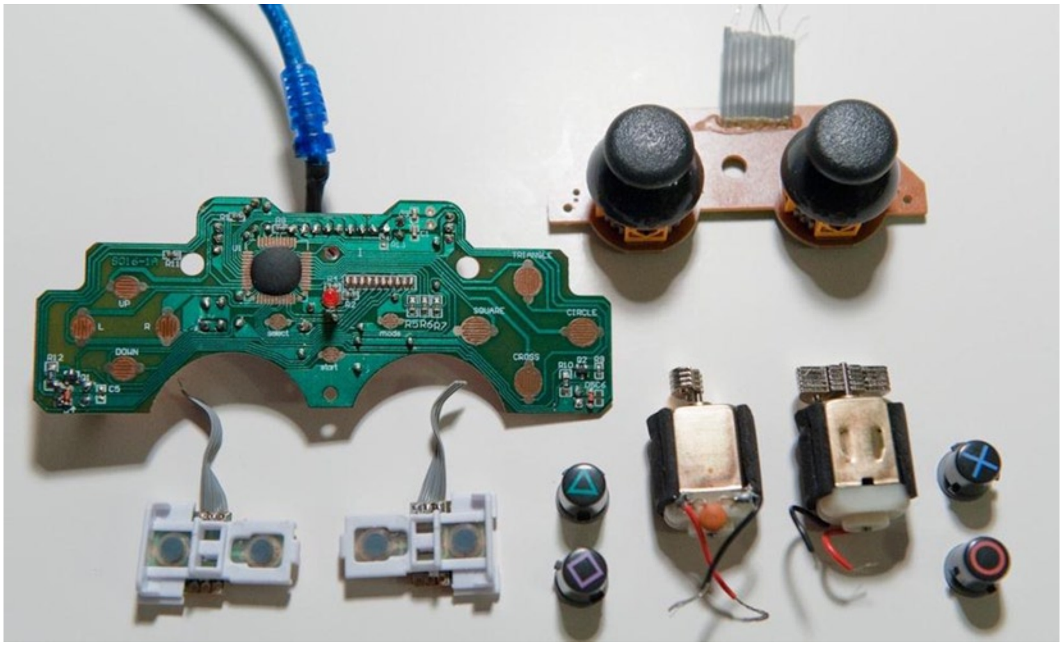


Figure 9-8. *All the great parts available inside a PS2 Controller (image by Mike Prevette)*

Enter the Code

Enter the code in Listing 9-3.

Listing 9-3. Code for Project 27

```
// Project 27
#include <Servo.h>

Servo servo1;  // Create a servo object
Servo servo2;  // Create a second servo object
int pot1, pot2;
```

```
void setup()
{
  servo1.attach(5);  // Attaches the servo on pin 5 to the servo1 object
  servo2.attach(6);  // Attaches the servo on pin 6 to the servo2 object

  servo1.write(90);  // Put servo1 at home position
  servo2.write(90);  // Put servo2 at home postion

}

void loop()
{
  pot1 = analogRead(3); // Read the X-Axis
  pot2 = analogRead(4); // Read the Y-Axis
  pot1 = map(pot1,0,1023,0,180);
  pot2=map(pot2,0,1023,0,180);
  servo1.write(pot1);
  servo2.write(pot2);
  delay(15);
}
```

When you run this program you will be able to use the servos as a pan/tilt head. Rocking the joystick backwards and forwards will cause the top servo's armature to rock back and forth, and moving the joystick from side to side will cause the bottom servo to rotate.

If you find that the servos are going in the opposite direction from what you expected, then you have the outer pins of the appropriate servo connected the wrong way. Just swap them around.

Project 27 – Joystick Servo Control – Code Overview

Again, this is a very simple project, but the effect of the two servos moving is quite compelling.

The Servo library is loaded:

```
#include <Servo.h>
```

Two servo objects are created and two sets of integers hold the values read from the two potentiometers inside the joystick:

```
Servo servo1;  // Create a servo object
Servo servo2;  // Create a second servo object
int pot1, pot2;
```

The setup loop attaches the two servo objects to Pins 5 and 6 and moves the servos into the central positions:

```
servo1.attach(5);  // Attaches the servo on Pin 5 to the servo1 object
servo2.attach(6);  // Attaches the servo on Pin 6 to the servo2 object

servo1.write(90);  // Put servo1 at home position
servo2.write(90);  // Put servo2 at home postion
```

In the main loop, the analog values are read from both the X and Y axis of the joystick:

```
pot1 = analogRead(3); // Read the X-Axis
pot2 = analogRead(4); // Read the Y-Axis
```

Those values are then mapped to be between 0 and 180 degrees

```
pot1 = map(pot1,0,1023,0,180);
pot2 = map(pot2,0,1023,0,180);
```

and then sent to the two servos

```
servo1.write(pot1);
servo2.write(pot2);
```

The range of motion available with this pan/tilt rig is amazing, and you can make the rig move in a very humanlike way. This kind of servo setup is often made to control a camera for aerial photography (see Figure 9-9).

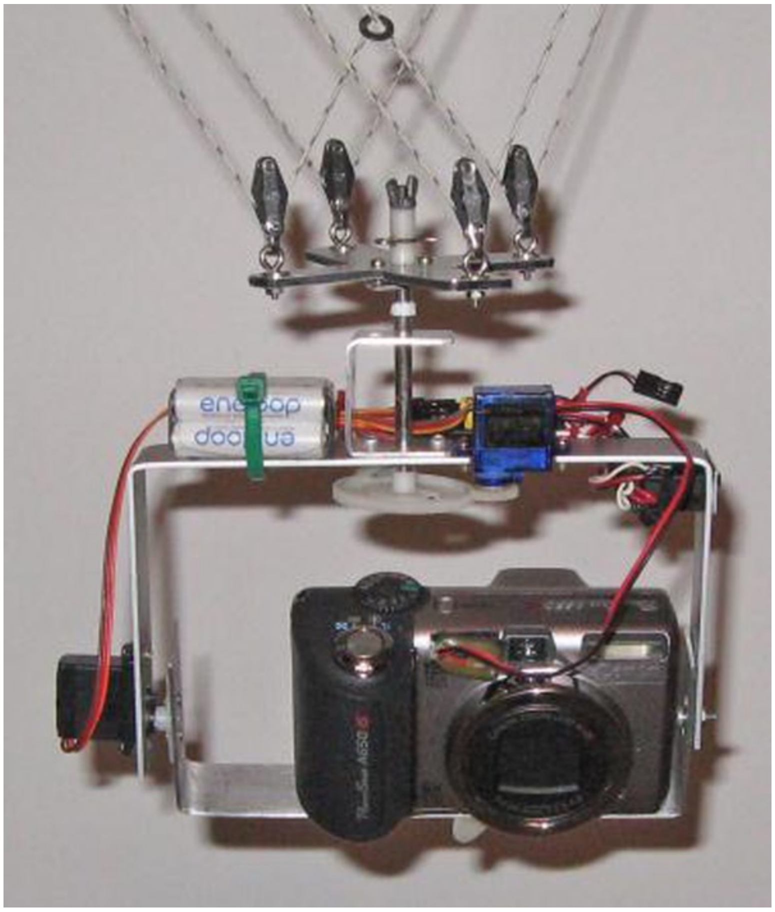


Figure 9-9. *A pan/tilt rig made for a camera using two servos (image by David Mitchell)*

Summary

In Chapter 9, you worked your way through three very simple projects that show how easily servos can be controlled using the servo.h library. You can easily modify these projects to add up to 12 servos to make a toy dinosaur, for example, move around in a realistic manner.

Servos are great for making your own RC vehicle or for controlling a robot. Furthermore, as seen in Figure 9-9, you can make a pan/tilt rig for camera control. A third servo could even be used to push the shutter button on the camera.

Subjects and Concepts Covered in Chapter 9.

- The many potential uses for a servo
- How to use the Servo library to control between 1 and 12 servos
- How to use a potentiometer as a controller for a servo
- How a servo works
- How to modify a servo to provide continuous rotation
- How to control a set of servos using serial commands
- How to use an analog joystick for dual axis servo control
- How to arrange two servos to create a pan/tilt head
- How a PS2 Controller makes a great source for joystick and button parts

CHAPTER 10

Steppers and Robots

You are now going to take a quick look at a new type of motor called a stepper motor. Project 28 is a simple project to show you how to control the stepper, make it move a set distance, and change its speed and direction. Stepper motors are different than standard motors in that their rotation is divided into a series of steps.

By controlling the timing and number of steps issued to the motor, you can control the speed of the motor and how far it turns fairly precisely.

Stepper motors come in different shapes and sizes and have four, five, six, or eight wires.

Stepper motors have many uses; they are used in flatbed scanners to position the scanning head and in inkjet printers to control the location of the print head and paper.

Another project in this chapter has you using a motor shield with geared DC motors to control a robot base. You'll end up getting the robot to follow a black line drawn on the floor!

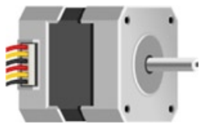
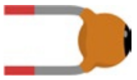
Project 28 – Basic Stepper Control

In this very simple project, you will connect up a stepper motor and then get the Arduino to control it in different directions and at different speeds for a set number of steps.

Parts Required

You will need either a bipolar or a unipolar DC stepper motor. I used a Sanyo 103H546-0440 unipolar stepper motor in the testing of this project.

Table 10-1. *The Parts for Project 28*

Stepper Motor	
L293D or SN754410 Motor Driver IC	
*2 × 0.01uf Ceramic Capacitors	
*Current-Limiting Resistor	

Connect It Up

Connect everything up as shown in Figure 10-1. See the variation for bipolar motors.

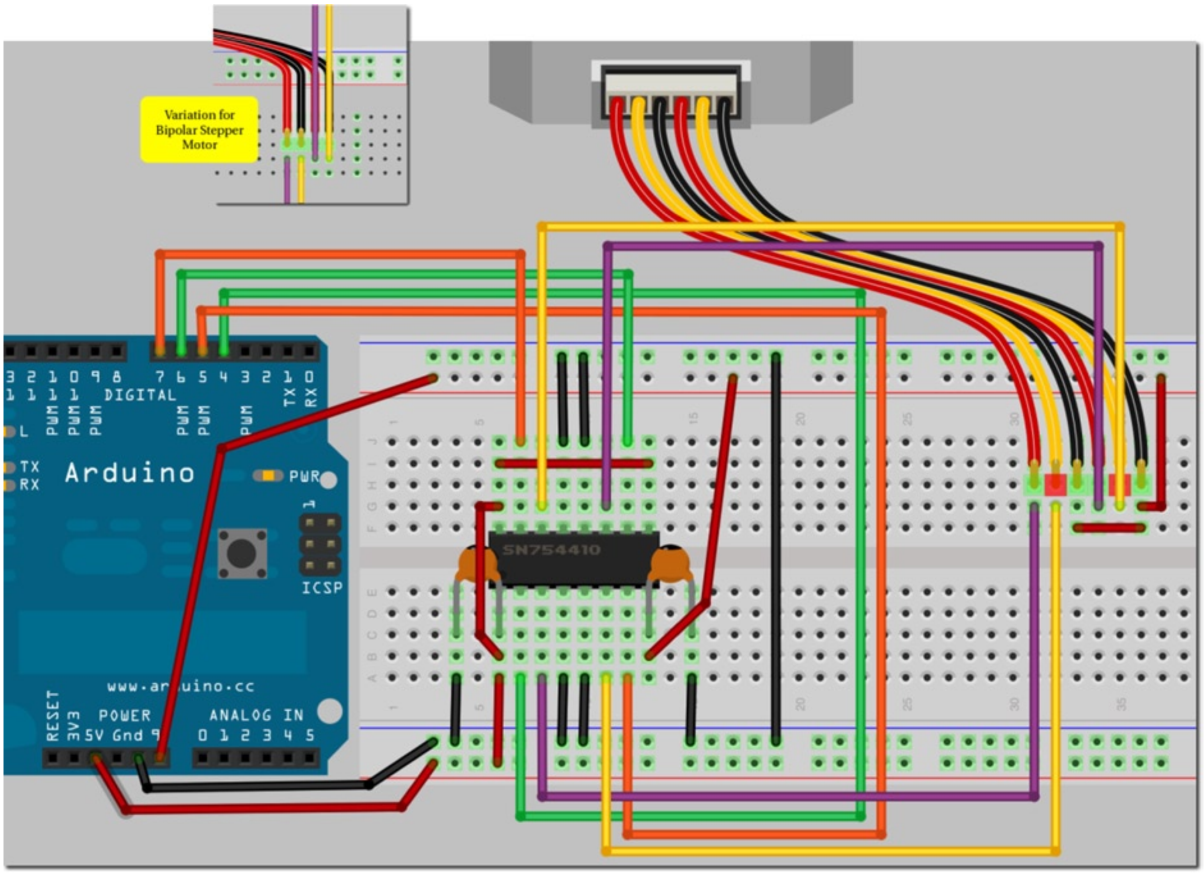


***Figure 10-1.** The circuit for Project 28—Basic Stepper Control*

Make sure the Arduino and the L293 are connected to an external DC power supply so as not to overload the Arduino. The Arduino does not need to be connected to an external supply, the L293 (or SN754410) does. It is best to run the connection to the external supply positive lead directly to the breadboard and pin 8 of the L293, and connected to the ground from the Arduino at the breadboard. By doing this rather than hooking pin 8 of the L293 to the Vin pin from the Arduino's shield connector, high currents (which can create electrical noise that can disrupt operation) won't be routed through the Arduino. (This advice holds true any time you are controlling high power devices like motors from the Arduino). The capacitors are optional and help to smooth out the current and prevent interference with the Arduino. These go between the +5v and ground and the motor power supply and ground. You can also use low value electrolytic capacitors, but make sure you connect them correctly (+ to supply, - to ground) as they are polarized and can be damaged or explode if hooked up backwards! I found that without them the circuit did not work.

You may also require a current-limiting resistor between the power supply and the power rail supplying the SN754410 or L293D chip. Use an appropriate power supply that is within the range of both voltage and current for your motor. Also ensure that the power rating of any current-limiting resistors you use is above the current required by the motor or the resistor will heat up and burn out. Note that the motor drive voltage will also be lower than the input voltage (even without a resistor) due to the internal voltage drops in the driver (which add up to 1.5 to 4 volts, depending on current and other conditions).

Digital pins 4, 5, 6 and 7 on the Arduino go to the input 1, 2, 3 and 4 pins on the motor driver (see Figure 10-2). Output pins 1 and 2 of the motor driver go across coil 1 of the motor and output pins 3 and 4 to coil 2. You will need to check the datasheet of your specific motor to see which colored wires go to coil 1 and which go to coil 2. A unipolar motor will also have a 5th and/or a 6th wire that go to ground.

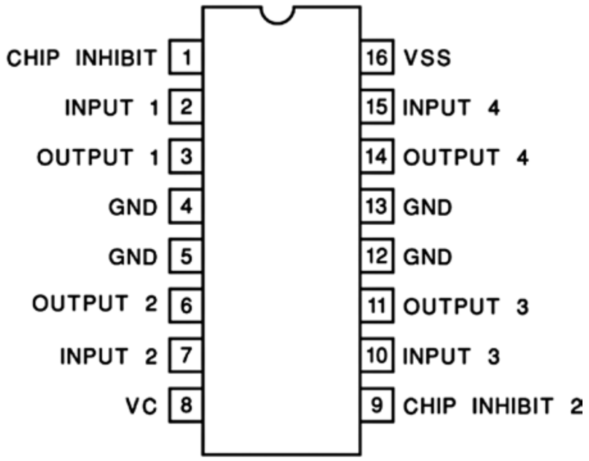


Figure 10-2. *Pin diagram for an L293D or SN754410 Motor Driver IC*

The 5v pin on the Arduino goes to pin 16 (VSS) of the motor driver pin and the two chip inhibit pins (1 and 9) are also tied to the 3.3v line to make them go HIGH, making sure neither half of the driver chip is inhibited.

The Vin (Voltage in) pin on the Arduino goes to pin 8 of the driver IC (VC). Pins 4, 5, 12, and 13 all go to ground (See Figure 10-2).

Once you are happy that everything is connected up as it should be, enter the code below.

Enter the Code

Enter the code in Listing 10-1.

Listing 10-1. Code for Project 28

```
// Project 28
#include <Stepper.h>

// steps value is 360 / degree angle of motor
#define STEPS 200

// create a stepper object on pins 4, 5, 6 and 7
Stepper stepper(STEPS, 4, 5, 6, 7);
```

```
void setup()
{
}

void loop()
{
    stepper.setSpeed(60);
    stepper.step(200);
    delay(100);
    stepper.setSpeed(20);
    stepper.step(-50);
    delay(100);
}
```

Make sure that your Arduino is powered by an external DC power supply before running the code. When the sketch runs, you will see the stepper motor rotate a full rotation, stop for a short time, then rotate backwards for a quarter rotation, stop a short time, then repeat. It may help to put a small tab of tape to the spindle of the motor so you can see it rotating.

Project 28 – Basic Stepper Control – Code Overview

The code for this project is, again, nice and simple, thanks to the stepper.h library that does all of the hard work for us. First, you include the library in the sketch:

```
#include <Stepper.h>
```

Then you need to define how many steps the motor requires to do a full 360 degree rotation. Typically, stepper motors will come in either a 7.5 degree or a 1.8 degree variety, but you may have a stepper motor with a different step angle. To work out the steps, just divide 360 by the step angle. In the case of the stepper motor I used, the step angle was 1.8 degrees, meaning 200 steps were required to carry out a full 360 degree rotation:

```
#define STEPS 200
```

Then you create a stepper motor object, call it `stepper` and assign the pins going to either side of the two coils:

```
Stepper stepper(STEPS, 4, 5, 6, 7);
```

The setup function does nothing at all, but must be included:

```
void setup()
{
}
```

In the main loop, you first set the speed of the motor in rpm (revolutions per minute). This is done with the `.setSpeed` command and the speed, in rpm, goes in the parenthesis:

```
stepper.setSpeed(60);
```

Then you tell the stepper how many steps it must carry out. You set this to 200 steps at 60 rpm, meaning it carries out a full revolution in one second:

```
stepper.step(200);
```

At the end of this revolution, a delay of 1/of a second is carried out:

```
delay(100);
```

Then the speed is slowed down to just 20 rpm:

```
stepper.setSpeed(20);
```

The motor is then made to rotate in the opposite direction (hence the negative value) for 50 steps, which on a 200 step motor is a quarter revolution:

```
stepper.step(-50);
```

Then another 100 millisecond delay is executed and the loop repeats.

As you can see, controlling a stepper motor is very easy and yet you have control over its speed, which direction it goes in, and exactly how much the shaft will rotate. You can therefore control exactly where the item attached to the shaft rotates and by how much. Using a stepper motor to control a robot wheel would give very precise control, and rotating a robot head or camera would allow you to position it to point exactly where you want it.

Project 28 – Basic Stepper Control – Hardware Overview

The new piece of hardware you are using in this project is the stepper motor. A stepper motor is a DC motor in which the rotation can be divided up into a number of smaller steps. This is done by having an iron gear-shaped rotor attached to the shafts inside the motor. Around the outside of the motor are electromagnets with teeth. One coil is energized, causing the teeth of the iron rotor to align with the teeth of the electromagnet. The teeth on the next electromagnet are slightly offset from the first; when it is energized and the first coil is turned off, this causes the shaft to rotate slightly more to toward the next electromagnet. This process is repeated by however many electromagnets are inside until the teeth are almost aligned with the first electromagnet, and the process is repeated.

Each time an electromagnet is energized and the rotor moves slightly, it is carrying out one step. By reversing the sequence of electromagnets energizing the rotor, it turns in the opposite direction.

The job of the Arduino is to apply the appropriate HIGH and LOW commands to the coils in the correct sequence and speed to enable the shaft to rotate. This is what is going on behind the scenes in the stepper.h library.

A unipolar motor has five or six wires, and these go to four coils. The center pins on the coils are tied together and go to the power supply. Bipolar motors usually have four wires and they have no common center connections. See Figure 10-3 for different types and sizes of stepper motors.

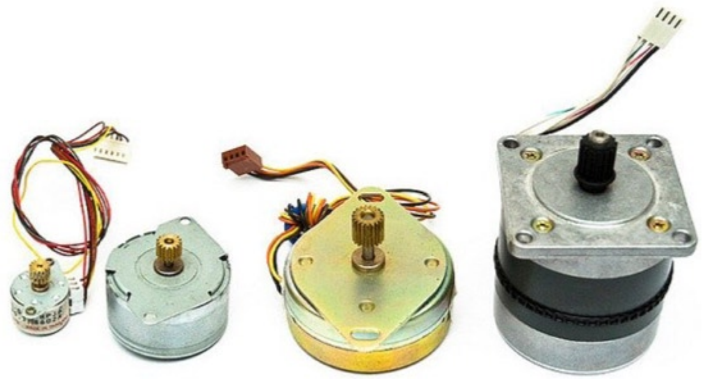


Figure 10-3. *Different kinds of stepper motors (image by Aki Korhonen)*

The step sequence for a stepper motor can be seen in Table 10-2.

Table 10-2. *Step Sequence for a Stepper Motor*

Step	Wire 1	Wire 2	Wire 4	Wire 5
1	HIGH	LOW	HIGH	LOW
2	LOW	HIGH	HIGH	LOW
3	LOW	HIGH	LOW	HIGH
4	HIGH	LOW	LOW	HIGH

Figure 10-4 shows the diagram for a unipolar stepper motor. As you can see, there are four coils (there are actually two coils, but they are divided by the center wires into two smaller coils). The center wire goes to the power supply and the two other wires go to the outputs on one side of the H-Bridge driver IC, and the other two wires for the second coil go to the last two outputs of the H-Bridge.

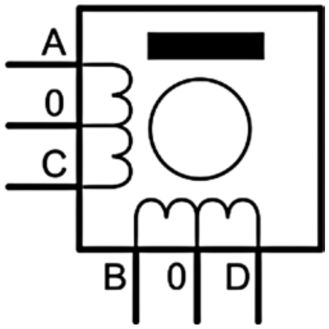


Figure 10-4. *Circuit diagram for a unipolar stepper motor*

Figure 10-5 shows the diagram for a bipolar stepper motor. This time there are just two coils with no centre pin. The step sequence for a bipolar motor is the same as for a unipolar. However, this time you reverse the current across the coils, i.e. a HIGH equates to a positive voltage and a LOW to a ground (or a pulled-down connection).

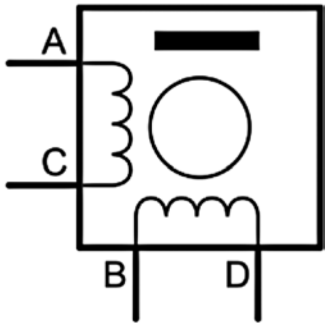


Figure 10-5. *Circuit diagram for a bipolar stepper motor*

It is possible to increase the resolution of the stepper motor by using special techniques for half stepping and microstepping. These can increase the resolution of the motor, but at the expense of reduced torque and reduced position reliability. The best way to increase resolution is through a gearing system. The garden-variety DC motor offers speed control (via voltage or pulse width modulation (PWM)) without position control. The servo motor offers position control, but not speed control. The stepper motor offers control of both speed and position. These projects have been designed to give a simple overview and introduction to driving these motors. The accuracy of speed and / or position control depends on staying within various design specifications of the respective motors. The more you increase the speed, acceleration, or load on the motor, the more likely you are to run into these design limits and find the motor will no longer behave quite as you expected. Fortunately, there are many fun and useful things that can be accomplished without pushing readily available motors to their limits. This book gives you an overview of them. But if you are doing anything in which human safety is involved, then it is essential that you should do more extensive learning about motors and control systems, and thoroughly verify your designs.

Now that you know all about controlling a stepper motor, you'll go back to regular motor control, but this time you'll be using a motor shield and controlling two motors connected to wheel on a robot base.

Project 29 – Using a Motor Shield

RequiredYou are now going to use a motor shield to control two motors separately. You'll learn how to control the speed and direction of each motor and apply that knowledge to control the wheels on a two-wheeled robot. When you move onto Project 30, these skills will be employed to make a line-following robot.

Parts Required

You can run this project using just two DC motors if that is all you have. However, that's not a lot of fun. So either obtain a two-wheeled robot base or make one out of two geared motors with wheels. The rotation at half speed needs to be about 20cm or 6 inches of forward movement per second.

The battery pack will need to have enough voltage to run your Arduino, shield and motors. I used 6 × AA batteries in a holder with a jack plug soldered to the wire to power my setup for testing this project.

Table 10-3. *The Parts Required for Project 29*

Motor Shield	
2 × DC Motors or …	
… a 2 wheeled robot base	
Battery pack	

Connect It Up

As this project requires nothing more than a motor shield plugged into an Arduino and your two motors wires hooked up to the four outputs from the shield, there is no need for a circuit diagram. Instead, in Figure 10-6, you have a pretty picture of a motor shield plugged into an Arduino. Power the shield and motor via an external power supply rather than using the power supply from the Arduino to ensure a noise and trouble-free circuit.

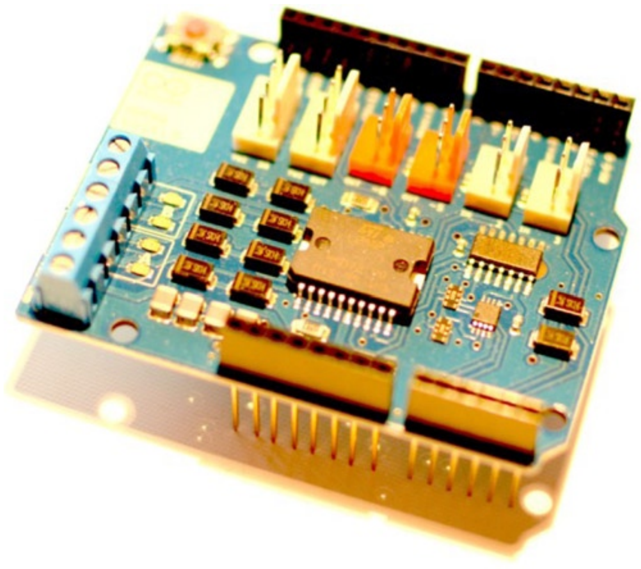


Figure 10-6. *An Arduino motor shield (image courtesy of Earthshine Electronics))*

For the testing of this project I used an official Arduino Motor Shield. This shield is designed to control two motors. There are also shields available from Adafruit that will let you control up to four DC motors, two stepper motors, or two servos. The Adafruit shield also comes with the excellent AFMotor.h library that makes controlling the motors, steppers, or servos easy. The choice is up to you. However, the code for this project was designed with the Arduino Motor Shield in mind, so if you use a different shield, you will need to modify the code accordingly.

Connect the left motor across the motor A terminals of the motor shield, and the right motor across the motor B terminals of the shield. I also used a two-wheeled robot base from DFRobot (see Figure 10-7). This is a nice inexpensive robot base that comes with attachments and mounting holes for fitting sensors. It's ideal for the line-following robot project that comes next. However, any robot base will do for this project; you could even use a four-wheeled robot base if you just remember to control both sets of motors on each side instead of just the one. Many similar robot bases are available from eBay and Amazon. Of course, if you don't want to get a robot base, you can test just using two DC motors instead.

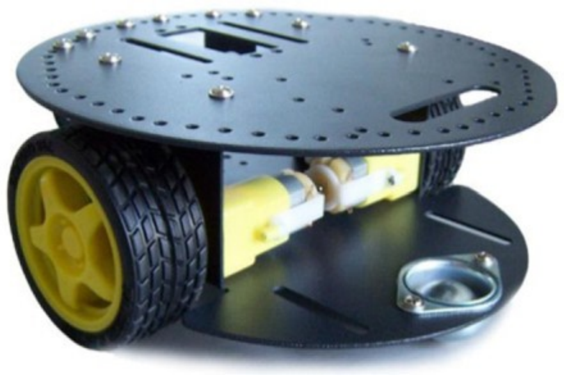


Figure 10-7. *A two-wheeled robot base (image courtesy of DFRobot)*

Enter the Code

Enter the code from Listing 10-2.

Listing 10-2. Code for Project 29

```
// Project 29 - Using a motor shield
#define QUARTER_SPEED 64
#define HALF_SPEED 128
#define FULL_SPEED 255

// Set the pins for speed and direction of each motor
int speed1 = 3;
int speed2 = 11;
int direction1 = 12;
int direction2 = 13;

void stopMotor() {
  // turn both motors off
  analogWrite(speed1, 0);
  analogWrite(speed2, 0);
}

void setup()
{
  // set all the pins to outputs
  pinMode(speed1, OUTPUT);
  pinMode(speed2, OUTPUT);
  pinMode(direction1, OUTPUT);
  pinMode(direction2, OUTPUT);
}
```

```
void loop()
{
  // Both motors forward at 50% speed for 2 seconds
  digitalWrite(direction1, HIGH);
  digitalWrite(direction2, HIGH);
  analogWrite(speed1, HALF_SPEED);
  analogWrite(speed2, HALF_SPEED);
  delay(2000);

  stopMotor(); delay(1000); // stop

  // Left turn for 1 second
  digitalWrite(direction1, LOW);
  digitalWrite(direction2, HIGH);
  analogWrite(speed1, HALF_SPEED);
  analogWrite(speed2, HALF_SPEED);
  delay(1000);

  stopMotor(); delay(1000); // stop

  // Both motors forward at 50% speed for 2 seconds
  digitalWrite(direction1, HIGH);
  digitalWrite(direction2, HIGH);
  analogWrite(speed1, HALF_SPEED);
  analogWrite(speed2, HALF_SPEED);
  delay(2000);

  stopMotor(); delay(1000); // stop

  // rotate right at 25% speed
  digitalWrite(direction1, HIGH);
  digitalWrite(direction2, LOW);
  analogWrite(speed1, QUARTER_SPEED);
  analogWrite(speed2, QUARTER_SPEED);
  delay(2000);

  stopMotor(); delay(1000); // stop

}
```

Project 29 – Using a Motor Shield – Code Overview

First, we define the three speeds at which we would like the motor to run. You can choose any of these speeds within your program.

```
#define QUARTER_SPEED 64
#define HALF_SPEED 128
#define FULL_SPEED 255
```

Next, assign which pins control the speed and direction. On the Arduino Motor Shield, these are fixed at pins 3, 11, 12, and 13:

```
int speed1 = 3;
int speed2 = 11;
int direction1 = 12;
int direction2 = 13;
```

Next, create a function to turn the motors off. The motor is turned off four times in the code, so it makes sense to create a function to do this:

```
void stopMotor() {
  // turn both motors off
  analogWrite(speed1, 0);
  analogWrite(speed2, 0);
}
```

To turn the motors off, you just need to set the speed of each motor to zero. Therefore, you write a value of zero to the speed1 (left motor) and speed2 (right motor) pins.

In the setup loop, the four pins are set to output mode:

```
pinMode(speed1, OUTPUT);
pinMode(speed2, OUTPUT);
pinMode(direction1, OUTPUT);
pinMode(direction2, OUTPUT);
```

In the main loop, you execute four separate movement routines. First, you move the robot forward at 50 percent speed for two seconds:

```
digitalWrite(direction1, HIGH);
digitalWrite(direction2, HIGH);
analogWrite(speed1, HALF_SPEED);
analogWrite(speed2, HALF_SPEED);
delay(2000);
```

This is done by first setting the direction pins to HIGH, which equates to forward (presuming your motors are wired the correct way around). The speed pins of both motors are then set to HALF_SPEED which is 128. The PWM values range from 0 to 255, so 128 is halfway between those. Therefore, the duty cycle is 50 percent and the motors will run at half speed. Whatever direction and speed you set the motors to run at, they will continue that way until you change them. Therefore, the two-second delay will ensure the robot moves forward at 50 percent speed for two seconds, which should be about a meter or three feet.

Next, you stop the motors from turning and wait one second:

```
stopMotor(); delay(1000); // stop
```

The next movement sequence changes the direction of the left hand motor to reverse and the right side motor to forward. To make a motor go forward, set its direction pin to HIGH; to make it go in reverse, set it to LOW. The speed remains at 50 percent, so the value of 128 is written to the speed pins of both motors (the lines for forward motion on the right wheel and the speeds are not strictly necessary, but I have left them in so that you can modify them as you wish to get different movements).

Making the right hand wheel go forward and the left hand wheel move in reverse causes the robot to rotate anti-clockwise (turn left). This movement is for one second, which should make it turn left by about 90 degrees. After this sequence, the motor is stopped for one second again:

```
digitalWrite(direction1, LOW);
digitalWrite(direction2, HIGH);
analogWrite(speed1, HALF_SPEED);
analogWrite(speed2, HALF_SPEED);
delay(1000);
```

The next sequence makes the robot move forward again at 50 percent speed for two seconds, followed by a one second stop:

```
digitalWrite(direction1, HIGH);
digitalWrite(direction2, HIGH);
analogWrite(speed1, HALF_SPEED);
analogWrite(speed2, HALF_SPEED);
delay(2000);
```

The final movement sequence changes the direction of the wheels so that the left hand wheel drives forward and the right hand wheel drives backwards. This will make the robot rotate clockwise (turn right). The speed this time is set at 64 or quarter speed, which is 25 percent duty cycle, making it rotate slower than before. The rotation lasts for two seconds, which should be enough to turn the robot by 180 degrees.

```
digitalWrite(direction1, HIGH);
digitalWrite(direction2, LOW);
analogWrite(speed1, QUARTER_SPEED);
analogWrite(speed2, QUARTER_SPEED);
```

The four sequences together should make your robot go forward for two seconds, stop, turn left by 90 degrees, stop, go forward for two seconds again, stop, then turn at a slower rate by 180 degrees, and repeat from the start.

You may need to adjust the timings and speeds according to your own robot as you may be using a different voltage, faster or slower geared motors, etc. Play around with the movement sequences to get the robot to move quicker or slower, or turn at different rates as you desire.

The whole idea behind Project 29 is to get you used to the necessary commands to move the two motors in different directions and different speeds. If you just have motors but no robot base, attach some tape to the spindle of the motors so you can see the speed and direction they are turning.

Project 29 – Using a Motor Shield – Hardware Overview

The new piece of hardware used in this project is the motor shield. A shield is simply a circuit board already made up and conveniently sized to fit over the Arduino. The shield has pins to connect itself to the pins on the Arduino; these have female sockets on the top so you can still access the pins through the shield. Of course, depending on what type of shield you have, some of the Arduino pins will be used by the shield and will therefore be out of bounds for use in your code. The Ardumoto shield, for example, uses digital pins 10, 11, 12, and 13.

Shields have the benefit of giving you added functionality to your Arduino by simply plugging it in and uploading the necessary code. You can get all kinds of shields to extend the function of the Arduino to include such things as Ethernet access, GPS, relay control, SD and Micro SD Card access, LCD displays, TV connectivity, wireless communications, touch screens, dot-matrix displays, and DMX lighting control. See Figure 10-8 for examples of different kinds of shields.

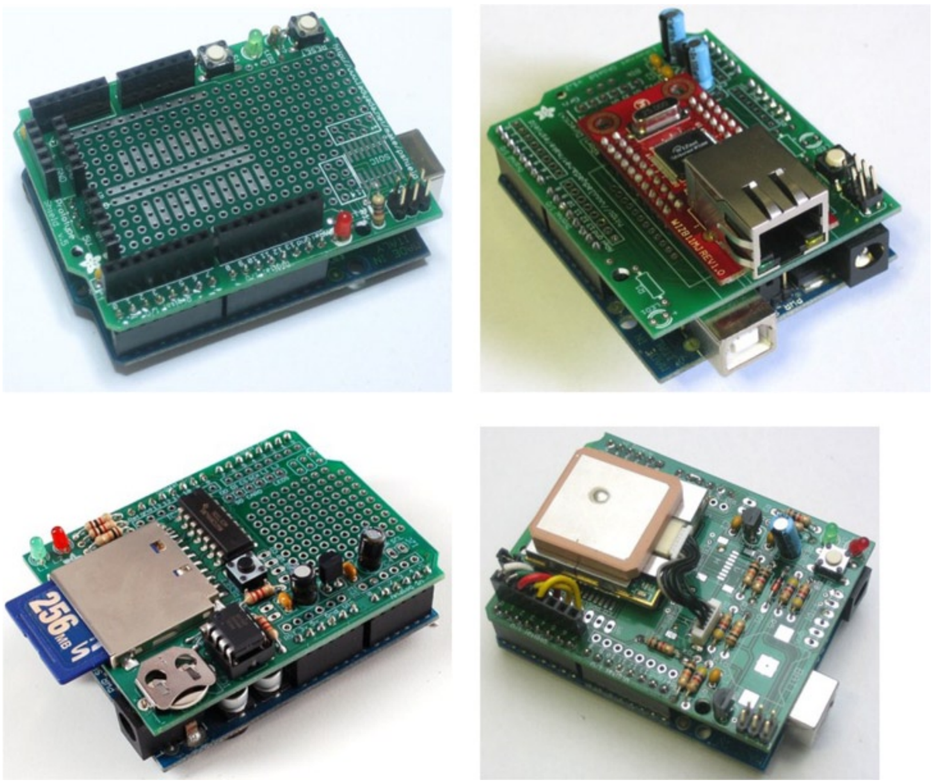


Figure 10-8. *A proto shield, ethernet shield, data logger shield, and GPS shield (courtesy of Adafruit Industries at* `www.adafruit.com`*)*

Project 30 – Line-Following Robot

Now that you know how to control the two motors or your robot base using a motor shield, you are going to put those skills to good use! In this project, you are going to build a robot that is capable of detecting and following a black line drawn on a floor. This kind of robot is still used today in factories to ensure the robot keeps to a set path across the factory floor. They are known as Automated Guided Vehicles (AGVs).

Parts Required

For this project, you will need a small lightweight robot base. These can be purchased ready-made or in kit form. Or you can build one from scratch (much more fun). Make sure the wheels are geared so that they rotate much slower than the motor because the robot must be able to move fairly slowly to enable it to navigate the line.

As the robot will be autonomous, it will need its own battery pack. I found that a six-AA battery holder provided enough power for the Arduino, sensors, LEDs, shield, and motors. Larger batteries will give you a longer life but at the expense of greater weight, so it will depend on the load bearing capacity of your robot. See Table 10-4 for a list of parts.

Table 10-4. *Parts Listing for Project 30*

Motor Shield	
4 × Current-Limiting Resistors	
3 × 1KΩ Resistors	
4 × White LEDs	
3 × Light-Dependent Resistors	
2 × DC Motors or...	
.... a 2 wheeled robot base	
Battery Pack	

Connect It Up

Connect the circuit up as in Figure 10-9.

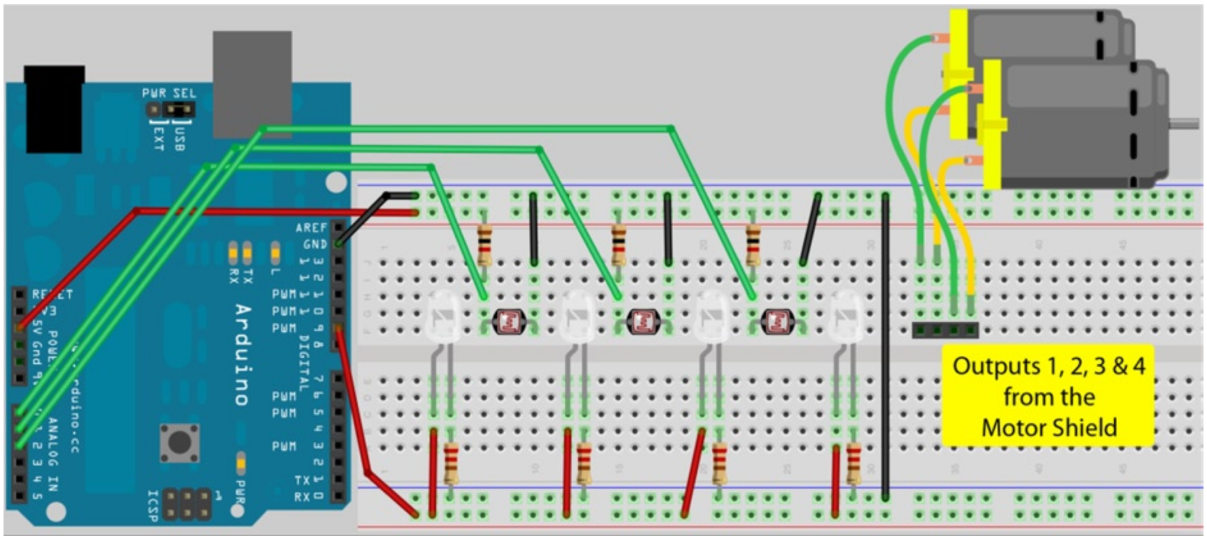


Figure 10-9. *The circuit for Project 30 – Line-Following Robot (see insert for color version)*

The shield is not shown, but the four outputs simply go to Motor A and Motor B.

The four LEDs can be any color as long as they have a reasonable enough brightness to illuminate the floor and create contrast between the floor and the black line for your sensor to detect. White LEDs are best as they output the greatest brightness for the lowest current. Four of these are connected in parallel with pin 9. Make sure that you use the appropriate current-limiting resistors for your LEDs.

Finally, connect up three light-dependent resistors to analog pins 0, 1, and 2. One side of the LDR goes to ground, the other to +5v via a 1K ohm resistor. The wire from the analog pins goes between the resistor and the LDR pin (you used LDRs in Project 14).

The breadboard above can be used for testing purposes. However, for making the robot, you will need to make up a sensor bar that houses the four LEDs and the three LDRs. These must be positioned so that they are all close together and pointing in the same direction. I soldered the LED and sensor circuit onto a small piece of stripboard and placed the three LDRs in-between the four LEDs and just slightly higher than the LEDs so that the light from them did not shine onto the sensor. Figure 10-10 shows how I did it.

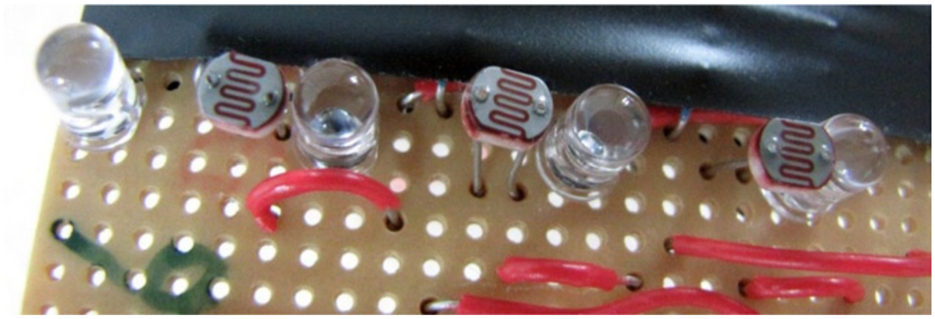


Figure 10-10. *The sensor bar consisting of four LEDs and three LDRs*

The sensor bar is then attached to the front side of your robot in such a way that the LEDs and LDRs are about 1 cm or half an inch above the floor (see Figure 10-11). I simply taped mine to the front of my robot, but for a more permanent solution, the sensor bar could be bolted to the chassis.

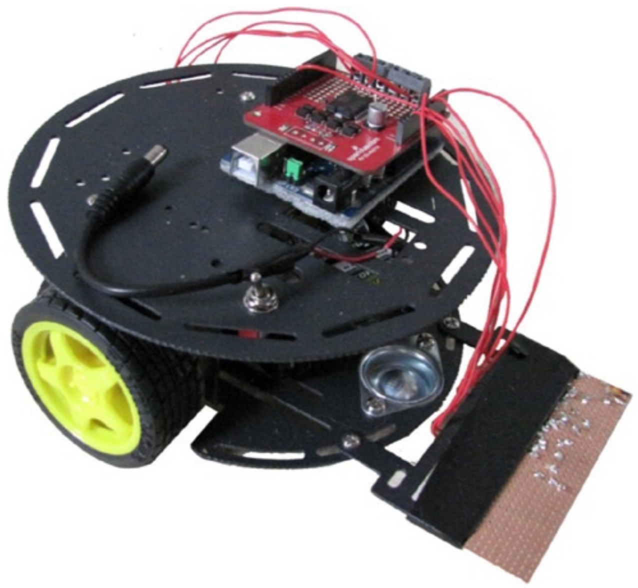


Figure 10-11. *The robot base fitted with the sensor bar*

The battery pack should be placed centrally inside the robot base so as to not upset the load balance between the wheels. Fix the pack to the chassis with some Velcro so it cannot move around as the robot navigates the course.

As you can see from Figure 10-11, wires runs between the motor shield and the two motors and also down to the sensor bar. One wire provides +5v, one is for ground, one is for +5v from digital pin 9, and the other three go to analog sensors 0, 1 and 2, with 0 being the left hand sensor (as you face the front of the robot), 1 being the center sensor, and 2 being the right hand one. It is critical that you get the order correct or the robot will not work as expected. See Figure 10-12 for an image of my robot in action on my kitchen floor (there is also a video of it in action on YouTube and Vimeo. Look for "Arduino Controlled Line-Following Robot" by Mike McRoberts).

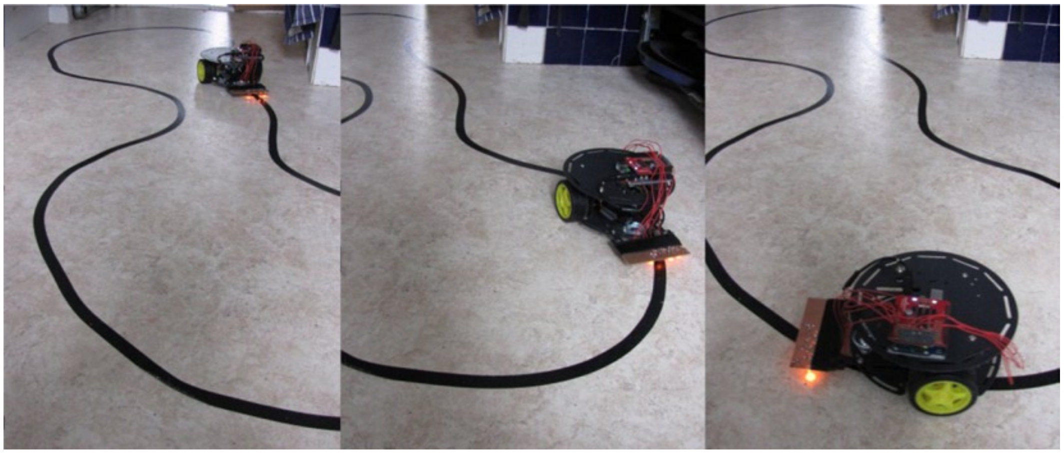


Figure 10-12. *My own line-following robot in action*

Enter the Code

When you have built your robot and everything has been checked, enter the code from Listing 10-3.

Listing 10-3. Code for Project 30

```
// Project 30 - Line Following Robot

int LDR1, LDR2, LDR3;              // sensor values

// calibration offsets
int leftOffset = 0, rightOffset = 0, center = 0;
// pins for motor speed and direction
int speed1 = 10, speed2 = 11, direction1 = 12, direction2 = 13;
// starting speed and rotation offset
int startSpeed = 70, rotate = 30;
// sensor threshold
int threshhold = 5;
int left = startSpeed, right = startSpeed;

int button = 2;                    // button pin (optional)
int motorsOff = true;              // motors initially off until button pressed.
                                   // change the initial value to false if you
                                   // don't have a pushbutton on your robot

// Sensor calibration routine
void calibrate() {

  for (int x=0; x<10; x++) {       // run this 10 times to obtain average
  digitalWrite(9, HIGH);           // lights on
  delay(100);
  LDR1 = analogRead(0);            // read the 3 sensors
  LDR2 = analogRead(1);
  LDR3 = analogRead(2);
  leftOffset = leftOffset + LDR1;  // add value of left sensor to total
  center = center + LDR2;          // add value of center sensor to total
  rightOffset = rightOffset + LDR3; // add value of right sensor to total

  delay(100);
  digitalWrite(9, LOW);            // lights off
  delay(100);
  }
  // obtain average for each sensor
  leftOffset = leftOffset / 10;
  rightOffset = rightOffset / 10;
  center = center /10;
  // calculate offsets for left and right sensors
  leftOffset = center - leftOffset;
  rightOffset = center - rightOffset;

}
```

```
void setup()
{
    pinMode(button, INPUT_PULLUP); // button on pin 2 (optional)
    // set the motor pins to outputs
    pinMode(9, OUTPUT);
    pinMode(speed1, OUTPUT);
    pinMode(speed2, OUTPUT);
    pinMode(direction1, OUTPUT);
    pinMode(direction2, OUTPUT);
    // calibrate the sensors
    calibrate();
    delay(3000);

    digitalWrite(9, HIGH);         // lights on
    delay(100);

    // set motor direction to forward
    digitalWrite(direction1, HIGH);
    digitalWrite(direction2, HIGH);
    // set speed of both motors
    analogWrite(speed1,left);
    analogWrite(speed2,right);
}

void loop() {
    if ( !digitalRead(button) ){
     // button is pressed. Toggle motors on or off.
     motorsOff = !motorsOff;
     if (motorsOff) {
        analogWrite(speed1, 0);    // turn motors off by setting speed to zero
        analogWrite(speed2, 0);
     }
     delay(500);                   // give time for human to let go of the buttton,
                                   // and to ignore any switch bounce.
  }

  // make both motors same speed
  left = startSpeed;
  right = startSpeed;

  // read the sensors and add the offsets
  LDR1 = analogRead(0) + leftOffset;
  LDR2 = analogRead(1);
  LDR3 = analogRead(2) + rightOffset;

  // if LDR1 is greater than the centre sensor + threshold turn right
  if (LDR1 > (LDR2+threshhold)) {
    left = startSpeed + rotate;
    right = startSpeed - rotate;
  }
  // if LDR3 is greater than the centre sensor + threshold turn left
  if (LDR3 > (LDR2+threshhold)) {
    left = startSpeed - rotate;
    right = startSpeed + rotate;
  }

  // send the speed values to the motors
  if (!motorsOff) {
     analogWrite(speed1, left);
     analogWrite(speed2,right);
  }

}
```

Lay out a course for your robot on a flat surface. I used the linoleum on my kitchen floor and made a course using black electrical tape. Turn the robot on, making sure it is sitting on a blank piece of floor next to, but not over, the back line. The calibration routine will run, making the LEDs flash ten times in quick succession. Once this stops, you have three seconds to pick up the robot and place it on the line. If all is well, the robot will now happily follow the line. Don't make the turns too sharp, as this rapid transition will not be detected with only three LDRs. See Figure 10-12 for an example of a course made with black electrical tape. Code is included for an optional button on pin 2. This will allow you to turn the power to the motors on and off to enable you to adjust values in your code to get the robot to work correctly.

Project 30 – Line-Following Robot – Code Overview

First, you define the pin for the lights, and then three integers are declared that will hold the values read from the three light-dependent resistors:

```
#define lights 9
int LDR1, LDR2, LDR3; // sensor values
```

Then another three integers are declared that will hold the offset values for the three sensors calculated in the calibration routine (this will be explained later):

```
int leftOffset = 0, rightOffset = 0, center = 0;
```

Next, you define the pins that will control the speed and direction of the two motors:

```
int speed1 = 3, speed2 = 11, direction1 = 12, direction2 = 13;
```

Then two integers are created to hold the initial speed of the motors and the speed offset for each wheel that you add or subtract to make the robot rotate:

```
int startSpeed = 70, rotate = 30;
```

The default speed is set to 70, which is around 27 percent duty cycle. You may need to adjust this value to suit your own robot. Too high a value will make the robot overshoot the line and too low will prevent the motors from turning fast enough to turn the wheels to overcome friction.

The `rotate` value is how much you will speed up or slow down the wheels to cause the robot to turn. In my case, the required value is 30. So when turning left, for example, the right wheel spins at speed 100 and the left wheel at a speed of 40 (70+30 and 70-30). The rotate value is another setting you may need to adjust for your own setup.

Another integer is created to hold the sensor threshold:

```
int threshhold = 5;
```

This is the difference in values required between the center sensor and the left or right sensors before the robot decides to turn. In my case, a setting of 5 works well. This means that the left and right sensors would need to detect a value greater than the value read from the center sensor plus the threshold value before action is taken. In other words, if the center sensor is reading a value of 600 and the left sensor is reading 603, then the robot will keep going straight. However, a left sensor value of 612 (which is higher than the center value plus threshold) means that the left sensor is detecting the back line, indicating that the robot is too far over to the left. So the motors would adjust to make the robot turn to the right to compensate.

The threshold value will vary depending on the contrast between your floor (or whatever surface you use) and the black line. This may need to be adjusted to ensure the robot only turns when it has detected enough of a difference between floor and line to ascertain it had moved too far left or right.

The final set of variables will store the speed values for the left and right motors. These are initially set to the value in `startSpeed`:

```
int left = startSpeed, right = startSpeed;
```

Next is an optional two lines of code if you have included a pushbutton to your circuit going to input pin 2. If you don't have a pushbutton on your circuit then change `motorsOff` to false.

```
int button = 2; // button pin (optional)
int motorsOff = true;
```

After the variables are all declared and initialized, you come to your first and only function, which is the calibration routine:

```
void calibrate() {
```

The purpose of this routine is two-fold. First, it obtains an average value from each of the three sensors over a sequence of ten reads. Second, it flashes the lights 10 times (while reading values from the three sensors) to show you that the robot is calibrating and nearly ready to run.

The sensors require calibration, as each one will read different values to the next one. Every LDR will give a slightly different reading and this will be affected by manufacturing tolerances, the tolerance of the voltage divider resistors used, and the resistance in the wires. You want all three sensors to read (roughly) the same value, so you take ten readings from each, average them out, and calculate the difference that the left and right sensors are from the center sensor (which is used as the baseline).

You start off with creating a `for` loop that runs ten times:

```
for (int x=0; x<10; x++) {
```

The LEDs attached to pin 9 are turned on, followed by a delay of 100 milliseconds:

```
digitalWrite(9, HIGH);
delay(100);
```

Next, the values from all three sensors are read in and stored in LDR1, LDR2, and LDR3:

```
LDR1 = analogRead(0);
LDR2 = analogRead(1);
LDR3 = analogRead(2);
```

Now you take those values and add them to the leftOffset, center, and rightOffset variables. These variables start off at zero, so after ten iterations they will contain a running total of all ten values read from the sensors.

```
leftOffset = leftOffset + LDR1;
center = center + LDR2;
rightOffset = rightOffset + LDR3;
```

Then you wait 100 milliseconds, turn the light off, wait another 100 milliseconds, and then repeat the process:

```
delay(100);
digitalWrite(9, LOW); // lights off
delay(100);
```

After this process has repeated ten times, you exit the for loop and then divide each of the running totals by ten. This gives you an average sensor reading for each of the three LDRs.

```
leftOffset = leftOffset / 10;
rightOffset = rightOffset / 10;
center = center /10;
```

You then calculate what the offset will be by deducting the left and right sensor values by the center one:

```
leftOffset = center - leftOffset;
rightOffset = center - rightOffset;
```

These values will be added to the sensor readings from the left and right sensors so that all three sensors will be giving approximately the same readings. This will make ascertaining the difference between the three readings a lot easier when detecting the difference between the floor and the black line.

Next, you have the setup routine, which starts off by setting the pin for the button, LEDs, and the pins for the motor speed and direction to OUTPUT. The mode for the button is INPUT_PULLUP. The chip on the Arduino has internal pull-up resistors that you can use instead of external pull-down resistors:

```
pinMode(button, INPUT_PULLUP);
pinMode(9, OUTPUT); // lights
pinMode(speed1, OUTPUT);
pinMode(speed2, OUTPUT);
pinMode(direction1, OUTPUT);
pinMode(direction2, OUTPUT);
```

The calibration routine is now run to ensure all three sensors issue similar readings, followed by a delay of three seconds. After the LEDs flash ten times during the calibration routine, you have three seconds to place the robot on the line it is to follow:

```
calibrate();
delay(3000);
```

The lights are turned on, followed by a brief delay:

```
digitalWrite(9, HIGH); // lights on
delay(100);
```

The direction of both motors is set to forward and the speeds are set to the values stored in the right and left variables (initially set to the value stored in startSpeed):

```
digitalWrite(direction1, HIGH);
digitalWrite(direction2, HIGH);
analogWrite(speed1,left);
analogWrite(speed2,right);
```

Now you move onto the main loop of the program. This starts with checking whether the optional button on pin 2 is pressed or not, and toggling the motor on or off accordingly. The button will allow you to turn the robot off to make adjustments to your code and to upload it.

```
if ( !digitalRead(button) ){
   // button is pressed. Toggle motors on or off.
   motorsOff = !motorsOff;
   if (motorsOff) {
      analogWrite(speed1, 0);  // turn motors off by setting speed to zero
      analogWrite(speed2, 0);
   }
   delay(500);  // give time for human to let go of the buttton,
                       // and to ignore any switch bounce.
}
```

Next, set the speed of the left and right motors to the value in startSpeed:

```
left = startSpeed;
right = startSpeed;
```

The motor speed is reset to these values at the start of each loop so that the robot is always going forward, unless the values are changed later in the loop to make the robot turn.

You now read the sensor values from each of the three LDRs and store the values in the LDR1, LDR2, and LDR3 integers. The offsets calculated for the left and right sensors are added to the value so that when all three sensors are looking at the same surface they will read approximately the same values.

```
LDR1 = analogRead(0) + leftOffset;
LDR2 = analogRead(1);
LDR3 = analogRead(2) + rightOffset;
```

Now you need to check those sensor values and see if the black line has moved too far from the center. You do this by checking the left and right sensors and seeing if the values read are greater than the value read from the center LDR, plus the threshold offset.

If the value from the left sensor is greater than the reading-plus-threshold-offset, then the black line has shifted from the centerline and is too far to the right. The motor speeds are then adjusted so that the left wheel spins faster than the right, thus turning the robot to the right, which will bring the black line back to the centre.

```
if (LDR1 > (LDR2+threshhold)) {
    left = startSpeed + rotate;
    right = startSpeed - rotate;
  }
```

The same is done for the right hand sensor, this time turning the robot to the left:

```
if (LDR3 > (LDR2+threshhold)) {
  left = startSpeed - rotate;
  right = startSpeed + rotate;
}
```

The speeds, which may or may not have been adjusted if the line was off center, are then sent out to the motors, if the motor is currently set to be on by the optional pushbutton:

```
if (!motorsOff) {
   analogWrite(speed1, left);
   analogWrite(speed2,right);

}
```

This whole process is repeated over and over many times per second and forms a feedback loop which makes the robot follow the line.

If your robot runs off the line, you may need to adjust the speed in `startSpeed` to a slower one. If it doesn't turn fast or far enough or if it turns too much, then the `rotate` value needs to be adjusted accordingly. The sensitivity of the LDRs can be adjusted by changing the value in the `threshold` variable. Play around with these values until you get the robot to follow the line successfully.

Summary

Chapter 10 started off with a simple introduction to stepper motors and how to control their speed and direction using a motor drive IC. You also learned the difference between a bipolar and a unipolar stepper motor and how they work. Then you moved onto using a simple motor shield to control two motors independently and ended up with connecting that shield to a robot base to make a two-wheeled line-following robot.

You have also learned about shields and their different uses, and you've put a motor shied to a practical application. Knowing how to control stepper motors and standard motors, as well as servos (Chapter 9), means you are well on your way to making your own advanced robot or animatronics device!

Subjects and Concepts covered in Chapter 10 include:

- The difference between unipolar and bipolar stepper motors
- How to connect a stepper to a motor driver IC
- Using capacitors to smooth a signal and reduce interference
- How to set up a stepper object in your code
- Setting the speed and number of steps of the motor
- Controlling the direction of the motor with positive or negative numbers
- How a stepper motor works
- The coil energizing sequence required to drive a stepper motor
- That resolution can be increased using half-stepping and microstepping
- How to use a motor shield

- Using a motor shield to control the speed and/or direction of several motors
- What an Arduino shield is and the different kinds available
- How to detect contrast variations in light using several LDRs
- How to obtain an average sensor reading
- Using offset values to balance sensor readings
- How to make an awesome line-following robot

CHAPTER 11

Pressure Sensors

Now you are going to take a look at pressure sensors, in particular, the MPL3115A2 digital pressure sensor from Freescale. This is a great sensor that is easily interfaced with an Arduino and provides accurate pressure and temperature readings. The device needs to be connected to the Arduino's I2C (inter-integrated circuit or I-Squared-C, also known as a two-wire interface (TWI)) bus for data to be exchanged. I2C is a new concept for you, and although this chapter will not cover it in great detail, you will learn the basic concepts of the I2C and how to use it to get data from the MPL3115A2 sensor.

Digital pressure sensors are ideal for making your own weather station. You'll start off in this chapter by learning how to interface the MPL3115A2 to the Arduino and read data from it using the serial monitor. You will then use the sensor and an LCD to display a pressure graph over a 24-hour period. Along the way, you will also learn how to control a GLCD (graphic LCD) display.

Project 31 – Digital Pressure Sensor

In this project we are going to learn how to use an MPL3115A2 pressure sensor. Then, in Project 32, we will use our knowledge of how to use this sensor to create a graph of pressure over time on a graphical LCD.

Parts Required

The tiny MPL3115A2 sensor can be purchased from Sparkfun or their distributors pre-soldered to a breakout board (see Figure 11-1) to make it easy to interface with an Arduino (or other microcontroller). You will need to solder some header pins onto the board if you wish to push it into a breadboard. Otherwise, solder some wires to it so it can be connected to the Arduino.

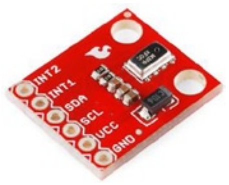


Figure 11-1. *The MPL3115A2 on a Sparkfun breakout board*

Table 11-1. *Parts Required for Project 32*

Arduino Mega or Uno	
MPL3115A2 Pressure Sensor	

The MPL3115A2 is a great sensor. It is able to calculate altitude to within 30 cm (1 ft.) and also has a 12-bit temperature sensor on board.

Connect It Up

Connect everything as shown in Figure 11-2. I have used an Arduino Mega instead of an Uno for these projects as it has extra pins free for adding additional devices, if you wish. If you do use an Arduino Uno for your project, the SDA and SCL signals are on the analog-in 4 and analog-in 5 pins, respectively. This is different from the Arduino Mega, which uses pins 20 and 21 instead.

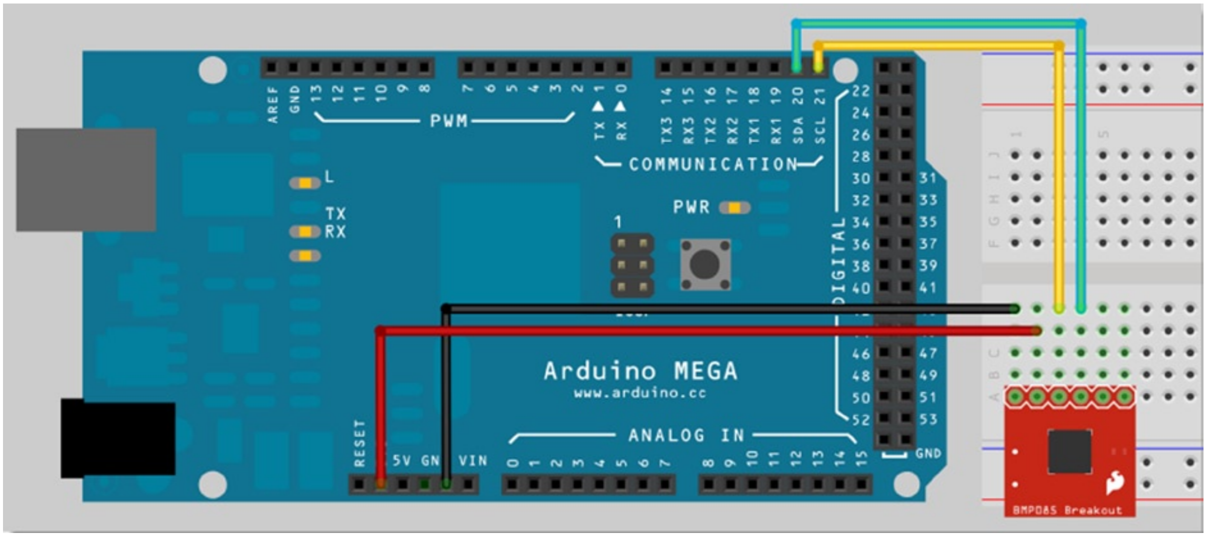


Figure 11-2. *The circuit for Project 31 – Digital Pressure Sensor*

This project simply requires the Uno, the MPL3115A2 on a breakout board, and a few wires. Connect the GND pin on the sensor to the GND pin on the Arduino. Connect the VCC pin on the sensor to the 3.3V pin on the Uno. Then connect the SDA (serial data) and SCL (serial clock) pins on the Uno to the corresponding SDA and SCL pins on the sensor.

Be aware that the wire library turns on the internal pull-ups to 5V and the MPL3115A2 requires a maximum voltage of 3.6 volts on the SDA and SCL pins. The Sparkfun breakout board has pull-up resistors to bring that down to a safe 3.3V. If the MPL3115A2 is the only device on the I2C bus, the pull-ups to 3.3V on the breakout board will keep the signal levels within safe range. If you add any other devices to the I2C bus that operate at 5V and have on-board pull-ups like the MPL3115A2 breakout board does, you could easily damage the MPL3115A2 and any other 3.3V-only I2C devices on the bus. Therefore, do not add any other I2C devices to this project without also providing some protection.

One way to avoid these problems is to clamp the SCL and SDA lines using low-capacitance 3.6-volt zener diodes to ground (3.6V because of the nonlinear knee on the zener curve). Another way is to use LEDs with a 3.1 – 3.4-volt forward voltage to clamp the lines. (Most blue and white LEDs have a forward voltage in that range.)

Enter the Code

Enter the code from Listing 11-1.

Listing 11-1. Code for Project 31

```
// Based on the One Shot example by Henry Lahr

#include <Wire.h> // so we can use I2C communication
#define MYALTITUDE 262 //define altitude at your location to calculate mean sea level pressure in meters

// Register addresses
const int SENSORADDRESS = 0x60; // MPL3115A1 address from the datasheet
#define SENSOR_CONTROL_REG_1 0x26
#define SENSOR_DR_STATUS 0x00  // Address of DataReady status register
#define SENSOR_OUT_P_MSB 0x01  // Starting address of Pressure Data registers

float baroAltitudeCorrectionFactor = 1/(pow(1-MYALTITUDE/44330.77,5.255877));

byte I2Cdata[5] = {0,0,0,0,0}; //buffer for sensor data

void setup(){
  Wire.begin(); // join i2c bus
  Serial.begin(9600); // start serial for output at 9600 baud
  Serial.println("Setup");
  I2C_Write(SENSOR_CONTROL_REG_1, 0b00000000); // put in standby mode
  // these upper bits of the control register
  // can only be changed while in standby
  I2C_Write(SENSOR_CONTROL_REG_1, 0b00111000); // set oversampling to 128
  Serial.println("Done.");
}

void loop(){
  float temperature, pressure, baroPressure;

  Read_Sensor_Data();
  temperature = Calc_Temperature();
  pressure = Calc_Pressure();
  baroPressure = pressure * baroAltitudeCorrectionFactor;
  Serial.print("Absolute pressure: ");
  Serial.print(pressure); // in Pascal
  Serial.print(" Pa,    Barometer: ");
  Serial.print(baroPressure); // in Pascal
  Serial.print(" Pa,    Temperature: ");
  Serial.print(temperature); // in degrees C
  Serial.println(" C");
  delay(1000);
}

  // Read the pressure and temperature readings from the sensor
void Read_Sensor_Data(){

  // request a single measurement from the sensor
  I2C_Write(SENSOR_CONTROL_REG_1, 0b00111010); //bit 1 is one shot mode

  // Wait for measurement to complete.
  // One-shot bit will clear when it is done.
  // Rread the current (sensor control) register
  // repeat until sensor clears OST bit
  do {
    Wire.requestFrom(SENSORADDRESS,1);
  } while ((Wire.read() & 0b00000010) != 0);

  I2C_ReadData(); //reads registers from the sensor
}

  // This function assembles the pressure reading
  // from the values in the read buffer
  // The two lowest bits are fractional so divide by 4
float Calc_Pressure(){
  unsigned long m_pressure = I2Cdata[0];
  unsigned long c_pressure = I2Cdata[1];
  float l_pressure = (float)(I2Cdata[2]>>4)/4;
  return((float)(m_pressure<<10 | c_pressure<<2)+l_pressure);
}

// This function assembles the temperature reading
// from the values in the read buffer
float Calc_Temperature(){
  int m_temp;
  float l_temp;
  m_temp = I2Cdata[3]; //temperature in whole degrees C
  l_temp = (float)(I2Cdata[4]>>4)/16.0; //fractional portion of temperature
  return((float)(m_temp + l_temp));
}

// Read Barometer and Temperature data (5 bytes)
void I2C_ReadData(){
  byte readUnsuccessful;
  do {
    byte i=0;
    byte dataStatus = 0;

    Wire.beginTransmission(SENSORADDRESS);
    Wire.write(SENSOR_OUT_P_MSB);
    Wire.endTransmission(false);
    // read 5 bytes. 3 for pressure, 2 for temperature.
    Wire.requestFrom(SENSORADDRESS,5);
    while(Wire.available()) I2Cdata[i++] = Wire.read();

    // in some modes it is possible for the sensor
    // to update the pressure reading
    // while we were in the middle of reading it,
    // in which case our copy is garbage
    // (parts of two different readings)
    // We can check bits in the DR (data ready)
    // register to see if this happened.

    Wire.beginTransmission(SENSORADDRESS);
    Wire.write(SENSOR_DR_STATUS);
    Wire.endTransmission(false);

    Wire.requestFrom(SENSORADDRESS,1); //read 5 bytes. 3 for pressure, 2 for temperature.
    dataStatus = Wire.read();
    readUnsuccessful = (dataStatus & 0x60) != 0;
    // This will be unsuccessful if overwrite happened
    // while we were reading the pressure or temp data.
    // So keep reading until we get a successful clean read
  } while (readUnsuccessful);
}

// This function writes one byte over I2C
void I2C_Write(byte regAddr, byte value){
  Wire.beginTransmission(SENSORADDRESS);
  Wire.write(regAddr);
  Wire.write(value);
  Wire.endTransmission(true);
}
```

After you have uploaded the code, open up the serial monitor window and ensure that your baud rate is set to 9600. You will see a stream of data from the sensor showing the pressure in Pa (Pascals) and the temperature in Celsius. Pascals are the unit of pressure commonly used in weather forecasts.

Project 31 – Digital Pressure Sensor – Code Overview

First we include the wire library. This library allows us to communicate with I2C / TWI devices.

```
#include <Wire.h>
```

Next we define MYALTITUDE as our altitude in meters. You will need to find this out using a GPS unit or from your local weather station. My altitude is 262 meters, but change it to suit your own location.

```
#define MYALTITUDE 262
```

According to the datasheet for the MPL3115A2, the address of the I2C device is 0x60 or 60 in hexadecimal, which is 196 in decimal. Each I2C device will have its own address.

```
const int SENSORADDRESS = 0x60;
```

Then we need to define the three registers within the sensor that we will address later in the code.

```
#define SENSOR_CONTROL_REG_1 0x26
#define SENSOR_DR_STATUS 0x00
#define SENSOR_OUT_P_MSB 0x01
```

To be able to calculate an accurate pressure reading calibrated to your altitude we will need to generate a correction factor that will be used later in the code to calibrate the reading. So, a variable called baroAltitudeCorrectionfactor is created here using your altitude to work out the correction factor. To work out the pressure at sea level based on our altitude we use the calculation in the datasheet

$$P = P_0 \cdot \left(1 - \frac{h}{44330.77}\right)^{5.255876}$$

Where P0 is the average sea level pressure of 101325 Pa (Pascals) and h is our altitude in meters. In code this translates to

```
float baroAltitudeCorrectionFactor = 1/(pow(1-MYALTITUDE/44330.77,5.255877));
```

The data from the sensor for the pressure takes up three bytes and the temperature takes up two bytes. We therefore create an array of five bytes in length to store the pressure and temperature readings from the sensor.

```
byte I2Cdata[5] = {0,0,0,0,0};
```

Next we go into the setup function and start communicating with the sensor. However, before we do that, let us take a brief look at the I2C bus hardware and how it works.

I2C Bus

I2C (sometimes referred to as I-squared-C) is a serial computer bus invented by Philips Semiconductors and is used for attaching low-speed peripherals to electronic devices. I2C uses two bi-directional lines, a serial data line (SDA) and a serial clock (SCL) pulled up with resistors. The I2C bus has a 7-bit or a 10-bit (depending on the device used) address space. Devices on the I2C bus comprise of a master device and one or more slave devices. These are all connected on the same data and clock lines as in Figure 11-3.

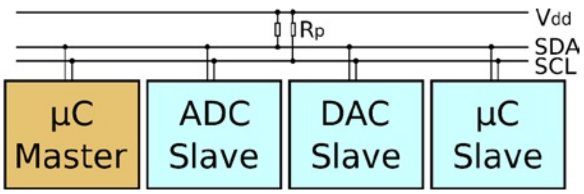


Figure 11-3. *An I2C sample schedmatic (image by Colin M.L. Burnett)*

The master node is the device that generates the clock signal and initiates communication with the slave devices. The slave node is the device that receives the clock signal and responds when addressed by the master.

It is possible to have multiple master devices. Additionally, master and slave devices can swap roles between messages (after a STOP command is sent). The slave devices have a 7-bit address and these will each be different.

Inside each I2C device are registers, which can be read or written to in order to control the device or change its configuration. Imagine the address of the device as being a building number and the internal registers as being apartment numbers. To read or write to a register you will need the address of the device (building number) and its register address (apartment number).

For further information, search on YouTube for "What is the I2C Bus? An Introduction from NXP." Now let's get back to our code. Next is the setup() function.

```
void setup(){
```

Now we need to initiate communication with the I2C device. We use the Wire.Begin() command to start this. This joins the I2C bus. I2C devices can be either master or slave devices. The master device generates the clock signal and communicates with the slaves. The slave device receives the clock signal from the master and responds when addressed by the master. In our case, the Arduino is the master and the MPL3115A2 is the slave.

```
Wire.begin();
```

We will be using the serial monitor window to view the data output from the sensor so we begin serial communications at 9600 baud.

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

Tell the user, via the serial monitor, that the program is running and the device is being set up.

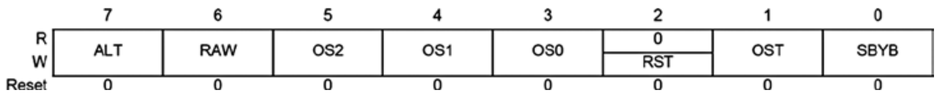
```
Serial.println("Setup");
```

Next, we need to set the sensor to Standby Mode and to 128x oversampling rate.

```
I2C_Write(SENSOR_CONTROL_REG_1, 0b00000000);
I2C_Write(SENSOR_CONTROL_REG_1, 0b00111000);
```

The sensor can either give us a data reading when we ask it to or be set to output data at a set interval between 1 second and 2^{15} seconds (equal to 32,768 seconds or just over 9 hours). For our purposes it is easier to read the data simply whenever we need it so we will use the OST mode. Oversampling is the process of sampling a signal numerous times and then obtaining the average of those signals. This helps reduce the signal-to-noise ratio of the device and give us a more accurate reading. 128x oversampling is the maximum the device will allow, so we will use that. This means 128 readings are taken and averaged out. The device then gives us that averaged reading.

Address 0x26 (SENSOR_CONTROL_REG_1) is Control Register 1 (CTRL-REG1) inside the device. This is simply an address space that allows us to change settings on the sensor by changing the value stored at that address. Each individual bit of CTRL-REG1 has a different purpose. According to the datasheet the bits are shown in Figure 12-4.



	7	6	5	4	3	2	1	0
R	ALT	RAW	OS2	OS1	OS0	0	OST	SBYB
W						RST		
Reset	0	0	0	0	0	0	0	0

Figure 12-4. *Individual bits of CTRL-REG1*

The first bit is the Standby Bit (SBYB). When set to 0, the device goes into STANDBY mode. When set to 1, it is in ACTIVE mode. If you look at the code, the first value we write to CTRL-REG1 has the SBYB bit (the far right bit) set to 0 and hence we are setting the device to standby mode. The next bit is the One Shot (OST) mode bit. Again, the first value we send to the register will set this bit to 0. This clears the One Shot Mode bit. The OST bit must be set to 0 to clear it so we can take another reading next time around. The second value we will write to CTRL_REG1 clears the

OST bit (sets it to 0), as you can see below. The next bit is the RST or software reset bit. We are not using this so leave it at 0. The next 3 bits are a 3-bit number that sets the oversampling rate. The oversampling rate goes from 1 when set to 000 bits through 2, 4, 8, 16, 32, 64, and finally, 128 when set to 111. The next two bits are not used and so left at 0.

So, with the information we have above, we can see that first we write a value to the CTRL-REG1 that sets the sensor to STANDBY mode and also clears the One Shot mode with the oversampling rate set to 128x.

We then notify the user that we have finished setting up the device.

```
Serial.println("Done.");
```

Next comes the main loop function of our sketch, which simply reads the three bytes from the device that make up the pressure reading plus the two bytes for the temperature reading. It converts these bytes into floating point numbers, and then displays the floating point values in the serial monitor window. First, we create three variables to store the temperature, pressure and baroPressure (the pressure corrected for altitude).

```
float temperature, pressure, baroPressure;
```

Next we call a function called Read_Sensor_Data() to obtain the readings from the device.

```
Read_Sensor_Data();
```

And then call two more functions, which will convert the three bits that store the pressure into a decimal figure and the two bytes that make up the temperature reading.

```
temperature = Calc_Temperature();
pressure = Calc_Pressure();
```

We then take the pressure reading and adjust it for our altitude and store this in the baroPressure variable.

```
baroPressure = pressure * baroAltitudeCorrectionFactor;
```

Then we use a series of Serial.print commands to display the absolute pressure, the adjusted pressure, and the temperature.

```
Serial.print("Absolute pressure: ");
Serial.print(pressure); // in Pascal
Serial.print(" Pa,    Barometer: ");
Serial.print(baroPressure); // in Pascal
Serial.print(" Pa,    Temperature: ");

Serial.print(temperature); // in degrees C
Serial.println(" C");
```

So as not to flood the serial monitor with readings, we take a reading and display it every second and so perform a 1000ms delay next.

```
delay(1000);
```

We declare our next function to read the sensor data.

```
void Read_Sensor_Data(){
```

Then we set bit 1 (One Shot Mode) to 1, which will request a single pressure and temperature reading from the sensor.

```
I2C_Write(SENSOR_CONTROL_REG_1, 0b00111010);
```

Then we use a do-while loop to request a single byte from the device using the Wire.requestFrom() function supplying the device address and the number of bytes to retrieve.

```
do {
  Wire.requestFrom(SENSORADDRESS,1);
```

The loop will keep reading a byte while there is data available and until the device clears the OST bit.

```
} while ((Wire.read() & 0b00000010) != 0);
```

The above will fill up the five registers inside the device that store the pressure (three bytes) and the temperature (two bytes) with the current readings from the sensor. We now call the I2C_ReadData() function that will read those five bytes from the device and store them in ourI2Cdata[] array.

```
I2C_ReadData(); //reads registers from the sensor
```

Now, let us look at the three functions called by the Read-Sensor_Data function. The first function is the one that takes the three bytes that make up the pressure reading and convert it to a floating-point number.

```
float Calc_Pressure(){
```

The integer (whole number) part of the pressure reading is stored in 18 bits, with two bits being the fractional part of the pressure reading. The first 16 bits are stored in the MSB and CSB (I2Cdata[0] and [1]) Then bits 7 and 6 of the LSB (I2Cdata[2]) make up the final two bits of the 18-bit number. Bits 5 and 4 of the LSB make up the fractional part of the pressure reading.

The most significant byte of the pressure reading is stored in the first element of the I2Cdata array so we store this in the m_pressure variable.

```
unsigned long m_pressure = I2Cdata[0];
```

The central byte of the three bytes that make up the pressure reading is stored in the second element of the I2C data array and we store that in the c_pressure variable.

```
unsigned long c_pressure = I2Cdata[1];
```

Then we take the last two bits of the number by bit shifting the LSB four places to the right, leaving us with the two bits that make up the last of the 18 bits of the integer part of the number and the two bits that make up the fractional part. As the two bits that make up the fractional part can store only four numbers (each bit is 1/4 or 0.25 of a Pascal), this is our resolution. So divide this by 4.

```
float l_pressure = (float)(I2Cdata[2]>>4)/4;
```

Finally we take all three of these bytes and combine them to create the final pressure reading. We do this by bitwise OR-ing the digits in the MSB and CSB, and then adding the digits in the LSB to this to give us our final pressure reading. The digits in m_pressure are bit shifted left 10 places and in c_pressure by two places. As the integer part of the pressure reading takes up 18 bits, this moves the bits into their correct position, leaving space to add the two final bits that make up the remainder of the integer part, and, as we have cast the number to a float, also the two bits that make up the fractional part of the number. We then return this number as the pressure from our function.

```
return((float)(m_pressure<<10 | c_pressure<<2)+l_pressure);
```

The next function does the same as above but for the two bytes that make up the temperature reading.

```
float Calc_Temperature(){
  int m_temp;
  float l_temp;
  m_temp = I2Cdata[3]; //temperature in whole degrees C
  l_temp = (float)(I2Cdata[4]>>4)/16.0; //fractional portion of temperature
  return((float)(m_temp + l_temp));
}
```

The next function simply reads the five bytes from the sensor that make up the pressure reading (three bytes) and the temperature reading (two bytes) and stores them in the I2Cdata[] array.

```
void I2C_ReadData(){
```

First, we create a variable of type byte that will be used to read the dataStatus register to determine if we have finished reading bytes from the sensor or not.

```
byte readUnsuccessful;
```

The rest of the function is inside a do-while loop that will keep repeating while data can be read successfully from the device.

```
do {
```

Next, we create an index for our array. We start at element 0.

```
byte i=0;
```

Then a variable of type byte called dataStatus is generated. This will be used later to determine if the data was successfully read from the device or not.

```
byte dataStatus = 0;
```

Next, we begin a transmission to the I2C slave device with the beginTransmission function in the Wire library. The parameter for the function is the address of the device, which is stored in SENSORADDRESS.

```
Wire.beginTransmission(SENSORADDRESS);
```

The next function queues bytes from the slave device starting at the address supplied in the parameter. In our case, the first byte of the pressure data starts at address 0x01 (SENSOR_OUT_P_MSB) so we tell the device to start to queue bytes from that address ready to be read.

```
Wire.write(SENSOR_OUT_P_MSB);
```

Next we use the endTransmission function to end a transmission to the slave device that was begun by beginTransmission and transmits the bytes that were queued by write(). The endTransmission function accepts a boolean argument of either true or false. If true, endTransmission() sends a stop message after transmission, releasing the I2C bus. If false, endTransmission() sends a restart message after transmission. The bus will not be released, which prevents another master device from transmitting between messages. This allows one master device to send multiple transmissions while in control.

```
Wire.endTransmission(false);
```

The requestFrom() function is used by the master to request bytes from the slave device. The bytes are then retrieved with the available() and read() functions. We pass two arguments to the function, the first being the address of the device and the second the number of bytes requested. We need the three bytes for the pressure reading and the three bytes for the temperature reading that all start at address 0x01 in the device, and so request five bytes.

```
Wire.requestFrom(SENSORADDRESS,5);
```

Next, we use a while statement to determine if there are bytes available to be read with the available() function. This function returns the number of bytes available for reading. In our case, that number will be five as this is the amount of bytes we requested from the device. So, if the value returned is above zero, the command after the available() function will run. In our case, we use the read() function to read a single byte from the device and store it in the I2Cdata[] array. The value in i is post-incremented with i++ at the same time. As long as data is still available to be read, the while statement will continue to read in a byte at a time and store it in the I2Cdata[] array. After five iterations, the available() function will return a 0 as we only requested five bytes. The program will then jump out of the while loop.

```
while(Wire.available()) I2Cdata[i++] = Wire.read();
```

In some modes, it is possible for the sensor to update the pressure reading in the middle of the reading it, resulting in a garbage copy (parts of two different readings). Therefore, we can check bits in the DR (data ready) register to see if this has happened. First, we ask for one byte from the DR register.

```
Wire.beginTransmission(SENSORADDRESS);
Wire.write(SENSOR_DR_STATUS); // Address of DataReady status register in sensor
Wire.endTransmission(false);
```

Then store it in dataStatus()

```
Wire.requestFrom(SENSORADDRESS,1);
dataStatus = Wire.read();
```

Now we need to check if the data status returns as successful or not. Bits 5 and 6 of the DR_STATUS register are set to 1 whenever new data is acquired before completing the retrieval of the previous set. So, if the byte returned from the DR_STATUS register is logically AND-ed with 0x60, the result will be a non-zero value if the read was unsuccessful. We store that result in readUnsuccessful.

```
    readUnsuccessful = (dataStatus & 0x60) != 0;
```

The while loop will keep executing until we get a clean read.

```
  } while (readUnsuccessful);   // keep reading until we get a successful clean read
}
```

The next function allows us to write a single byte to a single address within the device. We used this earlier in the program when we wrote our settings to CTRL_REG1 and also the calibration data to the device. The function accepts two arguments, the address of the register we wish to write to and the value we want to write to that address.

```
void I2C_Write(byte regAddr, byte value){
```

Again we begin transmission to the device using its address.

```
Wire.beginTransmission(SENSORADDRESS);
```

Then we transmit the address we want to write to followed by the value we want to write to that address using the write() function.

```
Wire.write(regAddr);
Wire.write(value);
```

Finally, we end transmission and this time, we pass the true argument to the function to send a stop message and release the I2C bus.

```
Wire.endTransmission(true);
```

Now you know the basics of communicating with an I2C device using the Wire library. However, some pressure sensors don't use the I2C bus and use SPI instead. SPI stands for serial peripherals interface. Just in case you come across a similar pressure sensor that uses SPI instead of I2C, let us take a little detour to look at SPI and how it works.

SPI – Serial Peripherals Interface

SPI can be a complicated subject and is far more complicated than using the much simpler I2C bus, so I am not going to go into it in any great depth. This is a beginner's book, after all. Instead, I am going to give you just enough knowledge so that you understand what SPI is, how it works, and what may be relevant to you if you were to use an SPI sensor instead of one with an I2C bus. You will then be able to understand how to interface with other devices that also use SPI.

SPI is a way for two devices to exchange information. It has the benefits of needing only four pins from the Arduino and it is fast. SPI is a synchronous protocol that allows a master device to communicate with a slave device. The data is controlled with a clock signal (CLK) which decides when data can change and when it is valid for reading. The clock rate can vary, unlike some other protocols in which the clock signal must be timed very accurately. This makes it ideal for microcontrollers that do not run particularly fast or whose internal oscillator is not clocked precisely.

SPI is a master-slave protocol (see Figure 11-5), meaning that a master device controls the clock signal. No data can be transmitted unless the clock is pulsed. SPI is also a data exchange protocol. This means that as data is clocked out, new data is being clocked in.

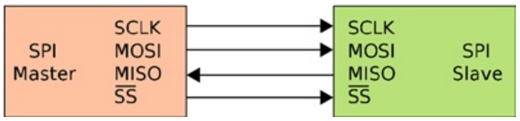


Figure 11-5. *SPU bus: single master and single slave (image courtesy of Colin M.L. Burnett)*

The slave select pin will control when a device can be accessed if more than one slave is attached to the master (see Figure 11-6). When there is only one slave device, the SS (sometimes denoted as CSB on some devices) is optional. However, as a rule, it should be used regardless as it is also used as a reset for the slave to make it ready to receive the next byte. The slave select signal is sent out by the master to tell the slave that it wishes to start an SPI data exchange. This signal is active when LOW, so when held HIGH, the slave device is not selected.

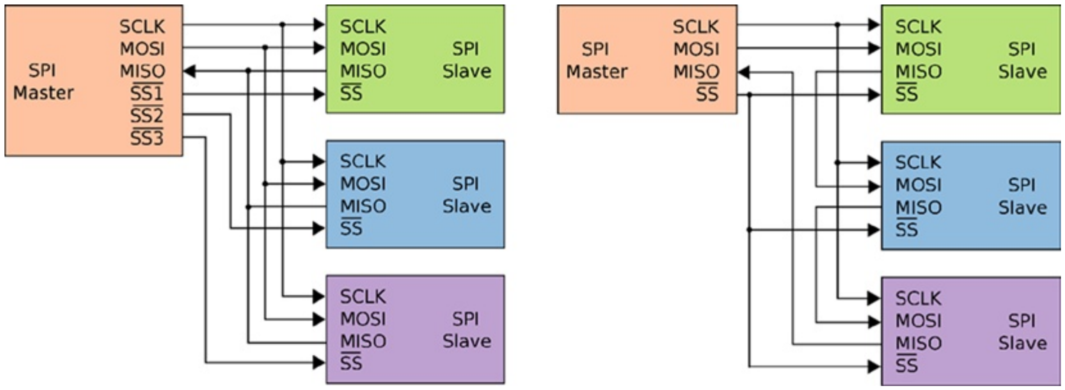


Figure 11-6. *Left: A master with three independent slaves. Right: daisy chained slaves. (Images courtesy of Colin M.L. Burnett)*

Data is only output during either the rising or falling edge of the clock signal on SCK. Data is latched during the opposite edge of SCK. The polarity of the clock is set by the master using one of the flags set in the SPCR register. The two data lines are known as MOSI (Master Output Slave Input) and MISO (Master Input Slave Output). So, if the device is set to send data from the master on the rising edge of the clock pulse, data would be sent back from the slave on the falling edge of the clock pulse. Data is therefore both sent (MOSI) and input (MISO) from the master during one clock pulse.

Remember that even if you only want to read data from a device (like you do mostly with a pressure sensor), you still need to send data both ways during one exchange.

The three registers used by the SPI bus are:

- SPCR – SPI control register
- SPDR – SPI data register
- SPSR – SPI status register

The control register has eight bits and each bit controls a particular SPI setting. These bits are listed in Table 11-2.

Table 11-2. *The SPI Control Register Settings*

7	6	5	4	3	2	1	0
SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

- SPIE – SPI Interrupt Enable - Enables the SPI interrupt if 1.
- SPE – SPI Enable - SPI enabled when set to 1.
- DORD – Data Order - LSB transmitted first if 1 and MSB if 0.
- MSTR – Master/Slave Select – Sets Arduino in Master Mode if 1, Slave Mode when 0.

- CPOL – Clock Polarity – Sets Clock to be idle when high if set to 1, idle when low if set to 0.
- CPHA – Clock Phase – Determines if data is sampled on the leading or trailing edge of the clock.
- SPR1/0 – SPI Clock Rate Select 1 & 0 – These two bits control the clock rate of the master device.

The reason you can change these settings is that different SPI devices expect the clock polarity, clock phase, data order, speed, etc. to be different. This is mainly due to the fact that there is no standard for SPI, therefore manufacturers create devices with minor differences. In code you could set the SPCR thus:

```
SPCR = B01010011;
```

So you have disabled the interrupt (SPIE = 0), enabled the SPI (SPE = 1), set the data order to MSB first (DORD = 0), set the Arduino as master (MSTR = 1), set the clock polarity to be idle when low (CPOL = 0), set the clock phase to sample on the rising edge (CPHA = 0), and set the speed to be 250kHz (SPR1/2 = 11, which is 1/64th of the Arduino's oscillator frequency (16,000/64)).

The SPI status register (SPSR) uses three of its bits for setting the status of the SPI transfer. You are only interested in bit 7 (SPIF – SPI Interrupt Flag), which tells you if a serial transfer has been completed or not. If a transfer has been completed, it is set to 1. This bit is cleared (set to 0) by first reading the SPSR with bit 7 set to 1 and then the SPI Data Register (SPDR) is accessed.

The SPDR simply holds the byte that is going to be sent out of the MOSI line and read in from the MISO line.

All of the above sounds pretty complicated, but most of it you do not need to know (just yet). The SPI bus can be explained in layman's terms as having a master and slave device that want to talk to each other. The clock pulse ensures that data is sent from the master to the slave as the clock pulse rises (or falls, depending on how you have set the control register) and from the slave to the master as the clock falls (or rises). SPIF is set to 1 after the transfer is complete.

Now you hopefully understand a little about how SPI works and can amend your code accordingly to use the SPI bus instead of the I2C bus if you have a pressure sensor that uses SPI instead of I2C.

Now you that you know how to read the pressure readings from an I2C-based digital pressure sensor, let's do something useful with the pressure readings. In the next project, we will take the pressure readings from the device and plot them over time on a GLCD (Graphical Liquid Crystal Display).

Project 32 – Digital Barograph

Now that you can hook up the MPL3115A2 pressure sensor and obtain data from it, you are going to put it to some use. This project is to make a digital barograph, which is a graph of pressure over time. To display the graph, you are going to use a GLCD (Graphic LCD). You'll learn how to display graphics as well as text.

Parts Required

The parts list is identical to Project 31, with the addition of a 128 x 64 GLCD, an extra resistor, and a potentiometer. You are going to use the GLCD library so the GLCD must have a KS0108 (or equivalent) driver chip. Check the datasheet before purchase.

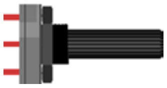
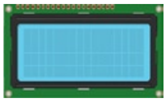
To get the GLCD library to work with an Uno:

In glcd/config/ks0108_Arduino.h

```
Change #define glcdEN   18
To     #define glcdEN   12
```

This modification tells the library to use digital pin 12 for the enable signal, instead of analog pin 5, which I2C uses for the SDA signal.

Table 11-3. *Parts Required for Project 31*

Arduino Uno or Mega	
MPL3115A2 Pressure Sensor	
150Ω Resistor	
10KΩ Potentiometer	
128x64 GLCD	

Connect It Up

Connect everything as shown in Figure 11-7. This circuit shows how to connect things up on an Arduino Mega. You will need to change some of the connections if you use a different kind of Arduino board.

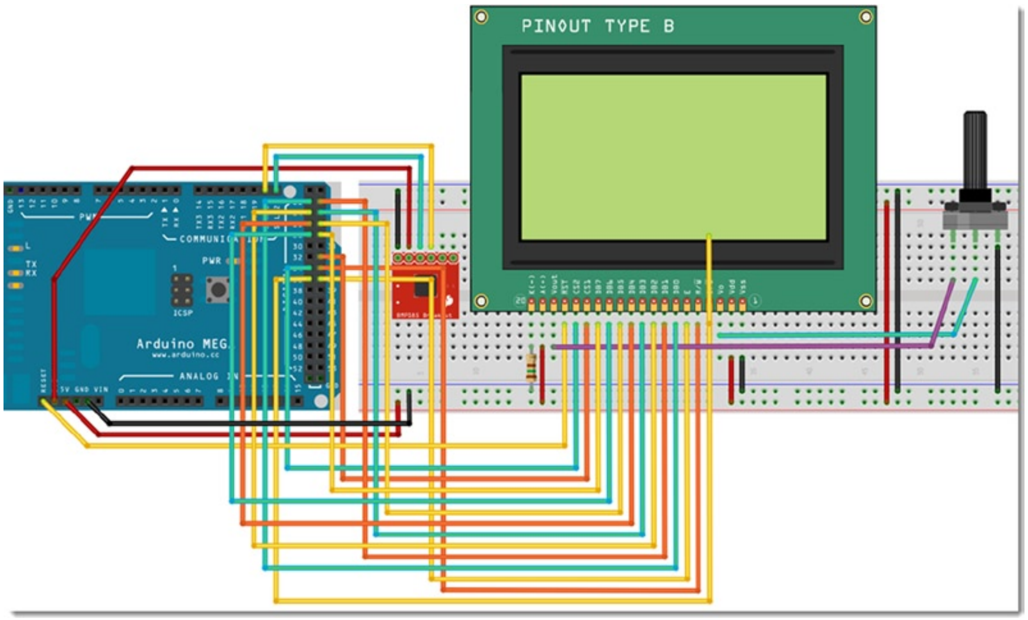


Figure 11-7. *The circuit for Project 32 – Digital Barograph (see insert for color version)*

The MPL3115A2 part of the circuit hasn't changed from Project 31. You are simply adding some extra components to that circuit. The GLCD you use may have different pinouts from the one from this project. Read the datasheet and make sure the pins match those in Table 11-4.

Table 11-4. *Pinouts between an Uno or Mega and the GLCD*

Uno	Mega	GLCD	Other
GND	GND	GND	
+5v	+5v	+5v	
		LCD Contrast	Centre pot pin
8	36	D/I	
9	35	R/W	
10	37	Enable	
11	22	DB0	
4	23	DB1	
5	24	DB2	
6	25	DB3	
7	26	DB4	
A0	27	DB5	
A1	28	DB6	
A2	29	DB7	
A3	33	CS1	
12	34	CS2	
Reset	Reset	Reset	
		VEE	Positive side of pot
		Backlight +5v	+5v
		Backlight +0v	GND via 150Ω Resistor

Note that the pinout for these LCDs is not standardized – there are at least four common variations. Check the data sheet for your particular model of LCD to make sure which pins are which. Some LCDs have power on pin 1 and ground on pin 2, others the opposite. If you hook the power up backward, you will damage the LCD as well. So, do not blindly follow the diagram above without first checking the datasheet for your LCD.

Do not power up the circuit until everything is connected up and you have double-checked the wiring. It is very easy to damage a GLCD by having it wired incorrectly. In particular, make sure that the potentiometer has one side ground to ground, with the center pin going to the LCD's contrast adjustment pin, and the other pin to VEE (LCD Voltage). The 150Ω resistor is to limit the current going to the LCD backlight; you may need to experiment with this value to get the correct brightness, but make sure you do not exceed the voltage in the datasheet for your LCD. The potentiometer is used to adjust the contrast on the display to get it so it can be viewed easily.

Also remember that the power going to the GLCD must be +5V and the power to the pressure sensor must be 3.3V. Don't mix them up or have the pressure sensor somehow connected to the 5V supply or you will damage the sensor. Check everything carefully before you power up your circuit.

Once you are confident everything is connected properly, power up your Arduino. You should see the GLCD light up ready to receive data. Now enter the code.

Enter the Code

Before you enter the code, you will need to download and install the GLCD library. You can find it at http://playground.arduino.cc/Code/GLCDks0108. This great library, written by Michael Margolis, is currently in its third version. It comes with a great document that shows how to connect up different types of GLCDs and full instructions for all of the commands in the library. Look on the Arduino Playground under LCDs to find it. Once you have downloaded the library, unzip it, and place the entire folder into the "libraries" folder inside your Arduino installation. The next time you fire up the Arduino IDE, it will be loaded and ready for use.

Enter the code from Listing 11-2.

Listing 11-2. Code for Project 32

```
// Project 32
#include <glcd.h>
#include "fonts/allFonts.h"
#include <Wire.h> // so we can use I2C communication

// The altitude at your location is needed to calculate the mean sea
// level pressure in meters
#define MYALTITUDE 262
#define SENSORADDRESS 0x60 // The address of the MPL3115A1
#define SENSOR_CONTROL_REG_1 0x26
#define SENSOR_DR_STATUS 0x00  // Address of DataReady status register
#define SENSOR_OUT_P_MSB 0x01  // Starting address of Pressure Data registers

  // Time interval in seconds (approx.) per graph tic
// Scale to deduct from hPa reading to fit within the 40 pixels graph 990-1030
#define INTERVAL 900

float baroAltitudeCorrectionFactor = 1/(pow(1-MYALTITUDE/44330.77,5.255877));

byte I2Cdata[5] = {0,0,0,0,0}; //buffer for sensor data
float pressure, temperature, buffer[30], baroPressure;
int dots[124], dotCursor = 0, counter = 0, index = 0, numReadingsBuffered = 0;

void setup()
{
  Wire.begin(); // join i2c bus
  I2C_Write(SENSOR_CONTROL_REG_1, 0b00000000); // put in standby mode
  // these upper bits of the control register
  // can only be changed while in standby

// Set oversampling to 128. Then trigger first reading
  I2C_Write(SENSOR_CONTROL_REG_1, 0b00111010);
  GLCD.Init();   // initialise the library
  GLCD.ClearScreen();
  GLCD.SelectFont(System5x7, BLACK); // load the font

  GLCD.DrawRect(1,1,125,44); // Draw a rectangle
  for (int x=0; x<46; x+=11) { // Draw vertical scale
    GLCD.SetDot(0,1+x, BLACK);
    GLCD.SetDot(127,1+x, BLACK);
  }
  for (int x=0; x<128; x+=5) { // Draw horizontal scale
    GLCD.SetDot(1+x,0, BLACK);
  }

// Clear the array to standard sea level
  for (byte x=0; x<124; x++) {dots[x]=1013;}

  Read_Sensor_Data();
  drawPoints(dotCursor);
}

void loop()
{
  Read_Sensor_Data();

  GLCD.CursorToXY(0, 49); // print pressure
  GLCD.print("hPa:");
  GLCD.CursorToXY(24,49);
  GLCD.print(baroPressure/100);
  GLCD.print("   ");  // erase any old value longer than new value

  float tempF = (temperature*1.8) + 32;

  GLCD.CursorToXY(0,57); // print temperature
  GLCD.print("Temp:");
  GLCD.CursorToXY(28, 57);
//  GLCD.print(temperature); // change to temperature for Centigrade
  GLCD.print(tempF);
  GLCD.print("   ");  // erase any old value longer than new value

  delay(1000);

  GLCD.CursorToXY(84,49); // print trend
  GLCD.print("TREND:");
  GLCD.CursorToXY(84,57);
  printTrend();

  counter++;
  if (counter==INTERVAL) {
    drawPoints(dotCursor);
    counter = 0;
  }
}
```

```
void drawPoints(int position) {
  dots[dotCursor] = int(baroPressure/100);
  int midscale = dots[dotCursor];  // center graph scale on current reading
  GLCD.FillRect(2, 2, 123, 40, WHITE); // clear graph area
  for (int x=0; x<124; x++) {
    // limit to graph boundary
    int y = constrain(22-((dots[position]- midscale)), 0,44);
    GLCD.SetDot(125-x,y, BLACK);
    position--;
    if (position<0) {position=123;}
  }
  dotCursor++;
  if (dotCursor>123) {dotCursor=0;}
}

// calculate trend since last data point and print
void printTrend() {
  int dotCursor2=dotCursor-1;
  if (dotCursor2<0) {dotCursor2=123;}
  int val1=dots[dotCursor2];
  int dotCursor3=dotCursor2-1;
  if (dotCursor3<0) {dotCursor3=123;}
  int val2=dots[dotCursor3];
  if (val1>val2) {GLCD.print("RISING ");}
  if (val1==val2) {GLCD.print("STEADY ");}
  if (val1<val2) {GLCD.print("FALLING");}
}

  // This function triggers a reading, waits for acquisition complete, and
  //  reads the pressure and temperature readings from the sensor
void Read_Sensor_Data(){
float tempPressure;

  // request a single measurement from the sensor
  I2C_Write(SENSOR_CONTROL_REG_1, 0b00111010); //bit 1 is one shot mode

  // wait for measurement to complete.
  // One-shot bit will clear when it is done.
  do {
   // read the current (sensor control) register
   // then repeat until sensor clears OST bit
    Wire.requestFrom(SENSORADDRESS,1);
  } while ((Wire.read() & 0b00000010) != 0);

  I2C_ReadData(); //reads registers from the sensor
  temperature = Calc_Temperature();
  tempPressure = Calc_Pressure();
  buffer[index++]=tempPressure;
  if (index > 29)  index = 0; // at end of buffer, wrap around to start.
  numReadingsBuffered++;
  if (numReadingsBuffered > 30)   numReadingsBuffered = 30;
  float mean = 0;
  for (int i=0; i<numReadingsBuffered; i++) {
    mean = mean+buffer[i];
  }
  pressure = mean/numReadingsBuffered;
  baroPressure = pressure*baroAltitudeCorrectionFactor;
}

// This function takes values from the read buffer
// and converts them to pressure units
float Calc_Pressure(){
  unsigned long m_pressure = I2Cdata[0];
  unsigned long c_pressure = I2Cdata[1];
  float l_pressure = (float)(I2Cdata[2]>>6)/4;
  return((float)(m_pressure<<10 | c_pressure<<2)+l_pressure);
}

// This function assembles the temperature reading
// from the values in the read buffer
float Calc_Temperature(){
  int m_temp;
  float l_temp;
  m_temp = I2Cdata[3]; //temperature in whole degrees C
  l_temp = (float)(I2Cdata[4]>>4)/16.0; //fractional portion of temperature
  return((float)(m_temp + l_temp));
}
void I2C_ReadData(){  //Read Barometer and Temperature data (5 bytes)
  byte readUnsuccessful;
  do {
    byte i=0;
    byte dataStatus = 0;

    Wire.beginTransmission(SENSORADDRESS);
    Wire.write(SENSOR_OUT_P_MSB);
    Wire.endTransmission(false);

    //read 5 bytes. 3 for pressure, 2 for temperature.
    Wire.requestFrom(SENSORADDRESS,5);
    while(Wire.available()) I2Cdata[i++] = Wire.read();

    // in some modes it is possible for the sensor to
    // update the pressure reading while we were in
    // the middle of reading it, in which case our copy is garbage
    // because it has parts of two different readings.
    // We can check some status bits in the DR (data ready)
    // register to see if this happened.

    Wire.beginTransmission(SENSORADDRESS);
    Wire.write(SENSOR_DR_STATUS);
    Wire.endTransmission(false);
    //read 1 byte to get status value
    Wire.requestFrom(SENSORADDRESS,1);
    dataStatus = Wire.read();
    readUnsuccessful = (dataStatus & 0x60) != 0;
    // unsuccessful if overwrite happened while we
    // were reading the pressure or temp data
   // keep reading until we get a successful clean read
  } while (readUnsuccessful);
}

// This function writes one byte over I2C
void I2C_Write(byte regAddr, byte value){
Wire.beginTransmission(SENSORADDRESS);
  Wire.write(regAddr);
  Wire.write(value);
  Wire.endTransmission(true);
}
```

When you run the code, you will see a graph of pressure over time as well as the current pressure and temperature. This isn't very exciting at first, as it needs over 24 hours to show the pressure changes. Each dot represents 15 minutes of time and the horizontal scale is hours. If you want to speed things up and fill the graph up more quickly, change the value in the INTERVAL define to something smaller. The 900 in INTERVAL is seconds (900 seconds – 15 minutes) so change it to a smaller interval setting to see a speeded up version of the graph. However, for the graph to be useful for weather prediction, leave it at 900 seconds and leave your circuit running for over 24 hours.

Project 32 – Digital Barograph – Code Overview

Most of the code for Project 32 is identical to Project 31. However, there are new sections to control the GLCD, and the parts for sending data to the serial monitor have been removed. I shall therefore gloss over code that is repeated from Project 31 and concentrate on the additions.

First, you need to include the GLCD.h library header file in your code, as well as the fonts you are going to use. We also include the wire library so that we can communicate with our I2C sensor.:

```
#include <glcd.h>
#include "fonts/allFonts.h"
#include <Wire.h>
```

As in Project 31, we now set up a value for your altitude in meters as well as the address of the sensor. This time we are using the #define statement which is more efficient as it doesn't require any memory to store the values. Instead, the compiler simply replaces every instance of the word MYALTITUDE with 83 (or whatever your own altitude is) and SENSORADDRESS with 0x60. The same goes for the last three define statements, which are for internal registers within the sensor.

```
#define MYALTITUDE 262
#define SENSORADDRESS 0x60
#define SENSOR_CONTROL_REG_1 0x26
#define SENSOR_DR_STATUS 0x00
#define SENSOR_OUT_P_MSB 0x01
```

In the addresses section, there is a new define for `INTERVAL`. This defines the interval, in seconds, in between data points stored and displayed on the graph. The interval is 900 seconds, which is 15 minutes between points.

```
#define INTERVAL 900  // Time interval in seconds (approx.)
```

We also define a scale for our graph.

```
#define SCALE 990
```

The graph shows a range of 40 hPa. You therefore want your plot to be somewhere in the center of the graph. Use Project 31 to determine your current hPa reading (hPa is simply the pressure in Pa divided by 100) and then deduct 22 from this reading to get your scale factor, e.g. if your pressure reading is 1,028 then the scale will be 1,008 which will start your plot off in the center of the 44 pixel graph.

The pressure reading from the sensor will need to be corrected according to your own altitude. We therefore need a correction factor and so create a variable of type float called baroAltitudeCorrectionFactor using the calculation in the MPL3115A2 datasheet.

```
float baroAltitudeCorrectionFactor = 1/(pow(1-MYALTITUDE/44330.77,5.255877));
```

Next, we create our array that will store the five bytes of the pressure and temperature readings.

```
byte I2Cdata[5] = {0,0,0,0,0};
```

We also create our pressure and temperature variables. In Project 31, these were inside the Read-Sensor_Data() function. However, we need that function to return the temperature and pressure readings to outside the function so it can be used elsewhere in the program. As it is not possible to easily return two separate values from a function (it can be done but complicates matters for beginners) we create the two variables outside of all other functions, which gives them Global Scope. This means the variables are accessible from any function within the program. This time around we increase the buffer array from 10 elements to 30 elements. We will use this array in Project 32 to store the pressure readings taken every second over 30 seconds and then average them out to give a more accurate pressure reading. You may have noticed that the readings in Project 31 fluctuated a lot from time to time. This was due to noise or other factors within your circuit making the readings inaccurate occasionally. By averaging readings over time we will get a much more accurate reading. Be aware, though, that this approach does take up a lot of memory. Each element is a float, which takes up four bytes each, and we have 30 elements. This means this array alone is taking up 120 bytes of space in memory. This may not sound a lot in the PC world where Gb of memory is common. However, on our humble little Arduino we have a limited amount of memory space to play with.

```
float pressure, temperature, buffer[30], baroPressure;
```

Five new integers are declared and initialized. One is the `dots[]` array that holds the pressure measurements read at fifteen minute intervals. The next is the `dotCursor` variable that stores the index of the array you are currently on. Next, the counter variable will be incremented every time the main loop is repeated and will be used to see how many seconds (roughly) have passed since the last data point was stored. The index variable will remember the current place within the buffer array that we are storing sensor data. Finally, numReadingsBuffered is used to store how many readings we have stored in the buffer[] array over time. Every 900 seconds this will increase by one until we have reached the maximum of 30 readings. We will average our readings later using numReadingsBuffered.

```
int dots[124], dotCursor = 0, counter = 0, index = 0, numReadingsBuffered = 0;
```

In the setup routine, you first initialize I2C communications and

```
Wire.begin();
```

Then, as in Project 31, the sensor is placed into Standby Mode and the oversampling rate is set to 128x.

```
I2C_Write(SENSOR_CONTROL_REG_1, 0b00000000);
I2C_Write(SENSOR_CONTROL_REG_1, 0b00111010);
```

Next, initialize the GLCD device:

```
GLCD.Init();
```

The functions to control the GLCD all come after the dot, such as the next command to clear the screen:

```
GLCD.ClearScreen();
```

Next, you choose which font you are going to use and if it is to be displayed as black or white pixels:

```
GLCD.SelectFont(System5x7, BLACK); // load the font
```

There are many different types and sizes of fonts that can be used. Read the instructions that come with the GLCD library and try out different fonts. Also, the library includes a cool piece of free software called FontCreator2 that can be used to create a font header file to include in your sketch. This program can convert PC fonts for use with the library.

Next, the box for the graph is displayed. This is done with the `DrawRect` command:

```
GLCD.DrawRect(1,1,125,44);
```

The coordinate system for the 128x64 display is 128 pixels wide and 64 pixels high with pixel 0 for both axes being in the top left corner. The X coordinate then stretches across from 0 to 127. The Y coordinate goes from 0 to 63 at the bottom. The `DrawRect` draws a rectangle from the X and Y coordinates that form the first two parameters. These are the upper left corner of the box. The next two parameters are the height and width extending from the X and Y co-ordinates. Your box goes from point 1,1 and stretches 125 pixels wide and 44 pixels high.

You can also create a filled rectangle with the `FillRect()` command, so

```
GLCD.FillRect(1,1,125,44, BLACK);
```

would create a solid black rectangle of the same dimensions. Next, you need the vertical and horizontal scale. You use a `for` loop to create dots at 0 pixels and 127 pixels across and at 11 pixel intervals starting at 1 pixel down:

```
  for (int x=0; x<46; x+=11) {
    GLCD.SetDot(0,1+x, BLACK);
    GLCD.SetDot(127,1+x, BLACK);
  }
```

You do this with the `SetDot` command that simply puts a single pixel at the X and Y co-ordinates in either BLACK or WHITE. White pixels will simply erase any black pixels already displayed.

Next, the vertical scale (hours) is drawn at five pixel intervals from pixel 1 across to the right hand side:

```
  for (int x=0; x<128; x+=5) {
  GLCD.SetDot(1+x,0, BLACK);
}
```

Now we iterate through the dots[] array and put a value of 1,013, which is the pressure at sea level, into each element as a starting point for the graph.

```
for (byte x=0; x<124; x++) {dots[x]=1013;} // clear the array to standard sea level
```

Next, we call the two functions that read the data from the sensor and then plot it on the graph.

```
Read_Sensor_Data();
drawPoints(dotCursor);
```

Now the main loop function of the program starts.

```
void loop()
{
```

First, the function for reading the sensor data is called.

```
Read_Sensor_Data();
```

Then we use the GLCD.CursorToXY() function to move the pointer to the X and Y coordinates specified, and then the GLCD.print() function to print the pressure and temperature.

```
  GLCD.CursorToXY(0, 49); // print pressure
  GLCD.print("hPa:");
  GLCD.CursorToXY(24,49);
  GLCD.print(baroPressure/100);
  GLCD.print("   ");  // erase any old value longer than new value

  float tempF = (temperature*1.8) + 32;

  GLCD.CursorToXY(0,57); // print temperature
  GLCD.print("Temp:");
  GLCD.CursorToXY(28, 57);
//  GLCD.print(temperature); // change to temperature for Centigrade
  GLCD.print(tempF);
  GLCD.print("   ");  // erase any old value longer than new value
```

Wait one second.

```
  delay(1000);
```

Then we print the trend using the printTrend() function. The trend shows if the pressure is rising, steady or falling.

```
GLCD.CursorToXY(84,49); // print trend
GLCD.print("TREND:");
GLCD.CursorToXY(84,57);
printTrend();
```

Next, the counter is incremented. Once the counter has reached the same value as our interval (in our case 900 seconds), then the drawPoints() function is used to plot the graph.

```
  counter++;
  if (counter==INTERVAL) {
    drawPoints(dotCursor);
    counter = 0;
  }
}
```

Next, you have added two new functions to draw the graph and print the current pressure trend. The first is the drawPoints() function. You pass it the dotCursor value as a parameter:

```
void drawPoints(int position) {
```

The current pressure reading is stored in the dots[] array at the current position. As you are only interested in a dot on a low resolution display, you don't need any numbers after the decimal point, so cast hPa to an integer. This also saves memory as an array of integer values takes up less space than an array of floats.

```
dots[dotCursor] = int(baroPressure/100);
```

Next, the graph needs to centered on the current reading.

```
int midscale = dots[dotCursor];
```

Now you need to clear the graph for the new data. This is done with a FillRect command, creating a white rectangle just inside the borders of the graph box:

```
GLCD.FillRect(2, 2, 123, 40, WHITE); // clear graph area
```

You now iterate through the 124 elements of the array with a for loop:

```
for (int x=0; x<124; x++) {
```

Next, we calculate the Y position of the graph. This is the pressure stored in the dots[] array. We take 22 (the height of the graph area divided by 2) and subtract the dot position which has had the midscale factor subtracted from that. This entire calculation is inside a constrain command which ensures that the result of the calculation doesn't go below 0 or above 44 or it will print outside the boundary of the graph area.

```
int y = constrain(22-((dots[position]-midscale)), 0,44);
```

Then, place a dot at the appropriate position on the graph using the `SetDot()` command:

```
GLCD.SetDot(125-x,y, BLACK);
```

You want the graph to draw from right to left so that the most current reading is in the right hand side and the graph scrolls to the left. So you start off with drawing the first point at the X coordinate of 125-x, which will move the points to the left as the value of X increases. The Y coordinate is calculated as above. Use Project 31 to determine the current hPa and subtract 22 from it to get your scale number.

These are typical hPa values for most locations. If you live in an area with generally higher or lower pressure, you can adjust the midscale value accordingly. You then deduct that number from 22 to give you the Y position of the dot for that particular pressure reading in the array.

The value of position, which was originally set as the current value of dotCursor, is then decremented

```
position--;
```

and, in case it goes below zero, is set back to 123.

```
if (position<0) {position=123;}
```

Once all 124 dots have been drawn, the value of `dotCursor` is incremented, ready to store the next pressure measurement in the array

```
dotCursor++;
```

and, in case it goes above the value of 123 (the maximum element of the array), is set back to zero

```
if (dotCursor>123) {dotCursor=0;}
```

The next new function is the printTrend() function. The job of this function is to simply find out if the current pressure measurement stored is higher, lower, or the same as the last reading stored and to print RISING, STEADY, or FALLING accordingly.

You start by storing the last position of dotCursor in dotCursor2. You deduct one from its value as it was incremented after the measurement was stored in the drawPoints() function.

```
int dotCursor2=dotCursor-1;
```

You check if that value is less than zero, and if so, set it back to 123:

```
if (dotCursor2<0) {dotCursor2=123;}
```

dotCursor2 now stores the position in the array of the last measurement taken. You now declare an integer called val1 and store the last pressure measurement taken in it.

```
int val1=dots[dotCursor2];
```

You now want the measurement taken BEFORE the last one, so you create another variable called dotCursor3 that will store the position in the array before the last one taken:

```
int dotCursor3=dotCursor2-1;
if (dotCursor3<0) {dotCursor3=123;}
int val2=dots[dotCursor3];
```

You now have val1 with the last pressure reading and val2 with the pressure reading before that one. All that is left to do is to decide if the last pressure reading taken is higher, lower, or the same as the one before that and print the relevant trend accordingly.

```
if (val1>val2) {GLCD.print("RISING ");}
if (val1==val2) {GLCD.print("STEADY ");}
if (val1<val2) {GLCD.print("FALLING");}
```

Finally, there are the five functions we used in Project 31: Read_Sensor_Data(), Calc_Pressure(), Calc_Temperature(), I2C_ReadData() and I2C_Write().

When you run this program, you will end up with a display similar to Figure 11-8. If the pressure has changed considerably over the last 24 hours, you will see a greatly varying line.

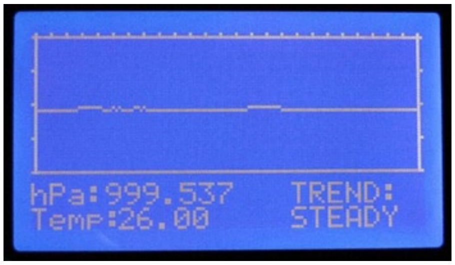


Figure 11-8. *The display for Project 32 – Digital Barograph*

The main purpose of Project 32 was to show you a practical use for the MPL31125A2 sensor and how to use a GLCD display. The commands for the GLCD introduced were just a taste for what you can do with a GLCD. You can display rectangles, circles, and lines, set dots, and even draw bitmaps. The screen can be divided up into text and graphics areas and you can access and print to them independently. The library comes with very good documentation by Michael Margolis and is very easy to use. Have a good read of the documentation and you will see there is a lot you can do with it. The library also comes with a whole set of example sketches including John Horton Conway's Game of Life and a great little rocket game.

Summary

Chapter 11 has shown you how to use a digital pressure sensor and communicate with it over a I2C bus. You have also been introduced to the basic concepts of SPI and how it works. Now that you know roughly how SPI works, you can use the great SPI library bundled with the Arduino library. This will do all of the hard work for you when communicating with any other SPI devices. You have learned how to use I2C to read data from the MPL3115A2 pressure sensor. If you wish to make your own weather station, this inexpensive sensor is an ideal choice. I choose this sensor for an amateur attempt at a high altitude balloon project, as it is small, accurate, and easy to use.

You have also learned how to connect a graphic LCD to the Arduino and how easy it is to print text and basic graphics on it using the GLCD library. By reading the documentation further, you can learn how to do cool things like display bitmaps or create your own games. An Arduino with a GLCD could easily be placed into a small box to make your own handheld game console.

Subjects and Concepts Covered in Chapter 11

- How to connect an MPL3115A2 pressure sensor to an Arduino
- How to use a #define to carry out bitwise operations on a set of numbers
- How to create larger bit length numbers by combining smaller bit length numbers together
- How the I2C bus works
- How to address an I2C device
- How to read data from an I2C device
- How to write data to an I2C device

- How to use the functions in the Wire library to initiate and end transmission
- How to use the functions in the Wire library to request bytes from a slave device
- How an SPI interface works
- That SPI devices can be controlled separately or daisy chained
- That an SPI device comprises a master and a slave
- How data is clocked in and out of an SPI device with the clock pulse
- The purpose and use of the three SPI bus registers
- Converting pressure in Pascals to Hectopascals and Atmospheres
- Using bitwise operators to check if a single bit is set or not
- How to connect a Graphic LCD to an Arduino
- Using the basic commands in the GLCD library to draw lines, dots, and print text

In the next chapter, you'll learn how to use a touch screen.

CHAPTER 12

Touch Screens

You are now going to take a look at a cool gadget that you can use easily with an Arduino—a touch screen. Since the advent of smart phones and handheld game consoles, touch screens are now inexpensive and readily available. A touch screen allows you to make an easy touch interface for a device, or can be overlaid onto an LCD screen to give a touch interface. Companies such as Sparkfun make it easy to interface with these devices by providing connectors and breakout units. A breakout unit allows you to take what would normally be a tiny set of connectors or a non-standard connector and "break it out" to something more user-friendly, such as header pins that you can use to push the unit into a breadboard. In this project, you're going to use the readily available Nintendo DS touch screen with a breakout unit from Sparkfun. You'll start off with a simple project that shows how to obtain readings from the touch screen before putting it to use.

Project 33 – Basic Touch Screen

For this project, you will need a Nintendo DS touch screen as well as a breakout module. The latter is essential as the output from the touch screen is a very thin and fragile ribbon connector, and it will be impossible to interface to the Arduino without additional components.

Parts Required

Nintendo DS touch screens are inexpensive and can be obtained from many suppliers. The XL version is about twice the size of the standard version; this is the recommended unit, if you can obtain one. The breakout module is from Sparkfun or their distributors.

Table 12-1. *Parts Required for Project 33*

Nintendo DS touch screen	
Touch screen breakout	
47K Resistors x 2	

Connect It Up

Connect everything as shown in Figure 12-1.

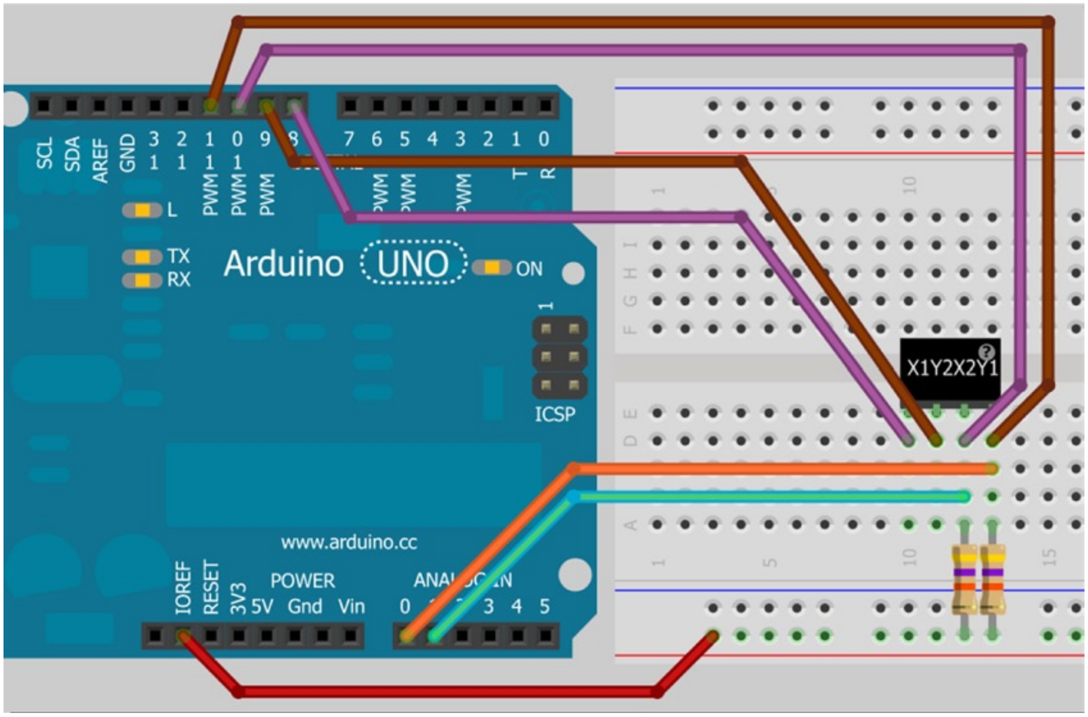


Figure 12-1. *The circuit for Project 33 – Basic Touch Screen*

The breakout unit has pins marked as X1, Y2, X2, and Y1. Connect the pins as described in Table 12-2. You will need a connection from the IOREF pin (or 5V pin if you have an older Arduino without an IOREF pin) to the breadboard and then suitably high value resistors (around 50K) between the X2 and Y1 pins and the IOREF line on the breadboard.

Table 12-2. *Pin Connections for Project 33*

Arduino	Breakout
Digital Pin 8	X1
Digital Pin 9	Y2
Digital Pin 10	X2
Digital Pin 11	Y1
Analog Pin 0	Y1
Analog Pin 1	X2
IOREF (or 5V) via a 50K resistor	Y1
IOREF (or 5V) via a 50K resistor	X2

You will need to solder some header pins to the breakout unit. The pins are soldered such that the Sparkfun logo is facing upward. The screen is connected to the breakout unit via the small connector. Pull back the tab and push the tiny ribbon cable into the connector, then push the tab closed to lock it in place. The screen goes with the ribbon connector at the top right when connecting. From now on, be very careful with the unit: it is very fragile and easily broken! I broke three screens and two breakouts in testing. If you can find a way of fixing the breadboard, breakout, and touch screen in place to prevent it from moving, you should do so.

Enter the Code

Enter the code in Listing 12-1.

Listing 12-1. Code for Project 33

```
// Project 33

// Power connections
#define Left 8                     // Left (X1) to digital pin 8
#define Bottom 9                   // Bottom (Y2) to digital pin 9
#define Right 10                   // Right (X2) to digital pin 10
#define Top 11                     // Top (Y1) to digital pin 11

// Analog connections
#define topInput 0                 // Top (Y1) to analog pin 0
#define rightInput 1               // Right (X2) to analog pin 1

int coordX = 0, coordY = 0;

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

void loop()
{
        if (touch())               // If screen touched, print co-ordinates
        {
                Serial.print(coordX);
                Serial.print("  ");
                Serial.println(coordY);
                delay(250);
        }
}

// return TRUE if touched, and set coordinates to touchX and touchY
boolean touch()
{
        boolean touch = false;

                                   // get horizontal co-ordinates
        pinMode(Top, INPUT);     // Top and Bottom to high impedance
        pinMode(Bottom, INPUT);

        pinMode(Left, OUTPUT);
        digitalWrite(Left, LOW); // Set Left to low

        pinMode(Right, OUTPUT);  // Set right to +5v
        digitalWrite(Right, HIGH);

        delay(3);
        coordX = analogRead(topInput);

                                 // get vertical co-ordinates

        pinMode(Right, INPUT);   // left and right to high impedance
        pinMode(Left, INPUT);

        pinMode(Bottom, OUTPUT); // set Bottom to Gnd
        digitalWrite(Bottom, LOW);

        pinMode(Top, OUTPUT);    // set Top to +5v
        digitalWrite(Top, HIGH);

        delay(3);
        coordY = analogRead(rightInput);

                                 // if co-ordinates read are less than 1000 and greater than 24
                                 // then the screen has been touched
        if(coordX < 1000 && coordX > 24 && coordY < 1000 && coordY > 24) {touch = true;}

          return touch;
}
```

Enter the code and upload it to your Arduino. Once it is running, open up the serial monitor, and then touch the touch screen. Whenever the screen is touched, the coordinates of your finger will be displayed on the serial monitor. The coordinates are *x* across the horizontal plane going from left to right and *y* across the vertical plane going from top to bottom.

Before you take a look at the code, it will help if you know how a touch screen works. I will therefore take a look at the hardware before examining the code.

Project 33 – Basic Touch Screen – Hardware Overview

The touch screen that you are using, from a Nintendo DS, is known as a Touch screen:resistive touch screen. It is a relatively simple construction made up of different layers. The bottom layer of the screen is made of glass that has been coated with a transparent film of metal oxide. This makes the coating both conductive and resistive. A voltage applied across the film has a gradient. On top of the rigid glass layer is a flexible top layer that is also covered in the transparent resistive film. These two layers are separated by a very thin gap kept apart by a grid of tiny insulating dots. These dots have the job of holding the two conductive layers apart so they don't touch.

If you examine your touch screen, you will see four connectors on the ribbon cable that lead to four metallic strips on the edges of the screen. Two of the metal strips are at the top and bottom of the screen, and if you flip the screen over, you will see the other two on the second layer and on the left and right hand sides of the screen.

When a finger or stylus is pressed against the top flexible layer, the layer bends down to touch the rigid layer, closing the circuit and creating a switch (see Figure 12-2).

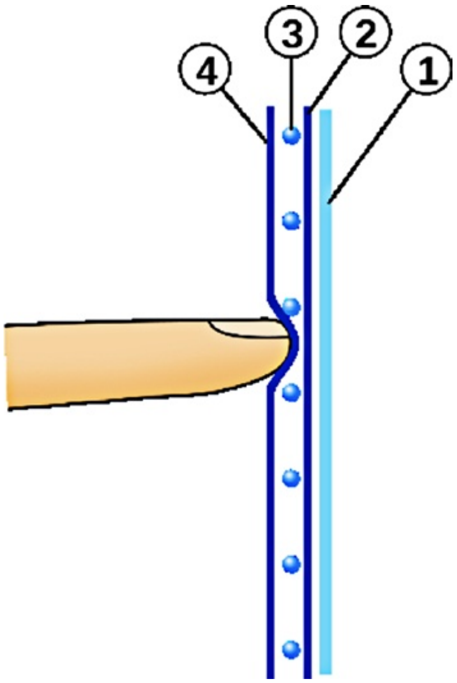


Figure 12-2. *How a touch screen works (courtesy of Mercury13 from Wikimedia Commons). 1: Rigid layer. 2: Metal oxide layer. 3: Insulating dots. 4: Flexible layer with metal oxide film*

To find the coordinates of the touched point, a voltage is first applied across the gradient from left to right. Making one side ground and the other 5V accomplishes this. Then, one of the strips on the opposite layer is read using an analog input to measure the voltage. The voltage when a point is pressed close to the five volts side will measure close to five volts; likewise, the voltage when pressed close to the ground side will measure close to zero.

Next, the voltage is applied across the opposing layer and read from the other. This is done in quick succession hundreds of times a second, so by reading first the X- and then the Y-axis quickly, you can obtain a voltage for each layer. This gives you the X- and Y- coordinates for the point on the screen that has been touched. If you touch two points on the screen at the same time, you get a reading equal to the halfway point between the two touched points.

There are other technologies used in touch screens, but the resistive type is cheap to manufacture and very easy to interface to the Arduino without needing other circuitry to make it work. Now that you know how the screen works, let's take a look at the code to see how to measure the voltages and obtain the coordinates.

Project 33 – Basic Touch Screen – Code Overview

The code for reading a touch screen is actually very simple. You start off by defining the four digital pins you will use for applying the power across the layers and the two analog pins you will use to measure the voltages:

```
// Power connections
#define Left 8       // Left (X1) to digital pin 8
#define Bottom 9     // Bottom (Y2) to digital pin 9
#define Right 10     // Right (X2) to digital pin 10
#define Top 11       // Top (Y1) to digital pin 11

// Analog connections
#define topInput 0   // Top (Y1) to analog pin 0
#define rightInput 1 // Right (X2) to analog pin 1
```

Then you declare and initialize the X and Y integers that will hold the coordinates, which are both initially set to zero:

```
int coordX = 0, coordY = 0;
```

Since you are going to read the coordinates using the serial monitor, in the setup procedure all you need to do is begin serial communication and set the baud rate. In this case, you'll use 38400 baud:

```
Serial.begin(38400);
```

The main program loop comprises nothing more than an `if` statement to determine of the screen has been touched or not:

```
if (touch())  // If screen touched, print co-ordinates
```

`touch()` is next. If the screen has been touched, you simply print out the X and Y coordinates to the serial monitor with a space in between, using the `Serial.print` commands:

```
Serial.print(coordX);
Serial.print("  ");
Serial.println(coordY);
```

After you have printed the coordinates, you wait a quarter of a second so the coordinates are readable if you keep your finger pressed down on the screen:

```
delay(250);
```

Next comes the function that does all of the hard work. The function will be returning a boolean true or false, so it is of data type boolean. You do not pass any parameters to the function, so the parameter list is empty.

```
boolean touch()
```

You declare a variable of type boolean and initialize it to false. This will hold a true or false value depending if the screen is touched or not.

```
boolean touch = false;
```

The top and bottom digital pins are then set to INPUT so that they become high impedance:

```
pinMode(Top, INPUT);
pinMode(Bottom, INPUT);
```

High impedance simply means that the pins are not driven by the circuit and are therefore *floating*, i.e. they are neither HIGH nor LOW. You do not want these pins to have a voltage across them or to be at ground, hence the high impedance state is perfect as you will want to read an analog voltage using one of these pins.

Next, you need to put a voltage across the left-right layer and read the voltage using the top input pin on the second layer. To do this, you set the left and right pins to outputs and then make the left pin LOW so it becomes ground and the right pin HIGH so it has five volts across it:

```
pinMode(Left, OUTPUT);
digitalWrite(Left, LOW); // Set Left to Gnd

pinMode(Right, OUTPUT);  // Set right to +5v
digitalWrite(Right, HIGH);
```

Next, you wait a short delay to allow the above state changes to occur and then read the analog value from the top input pin. This value is then stored in coordX to give you the X coordinate.

```
delay(3);
coordX = analogRead(topInput);
```

You now have your X coordinate. So next you do exactly the same thing but this time you set the voltage across the top-bottom layer and read it using the rightInput pin on the opposing layer:

```
pinMode(Right, INPUT);   // left and right to high impedance
pinMode(Left, INPUT);

pinMode(Bottom, OUTPUT); // set Bottom to Gnd
digitalWrite(Bottom, LOW);

pinMode(Top, OUTPUT);    // set Top to +5v
digitalWrite(Top, HIGH);

delay(3);
coordY = analogRead(rightInput);
```

You set the boolean variable touch to true only if the values read are greater than 24 and less than one thousand. This is to ensure you only return a true value if the readings are within acceptable values.

```
if(coordX < 1000 && coordX > 24 && coordY < 1000 && coordY > 24) {touch = true;}
```

You will find the values range from approximately 100 at the lowest scale to around 900 at the top end. Finally, you return the value of touch, which will be false if the screen is not pressed and true if it is:

```
return touch;
```

As you can see, reading values from the touch screen is very simple and allows for all kinds of uses. You can put a picture or diagram behind the screen relating to buttons or other controls or overlay the screen onto an LCD display, as in the Nintendo DS, which will allow you to change the user interface below the screens as required.

You'll do a simple demonstration of this by printing out a keypad that can be placed underneath the touch screen and reading the appropriate values to work out which key has been pressed.

Project 34 – Touch Screen Keypad

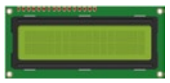
You'll now place a user interface underneath the touch screen in the form of a printed keypad and determine from the touch locations which key has been pressed. Once you understand the basics of doing this, you can go on to replace the printed keypad with one displayed on an LCD or OLED display.

You will output the key pressed on an LCD display so you'll need to add one to the parts list for this project.

Parts Required

You'll be using the exact same parts and circuit as in Project 33 with the addition of a 16×2 LCD display.

Table 12-3. *Parts Required for Project 34*

Nintendo DS touch screen	
Touch screen breakout	
16×2 LCD Display	
47K Resistors x 2	

The other difference is that you will create and print out a keypad to place underneath the touch screen. The standard DS touch screen is 70mm × 55mm (2.75" × 2.16") so you will need to create a template of this size using an art or word-processing package, and then place a set of evenly spaced keys on the rectangle so it resembles a phone keypad. Figure 12-3 shows the keypad I created. Feel free to use it.

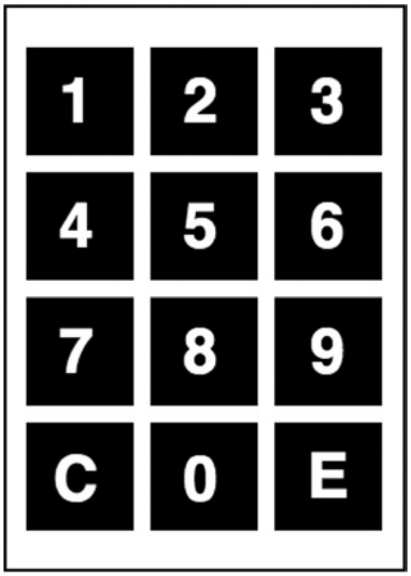


Figure 12-3. *The keypad diagram for Project 34*

Connect It Up

Connect everything as shown in Figure 12-4.

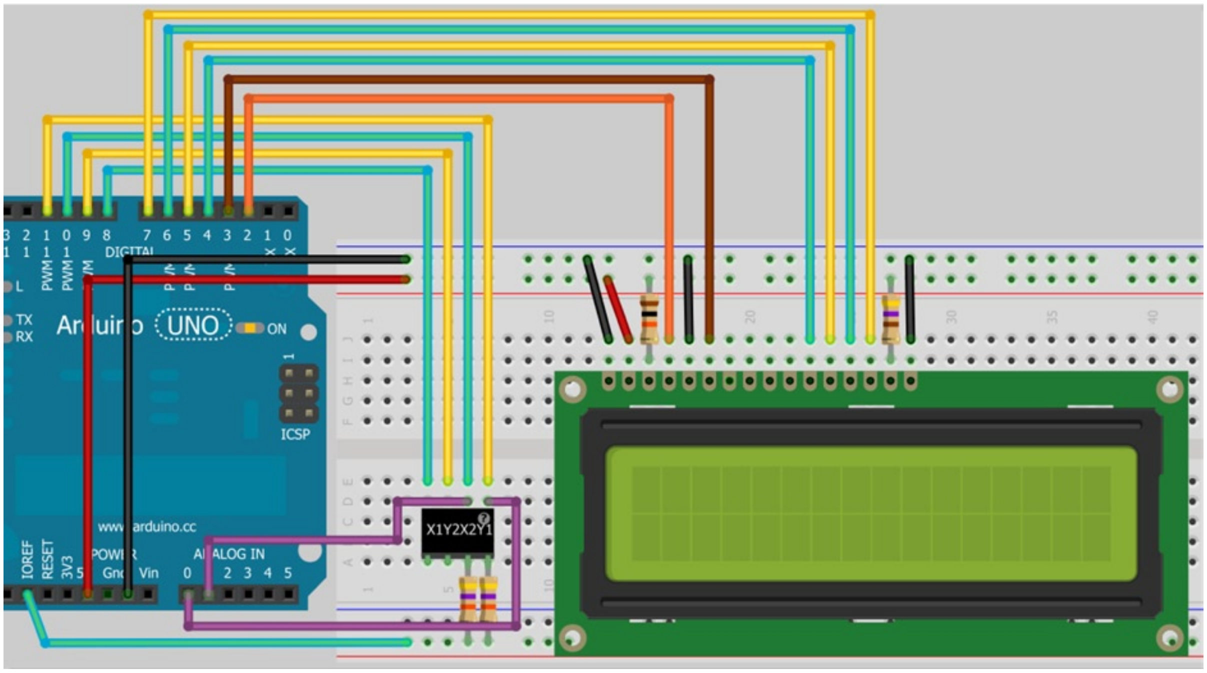


Figure 12-4. *The circuit for Project 34 – Touch Screen Keypad*

Refer to Table 12-4 for the pin outs for the LCD.

Table 12-4. *Pinouts for the LCD in Project 34*

Arduino	Other	Matrix
Digital 2		RS (Register Select
Digital 3		Enable
Digital 4		DB4 (Data Pin 4)
Digital 5		DB5 (Data Pin 5)
Digital 6		DB6 (Data Pin 6)
Digital 7		DB7 (Data Pin 7)
	Gnd	Vss (GND)
	Gnd	R/W (Read/Write)
	+5V	Vdd
	+5V via resistor	Vo (Contrast)
	+5V via resistor	A/Vee (Power for LED)
	Gnd	Gnd for LED

Enter the Code

Enter the code in Listing 12-2.

Listing 12-2. Code for Project 34

```
// Project 34

#include <LiquidCrystal.h>

LiquidCrystal lcd(2, 3, 4, 5, 6, 7); // create an lcd object and assign the pins

// Power connections
#define Left 8                      // Left (X1) to digital pin 8
#define Bottom 9                    // Bottom (Y2) to digital pin 9
#define Right 10                    // Right (X2) to digital pin 10
#define Top 11                      // Top (Y1) to digital pin 11

// Analog connections
#define topInput 0                  // Top (Y1) to analog pin 0
#define rightInput 1                // Right (X2) to analog pin 1

int coordX = 0, coordY = 0;
char buffer[16];
```

```
void setup()
{
  lcd.begin(16, 2);                  // Set the display to 16 columns and 2 rows
  lcd.clear();
}

void loop()
{
  if (touch())
  {
    if ((coordX>110 && coordX<300) && (coordY>170 && coordY<360)) {lcd.print("3");}
    if ((coordX>110 && coordX<300) && (coordY>410 && coordY<610)) {lcd.print("2");}
    if ((coordX>110 && coordX<300) && (coordY>640 && coordY<860)) {lcd.print("1");}
    if ((coordX>330 && coordX<470) && (coordY>170 && coordY<360)) {lcd.print("6");}
    if ((coordX>330 && coordX<470) && (coordY>410 && coordY<610)) {lcd.print("5");}
    if ((coordX>330 && coordX<470) && (coordY>640 && coordY<860)) {lcd.print("4");}
    if ((coordX>490 && coordX<710) && (coordY>170 && coordY<360)) {lcd.print("9");}
    if ((coordX>490 && coordX<710) && (coordY>410 && coordY<610)) {lcd.print("8");}
    if ((coordX>490 && coordX<710) && (coordY>640 && coordY<860)) {lcd.print("7");}
    if ((coordX>760 && coordX<940) && (coordY>170 && coordY<360)) {scrollLCD();}
    if ((coordX>760 && coordX<940) && (coordY>410 && coordY<610)) {lcd.print("0");}
    if ((coordX>760 && coordX<940) && (coordY>640 && coordY<860)) {lcd.clear();}
    delay(250);
  }
}

// return TRUE if touched, and set coordinates to touchX and touchY
boolean touch()
{
        boolean touch = false;

                                     // get horizontal co-ordinates

        pinMode(Top, INPUT);         // Top and Bottom to high impedance
        pinMode(Bottom, INPUT);

        pinMode(Left, OUTPUT);
        digitalWrite(Left, LOW);     // Set Left to low

        pinMode(Right, OUTPUT);      // Set right to +5v
        digitalWrite(Right, HIGH);

        delay(3);
        coordX = analogRead(topInput);

                                     // get vertical co-ordinates

        pinMode(Right, INPUT);       // left and right to high impedance
        pinMode(Left, INPUT);

        pinMode(Bottom, OUTPUT);     // set Bottom to Gnd
        digitalWrite(Bottom, LOW);
        pinMode(Top, OUTPUT);        // set Top to +5v
        digitalWrite(Top, HIGH);


        delay(3);
        coordY = analogRead(rightInput);

                                     // if co-ordinates read are less than 1000 and greater than 24
                                     // then the screen has been touched
        if(coordX < 1000 && coordX > 24 && coordY < 1000 && coordY > 24) {touch = true;}

          return touch;
}

void scrollLCD() {
  for (int scrollNum=0; scrollNum<16; scrollNum++) {
    lcd.scrollDisplayLeft();
    delay(100);
  }
  lcd.clear();
}
```

Enter the code and upload it to your Arduino. Slide the keypad template underneath the keypad with the ribbon cable at the bottom right (next to the E). You can now press the keys on the touch screen and what you press is displayed on the LCD. When you press the C (for Clear) button, the display will clear. When you press the E (for Enter) key, the numbers displayed will scroll to the left until they disappear.

You already know how the LCD screen and the touch screen work, so I will skip the hardware overview in this project and just look at the code.

Project 34 –Touch Screen Keypad – Code Overview

You start off by including the LiquidCrystal library and creating an `lcd` object:

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(2, 3, 4, 5, 6, 7); // create an lcd object and assign the pins
```

This time, you are using pins 2 and 3 for the RS and enable on the LCD and pins 4 to 7 for the data lines. Next, the pins for the touch screens are defined and the X and Y variables initialized:

```
// Power connections
#define Left 8                       // Left (X1) to digital pin 8
#define Bottom 9                     // Bottom (Y2) to digital pin 9
#define Right 10                     // Right (X2) to digital pin 10
#define Top 11                       // Top (Y1) to digital pin 11

// Analog connections
#define topInput 0                   // Top (Y1) to analog pin 0
#define rightInput 1                 // Right (X2) to analog pin 1

int coordX = 0, coordY = 0;
```

In the setup routine, you begin the LCD object and set it to 16 columns, 2 rows, then clear the display so you're ready to begin:

```
lcd.begin(16, 2); // Set the display to 16 columns and 2 rows
lcd.clear();
```

In the main loop, you have an `if` statement as you did in Project 33, but this time you need to check that the coordinates touched are within a rectangle that defines the boundary of each button. If the coordinate is within the relevant button boundary, the appropriate number is displayed on the LCD. If the C button is pressed, the display is cleared; if the E button is pressed, the `scrollLCD` function is called.

```
if (touch())
  {
    if ((coordX>110 && coordX<300) && (coordY>170 && coordY<360)) {lcd.print("3");}
    if ((coordX>110 && coordX<300) && (coordY>410 && coordY<610)) {lcd.print("2");}
    if ((coordX>110 && coordX<300) && (coordY>640 && coordY<860)) {lcd.print("1");}
    if ((coordX>330 && coordX<470) && (coordY>170 && coordY<360)) {lcd.print("6");}
    if ((coordX>330 && coordX<470) && (coordY>410 && coordY<610)) {lcd.print("5");}
    if ((coordX>330 && coordX<470) && (coordY>640 && coordY<860)) {lcd.print("4");}
    if ((coordX>490 && coordX<710) && (coordY>170 && coordY<360)) {lcd.print("9");}
    if ((coordX>490 && coordX<710) && (coordY>410 && coordY<610)) {lcd.print("8");}
    if ((coordX>490 && coordX<710) && (coordY>640 && coordY<860)) {lcd.print("7");}
    if ((coordX>760 && coordX<940) && (coordY>170 && coordY<360)) {scrollLCD();}
    if ((coordX>760 && coordX<940) && (coordY>410 && coordY<610)) {lcd.print("0");}
    if ((coordX>760 && coordX<940) && (coordY>640 && coordY<860)) {lcd.clear();}
    delay(250);
  }
```

Each `if` statement is a set of conditional and logical operators. If you look at the statement for button three

```
if ((coordX>110 && coordX<300) && (coordY>170 && coordY<360)) {lcd.print("3");}
```

you can see that the first logical AND condition is checking that the touched position is within position 110 and 300 from the left and the second is within position 170 and 360 from the top. All conditions must be met for the button to be pressed, hence the AND (&&) logical operators are used.

To find out your button coordinates, simply press gently using a pointed stylus on the left and right hand side of the button to get the X coordinates. Then repeat with the top and bottom sides to get the Y coordinates. If you use Project 33 to print out the coordinates to the serial monitor, you can use it to determine the exact coordinates for your button locations if you need to adjust the code or if you want to make your own button layout.

Next is the touch function; you already know how it works. Finally, there's the `scrollLCD` function that does not return any data nor takes any parameters and so is of type void:

```
void scrollLCD() {
```

Then you have a `for` loop that repeats 16 times, which is the maximum number of characters that can be entered and displayed:

```
for (int scrollNum=0; scrollNum<16; scrollNum++) {
```

Inside the `for` loop, you use the `scrollDisplayLeft()` function from the LiquidCrystal library to scroll the displayed characters one space to the left. This is followed by a 100 millisecond delay.

```
lcd.scrollDisplayLeft();
delay(100);
```

Once this has been done 16 times, the numbers entered will slide off to the left, giving the impression they have been entered into the system. You can write your own routines to do whatever you want with the data once entered.

Finally, you clear the display to ensure it is ready for new data before exiting the function back to the main loop:

```
lcd.clear();
```

This project gives you an idea how to zone off parts of a touch screen so that you can select areas for buttons, etc. The paper can be substituted with a graphic LCD or an OLED display on which you can draw buttons. The advantage of this is that different menus and different buttons can be drawn depending on what the user has selected. Using this technique, you could create a really fancy touch screen user interface for your project.

You'll now move on to controlling an RGB LED and sliding the touch screen instead of pressing it to control the colors.

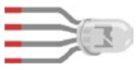
Project 35 – Touch Screen Light Controller

In this project, you will use the touch screen to turn an RGB LED lamp on and off and to control the color of the LED.

Parts Required

You will be using the exact same parts and circuit as in Project 33 with the addition of an RGB LED. The RGB LED needs to be of the common cathode type. This means that one of the pins is connected to ground (the cathode) and the other three pins go separately to the control pins for the red, green, and blue voltages.

Table 12-5. *Parts Required for Project 35*

Nintendo DS touch screen	
Touch screen breakout	
RGB LED (common cathode)	
Current-Limiting Resistors x 3	
47k resistors x 2	

You will also need a keypad template as in Project 34. This time it needs to have areas for the color sliders and the on/off buttons. Feel free to use the image in Figure 12-5.

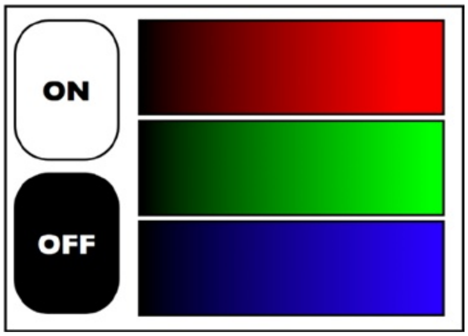


***Figure 12-5.** The keypad diagram for Project 35*

Connect It Up

Connect everything up as in Figure 12-6.

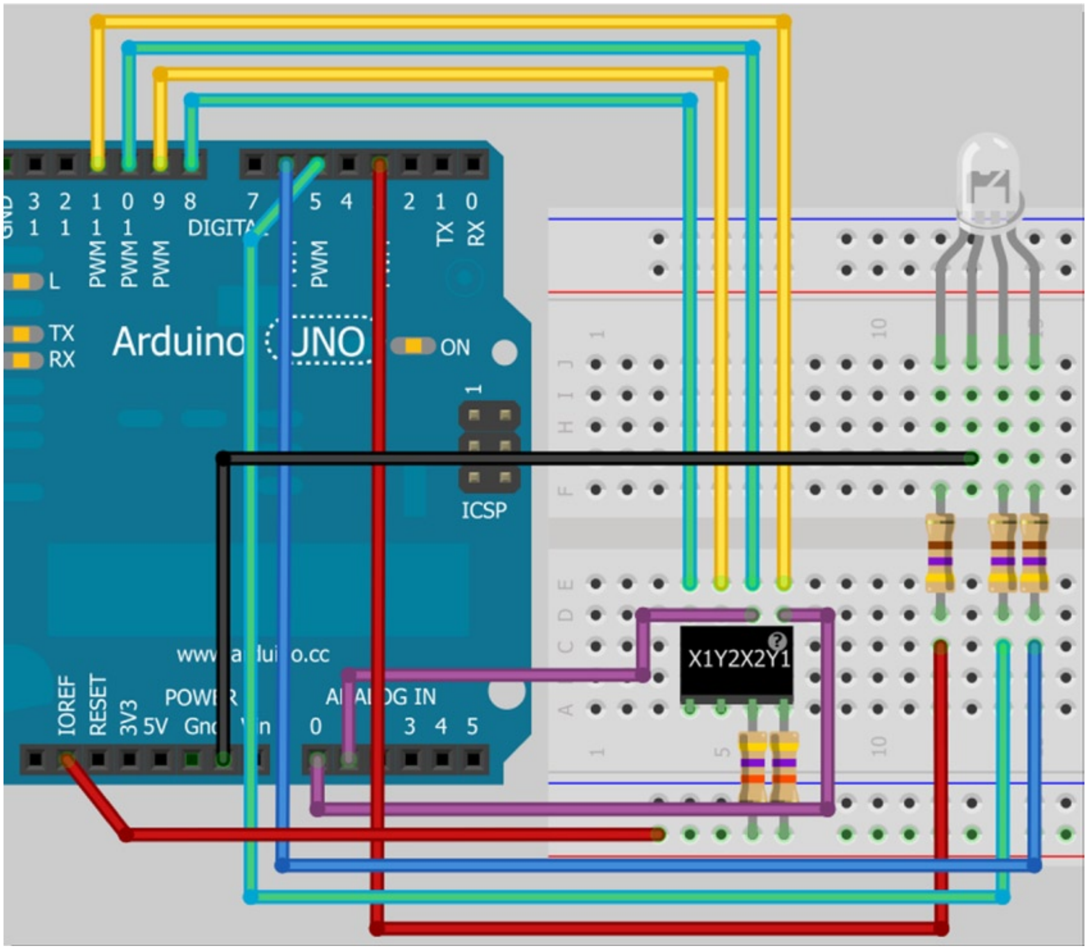


***Figure 12-6.** The circuit for Project 35 – Touch Screen Light Controller*

The ground wire goes to the common cathode pin of the LED. PWM pin 3 goes to the red anode, PWM pin 5 to the green anode, and PWM pin 6 to the blue anode. Make sure that you place current-limiting resistors on the color pins.

Enter the Code

Enter the code in Listing 12-3.

Listing 12-3. Code for Project 35

```
// Project 35

// Power connections
#define Left 8                       // Left (X1) to digital pin 8
#define Bottom 9                     // Bottom (Y2) to digital pin 9
#define Right 10                     // Right (X2) to digital pin 10
#define Top 11                       // Top (Y1) to digital pin 11

// Analog connections
#define topInput 0                   // Top (Y1) to analog pin 0
#define rightInput 1                 // Right (X2) to analog pin 1

// RGB pins
#define pinR 3
#define pinG 5
#define pinB 6

int coordX = 0, coordY = 0;
boolean ledState = true;
int red = 100, green = 100, blue = 100;

void setup()
{
  pinMode(pinR, OUTPUT);
  pinMode(pinG, OUTPUT);
  pinMode(pinB, OUTPUT);
}

void loop()
{
        if (touch()) {
                if ((coordX>0 && coordX<270) && (coordY>0 && coordY<460))
                {
                    ledState = true; delay(50);
                }

                if ((coordX>0 && coordX<270) && (coordY>510 && coordY< 880))
                {
                    ledState = false; delay(50);
                }
                if ((coordX>380 && coordX<930) && (coordY>0 && coordY<300))
               {
                  red=map(coordX, 380, 930, 0, 255);
               }

                if ((coordX>380 && coordX<930) && (coordY>350 && coordY<590))
                {
                   green=map(coordX, 380, 930, 0, 255);
                }

                if ((coordX>380 && coordX<930) && (coordY>640 && coordY<880))
                {
                   blue=map(coordX, 380, 930, 0, 255);
                }

                delay(10);
}

        if (ledState) {
                analogWrite(pinR, red);
                analogWrite(pinG, green);
                analogWrite(pinB, blue);
        }
        else {
                analogWrite(pinR, 0);
                analogWrite(pinG, 0);
                analogWrite(pinB, 0);
        }
}

// return TRUE if touched, and set coordinates to touchX and touchY
boolean touch()
{
        boolean touch = false;

                                 // get horizontal co-ordinates

        pinMode(Top, INPUT);     // Top and Bottom to high impedance
        pinMode(Bottom, INPUT);

        pinMode(Left, OUTPUT);
        digitalWrite(Left, LOW); // Set Left to low

        pinMode(Right, OUTPUT);  // Set right to +5v
        digitalWrite(Right, HIGH);


        delay(3);
        coordX = analogRead(topInput);

                                 // get vertical co-ordinates
        pinMode(Right, INPUT);   // left and right to high impedance
        pinMode(Left, INPUT);

        pinMode(Bottom, OUTPUT); // set Bottom to Gnd
        digitalWrite(Bottom, LOW);

        pinMode(Top, OUTPUT);    // set Top to +5v
        digitalWrite(Top, HIGH);


        delay(3);
        coordY = analogRead(rightInput);

                             // if co-ordinates read are less than 1000 and greater than 24
                             // then the screen has been touched
        if(coordX < 1000 && coordX > 24 && coordY < 1000 && coordY > 24) {touch = true;}

          return touch;
}
```

Project 35 – Touch Screen Controller – Code Overview

The initial defines are the same as in Projects 33 and 34 with the addition of a set of defines for the three PWM pins used to control the R, G, and B components of the RGB LED:

```
// RGB pins
#define pinR 3
#define pinG 5
#define pinB 6
```

You add a boolean called `ledState` and set it to true. This boolean will hold the state of the LEDs, i.e. true = on and false = off.

```
boolean ledState = true;
```

A set of three integers are declared and initialized with the value of 100 each:

```
int red = 100, green = 100, blue = 100;
```

These three integers will hold the separate color values for the LED. These will equate to the PWM values output from pins 3, 5, and 6.

In the main setup routine, the three LED pins you have defined are all set to outputs:

```
pinMode(pinR, OUTPUT);
pinMode(pinG, OUTPUT);
pinMode(pinB, OUTPUT);
```

In the main loop, you have an `if` statement again to check if the value returned from `touch()` is true. Inside it are more `if` statements to decide which parts of the touch screen have been pressed. The first two define the borders of the ON and OFF buttons and change the `ledState` to true if a touch is detected within the border of the ON button and to false if it is within the borders of the OFF button. A short delay is included after this to prevent false readings from the buttons.

```
if ((coordX>0 && coordX<270)
        && (coordY>0 && coordY<460))
                {ledState = true; delay(50);}

if ((coordX>0 && coordX<270)
        && (coordY>510 && coordY< 880))
                {ledState = false; delay(50);}
```

Next, you check if a touch has been detected within the borders of the slider areas for the red, green, and blue controls. If a touch has been detected, then the value in the red, green, or blue integer is changed to match which part of the slider has been touched.

```
if ((coordX>380 && coordX<930)
        && (coordY>0 && coordY<300))
                {red=map(coordX, 380, 930, 0, 255);}

if ((coordX>380 && coordX<930)
        && (coordY>350 && coordY<590))
                {green=map(coordX, 380, 930, 0, 255);}

if ((coordX>380 && coordX<930)
        && (coordY>640 && coordY<880))
                {blue=map(coordX, 380, 930, 0, 255);}
```

You accomplish this using a `map()` function, which takes five parameters. The first is the variable you are checking followed by the upper and lower ranges of the variable (all others are ignored). The final two parameters are the upper and lower ranges you wish to map the values to. In other words, you take the X coordinates within the slider area and map that value to go from 0 at the far left of the slider to 255 at the far right. By sliding your finger from left to right, you can make the relevant color component change from 0 at its dimmest, which is off, to 255 at its maximum brightness.

Finally, you have another `if` statement to set the PWM values of the R, G, and B pins to the appropriate values stored in red, green, and blue, but only if `ledState` is true. An `else` statement sets the PWM values all to 0, or off, if `ledState` is false.

```
if (ledState) {
        analogWrite(pinR, red);
        analogWrite(pinG, green);
        analogWrite(pinB, blue);
}
else {
        analogWrite(pinR, 0);
        analogWrite(pinG, 0);
        analogWrite(pinB, 0);
}
```

The remainder of the program is the `touch()` function which has already been covered.

Summary

Project 35 has introduced the concepts of buttons and slider controls controlling a touch screen. Again, using a GLCD or OLED display would give you greater control over the lighting system. Project 35 could, relatively easily, be extended to control wall-powered RGB lighting around a house with the standard light switches replaced with color OLED displays and touch screens for versatile lighting control.

Chapter 12 has shown that resistive touch screens are very easy to interface with the Arduino and use. With only a short and simple program, a touch screen and an Arduino can give provide flexibility for user control. Coupled with graphic displays, a touch screen becomes a very useful tool for controlling systems.

Subjects and Concepts covered in Chapter 12

- How to use a breakout module to make interfacing with non-standard connectors easier
- How a resistive touch screen works
- The correct power and voltage measurement cycle to obtain the X & Y co-ordinates
- The meaning of *high impedance*
- That touch screens can be overlaid onto graphic displays to create interactive buttons
- How to define a button area using coordinates and logical AND operators
- How touch screen areas can be zoned into buttons or sliders

CHAPTER 13

Temperature Sensors

The two projects in this chapter will demonstrate how to hook up analog and digital temperature sensors to an Arduino and how to get readings from them. Temperature sensors are used a lot in Arduino projects, from weather stations to brewing beer to high altitude balloon projects. You are going to take a look at two sensors, the analog LM335 sensor and the digital DS18B20.

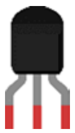
Project 36 – Serial Temperature Sensor

This project uses the LM335 analog temperature sensor. This sensor is part of the LM135 range of sensors from National Semiconductors. It has a range from –40°C to +100°C (–40°F to +212°F) and so is ideal for using in a weather station, for example.

Parts Required

The circuit and code is designed for an LM335 sensor, but you can just as easily substitute an LM135 or LM235 if you wish. You will need to adjust your code accordingly to the relevant sensor. You can substitute a standard rotary potentiometer of a similar value for the 5K ohm trim pot (potentiometer). A trim pot, or trimmer potentiometer, is simply a small potentiometer designed to adjust, or trim, part of a circuit and then, once calibrated, be left alone. Any value trimmer or potentiometer with a value between 5K ohm and 10K ohm will do.

Table 13-1. *Parts Required for Project 36*

LM335 Temperature Sensor	
5K ohm Trim Pot	
2.2K ohm Resistor	

Connect It Up

Connect everything as shown Figure 13-1.

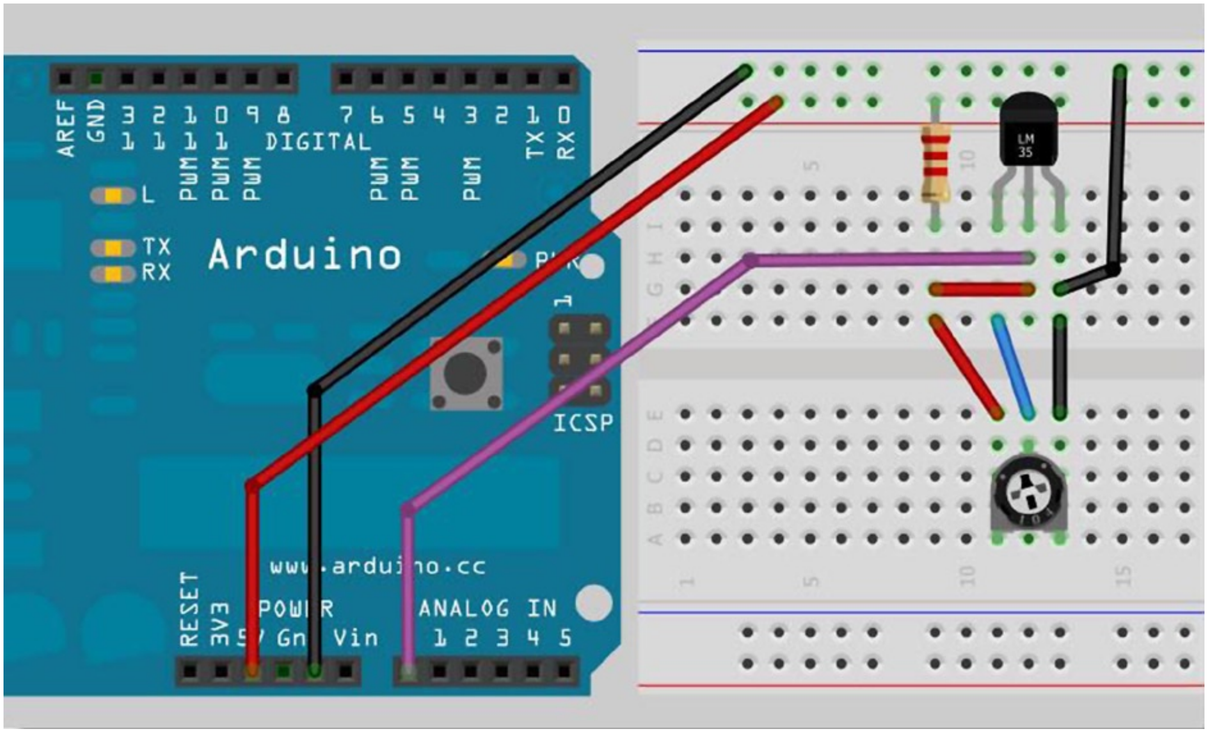


Figure 13-1. The circuit for Project 36 – Serial Temperature Sensor

If you have the flat side of the LM335 temperature sensor facing you, the left hand leg is the adjustment pin that goes to the center pin of the pot, the middle leg is the positive supply pin, and the right hand leg is the ground pin. See Figure 13-2 for the diagram from the National Semiconductor's datasheet. The center pin goes to analog pin 0 on the Arduino.

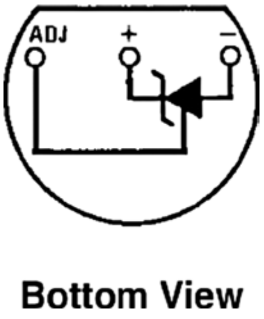


Figure 13-2. Pin diagram for the LM335 temperature sensor

Enter the Code

Enter the code in Listing 13-1.

Listing 13-1. Code for Project 36

```
// Project 36

#define sensorPin 0

float Celsius, Fahrenheit, Kelvin;
int sensorValue;

void setup() {
Serial.begin(9600);
Serial.println("Initialising.....");
}

void loop() {

  GetTemp();
  Serial.print("Celsius: ");
  Serial.println(Celsius);
  Serial.print("Fahrenheit: ");
  Serial.println(Fahrenheit);
  Serial.println();

  delay(2000);
}

void GetTemp()
{
  sensorValue = analogRead(sensorPin);                // read the sensor
  Kelvin = (((float(sensorValue) / 1023) * 5) * 100); // convert to Kelvin
  Celsius = Kelvin - 273.15;                          // convert to Celsius
  Fahrenheit = (Celsius * 1.8) +32;                   // convert to Fahrenheit
}
```

Enter the code and upload it to your Arduino. Once the code is running, open the serial monitor and make sure your baud rate is set to 9600. You will see the temperature displayed in both Fahrenheit and Celsius. The temperature may look incorrect to you. This is where the trimmer comes in; you must first calibrate your sensor. For proper calibration, you should be using a mixture of ice and water that has had time to stabilize at the temperature at which the ice melts. Place chipped or crushed ice in a Styrofoam cup (to limit outside influences) and either let it thaw until partially melted (in a fridge), or add some clean (ideally, distilled) water. The mixture should be at least 50 percent ice. Stir to mix well, and wait at least several minutes to make sure the water has had a chance to cool to the just freezing point. Then put the sensor (protected by a thin plastic bag with as little air as possible) in the water-ice slurry and wait until the reading stops changing. Then adjust the trimmer for a reading of 0°C. Now turn your trimmer or pot until the reading in the serial monitor shows the correct temperature. Your sensor is now calibrated. If you are using heat shrink tubing for the purpose of waterproofing the sensor, you should use a dual wall or filled-core heat type of heat-shrink tubing.

You can remove the trimmer part of the circuit and it will run just fine. However, the temperature will be a close approximation, within 1°C. How the sensor works is not important (and is in fact pretty complicated) so I will simply look at how the code works for this project. If you do want to learn more about how this kind of sensor works, read *The Art of Electronics* by Paul Horowitz and Winfield Hill. This book is often referred to as the "Electronics Bible."

Project 36 – Serial Temperature Sensor – Code Overview

The code for this project is short and simple. You start off by defining the sensor pin. In this case, you are using analog pin 0.

```
#define sensorPin 0
```

You then need some variables to store the temperatures in Celsius, Fahrenheit, and Kelvin. Since you want to be able to represent fractions of a degree, you use variables of type float.

```
float Celsius, Fahrenheit, Kelvin;
```

Then you create an integer to hold the value read from the analog pin:

```
int sensorValue;
```

The setup function begins serial communication at a baud rate of 9600:

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

Then you display "Initializing....." to show the program is about to start:

```
Serial.println("Initializing.....");
```

In the main program loop, you call the `GetTemp()` function that reads the temperature from the sensor and converts it to Celsius and Fahrenheit. Then it prints out the temperatures in the serial monitor window.

```
GetTemp();
Serial.print("Celsius: ");
Serial.println(Celsius);
Serial.print("Fahrenheit: ");
Serial.println(Fahrenheit);
Serial.println();
```

Now you create the `GetTemp()` function:

```
void GetTemp()
```

First, the sensor is read and the value stored in `sensorValue:`

```
sensorValue = analogRead(sensorPin); // read the sensor
```

The output from the sensor is in Kelvin, with every 10mV being one K. Kelvin starts at zero degrees K when the temperature is at absolute zero, or the lowest possible temperature in the universe. So at absolute zero, the sensor will be outputting 0 volts. According to the datasheet, the sensor can be calibrated by checking that the voltage from the sensor is 2.98 volts at 25°C. To convert from Kelvin to Celsius, you simply subtract 273.15 from the Kelvin temperature to get the Celsius temperature. So 25°C in Kelvin is 298.15, and if every degree is 10mV, then you simply move the decimal point two places to the left to get the voltage at that temperature, which is indeed 2.98 volts.

So, to get the temperature in Kelvin, you read the value from the sensor, which will range from 0 to 1,023, and then divide it by 1,023, and multiply that result by 5. This will effectively map the range from 0 to 5 volts. As each degree K is 10mV, you then multiply that result by 100 to get degrees K.

```
Kelvin = (((float(sensorValue) / 1023) * 5) * 100); // convert to Kelvin
```

The sensor value is an integer so it is cast to a float to ensure the result is a float, too.

Now that you have your reading in K, it's easy to convert to Celsius and Fahrenheit. To convert to Celsius, subtract 273.15 from the temperature in K:

```
Celsius = Kelvin - 273.15;                          // convert to Celsius
```

And to convert to Fahrenheit, multiply the Celsius value by 1.8 and add 32:

```
Fahrenheit = (Celsius * 1.8) +32;                   // convert to Fahrenheit
```

The LM135 range of sensors is nice in that they can be easily calibrated so you can ensure an accurate reading every time. They are also cheap so you can purchase a whole bunch of them and obtain readings from different areas of your house, or the internal and external temperatures from a high altitude balloon project.

Other analog sensors can be used. You may find that the third pin on some sensors, which is the adj (adjustment) pin in the LM335, is the temperature output pin. Therefore, you should use this third pin to read the temperature instead of the supply voltage pin. Calibrating these types of sensors can be done easily in software.

You will next look at a digital temperature sensor. By far the most popular of these types is the DS18B20 from Dallas Semiconductor (Maxim).

Project 37 – One-Wire Digital Temperature Sensor

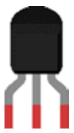
You are now going to take a look at the DS18B20 digital temperature sensor. These sensors send the temperature as a serial data stream over a single wire, which is why the protocol is called one-wire. Each sensor also has a unique serial number, allowing you to query different sensors using its ID number. As a result, you can connect many sensors on the same data line. This makes them very popular to use with an Arduino because an almost unlimited amount of temperature sensors can be daisy chained together and all connected to just one pin on the Arduino. The temperature range is also wide at –55°C to +125°C.

You'll use two sensors in this project to demonstrate not only how to connect and use this type of sensor but also how to daisy chain two or more together.

Parts Required

You will need two DS18B20 sensors in the TO-92 format (this just means it has three pins and so can easily be inserted into a breadboard or soldered onto a PCB). Some are marked DS18B20+, which means they are lead free.

Table 13-2. *Parts Required for Project 37*

2 × DS18B20 Temperature Sensor	
4.7K ohm Resistor	

Connect It Up

Connect everything as shown in Figure 13-3.

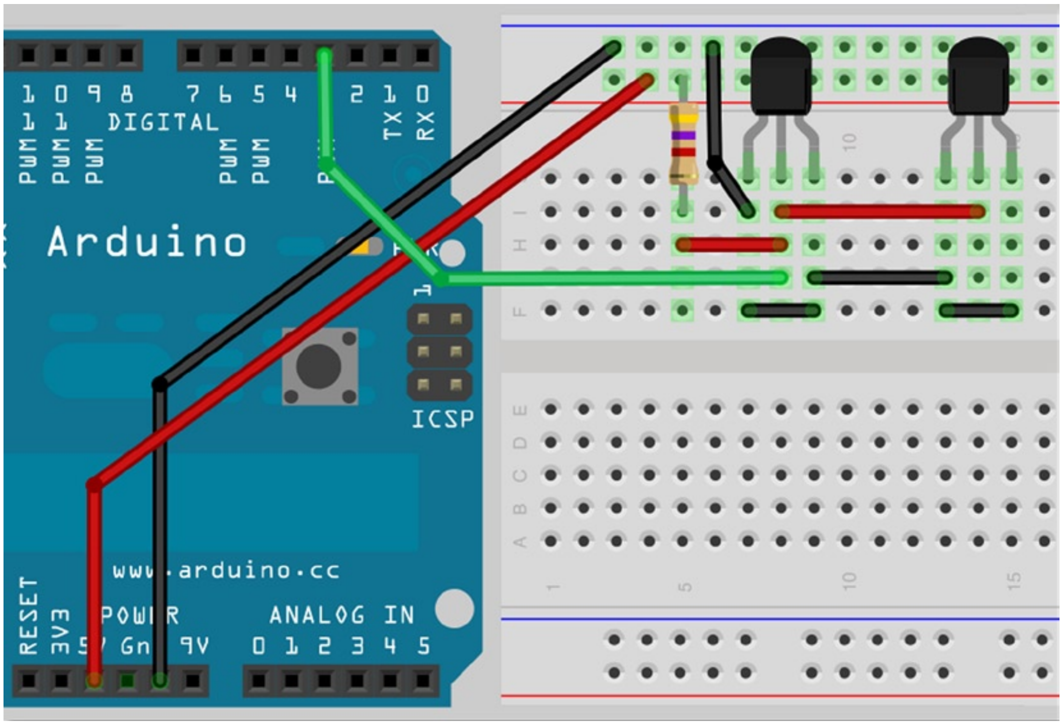


***Figure 13-3.** The circuit for Project 37 – One-Wire Digital Temperature Sensor*

I am going to do the code in two parts. The first part will find out the addresses of the two sensors. Once you know those addresses, you will move onto part two, in which the addresses will be used to obtain the temperatures directly from the sensors.

Figure 13-4 shows the pin diagram for the DS18B20 sensor. The device can be powered in two separate ways. You can either provide power on the V_{DD} pin in the range from 3.3 to 5V, or you can use what is known as "parasitic mode" and "steal" power from the one-wire bus via the DQ pin when the bus is high. In this case, the V_{DD} pin is connected to ground.

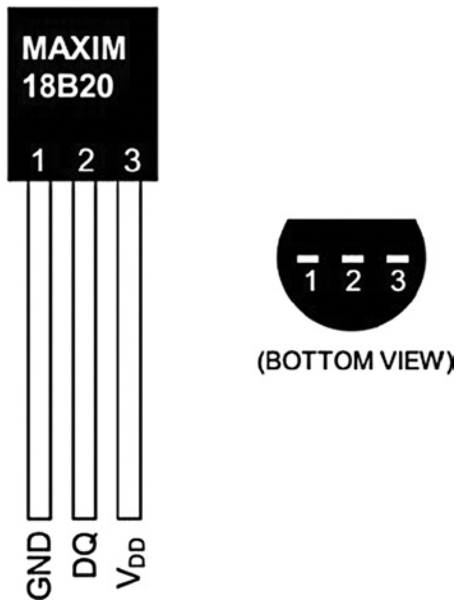


Figure 13-4. *The pin diagram for the DS18B20*

One disadvantage of the one-wire bus is that it depends on precise timing. It is vital, therefore, that your processor is only communicating with the one-Wire device during communication and is not carrying out other tasks at the same time (via interrupts, for example).

Enter the Code

Before you enter the code, you need to download and install two libraries. The first is the OneWire library. Download it from `www.pjrc.com/teensy/td_libs_OneWire.html` and unzip it. The OneWire library was first written by Jim Studt, with further improvements by Robin James, Paul Stoffregen, and Tom Pollard. This library can be used to communicate with any one-wire device. Place it in the "libraries" folder of your Arduino installation.

Next, download and install the DallasTemperature library from `http://milesburton.com/index.php?title=Dallas_Temperature_Control_Library` and again install it in the "libraries" folder. The directory, after it is unzipped, has spaces in the name. The Arduino won't accept this, so change the folder name to DallasTemperature before placing it in the libraries folder. This library is an offshoot of the OneWire library and was developed by Miles Burton, with improvements by several other contributors to the project afterward. This project is based on code from the examples included with this library.

Once you have installed both libraries, restart your Arduino IDE and then enter the code from the program in Listing 13-2.

Listing 13-2. Code for Project 37 (Part 1)

```
// Project 37 - Part 1

#include <OneWire.h>
#include <DallasTemperature.h>
```

```
// Data line goes to digital pin 3
#define ONE_WIRE_BUS 3

// Setup a oneWire instance to communicate with any
// OneWire devices (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device addresses
DeviceAddress insideThermometer, outsideThermometer;

void setup()
{
  // start serial port
  Serial.begin(9600);

  // Start up the library
  sensors.begin();

  // locate devices on the bus
  Serial.print("Locating devices...");
  Serial.print("Found ");
  Serial.print(sensors.getDeviceCount(), DEC);
  Serial.println(" devices.");

  if (!sensors.getAddress(insideThermometer, 0))
 {
    Serial.println("Unable to find address for Device 0");
  }

  if (!sensors.getAddress(outsideThermometer, 1))
 {
    Serial.println("Unable to find address for Device 1");
  }

  // print the addresses of both devices
  Serial.print("Device 0 Address: ");
  printAddress(insideThermometer);
  Serial.println();

  Serial.print("Device 1 Address: ");
  printAddress(outsideThermometer);
  Serial.println();
  Serial.println();
}
```

```
// function to print a device address
void printAddress(DeviceAddress deviceAddress)
{
  for (int i = 0; i < 8; i++)
  {
    // zero pad the address if necessary
    if (deviceAddress[i] < 16) Serial.print("0");
    Serial.print(deviceAddress[i], HEX);
  }
}

// function to print the temperature for a device
void printTemperature(DeviceAddress deviceAddress)
{
  float tempC = sensors.getTempC(deviceAddress);
  Serial.print("Temp C: ");
  Serial.print(tempC);
  Serial.print(" Temp F: ");
  Serial.print(DallasTemperature::toFahrenheit(tempC));
}

// main function to print information about a device
void printData(DeviceAddress deviceAddress)
{
  Serial.print("Device Address: ");
  printAddress(deviceAddress);
  Serial.print(" ");
  printTemperature(deviceAddress);
  Serial.println();
}

void loop()
{
  // call sensors.requestTemperatures() to issue a global temperature
  // request to all devices on the bus
  Serial.print("Requesting temperatures...");
   sensors.requestTemperatures();
  Serial.println("DONE");
  delay(750);

  // print the device information
  printData(insideThermometer);
  printData(outsideThermometer);
  Serial.println();
  delay(1000);
}
```

Once the code has been uploaded, open up the serial monitor. You will have a display similar to this:

```
Locating devices...Found 2 devices.

Device 0 Address: 28CA90C202000088
Device 1 Address: 283B40C202000093

Requesting temperatures...DONE
Device Address: 28CA90C202000088 Temp C: 31.00 Temp F: 87.80
Device Address: 283B40C202000093 Temp C: 25.31 Temp F: 77.56
```

The program gives you the two unique ID numbers of the DS18B20 sensors you are using. You can find out which sensor is which by varying the temperature between the two. I held onto the right hand sensor for a few seconds and, as you can see, the temperature increased on that one. This tells me that the right sensor has address 28CA90C202000088 and the left one has address 283B40C202000093. The addresses of your sensors will obviously differ. Write them down or copy and paste them into your text editor.

Now that you know the ID numbers of the two devices, you can move onto part two. Enter the code from Listing 13-3.

Listing 13-3. Code for Project 37 (Part 2)

```
// Project 37 - Part 2

#include <OneWire.h>
#include <DallasTemperature.h>

// Data wire is plugged into pin 3 on the Arduino
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12

// Setup a oneWire instance to communicate with any
// OneWire devices (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device addresses - replace with your sensors addresses
DeviceAddress insideThermometer = { 0x28, 0xCA, 0x90, 0xC2, 0x2, 0x00, 0x00, 0x88 };
DeviceAddress outsideThermometer = { 0x28, 0x3B, 0x40, 0xC2, 0x02, 0x00, 0x00, 0x93 };

void setup()
{
  // start serial port
  Serial.begin(9600);

  // Start up the library
  sensors.begin();

  Serial.println("Initialising...");
  Serial.println();
// set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
  sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
}

// function to print the temperature for a device
void printTemperature(DeviceAddress deviceAddress)
{
  float tempC = sensors.getTempC(deviceAddress);
  Serial.print(" Temp C: ");
  Serial.print(tempC);
  Serial.print("  Temp F: ");
  Serial.println(DallasTemperature::toFahrenheit(tempC));
}

  // Measure and then print the temperatures to the serial output port
void loop()
{
  sensors.requestTemperatures();
  delay(750);
  Serial.print("Inside Temp:");
  printTemperature(insideThermometer);
  Serial.print("Outside Temp:");
  printTemperature(outsideThermometer);
  Serial.println();
  delay(3000);
}
```

Replace the two sensor addresses with those you discovered using the code from part one, and then upload this code. Open the serial monitor and you will get a readout like this:

```
Initialising...

Inside Temp: Temp C: 24.25  Temp F: 75.65
Outside Temp: Temp C: 19.50  Temp F: 67.10

Inside Temp: Temp C: 24.37  Temp F: 75.87
Outside Temp: Temp C: 19.44  Temp F: 66.99

Inside Temp: Temp C: 24.44  Temp F: 75.99
Outside Temp: Temp C: 19.37  Temp F: 66.87
```

If you solder the outside sensor to a long twin wire (solder pins 1 and 3 together for one wire and pin 2 for the second wire), and then waterproof it by sealing it in heat-shrink tubing, it can be placed outside to gather external temperatures. The other sensor can obtain the internal temperature.

Project 37 – 1-Wire Digital Temperature Sensor – Code Overview

First the two libraries are included:

```
#include <OneWire.h>
#include <DallasTemperature.h>
```

Then the digital pin you will be using for reading the data from the sensors is defined

```
#define ONE_WIRE_BUS 3
```

followed by a definition for the precision required, in bits

```
#define TEMPERATURE_PRECISION 12
```

The precision can be set between 9 and 12 bits resolution. This corresponds to increments of 0.5°C, 0.25°C, 0.125°C, and 0.0625°C, respectively. The default resolution is 12 bit. The maximum resolution of 12 bit gives the smallest temperature increment, but at the expense of speed. At maximum resolution, the sensor takes 750ms to convert the temperature. At 11 bit, it is half that at 385ms; 10 bit is half again at 187.5ms; and finally 9 bit is half again at 93.75ms. 750ms is fast enough for most purposes. However, if you need to take several temperature readings a second for any reason, then 9 bit resolution would give the fastest conversion time.

Next, you create an instance of a OneWire object and call it `oneWire`:

```
OneWire oneWire(ONE_WIRE_BUS);
```

You also create an instance of a DallasTemperature object, call it `sensors`, and pass it a reference to the object called `oneWire`:

```
DallasTemperature sensors(&oneWire);
```

Next, you need to create the arrays that will hold the sensor addresses. The DallasTemperature library defines variables of type DeviceAddress (which are just byte arrays of eight elements). We create two variables of type DeviceAddress, call them `insideThermometer` and `outsideThermometer` and assign the addresses found in part one to the arrays.

Simply take the addresses you found in part one, break them up into units of two hexadecimal digits and add 0x (to tell the compiler it is a hexadecimal number and not standard decimal), and separate each one by a comma. The address will be broken up into eight units of two digits each.

```
DeviceAddress insideThermometer = { 0x28, 0xCA, 0x90, 0xC2, 0x2, 0x00, 0x00, 0x88 };
DeviceAddress outsideThermometer = { 0x28, 0x3B, 0x40, 0xC2, 0x02, 0x00, 0x00, 0x93 };
```

In the setup function, you begin serial communications at 9600 baud:

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

Next, the communication with the sensors object is started using the `begin()` command:

```
sensors.begin();
```

You print "Initializing…" to show the program has started, followed by an empty line:

```
Serial.println("Initialising...");
Serial.println();
```

Next, you set the resolution of each sensor using the setResolution command. This command requires two parameters with the first being the device address and the second being the resolution. You have already set the resolution at the start of the program to 12 bits.

```
sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
```

Next, you create a function called `printTemperature()` that will print out the temperature in both degrees Celsius and Fahrenheit from the sensor address set in its single parameter:

```
void printTemperature(DeviceAddress deviceAddress)
```

Next, you use the `getTempC()` command to obtain the temperature in Celsius from the device address specified. You store the result in a float called `tempC`.

```
float tempC = sensors.getTempC(deviceAddress);
```

You then print that temperature

```
Serial.print(" Temp C: ");
Serial.print(tempC);
```

followed by the temperature in Fahrenheit

```
Serial.print("  Temp F: ");
Serial.println(DallasTemperature::toFahrenheit(tempC));
```

You use :: to access the `toFahrenheit` function that is inside the DallasTemperature library. This converts the value in `tempC` to Fahrenheit.

In the main loop, you simply request the temperature using the sensors.requestTemperatures() function, followed by a delay of 750ms to allow the sensor to read the temperature and transmit it to the Arduino. Then it is safe to call the `printTemperature()` function twice, passing the address of the inside and then the outside sensor each time followed by a three second delay. Without the delays, the Arduino moves more quickly than the sensors can respond and you miss some of the data they send back:

```
sensors.requestTemperatures();
delay(750);

Serial.print("Inside Temp:");
printTemperature(insideThermometer);
Serial.print("Outside Temp:");
printTemperature(outsideThermometer);
Serial.println();
delay(3000);
```

I recommend you try out the various examples that come with the DallasTemperature library as these will give a greater understanding of the various functions available within the library. I also recommend that you read the datasheet for the DS18B20. This sensor can also have alarms set inside it to trigger when certain temperature conditions are met that could be useful for sensing conditions that are too hot or cold.

The DS18B20 is a very versatile sensor that has a wide temperature-sensing range and has the advantage over an analog sensor in that many can be daisy chained along the same data line; thus, only one pin is needed no matter how many sensors you have.

Next, you are going to take a look at a totally different kind of sensor that uses sound waves.

Summary

In this chapter, you have worked through two simple projects that showed you how to connect analog and digital temperature sensors to your Arduino. The projects showed you the basics of reading data from each sensor and displaying it in the serial monitor. Once you know how to do that, it's a relatively easy step to get that data displayed on an LCD or LED dot-matrix display.

Knowing how to obtain temperature readings from sensors opens up a whole new range of projects to the Arduino enthusiast. You will revisit temperature sensors later in the book when they are put to practical use in Chapter 17.

Subjects and Concepts Covered in Chapter 13

- How to wire up an analog temperature sensor to an Arduino
- How to use a trimmer to calibrate an LM135 series sensor
- How to convert the voltage from the sensor to Kelvin
- How to convert Kelvin to Celsius and Celsius to Fahrenheit
- How to waterproof sensors using heat-shrink tubing
- How to wire up a one-wire temperature sensor to an Arduino
- That one-wire devices can be daisy chained
- That one-wire devices have unique ID numbers
- How to set the resolution of a DS18B20 sensor
- That higher resolutions equal slower conversion speeds

CHAPTER 14

Ultrasonic Rangefinders

You are now going to take a look at a different kind of sensor, one that is used frequently in robotics and industrial applications. The ultrasonic rangefinder is designed to detect the distance to an object by bouncing an ultrasonic sound pulse off it and listening for the time it takes for the pulse to return. You are going to use a popular type of ultrasonic rangefinder, from the Maxbotix LV-MaxSonar range of sensors, but the concepts learned in this chapter can be applied to any other make of ultrasonic range finder. You'll learn the basics of connecting the sensor to the Arduino first, then move on to putting the sensor to use.

Project 38 – Simple Ultrasonic Rangefinder

The LV-MaxSonar ultrasonic rangefinder comes in EZ1, EZ2, EZ3, and EZ4 models. All have the same range, but they come in progressively narrower beam angles to allow you to match your sensor to your particular application. I used an EZ3 in the creation of this chapter, but you can choose any model.

Parts Required

Table 14-1. *Parts required for Project 38*

LV-MaxSonar EZ3*	
100µF Electrolytic Capacitor	
100 ohm Resistor	

**or any from the LV range (image courtesy of Sparkfun)*

Connect It Up

Connect everything as shown in Figure 14-1.

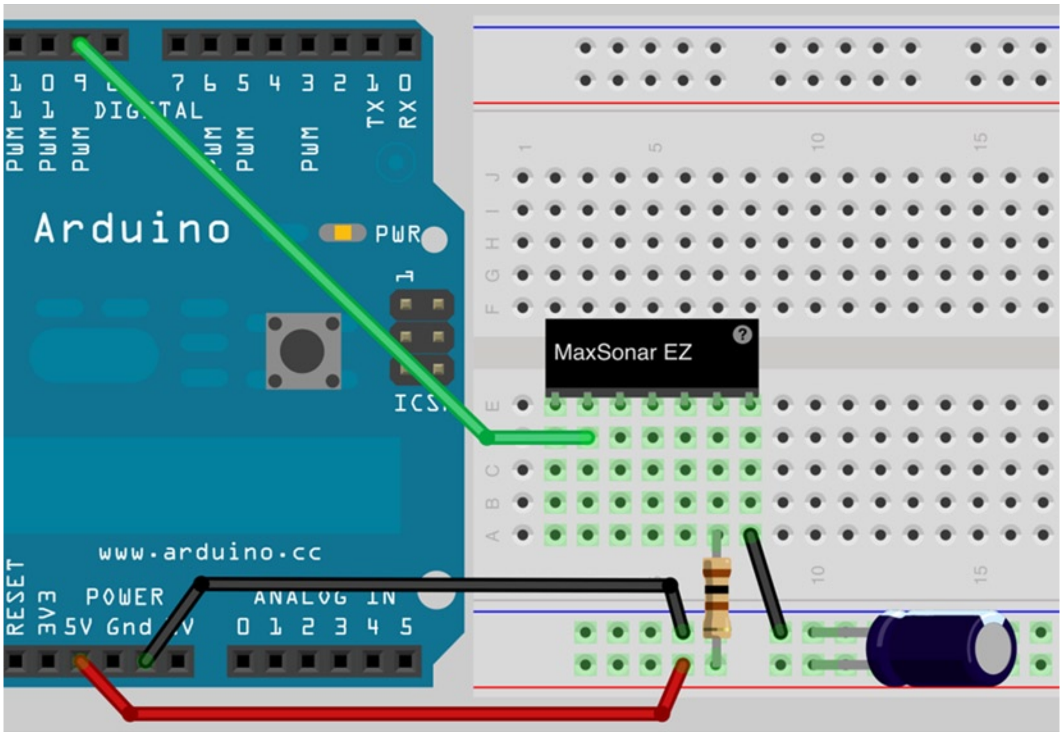


Figure 14-1. *The circuit for Project 38—Simple Ultrasonic Rangefinder*

As Fritzing (the software used to create the breadboard diagrams in this book) does not have a LV-MaxSonar in its parts library, I have used a “mystery part” as a substitute. Connect the +5v and ground to the two power rails on the breadboard. Place a 100µF electrolytic capacitor across the power rails, ensuring you get the longer leg connected to the +5v and the shorter leg (also with a white band and minus signs across it) to the ground rail. Then connect a jumper wire between ground and the Gnd pin on the sensor. It is essential that you get the polarity right as it can explode if connected the wrong way! Then connect a 100 ohm resistor between the +5v rail and the +5v pin on the sensor. Finally, connect a wire between the PW pin on the sensor and digital pin 9.

Enter the Code

Once you have checked that your wiring is correct, enter the code in Listing 14-1 and upload it to your Arduino.

Listing 14-1. Code for Project 38

```
// Project 38

#define sensorPin 9

long pwmRange, inch, cm;

void setup() {
  // Start serial communications
  Serial.begin(115200);
  pinMode(sensorPin, INPUT);
}
```

```
void loop() {
pwmRange = pulseIn(sensorPin, HIGH);

  // 147uS per inch according to datasheet
  inch = pwmRange / 147;
  // convert inch to cm
  cm = inch * 2.54;

  Serial.print(inch);
  Serial.print(" inches    ");
  Serial.print(cm);
  Serial.println(" cm");
}
```

Once you have uploaded the code, power the Arduino down for a second. Then make sure that your ultrasonic sensor is still and pointing at something that is not moving. Putting it flat on a table and pointing it at your ceiling will work best. Make sure that nothing is near the sensor when you power the Arduino back up. When the device is first powered up, it runs through a calibration routine for the first read cycle. Make sure nothing is moving around in its beam while this takes place, otherwise you will get inaccurate readings. This information is then used to determine the range of objects in the line of sight of the sensor. Measure the distance between the sensor and the ceiling, and this distance (roughly) will be output from the serial monitor when you open it up. If the distance is inaccurate, power the Arduino down and back up, allowing the device to calibrate without obstacles. By moving the sensor around or by raising and lowering your hand over the sensor, the distance to the object placed in its path will be displayed on the serial monitor.

Project 38 – Simple Ultrasonic Rangefinder – Code Overview

Again, you have a short and simple piece of code to use with this sensor. First, you start of by defining the pin you will use to detect the pulse. You are using digital pin 9:

```
#define sensorPin 9
```

Then three variables of type long are declared:

```
long pwmRange, inch, cm;
```

These will be used to store the range read back from the sensor, the range converted into inches, and then into centimeters, respectively.

In the setup routine, you simply begin serial communications at 115200 baud and set the sensor pin to an input:

```
Serial.begin(115200);
pinMode(sensorPin, INPUT);
```

In the main loop, you start by reading the pulse from the sensor pin and storing it in pwmRange:

```
pwmRange = pulseIn(sensorPin, HIGH);
```

To accomplish this, you use the new command, `pulseIn`. This new command is tailor-made for this use, as it is designed to measure the length of a pulse, in microseconds, on a pin. The PW pin of the sensor sends a HIGH signal when the ultrasonic pulse is sent from the device, and then a LOW signal once that pulse is received back. The time in-between the pin going high and low will give you the distance, after conversion. The pulseIn command requires two parameters. The first is the pin you want to listen to and the second is either a HIGH or a LOW to define at what state the pulseIn command will commence timing the pulse. In your case, you have this set to HIGH, so as soon as the sensor pin goes HIGH, the pulseIn command will start timing; once it goes LOW, it will stop timing and then return the time in microseconds.

According to the datasheet for the LV-MaxSonar range of sensors, the device will detect distances from 0 to 254 inches (6.45 meters) with distances below 6 inches being output as 6 inches. Each 147µS (micro-seconds) equates to one inch. So, to convert the value returned from the pulseIn command to inches, you simply need to divide it by 147. This value is then stored in inch.

```
inch = pwmRange / 147;
```

Next, that value is multiplied by 2.54 to give you the distance in centimeters:

```
cm = inch * 2.54;
```

Finally, the values in inches and centimeters are printed to the serial monitor:

```
Serial.print(inch);
Serial.print(" inches    ");
Serial.print(cm);
Serial.println(" cm");
```

Project 38 – Simple Ultrasonic Rangefinder – Hardware Overview

The new component introduced in this project is the ultrasonic rangefinder. This device uses ultrasound, which is a very high frequency sound above the upper limit of human hearing. In the case of the MaxSonar, it sends a pulse at 42KHz (the average human has an upper hearing limit of around 20KHz). A pulse of ultrasonic sound is sent out by the device from a transducer and is then picked up again, by the same transducer when it reflects off an object. By calculating the time it takes for the pulse to return, you can work out the distance to the reflected object (See Figure 14-2). Sound waves travel at the speed of sound which, in dry air at 20°C (68°F) is 343 meters per second, or 1,125 feet per second. Knowing this, you can work out the speed, in microseconds, that the sound wave takes to return to the sensor. As it happens, the datasheet tells you that every inch takes 147µS for the pulse to return. So taking the time in microseconds and dividing it by 147 gives us the distance in inches, and then you can convert that to centimeters, if necessary.

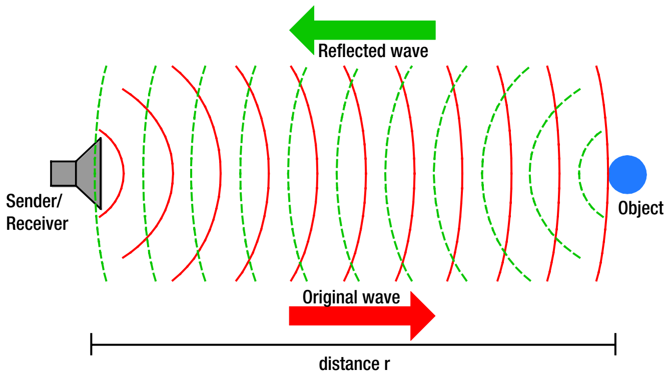


***Figure 14-2.** The principle of sonar or radar distance measurement (Image by Georg Wiora)*

This principle is also called SONAR (sound navigation and ranging) and is used in submarines to detect distances to other marine craft or nearby hazards. It is also used by bats to detect their prey.

The MaxSonar devices have three ways to read the data from the sensor. One is an analog input, the second is a PWM input, and the final one is a serial interface. The PWM input is probably the easiest to use with the most reliable data, hence this is what I have used here. Feel free to research and use the other two pins if you wish, although there will be no real benefit from doing so unless you specifically need to have an analog or serial data stream.

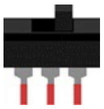
Now that you know how the sensor works, let's put it to a practical use and make an ultrasonic tape measure or distance display.

Project 39 – Ultrasonic Distance Display

Now you're going to use the ultrasonic sensor to create a (fairly) accurate distance display. You are going to use the MAX7219 LED driver IC used back in Chapter 7 to display the distance measured. Instead of a dot-matrix display, however, you're going to use what the MAX7219 was designed for, a set of 7-segment LED displays.

Parts Required

Table 14-2. *Parts required for Project 39*

LV-MaxSonar EZ3*	
100µF Electrolytic Capacitor	
100 ohm Resistor	
2 x 10K ohm Resistor	
Switch	
5 × 7-Segment LED displays (Common Cathode)	
MAX7219 LED Driver IC	

**or any from the LV range (image courtesy of Sparkfun)*

The switch must be the single-pole, single-throw type (SPST). A single-pole single-throw switch only has two terminals, which are connected in one switch position and unconnected in the other position. You will use one of those positions to switch the display between inches and centimeters. The 7-segment LED displays must be the common cathode type. Make sure to get the datasheet for the type you purchase so that you can ascertain how to connect it, as it may differ from mine.

Connect It Up

Connect everything as shown in Figure 14-3.

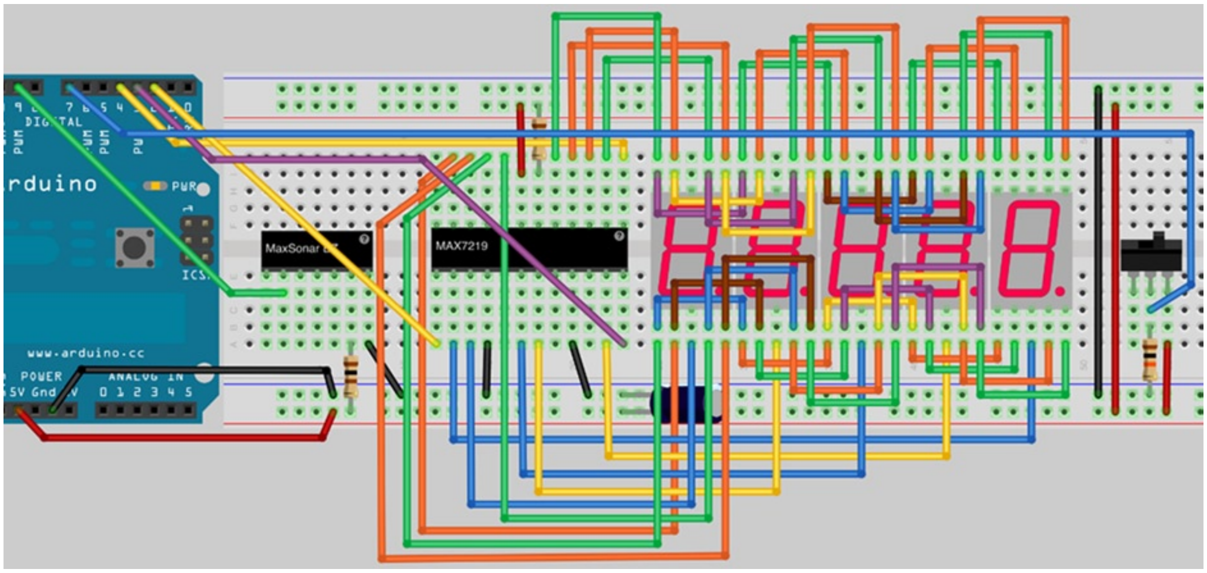


Figure 14-3. *The circuit for Project 39—Ultrasonic Distance Display*

This circuit is pretty complex, so below I have also provided a table of pins (Table 14-3) for the Arduino, Max7219, and 7-Segment display so you can match them to the diagram. The displays I used had the code 5101AB, but any common cathode 7-segment display will work. Make sure the pins are across the top and bottom of the display and not along the sides, otherwise you will not be able to insert them into a breadboard.

Table 14-3. *Pin Outs Required for Project 39*

Arduino	MaxSonar	MAX7219	7-Segment	Other
Digital Pin 2		Pin 1 (DIN)		
Digital Pin 3		Pin 12 (LOAD)		
Digital Pin 4		Pin 13 (CLK)		
Digital Pin 7				Switch
Digital Pin 9	PW	Pin 4 (Gnd)		Gnd
		Pin 9 (Gnd)		Gnd
		Pin 18 (ISET)		VDD via 10KΩ Resistor
		Pin 19 (VDD)		+5 volts
		Pin 2 (DIG 0)	Common on Display 0	
		Pin 11 (DIG 1)	Common on Display 1	

(*continued*)

Table 14-3. (*continued*)

Arduino	MaxSonar	MAX7219	7-Segment	Other
		Pin 6 (DIG 2)	Common on Display 2	
		Pin 7 (DIG 3)	Common on Display 3	
		Pin 3 (DIG 4)	Common on Display 4	
		Pin 14	SEG A	
		Pin 16	SEG B	
		Pin 20	SEG C	
		Pin 23	SEG D	
		Pin 21	SEG E	
		Pin 15	SEG F	
		Pin 17	SEG G	
		Pin 22	SEG DP	

Once you have connected the MAX7219 to the SEG A-G and DP pins of the first 7-segment display, i.e. the one nearest the chip (see Figure 14-4), connect the SEG pins on the first display to the second, and then the second to the third, and so on. All of the SEG pins are tied together on each display with the ground pins separate and going to the relevant DIG pins on the MAX7219. Make sure you read the datasheet for your 7-segment display as its pins may differ from mine.

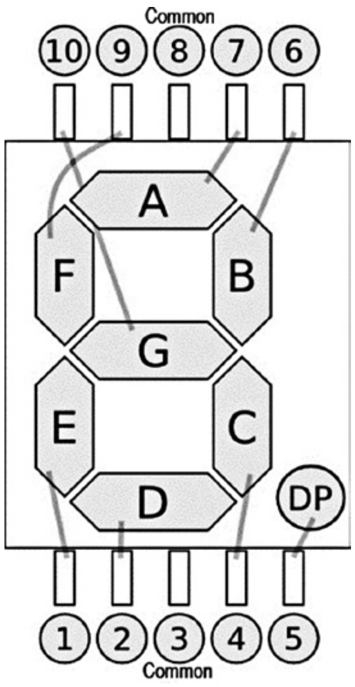


Figure 14-4. *A typical common cathode 7-segment LED display with pin assignments (image courtesy of Jan-Piet Mens)*

The MaxSonar is connected the same way as before, except for the PW pin going to digital pin 9 instead of 3. Finally, digital pin 7 goes to the toggle switch.

Note that you may need to use an external power supply for this project if you find it is erratic—it may draw too much power from the USB port.

Enter the Code

Once you have checked that your wiring is correct, power up the Arduino and enter the code in Listing 14-2, then upload it to your Arduino. Make sure you have LedControl.h in your libraries folder (see Chapter 7 for instructions).

Listing 14-2. Code for Project 39

```
// Project 39

#include "LedControl.h"

#define sensorPin 9
#define switchPin 7
#define DataIn 2
#define CLK 4
#define LOAD 3
#define NumChips 1
#define samples 5.0

float pwmRange, averageReading, inch, cm;
LedControl lc=LedControl(DataIn,CLK,LOAD,NumChips);

void setup() {
        // Wakeup the MAX7219
        lc.shutdown(0,false);
        // Set it to medium brightness
        lc.setIntensity(0,8);
        // clear the display
        lc.clearDisplay(0);
        pinMode(sensorPin, INPUT);
        pinMode(switchPin, INPUT);
}

void loop() {
        averageReading = 0;
        for (int i = 0; i<samples; i++) {
                pwmRange = pulseIn(sensorPin, HIGH);
                averageReading += pwmRange;
        }

        averageReading /= samples;
        // 147uS per inch according to datasheet
        inch = averageReading / 147;
        // convert inch to cm
        cm = inch * 2.54;
        if (digitalRead(switchPin)) {
                displayDigit(inch);
        }
        else {
                displayDigit(cm);
        }
}

void displayDigit(float value) {
        int number = value*100;
        lc.setDigit(0,4,number/10000,false);        // 100s digit
        lc.setDigit(0,3,(number%10000)/1000,false); // 10s digit
        lc.setDigit(0,2,(number%1000)/100,true);    // first digit with DP on
        lc.setDigit(0,1,(number%100)/10,false);     // 10th digit
        lc.setDigit(0,0,number%10,false);           // 100th digit
}
```

Project 39 – Ultrasonic Distance Display – Code Overview

The project starts by including the LedControl.h library:

```
#include "LedControl.h"
```

You then define the pins you will require for the sensor and the MAX7219 chip:

```
#define sensorPin 9
#define switchPin 7
#define DataIn 2
#define CLK 4
#define LOAD 3
#define NumChips 1
```

The sensor readings are smoothed out using a simple running average algorithm, so you need to define how many samples you take to do that:

```
#define samples 5.0
```

You will be using this number with floats later, so to avoid errors, the number is defined as 5.0 rather than a 5 to make sure it is forced as a float and not an int.

Next, the floats for the sensor are declared as in Project 38, but with the addition of `averageReading`, which you will use later on in the program:

```
float pwmRange, averageReading, inch, cm;
```

You create a `LedControl` object and set the pins used and the number of chips:

```
LedControl lc=LedControl(DataIn,CLK,LOAD,NumChips);
```

As in Project 21, you ensure the display is enabled, the intensity is set to medium, and the display is cleared and ready for use:

```
lc.shutdown(0,false);
lc.setIntensity(0,8);
lc.clearDisplay(0);
```

The pins for the sensor and the switch are both set to INPUT:

```
pinMode(sensorPin, INPUT);
pinMode(switchPin, INPUT);
```

Then you reach the main loop. First the variable `averageReading` is set to zero:

```
averageReading = 0;
```

Next, a `for` loop runs to collect the samples from the sensor. The sensor value is read into `pwmRange` as before, but it is then added to `averageReading` each time the loop runs. The `for` loop will reiterate the number of times defined in samples at the start of the program.

```
for (int i = 0; i<samples; i++) {
        pwmRange = pulseIn(sensorPin, HIGH);
        averageReading += pwmRange;
}
```

Then you take the value in `averageReading` and divide it by the number in samples. In your case, the sample number is set to 5, so five samples are taken, added to `averageReading`, which is initially zero, and then divided by five to give you an average reading. This ensures you have a more accurate reading by averaging out some of the noise in the readings from various causes.

```
averageReading /= samples;
```

As before, the timing of the pulse is converted into inches and centimeters:

```
inch = averageReading / 147;
cm = inch * 2.54;
```

Next, you use an `if` statement to check if the switch is HIGH or LOW. If it is HIGH, then the `displayDigit()` function (explained shortly) is run and the value in inches is passed to it. If the switch is LOW, the `else` statement runs the function, but using centimeters instead.

```
if (digitalRead(switchPin)) {
        displayDigit(inch);
}
else {
        displayDigit(cm);
}
```

This `if-else` statement ensures that either inches or centimeters are displayed depending on the position of the toggle switch.

Finally, you define the `displayDigit()` function. This function simply prints the number passed to it on the 7-segment LED display. A floating point number must be passed to the function as a parameter. This will be either inches or centimeters.

```
void displayDigit(float value) {
```

The number passed to this function is a floating point number and will have digits after the decimal point. You are only interested in the first two digits after the decimal point, so it is multiplied by 100 to shift those two digits two places to the left:

```
int number = value*100;
```

This is because you will be using the modulo % operator, which requires integer numbers, and so you must convert the floating point number to an integer. Multiplying it by 100 ensures that the two digits after the decimal point are preserved and anything else is lost. You now have the original number, but without the decimal point. This does not matter as you know there are two digits after the decimal point.

Next, you need to take that number and display it one digit at a time on the 7-segment displays. Each digit is displayed using the `setDigit` command, which requires four parameters. These are

```
setDigit(int addr, int digit, byte value, boolean dp);
```

with `addr` being the address of the MAX7219 chip. You have just one chip, so this value is zero. If a second chip were added, its address would be 1, and so on. Digit is the index of the 7-segment display being controlled. In your case the right hand display is digit 0, the one to its left is 1, and so on. Value is the actual digit, from 0 to 9, that you wish to display on the 7-segment LED. Finally, a boolean value of false or true decides if the decimal point on that display is on or off.

So, using the `setDigit` command, you take the value stored in the integer called number and do division and modulo operations on it to get each digit separately and then display them on the LED:

```
lc.setDigit(0,4,number/10000,false);        // 100s digit
lc.setDigit(0,3,(number%10000)/1000,false); // 10s digit
lc.setDigit(0,2,(number%1000)/100,true);    // first digit with DP on
lc.setDigit(0,1,(number%100)/10,false);     // 10th digit
lc.setDigit(0,0,number%10,false);           // 100th digit
```

Digit 2 has its decimal point turned on, as you want two digits after the decimal point, so the DP flag is true.

You can see how the above works with the following example. Let's say the number to be displayed was 543.21. Remember that the number is multiplied by 100, so you then have 54,321. For Digit 0, you take the number and do a modulo 10 operation on it. This leaves you with the first digit (at the farthest right) which is 1.

543.21 * 100 = 54321
54321 % 10 = 1

Remember that the modulo % operator divides an integer by the number after it, but only leaves you with the remainder. 54321 divided by 10 would be 5432.1 and the remainder is 1. This gives you the first digit (at the farthest right) to be displayed.

The second digit (the 10s column) is modulo 100 and then divided by 10 to give you the second digit.

54321 % 100 = 21

21 / 10 = 2 (remember this is integer arithmetic, and so anything after the decimal point is lost) and so on.

If you follow the calculations using 543.21 as your original number, you will see that the set of modulo and division operations leave you with each individual digit of the original number. The addition of the decimal point on digit 2 (third from right) makes sure the number is displayed with two digits after the decimal point.

You end up with an ultrasonic tape measure that is pretty accurate and displayed to a resolution of 100th of an inch or centimeter. Be aware that the results may not be exactly spot on as the sound waves will move faster or slower due to different temperatures or air pressures. Also, sound waves are reflected off different surfaces differently. A perfectly flat surface perpendicular to the plane of the sensor will reflect the sound well and will give the most accurate reading. A surface with bumps on it or one that absorbs sound or one that is at an angle will give an inaccurate reading. Experiment with different surfaces and compare the readings with a real tape measure.

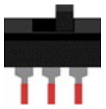
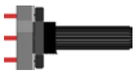
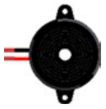
Let's use the ultrasonic sensor for something different now.

Project 40 – Ultrasonic Alarm

You will now build upon the circuit from the last project and turn it into an alarm system.

Parts Required

Table 14-4. *Parts required for Project 40*

LV-MaxSonar EZ3*	
100µF Electrolytic Capacitor	
2 × 100 ohm Resistor	
3 × 10K ohm Resistor	
Switch	
Pushbutton	
5 × 7-Segment LED displays (Common Cathode)	
MAX7219 LED Driver IC	
5-10K ohm Potentiometer	
Piezo Sounder or 8 ohm Speaker	

**or any from the LV range (image courtesy of Sparkfun)*

Connect It Up

Connect everything as shown in Figure 14-5.

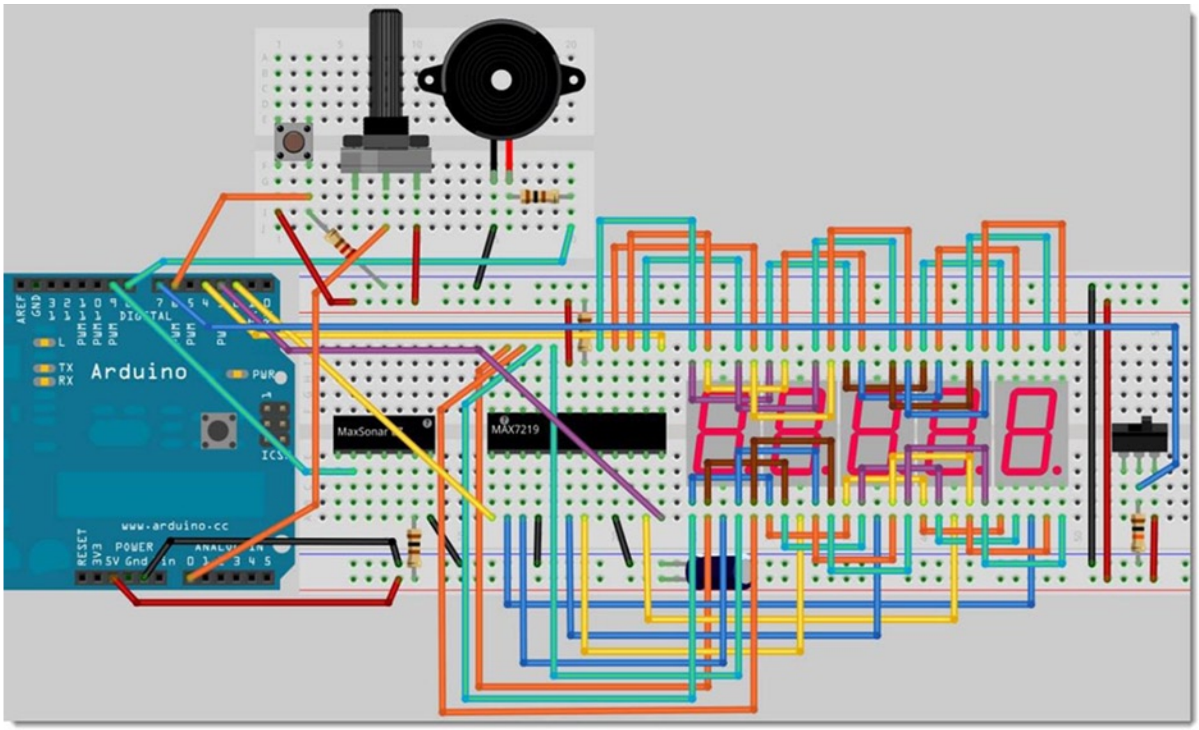


Figure 14-5. *The circuit for Project 40—Ultrasonic Alarm*

The circuit is the same as for Project 39 but with the addition of a pushbutton, a potentiometer, and a piezo sounder (or speaker). The button has both terminals connected to +5v and ground, with the Gnd pin connected via a 10K ohm resistor. A wire goes from this same pin to digital pin 6. The potentiometer has +5v and ground connected to its outer pins and the center pin goes to analog pin 0. The speaker has its negative terminal connected to ground and the positive terminal, via a 100 ohm resistor, to digital pin 8. The potentiometer will be used to adjust the alarm sensor range and the button will reset the system after an alarm activation. The piezo will obviously sound the alarm.

Enter the Code

After checking your wiring is correct, power up the Arduino and upload the code from Listing 14-3.

Listing 14-3. Code for Project 40

```
// Project 40

#include "LedControl.h"

#define sensorPin 9
#define switchPin 7
#define buttonPin 6
```

```
#define potPin 0
#define DataIn 2
#define CLK 4
#define LOAD 3
#define NumChips 1
#define samples 5.0

float pwmRange, averageReading, inch, cm, alarmRange;
LedControl lc=LedControl(DataIn,CLK,LOAD,NumChips);

void setup() {

 // Wakeup the MAX7219
  lc.shutdown(0,false);
  // Set it to medium brightness
  lc.setIntensity(0,8);
  // clear the display
  lc.clearDisplay(0);
  pinMode(sensorPin, INPUT);
  pinMode(switchPin, INPUT);
}

void loop() {
  readPot();
  averageReading = 0;
  for (int i = 0; i<samples; i++) {
   pwmRange = pulseIn(sensorPin, HIGH);
  averageReading += pwmRange;
  }

  averageReading /= samples;
  // 147uS per inch according to datasheet
  inch = averageReading / 147;
  // convert inch to cm
  cm = inch * 2.54;

  if (digitalRead(switchPin)) {
  displayDigit(inch);
  }
  else {
    displayDigit(cm);
  }

        // if current range smaller than alarmRange, set off alarm
        if (inch<=alarmRange) {startAlarm();}
}

void displayDigit(float value) {
    int number = value*100;
    lc.setDigit(0,4,number/10000,false);        // 100s digit
    lc.setDigit(0,3,(number%10000)/1000,false); // 10s digit
    lc.setDigit(0,2,(number%1000)/100,true);    // first digit
    lc.setDigit(0,1,(number%100)/10,false);     // 10th digit
    lc.setDigit(0,0,number%10,false);           // 100th digit
}

// read the potentiometer
float readPot()  {
  float potValue = analogRead(potPin);
  alarmRange = 254 * (potValue/1024);
  return alarmRange;
}

// set off the alarm sound till reset pressed
void startAlarm() {
  while(1) {
    for (int freq=800; freq<2500;freq++) {
      tone(8, freq);
      if (digitalRead(buttonPin)) {
        noTone(8);
        return;
      }
    }
  }
}
```

Once the code is entered, upload it to your Arduino, and then power down the device. Power back up, making sure the sensor is able to calibrate properly. Now you can turn the potentiometer to adjust the range of the alarm. Put a hand into the beam and steadily move closer until the alarm goes off. The reading once the alarm is activated will remain still and show you the last distance measured. This is your alarm range. Press the reset button to silence the alarm, reset the system, and then keep adjusting the potentiometer until you get a range you are happy with. Your alarm is now ready to protect whatever it is near. Anything that comes within the range of the sensor that you have set will activate the alarm until reset.

Project 40 – Ultrasonic Alarm – Code Overview

Most of the code is the same as explained in Project 39, so I will skip over explaining those sections. The LedControl library is loaded in:

```
#include "LedControl.h"
```

Then the pins used are defined as well as the number of chips and samples, as before:

```
#define sensorPin 9
#define switchPin 7
#define buttonPin 6
#define potPin 0
#define DataIn 2
#define CLK 4
#define LOAD 3
#define NumChips 1
#define samples 5.0
```

You add a definition for the buttonPin and potPin. The variables are declared, including the new variable called alarmRange that will hold the distance threshold after which the alarm will sound if a person moves closer than the range set:

```
float pwmRange, averageReading, inch, cm, alarmRange;
```

You create an LedControl object called lc and define the pins:

```
LedControl lc=LedControl(DataIn,CLK,LOAD,NumChips);
```

The setup() loop is the same as before with the addition of setting the pinMode of the buttonPin to INPUT:

```
lc.shutdown(0,false);
lc.setIntensity(0,8);
lc.clearDisplay(0);
pinMode(sensorPin, INPUT);
pinMode(switchPin, INPUT);
pinMode(buttonPin, INPUT);
```

The main loop starts with calling a new function called readPot(). This function reads the value from the potentiometer that you will use to adjust the alarm range (discussed later):

```
readPot();
```

The rest of the main loop is the same as in project 39

```
averageReading = 0;
for (int i = 0; i<samples; i++) {
        pwmRange = pulseIn(sensorPin, HIGH);
        averageReading += pwmRange;
 }

averageReading /= samples;
inch = averageReading / 147;
cm = inch * 2.54;

if (digitalRead(switchPin)) {
        displayDigit(inch);
}
else {
        displayDigit(cm);
}
```

until you reach the next if statement

```
if (inch<=alarmRange) {startAlarm();}
```

which simply checks if the current measurement from the sensor is smaller or equal to the value in alarmRange that has been set by the user and if so, calls the startAlarm() function.

The displayDigit() function is the same as in Project 39:

```
void displayDigit(float value) {
    int number = value*100;
    lc.setDigit(0,4,number/10000,false);        // 100s digit
    lc.setDigit(0,3,(number%10000)/1000,false); // 10s digit
    lc.setDigit(0,2,(number%1000)/100,true);    // first digit
    lc.setDigit(0,1,(number%100)/10,false);     // 10th digit
    lc.setDigit(0,0,number%10,false);           // 100th digit
}
```

Next is the first of the two new functions. This one is designed to read the potentiometer and convert its value into inches to set the range of the alarm. The function has no parameters but is of type float as it will be returning a float value in alarmRange.

```
float readPot()
```

Next, you read the analog value from the potPin and store it in potValue:

```
float potValue = analogRead(potPin);
```

You then carry out a calculation on this value to convert the values from 0 to 1,023 that is read in from the potentiometer and converts it to the maximum and minimum range of the sensor, i.e. 0 to 254 inches.

```
alarmRange = 254 * (potValue/1024);
```

Then you return that value to the point at which the function was called:

```
return alarmRange;
```

The next function is responsible for setting off the alarm sound. This is the startAlarm() function:

```
void startAlarm() {
```

Next you have a while loop. You came across the while loop in Chapter 3. The loop will run while the statement in the brackets is true. The parameter for the while loop is 1. This simply means that while the value being checked is true, the loop will run. In this case, the value being checked is a constant value of 1, so the loop always runs forever. You will use a return command to exit the loop.

```
while(1) {
```

Now you have a for loop that will sweep up through the frequencies from 800 to 2500Hz:

```
for (int freq=800; freq<2500;freq++) {
```

You play a tone on pin 8, where the piezo sounder is, and play the frequency stored in freq:

```
tone(8, freq);
```

Now you check the buttonPin using digitalRead to see if the button has been pressed or not:

```
if (digitalRead(buttonPin)) {
```

If the button has been pressed, the code inside the brackets is run. This starts with a `noTone()` command to cease the alarm sound and then a return to exit out of the function and back to the main loop of the program:

```
noTone(8);
return;
```

In the next project, you will keep the same circuit, but upload some slightly different code to use the sensor for a different purpose.

Project 41 – Ultrasonic Theremin

For this project, you are going to use the same circuit. Although you won't be using the potentiometer, switch, or reset button in this project I am leaving them in to give you the flexibility to modify the project if you wish—plus, this means you can jump back to using Project 40 if you wish later.

This time, you are going to use the sensor to create a theremin that uses the sensor ranging instead of the electrical field that a real theremin uses. If you don't know what a Theremin is, look it up on Wikipedia. It is basically an electronic instrument that you can play without touching it by placing your hands inside an electrical field and by moving your hands inside that field. The device senses changes in the field and plays a note that relates to the distance to the coil. It is difficult to explain, so check out some videos of it being used on YouTube. As the circuit is the same, I will jump right to the code.

Enter the Code

Enter the code in Listing 14-4.

Listing 14-4. Code for Project 41

```
// Project 41

#define sensorPin 9

#define lowerFreq 123  // C3
#define upperFreq 2093 // C7
#define playHeight 36

float pwmRange, inch, cm, note;

void setup() {
        pinMode(sensorPin, INPUT);
}

void loop() {
        pwmRange = pulseIn(sensorPin, HIGH);

        inch = pwmRange / 147;
        // convert inch to cm
        cm = inch * 2.54;
        // map the playHeight range to the upper and lower frequencies
        note = map(inch, 0, playHeight, lowerFreq, upperFreq);
        if (inch<playHeight) {tone(8, note); }
        else {noTone(8);}
}
```

Once you upload it to the Arduino, you can now enter your hand into the sensor's beam and the theremin will play the note mapped to that height from the sensor. Move your hand up and down in the beam and the tones played will also move up and down the scale. You can adjust the upper and lower frequency ranges in the code if you wish.

Project 41 – Ultrasonic Theremin – Code Overview

This code is a stripped-down version of Project 40 with some code to turn the sensor range into a tone to be played on the piezo sounder or speaker. You start off by defining the sensor pin as before.

```
#define sensorPin 9
```

Then you have some new definitions for the upper and lower notes to be played and the `playHeight` in inches. The `playHeight` is the range between the sensor and as far as your arm will reach while playing the instrument. You can adjust this range to something more or less, if you wish.

```
#define lowerFreq 123  // C3
#define upperFreq 2093 // C7
#define playHeight 36
```

The variables are declared with one for `note`, which will be the note played through the speaker:

```
float pwmRange, inch, cm, note;
```

The setup routine simply sets the sensor pin to be an input:

```
pinMode(sensorPin, INPUT);
```

In the main loop, the code is just the essentials. The value from the sensor is read and converted to inches:

```
pwmRange = pulseIn(sensorPin, HIGH);
inch = pwmRange / 147;
```

Next, the inch values from zero to the value stored in `playHeight` are mapped to the upper and lower frequencies defined at the start of the program:

```
note = map(inch, 0, playHeight, lowerFreq, upperFreq);
```

You only want the tone to play when your hand is inside the beam, so you check if the value from the sensor is less than or equal to the play height. If so, a hand must be within the play area, and then a tone is played.

```
if (inch<=playHeight) {tone(8, note); }
```

If a hand is not in the beam or removed from the beam, the tone is stopped:

```
else {noTone(8);}
```

Play around with the `playHeight`, `upperFreq`, and `lowerFreq` values to get the sound you want.

Summary

In this chapter you learned how to interface an ultrasonic sensor. I have also introduced a few uses of the sensor to give you a feel for how it can be used in your own projects. Sensors such as these are often used in hobby robotics projects in which the robot senses if it is near a wall or other obstacle. I have also seen them used in gyrocopter projects to ensure the craft does not bump into any walls or people. Another common use is to detect the height of a liquid inside a tank or a tube. I am sure you will think of other great uses for these kinds of sensors.

Subjects and Concepts covered in Chapter 14:

- How an ultrasonic sensor works
- How to read the PW output from the MaxSonar devices
- Using a capacitor to smooth the power line
- How to use the `pulseIn` command to measure pulse widths
- Various potential uses for an ultrasonic range finder
- How to use the MAX7219 to control 7-segment displays
- How to wire up a common cathode 7-segment display
- Using a running average algorithm to smooth data readings
- How to use the `setDigit()` command to display digits on 7-segment LEDs
- Using division and modulo operators to pick out digits from a long number
- How to make an infinite loop with a `while(1)` command

CHAPTER 15

Reading and Writing to an SD Card

Now you are going to learn the basics of writing to and reading from an SD Card. SD Cards are a small and cheap method of storing data, and an Arduino can communicate relatively easily with one using its SPI interface. You will learn enough to be able to create a new file, append to an existing file, timestamp a file, and write data to that file. This will allow you to use an SD Card and an Arduino as a data-logging device to store whatever data you wish. Once you know the basics, you will put that knowledge to use to create a time-stamped temperature data logger.

Project 42 – Simple SD Card/Read Write

In this project, we will connect up an SD Card to an Arduino and by using the SD.h library to access the card, we will create a new file on the card, write some text to that file, print out the contents of that file, then delete the file. This will teach you the basic concepts of accessing an SD card and reading and writing files to it. You will need an SD Card and some way of connecting it to an Arduino. The easiest way is get an SD/MMC Card Breakout Board from various electronics hobbyist suppliers. I used one from Sparkfun.

Parts Required

Table 15-1. *Parts Required for Project 42*

SD Card & Breakout*	
3 × 3.3K ohm Resistors	
3 × 1.8K ohm Resistors	

**image courtesy of Sparkfun*

The resistors create a voltage divider and drop the 5V logic levels down to 3.3V. (Note that a safer way would be to use a dedicated logic level converter. Again, these can be purchased from stockists such as Sparkfun). Never connect the Arduino's output pins directly to the SD Card without first dropping the voltage from them from 5V down to 3.3V or you will damage the SD Card.

Connect It Up

Connect everything as shown in Figure 15-1.

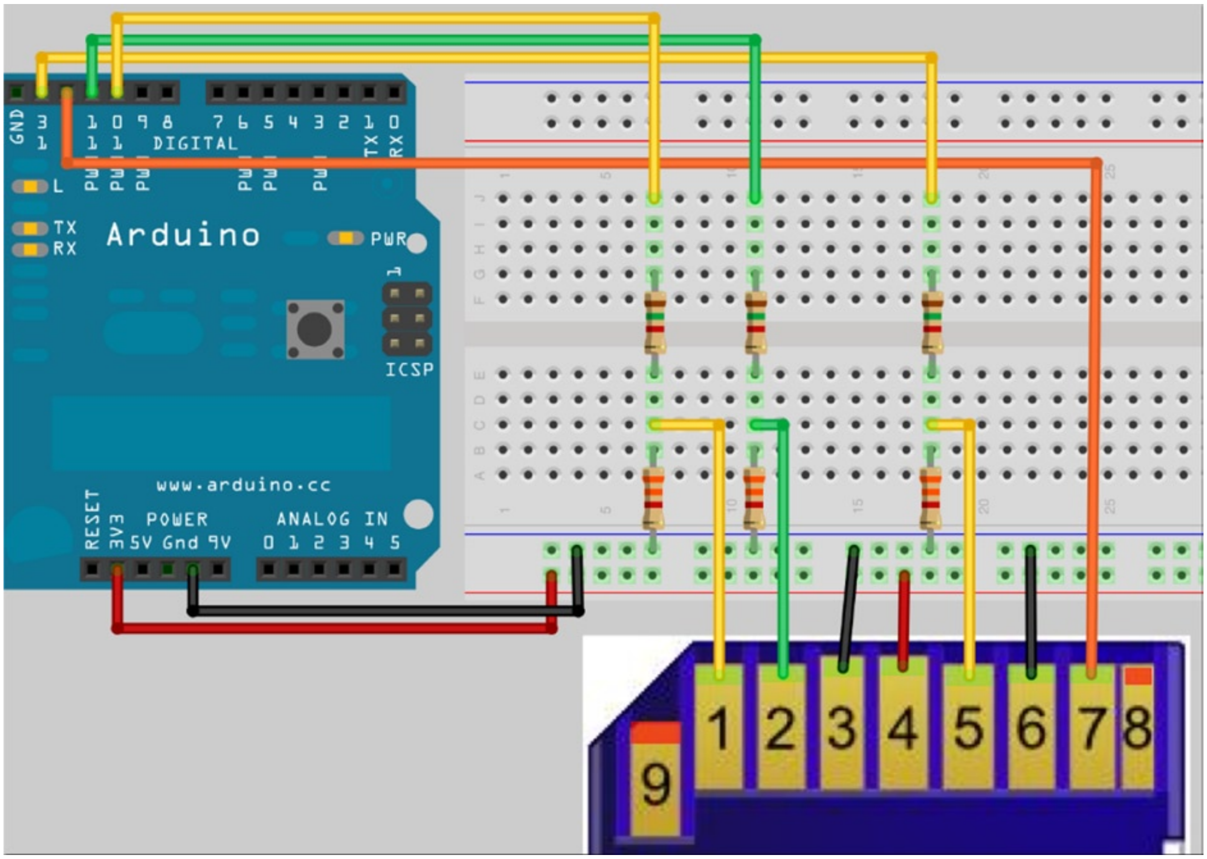


Figure 15-1. The circuit for Project 42 – Simple SD Card Read/Write (see insert for color version)

Refer to Table 15-2 for the correct pin outs. Digital pin 12 on the Arduino goes straight into pin 7 (DO) on the SD Card. Digital pins 13, 11, and 10 go via the resistors to drop the logic levels to 3.3V. The remaining connections to the SD card supply 3.3V to power the SD card via pin 4 and ground pins 3 and 6. Refer to the datasheet for your particular SD Card breakout board in case the pin outs differ from the circuit board above.

Table 15-2. Pin Connections between the Arduino and SD Card

Arduino	SD Card
+3.3V	Pin 4 (VCC)
Gnd	Pins 3 & 6 (GND)
Digital Pin 13 (SCK)	Pin 5 (CLK)
Digital Pin 12 (MISO)	Pin 7 (DO)
Digital Pin 11 (MOSI)	Pin 2 (DI)
Digital Pin 10 (SS)	Pin 1 (CS)

Enter the Code

This code will use the SD.h library included with your Arduino. This library was written by William Greiman. You can find the library at https://code.google.com/p/sdfatlib/downloads/list. Download the latest version. Use a formatted card (Fat16 or Fat32). Once you have checked your wiring is correct, enter the code in Listing 15-1 and upload it to your Arduino.

Listing 15-1. Code for Project 42

```
// Project 42

#include <SD.h>

File File1;

void setup()
{
  Serial.begin(9600);
   while (!Serial) { } // wait for serial port to connect.
                       // Needed for Leonardo only

  Serial.println("Initializing the SD Card...");

  if (!SD.begin()) {
    Serial.println("Initialization Failed!");
    return;
  }
  Serial.println("Initialization Complete.\n");

  Serial.println("Looking for file 'testfile.txt'...\n");

  if (SD.exists("testfile.txt")) {
    Serial.println("testfile.txt already exists.\n");
  }
  else {
    Serial.println("testfile.txt doesn't exist.");
    Serial.println("Creating file testfile.txt...\n");
  }

  File1 = SD.open("testfile.txt", FILE_WRITE);
  File1.close();

  Serial.println("Writing text to file.....");
  String dataString;
  File1 = SD.open("testfile.txt", FILE_WRITE);
  if (File1) {
      for (int i=1; i<11; i++) {
        dataString = "Test Line ";
        dataString += i;
        File1.println(dataString);
      }
      Serial.println("Text written to file.....\n");
  }
  File1.close();

  Serial.println("Reading contents of textfile.txt...");
  File1 = SD.open("testfile.txt");

   if (File1) {
    while (File1.available()) {
      Serial.write(File1.read());
    }
    File1.close();
  }
  // if the file isn't open, pop up an error:
  else {
    Serial.println("error opening testfile.txt");
  }

  // delete the file:
  Serial.println("\nDeleting testfile.txt...\n");
  SD.remove("testfile.txt");

  if (SD.exists("testfile.txt")){
    Serial.println("testfile.txt still exists.");
  }
  else {
    Serial.println("testfile.txt has been deleted.");
  }
}

void loop()
{
  // Nothing to see here
}
```

Make sure that your SD card has been freshly formatted in the FAT or FAT32 format. On a Mac, ensure the partition scheme is set to "Master Boot Record." Run the program and open the serial monitor. The program will now attempt to write a file to the SD card, write some text to that file, read back the contents of the file and then finally delete the file. This will all be displayed on the serial monitor window. If everything goes well, you will get a readout like this:

```
Initializing the SD Card...
Initialization Complete.

Looking for file 'testfile.txt'...

testfile.txt doesn't exist.
Creating file testfile.txt...

Writing text to file.....
Text written to file.....
```

```
Reading contents of textfile.txt...
Test Line 1
Test Line 2
Test Line 3
Test Line 4
Test Line 5
Test Line 6
Test Line 7
Test Line 8
Test Line 9
Test Line 10

Deleting testfile.txt...

testfile.txt has been deleted.
```

Make sure that the card is either SD or SDHC format and that it is formatted with “Master Boot Record” or MBR partitions.

Let’s see how the code works.

Project 42 – Simple SD Card Read/Write – Code Overview

First we need to include the SD.h library in our code. This gives us access to all of the card and file functions in the library that let us access the SD card, read and write files to it, etc.

```
#include <SD.h>
```

Next we use the File command, which is part of the SD.h library, to create an instance of a File object. We need a File object to be able to access, create, close, and delete files. We will name our File object File1.

```
File File1;
```

As we only want our program to run once and only once, we will put all of our code into the setup() function.

```
void setup()
```

We will be outputting some feedback data to the user with the serial monitor window, so we need to begin serial communications. We leave the baud rate at default.

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

The next line is only required if you have a Leonardo, as it may require some time longer to connect to the serial port. The While statement checks if the serial port is ready and while it is NOT (!) ready it waits, as the While statement will only end when the serial port is ready.

```
while (!Serial) { } // wait for serial port to connect.
                    // Needed for Leonardo only
```

We inform the user that the program is attempting to initialize (start communications with) the SD card.

```
Serial.println("Initializing the SD Card...");
```

The `SD.Begin` command initializes the SD card and the SD library. It returns a true if successful and false if it fails. We therefore need to inform the user if the initialization process failed by checking if `!SD.Begin` (NOT SD.Begin) or if the statement return a false. If so, inform the user and return. The return statement will exit the setup() function.

```
if (!SD.begin()) {
  Serial.println("Initialization Failed!");
  return;
}
```

If the initialization process was a success, then inform the user "Initialization Complete."

```
Serial.println("Initialization Complete.\n");
```

Next, we are going to create a text file on the SD card called testfile.txt. First we check if this file already exists or not, so we start by informing the user that we are looking for the file.

```
Serial.println("Looking for file 'testfile.txt'...\n");
```

Next, the `SD.exists` function will return a true if the file or directory we are looking for exists and a false if not. The parameter for the function is the name of the file we want to check for. The following if statement checks that and informs the user if the file exists or not. If it doesn't, the user is then informed that the program is about to create the file.

```
if (SD.exists("testfile.txt")) {
  Serial.println("testfile.txt already exists.\n");
}
else {
  Serial.println("testfile.txt doesn't exist.");
  Serial.println("Creating file testfile.txt...\n");
}
```

Next, we use the SD.open() function to open the file. If the file it is trying to open does not exist, then it will create it. The function required either one or two parameters. The first parameter is the name of the file we wish to open and the second optional parameter is either FILE_READ, to open the file for reading, starting at the beginning of the file, or FILE_WRITE, to open the file for reading and writing, starting at the end of the file. We assign this open or created file to the File object we named File1 earlier.

```
File1 = SD.open("testfile.txt", FILE_WRITE);
```

Once the file has been opened, or created, the `*file*.close` command is used to close the file and ensure that it and any data written to it are saved to the SD card.

```
File1.close();
```

Next, the user is informed that the program is about to write some text to the file.

```
Serial.println("Writing text to file.....");
```

We need a string of characters to write to the text file, so the String data type (array of characters) is used and we initialize this variable with the name dataString.

```
String dataString;
```

Next, we open the file again for writing and assign it to the File object File1.

```
File1 = SD.open("testfile.txt", FILE_WRITE);
```

Next, we use an if statement to check that the File has been opened successfully and if so, use a for statement to loop through the digits 1 to 10, which are concatenated (appended to) the end of the string "Test Line"; then, we print this string to the File in the same way we would write strings to the serial monitor window using the `File1.println(dataString)` command. File1.println() will add a line feed and new line marker to the end of the dataString. If you did not want to do this, then use the command `File1.print()` instead of `File1.println()`.

```
if (File1) {
    for (int i=1; i<11; i++) {
      dataString = "Test Line ";
      dataString += i;
      File1.println(dataString);
    }
```

If the file was opened and written to successfully, then we inform the user (this is inside of the if statement, so will only run if true).

```
    Serial.println("Text written to file.....\n");
}
```

Finally, the file is closed. Data is only written once it is closed. If you want to write to the file while still keeping it open, you could use the `.flush()` function instead.

```
File1.close();
```

Now, to confirm that the data was written to the text file, the program will print off the data inside the file for the user to see. We start by informing the user we are doing this.

```
Serial.println("Reading contents of textfile.txt...");
```

Then the file is opened again.

```
File1 = SD.open("testfile.txt");
```

If the file opening was successful

```
if (File1) {
```

Check that there are some bytes available to be read from the file (i.e. it isn't empty) first and while there are bytes available.

```
while (File1.available()) {
```

Read the next available character, using the `.read()` function and write it to the serial monitor window.

```
  Serial.write(File1.read());
}
```

If the file opening was successful, and after the data has been read, close the file.

```
  File1.close();
}
```

If the file didn't open, then inform the user.

```
else {
  Serial.println("error opening testfile.txt");
}
```

Now that we have created a file and written and read from it, the file will be deleted. First the user is informed.

```
Serial.println("\nDeleting testfile.txt...\n");
```

Then we use the `SD.remove()` function, with its parameter being the name of the file we want to remove, to delete the file.

```
SD.remove("testfile.txt");
```

Then finally, the program will check if the file exists any longer or not and inform the user.

```
  if (SD.exists("testfile.txt")){
    Serial.println("testfile.txt still exists.");
  }
  else {
    Serial.println("testfile.txt has been deleted.");
  }
}
```

As we only wanted the commands above to execute once and only once, the entire code was placed in the `setup()` function. The main `loop()` function is therefore left empty.

```
void loop()
{
  // Nothing to see here
}
```

The above example shows you the basic method of creating a file, writing some strings to the file, reading the file, closing the file, and then deleting it. You will now expand on that knowledge and put it to a practical use by using the SD Card to log some data readings from a temperature sensor.

Project 43 – Temperature SD Datalogger

Now you'll add some DS18B20 temperature sensors to the circuit along with a DS1307 RTC (Real Time Clock) chip. The readings from the temperature sensors will be logged onto the SD Card, and you'll use the RTC chip to read the date and time so that the sensor readings and the file modification can all be time stamped. We use two temperature sensors, one designed to be inside your house and the other via a long cable placed outside. For testing purposes, however, you can leave them both on the breadboard. Once the circuit and code are working, you can solder a long set of cables to the second sensor and place it outside to get external temperatures.

Parts Required

Table 15-3. *Parts Required for Project 43*

SD Card & Breakout*	
3 × 3.3K ohm Resistors	
3 × 1.8K ohm Resistors	
4.7K ohm Resistor	
2 × 1K ohm Resistors	
DS1307 RTC IC	
32.768khz 12.5pF Watch Crystal	
2 × DS18B20 Temp. Sensors	
Switch	
*Coin Cell Holder**	

**images courtesy of Sparkfun*
***Optional*

The coin cell holder is optional. Having a coin cell as a battery backup will allow you to retain the time and date in the RTC even when you power down your project. You can also substitute DS18S20 sensors for the DS18B20s. The only difference is that the DS18S20 has a 9-bit temperature resolution, whereas the DS18B20 has 9 to 12 bits of resolution. Either is accurate enough for our purposes.

The switch is to enable you to take the card in and out of the reader safely. The program will display, via the serial monitor window, when it is safe and unsafe to power off, and then disconnect the card. When reinserting the card, put the card into the slot before powering it back up with the switch.

Connect It Up

Connect everything as shown in Figure 15-2.

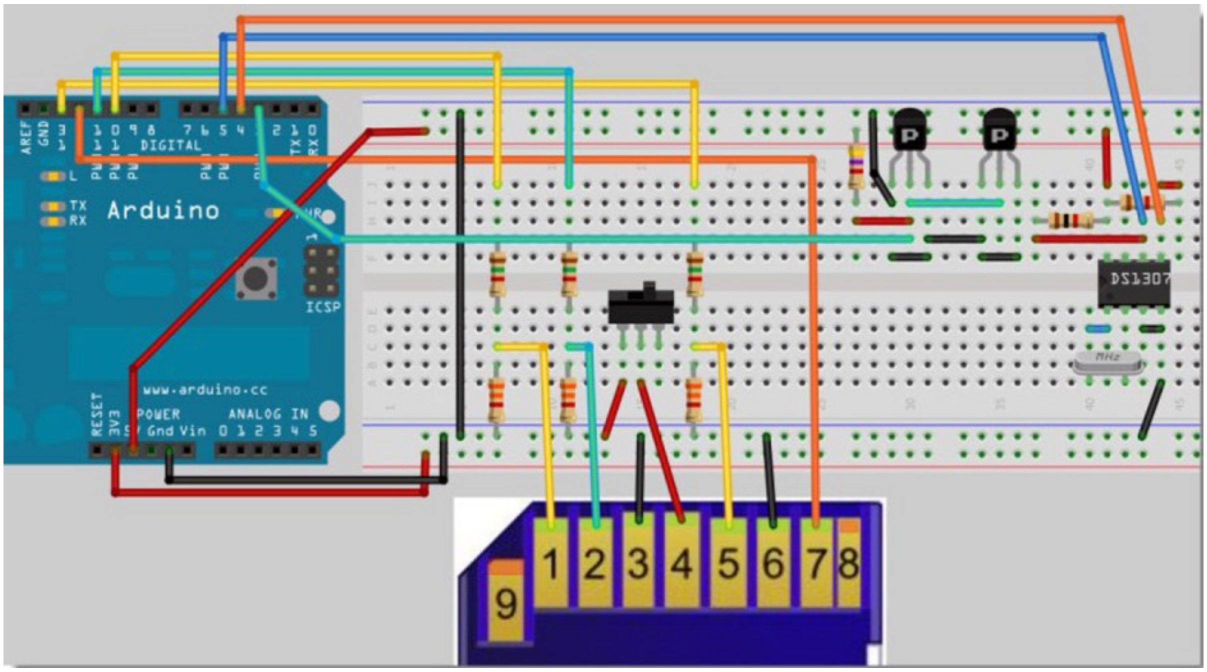


Figure 15-2. *The circuit for Project 43 – Temperature SD Datalogger*

Refer to Table 15-4 for the correct pin outs. Connect the DS18B20s exactly as in Project 37. If you are using the coin cell battery backup, do not install the wire between pins 3 and 4 on the RTC as in the diagram. Instead, connect the positive terminal of the battery to pin 3 of the chip and the negative to pin 4.

Table 15-4. *Pin Connections between the Arduino, SD Card, and RTC*

Arduino	SD Card	RTC
+5V	-	Pin 8
+3.3V	Pin 4 (VCC)	
Gnd	Pins 3 & 6 (GND)	Pins 3 & 4
Digital Pin 13 (SCK)	Pin 5 (CLK)	
Digital Pin 12 (MISO)	Pin 7 (DO)	
Digital Pin 11 (MOSI)	Pin 2 (DI)	
Digital Pin 10 (SS)	Pin 1 (CS)	
Digital Pin 4 (SDA)		Pin 5
Digital Pin 5 (SCL)		Pin 6

Place 1K ohm resistors between pin 8 and pins 5 and 6 on the RTC.

Enter the Code

Make sure that the OneWire.h and DallasTemperature.h libraries used in Project 37 are installed for this project: www.pjrc.com/teensy/arduino_libraries/OneWire.zip and http://download.milesburton.com/Arduino/MaximTemperature/DallasTemperature_372Beta.zip. We will need the SD.h library too. You will also be using the DS1307.h library by Henning Karlsen at henningkarlsen.com to control the DS1307 chip: http://www.henningkarlsen.com/electronics/ Make sure that your library folder name is consistent with the headers and code files within them.

Listing 15-2. Code for Project 43

```
// Project 43
// DS1307 library by Henning Karlsen

#include <SD.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <DS1307.h> // written by Henningh Karlsen

#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12

File File1;

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas
temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device addresses
DeviceAddress insideThermometer = { 0x28, 0x44, 0x12, 0xC2, 0x03, 0x00, 0x00, 0x92 };
DeviceAddress outsideThermometer = { 0x28, 0xA5, 0x02, 0xC2, 0x03, 0x00, 0x00, 0xF0 };

  float tempC, tempF;

  // Init the DS1307
DS1307 rtc(4, 5);

void setup() {
 Serial.println("Initializing the SD Card...");

  if (!SD.begin()) {
    Serial.println("Initialization Failed!");
    return;
  }
  Serial.println("Initialization Complete.\n");

  // Set the clock to run-mode
  rtc.halt(false);
  Serial.begin(9600);
  Serial.println("Type any character to start");
  while (!Serial.available());
  Serial.println();

  // Start up the sensors library
  sensors.begin();
  Serial.println("Initialising Sensors.\n");

  // set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
  sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
  delay(100);

  // Set the time on the RTC.
  // Comment out this section if you have already set the time and have a battery backup
  // The following lines can be commented out to use the values already stored in the DS1307
  rtc.setDOW(TUESDAY);         // Set Day-of-Week to TUESDAY
  rtc.setTime(9, 27, 00);      // Set the time HH,MM,SS
  rtc.setDate(30, 04, 2013);   // Set the date DD,MM,YYYY
}

void getTemperature(DeviceAddress deviceAddress)
{
  sensors.requestTemperatures();
  tempC = sensors.getTempC(deviceAddress);
  tempF = DallasTemperature::toFahrenheit(tempC);
}

void loop() {
    File1 = SD.open("TEMPLOG.txt", FILE_WRITE);
    Serial.println("File Opened.");
    if (File1) {
          File1.print(rtc.getDateStr());
          Serial.print(rtc.getDateStr());
          File1.print(", ");
          Serial.print(", ");
          File1.print(rtc.getTimeStr());
          Serial.print(rtc.getTimeStr());
          File1.print(":  Inside: ");
          Serial.print(":  Inside: ");
          getTemperature(insideThermometer);
          File1.print(tempC);
          Serial.print(tempC);
          File1.print("C  Outside: ");
          Serial.print("C  Outside: ");
          getTemperature(outsideThermometer);
          File1.print(tempC);
          Serial.print(tempC);
          File1.println(" C");
```

```
            Serial.println(" C");
            Serial.println("Data written to file.");
        }
        File1.close();
        Serial.println("File Closed.\n");
        Serial.println("Safe to disconnect card");
        delay(10000);
        Serial.println("Card in use, do not disconnect!!");
}
```

Open the serial monitor and the program will ask you to enter a character to start the code. You will then get an output similar to this:

```
Type any character to start

Initialising Sensors.

File Opened.
30.04.2013, 10:11:27:  Inside: 25.94C  Outside: 26.00 C
Data written to file.
File Closed.

File Opened.
30.04.2013, 10:11:39:  Inside: 25.94C  Outside: 26.00 C
Data written to file.
File Closed.
```

If you wait till the output says “File Closed,” then power off the Arduino and eject the SD Card, then insert it into your PC or Mac, you will see a file called TEMPLOG.TXT. Open up this file in a text editor and you will see the time stamped sensor readings looking like:

```
30.04.2013, 09:49:06:  Inside: 25.56C  Outside: 25.56 C
30.04.2013, 09:49:18:  Inside: 25.56C  Outside: 25.62 C
30.04.2013, 09:49:29:  Inside: 25.62C  Outside: 25.62 C
30.04.2013, 09:49:41:  Inside: 25.62C  Outside: 25.69 C
30.04.2013, 10:11:27:  Inside: 25.94C  Outside: 26.00 C
30.04.2013, 10:11:39:  Inside: 25.94C  Outside: 26.00 C
30.04.2013, 10:11:50:  Inside: 26.81C  Outside: 27.00 C
30.04.2013, 10:12:02:  Inside: 28.00C  Outside: 27.94 C
```

Let’s see how the program works.

Project 43 – Temperature SD Datalogger – Code Overview

As a lot of this code is covered in Project 37 and 42, I will concentrate on the new bits.

First, the appropriate libraries are included:

```
#include <SD.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <DS1307.h> // written by Henningh Karlsen
```

The new inclusion here is the DS1307.h library by Henning Karlsen. This library allows us to read and write date and time data to the DS1307 RTC chip. We will need this to first set the date and time on the chip. Once set, the chip will tick away every second and update its internal time and date settings accordingly. We can then read that data from the chip to get an accurate timestamp for our temperature log.

Next, we define which pin we are using for the One Wire bus; this pin is used to communicate with the temperature sensors. Then we define what precision we want the temperature to be set at in the DS18B20 sensor (not valid for the DS18S20 which has fixed precision).

```
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12
```

Next, we create a file instance as we want to write to the SD card and call it File1.

```
File File1;
```

Next, an instance of a oneWire object is created and called OneWire. This allows us to communicate with the OneWire devices, in this case the temperature sensors.

```
OneWire oneWire(ONE_WIRE_BUS);
```

Next, a reference to the oneWire bus is passed to the Dallas Temperature library so it can access the sensors.

```
DallasTemperature sensors(&oneWire);
```

The addresses of the two temperature sensors as previously discovered in Project 37 Part 1 are then stored within DeviceAddress variables. DeviceAddress is a data type created within the OneWire library to hold the addresses of the One Wire devices.

```
DeviceAddress insideThermometer = { 0x28, 0x44, 0x12, 0xC2, 0x03, 0x00, 0x00, 0x92 };
DeviceAddress outsideThermometer = { 0x28, 0xA5, 0x02, 0xC2, 0x03, 0x00, 0x00, 0xF0 };
```

We need to store the temperatures from the sensors, so we create variables of type float to store that temperature in both Celsius and Fahrenheit.

```
float tempC, tempF;
```

We need to tell the DS1307 library which pins are being used to communicate with the RTC chip. These are the SDA (Serial Data) and SCL (Serial Clock) pins on the DS1307. These are digital pins 4 and 5.

```
DS1307 rtc(4, 5);
```

Now in the setup function, we ready the temperature sensors and RTC chip. We start off by informing the user via the serial monitor that the program is about to initialize the SD Card.

```
void setup() {
 Serial.println("Initializing the SD Card...");
```

If the SD card does not initialize successfully, inform the user and return from the function, which will exit the program.

```
  if (!SD.begin()) {
    Serial.println("Initialization Failed!");
    return;
  }
```

Otherwise, inform the user that the SD Card initialization was successful.

```
Serial.println("Initialization Complete.\n");
```

The DS1307 chip has a CH (Clock Halt) flag which needs to be set to either TRUE to stop the clock from ticking or FALSE to start it ticking. As we need to ensure the device is ticking away merrily, the flag is set to FALSE.

```
rtc.halt(false);
```

Serial communications are begun to allow us to write data to the serial monitor.

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

Next the user is asked to type any character to start.

```
Serial.println("Type any character to start");
```

Then we check if any data is available on the serial data line, i.e. the user has entered a character. If not, the while function will keep checking until there is.

```
while (!Serial.available());
```

Print a line return.

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

The sensor library is initialized with the `Sensors.begin()` function.

```
sensors.begin();
```

and the user is informed.

```
Serial.println("Initialising Sensors.\n");
```

The DS18B20 sensor can have its resolution set to between 9 and 12 bits. This is not possible with the DS18S20 sensor which has a fixed bit resolution. We use the `.setResolution()` function of the Dallas Temperature library to set the resolution by passing the function the sensor address and the resolution in bits and then delay a small amount to allow the data to be written.

```
sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
delay(100);
```

The day of the week, time and date need to be set on the DS1307 chip. We will use the setDOW() function to set the day of the week. Enter the day in capitals into the functions parameter brackets.

```
rtc.setDOW(TUESDAY);
```

Next, the setTime() function is used to set the time in HH, MM, SS format.

```
rtc.setTime(9, 27, 00);
```

Then, set the date in DD, MM, YYYY format using the setDate() function.

```
  rtc.setDate(30, 04, 2013);
}
```

Now, we create a function to get the temperatures in Celsius and Fahrenheit from the sensors. The function requires the address of the device.

```
void getTemperature(DeviceAddress deviceAddress)
{
```

Now we use the `requestTemperatures()` function of the Dallas Temperature library to retrieve the temperature.

```
sensors.requestTemperatures();
```

We store in tempC the temperature using the `getTempC()` function which retrieves the temperature in Celsius.

```
tempC = sensors.getTempC(deviceAddress);
```

and then use the `toFahrenheit()` function of the Dallas Temperature library to convert that temperature to Fahrenheit and store it in tempF.

```
  tempF = DallasTemperature::toFahrenheit(tempC);
}
```

Now we have our main program loop.

```
void loop() {
```

We start by opening, or creating, a file on the SD card, which we will call TEMPLOG.txt.

```
File1 = SD.open("TEMPLOG.txt", FILE_WRITE);
```

Inform the user the file has been opened.

```
Serial.println("File Opened.");
```

If the file has opened or been created successfully

```
if (File1) {
```

then we write the relevant strings to the text file. We start by getting the date string with the `rtc.getDateStr()` function of the DS1307 library and writing it to the file.

```
File1.print(rtc.getDateStr());
```

The same data is printed to the serial monitor so the user can see what is going on.

```
Serial.print(rtc.getDateStr());
```

This date is followed by a comma and a space.

```
File1.print(", ");
Serial.print(", ");
```

Then, we retrieve the time string and write that to the file.

```
File1.print(rtc.getTimeStr());
Serial.print(rtc.getTimeStr());
```

Next comes the inside temperature, so we give it a label.

```
File1.print(":  Inside: ");
Serial.print(":  Inside: ");
```

Retrieve the latest temperature from the internal temperature sensor.

```
getTemperature(insideThermometer);
```

then write it to the file.

```
File1.print(tempC);
Serial.print(tempC);
```

If you want your temperature logged in Fahrenheit, simply change tempC to tempF. Next, we print C (or optionally F) and then label the outside temperature.

```
File1.print("C  Outside: ");
  Serial.print("C  Outside: ");
```

Next, we get the reading from the external sensor and write that to the file.

```
getTemperature(outsideThermometer);
File1.print(tempC);
Serial.print(tempC);
File1.println(" C");
Serial.println(" C");
```

All of the above print commands enter the string right after the last one. When we use println, then a newline command is entered at the end of the string to ensure the next set of data is on the next line.

Finally, the user is informed that the data has been written to the file.

```
Serial.println("Data written to file.");
    }
```

The File is closed, which has the effect of writing all of the above data to the file and the user is informed.

```
File1.close();
Serial.println("File Closed.\n");
```

And, finally, the program will wait 10 seconds before the next reading. Obviously you can change this to whichever interval you wish.

```
    delay(10000);
}
```

You now have the basic idea of how to write sensor or other data to an SD card. For more advanced functionality, read the documentation that comes with the SD.h library. You will now take a quick look at the RTC chip before moving onto the next chapter.

Project 43 – Temperature SD Datalogger – Hardware Overview

In Project 43, you were introduced to a new IC, the DS1307 real time clock chip. This is a great little IC that allows you to easily add a clock to your projects. With the addition of the coin cell battery backup, you can disconnect your Arduino from the power and the chip will automatically switch over to the battery backup and keep its data and time updated using the battery. With a good quality crystal, the device will keep reasonably accurate time. The device even adjusts itself for leap years and for months with days less than 31. It can also be set to operate in either 24-hour or 12-hour modes. Communication with the device is via an I^{2}C interface, which I will explain shortly. The chip is interesting in that it also has a square wave output on pin 7. This can be 1Hz (once per second), 4.096kHz, 8.192kHz, or 32.768kHz. You could therefore also use it as an oscillator for generating sound or other purposes that require a pulse. You could easily add an LED to this pin to indicate the seconds as they go by if set at 1Hz.

The communication with the chip is over a protocol called I^{2}C. You have come across one-wire and SPI so far, but this is a new protocol. To use the I^{2}C protocol, you need to include the Wire.h library in your code. However, the DS1307.h has been written to use the protocol without needing the Wire.h library, so in this case we won't be using it.

The I^{2}C protocol (sometimes also called TWI or Two Wire Interface)) was developed by Philips Semiconductors (now known as NXP) to create a simple bidirectional bus using just two wires for inter-IC control. The protocol uses just two wires: the serial data line (SDA) and the serial clock line (SCL). However, the DS1307 library uses its own set of user-defined pins. The only other external hardware required is a pull-up resistor on each of the bus lines. You can have up to 128 I^{2}C devices (or nodes) connected on the same two wires. Some I^{2}C devices use +5V and others +3.3V, so this is something to watch out for when using them. Make sure you read the datasheet and use the correct voltage before wiring up an I^{2}C device. If you ever wanted two Arduinos to talk to each other, then I^{2}C would be a good protocol to use.

The I^{2}C protocol is similar to SPI in that there are master and slave devices. The Arduino is the master and the I^{2}C device is the slave. Each device has its own I^{2}C address. Each bit of the data is sent on each clock pulse. Communication commences when the master issues a START condition on the bus and is terminated when the master issues a STOP condition. To start I^{2}C communication on an Arduino, you issue the `Wire.begin()` command. This will initialize the Wire library and the Arduino as the master I^{2}C device. It also reconfigures analog pins 4 and 5 to be the I^{2}C pins. To initiate communications as a slave (i.e., for two Arduinos connected via I^{2}C), the address of the slave device must be included in the parenthesis. In other words,

```
Wire.begin(5);
```

will cause the Arduino to join the I^{2}C bus as the slave device on address 5. A byte can be received from an I^{2}C device using

```
Int x = Wire. Receive();
```

Before doing so you must request the number of bytes using `Wire.requestFrom(address, quantity)` so

```
Wire.requestFrom(5,10);
```

would request 10 bytes from device 5. Sending to the device is just as easy with

```
Wire.beginTransmission(5);
```

which sets the device to transmit to device number 5 and then

```
Wire.send(x);
```

to send one byte or

```
Wire.send("Wire test.");
```

to send 10 bytes.

You can learn more about the Wire library at `www.arduino.cc/en/Reference/Wire` and about I^{2}C from Wikipedia or by reading the excellent explanation of I^{2}C in the Atmega datasheets on the Atmel website. The DS1307 library by Henning Karlsen uses its own internal set of functions to communicate via One Wire and so doesn't require these functions.

Summary

In Chapter 15, you have learned the basics of reading and writing to an SD Card. You can learn more about using the SD Card with the SD.h library by reading the documentation that comes with it. You have just scratched the surface with the projects in this chapter! Along the way, you have been introduced to the I^{2}C protocol. You have also learned how to connect a DS1307 real-time clock chip, which will be very useful for your own clock-based projects in future. Another great way of obtaining a very accurate time signal is using a cheap GPS device with a serial output; you can also use time receiver modules such as the MSF (UK), WWVB (USA) or DCF (Europe) time receivers for perfectly accurate time.

Knowing how to read and write to an SD card is a vital piece of knowledge for making data loggers, especially for remote battery operated devices. Many of the High Altitude Balloon (HAB) projects based on the Arduino or AVR chips use SD cards for logging GPS and sensor data for retrieval once the balloon is on the ground.

Subjects and Concepts covered in Chapter 15

- How to connect an SD card to an Arduino
- Using voltage divider circuits with resistors to drop voltage levels from +5V to +3.3V
- How to use the SD.h library
- Writing strings and numbers to files
- Opening and closing files
- Naming files
- Creating file timestamps
- Catching file errors

- The concept of the `do-while` loop
- How to connect a DS1307 real-time clock chip to an Arduino
- How to use the DS1307.h library to set and get time and dates
- Using a battery backup on the DS1307 to retain data after power loss
- An introduction to the I^{2}C protocol

CHAPTER 16

Making an RFID Reader

RFID (Radio Frequency Identification) readers are quite common today. They are the method of choice for controlling access in office blocks and for entry systems for public transport. Small RFID tags are injected into animals for identification if they get lost. Vehicles have tags on them for toll collection. They are even used in hospitals to tag disposable equipment in operating rooms to ensure that no foreign objects are left inside patients. The cost of the technology has come down drastically over the years to the point where readers can now be purchased for less than $10. They are easy to connect to an Arduino and easy to use. As a result, all kinds of cool projects can be created out of them.

You will be using the easily obtainable and cheap ID-12 reader from Innovations. These readers use 125KHz technology to read tags and cards up to 180mm away from the reader. There are other readers in the same range that give greater range, and with the addition of an external antenna you can increase the range further still.

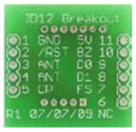
You'll start off by connecting one up and learning how easy it is to obtain serial data from it. You'll then make a simple access control system.

Project 44 – Simple RFID Reader

The pins on the ID12 readers are non-standard spacing so they will not fit into a breadboard. You will need to obtain a breakout board from a supplier such as Sparkfun. The cards or tags can be purchased from all kinds of sources and are very cheap; I got a bag of small key chain style tags on eBay for a few dollars. Make sure the tag or card is of 125KHz technology, otherwise it will not work with this reader.

Parts Required

Table 16-1. *Parts Required for Project 44*

ID-12 RFID Reader	
ID12 Breakout Board*	

(*continued*)

Table 16-1. (*continued*)

Current-Limiting Resistor	
5mm LED	
125KHz RFID tags or cards* (At least 4)	

**image courtesy of Sparkfun*

Connect It Up

Connect everything as shown in Figure 16-1.

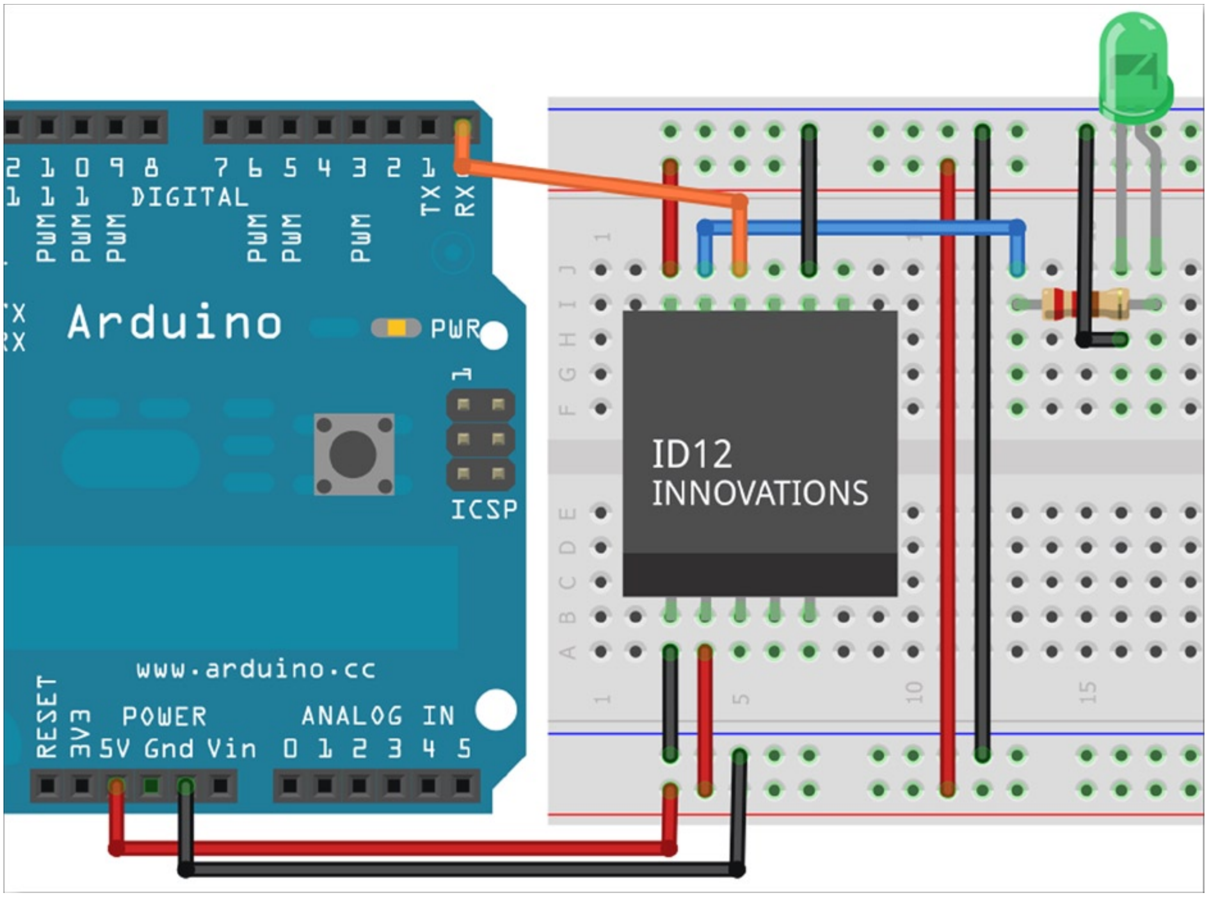


Figure 16-1. *The circuit for Project 44 – Simple RFID Reader*

Connect an LED via a current-limiting resistor to pin 10 (BZ) on the reader.

Table 16-2. *Pin Connections from the Arduino to the ID12*

Arduino	ID12
5v	Pin 2 (RST)
5v	Pin 11 (V)
Gnd	Pin 1 (GND)
Gnd	Pin 7 (FS)
RX	Pin 9 (DO)

Also connect an LED, via a 1k resistor, to pin 10 (BZ) in the ID12.

Enter the Code

Enter the code in Listing 16-1.

Listing 16-1. Code for Project 44.

```
// Project 44

char val = 0; // value read for serial port

void setup() {
    Serial.begin(9600);
}

void loop () {

  if(Serial.available() > 0) {
  val = Serial.read(); // read from the serial port
  Serial.write(val);   // and print it to the monitor
  }
}
```

Before you upload the code disconnect the RX wire or it won't work. Reconnect it before running the code. Run the code, open up the serial monitor, and hold your RFID tag or card up to the reader. The LED will flash to show it has read the card, and the serial monitor window will show you a 12-digit number which makes up the unique ID of the card. Make a note of the IDs of your tags as you will need them in the next project.

Project 44 – Simple RFID Reader – CODE Overview

Let's see how this simple program works. We will need to store the characters read from the ID12 reader somewhere so we create a variable of type char to store it in.

```
char val = 0; // value read for serial port
```

The setup function simply readies serial communication at baud rate 9600.

```
void setup() {
    Serial.begin(9600);
}
```

In the loop function, we check if data is available to be read with the Serial.available command.

```
void loop () {

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

If so, store the byte read in val.

```
  val = Serial.read(); // read from the serial port
```

Then print this to the serial monitor window.

```
  Serial.write(val); // and print it to the monitor
  }
}
```

This program is very simple. It reads one byte from the ID12 reader (when a card is held to the reader) and then prints that value to the serial monitor window.

Project 44 – Simple RFID Reader – Hardware Overview

RFID is everywhere, from your bus pass to the doors that let you into your office or college. The tags or cards come in all kinds of shapes and sizes (see Figure 16-2) and can be made so small that scientists have even attached RFID tags to ants to monitor their movements. They are simple devices that do nothing but transmit a unique serial code via radio waves to the reader. Most of the time, the cards or tags are *passive*, meaning they have no battery and need power from an external source. Other options include *passive RFID*, which has its own power source, and *battery assisted passive* (BAP), which waits for an external source to wake it up and then uses its own power source to transmit, giving greater range.

Figure 16-2. *RFID tags and cards come in all shapes and sizes (image courtesy of Timo Arnall)*

The kind you are using are of the passive type that have no battery. They get their power from a magnetic field transmitted by the reader. As the tag passes into the magnetic field, it inducts a current in the wires inside the tag. This current is used to wake up a tiny chip, which transmits the serial number of the tag. The reader then sends that data in serial format to the PC or microcontroller connected to it. The format of the data sent from the ID12 reader is as follows:

STX (02h)	DATA (10 ASCII)	CHECKSUM (2 ASCII)	CR	LF	ETX (03H)

An STX or transmission start character is sent first (ASCII 02), followed by 10 bytes that make up the individual HEX (hexadecimal) digits of the number. The next two HEX digits are the checksum of the number (this will be explained in the next project), then there's a carriage return (CR) and line feed (LF), followed by an ETX or transmission end code.

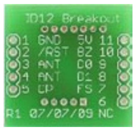
Only the 12 ASCII digits will show up on the serial monitor as the rest are non-printable characters. In the next project, you'll use the checksum to ensure the received string is correct and the STX code to tell you that a string is being sent.

Project 45—Access Control System

You're now going to create an access control system. You'll read tags using the RFID reader and validate select tags to allow them to open a door. The Arduino, via a transistor, will then operate an electric strike lock.

Parts Required

Table 16-3. *Parts Required for Project 45*

ID12 RFID Reader	
ID12 Breakout Board*	 ID12 Breakout 1 GND 5V 11 2 /RST BZ 10 3 ANT D0 9 4 ANT D1 8 5 CP FS 7 6 R1 07/07/09 NC
Current-Limiting Resistor	
2.2k Resistor	
5mm LED	

(*continued*)

Table 16-3. *(continued)*

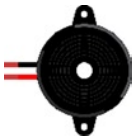
125KHz RFID tags or cards* (At least 4)	
1N4001 Diode	
TIP-120 NPN Transistor	
2.1mm Power Jack	
12V DC Power Supply	
8 ohm Speaker or a Piezo Sounder	
12V Electric Strike Lock	

**image courtesy of Sparkfun*

Connect It Up

Connect everything as shown in Figure 16-3.

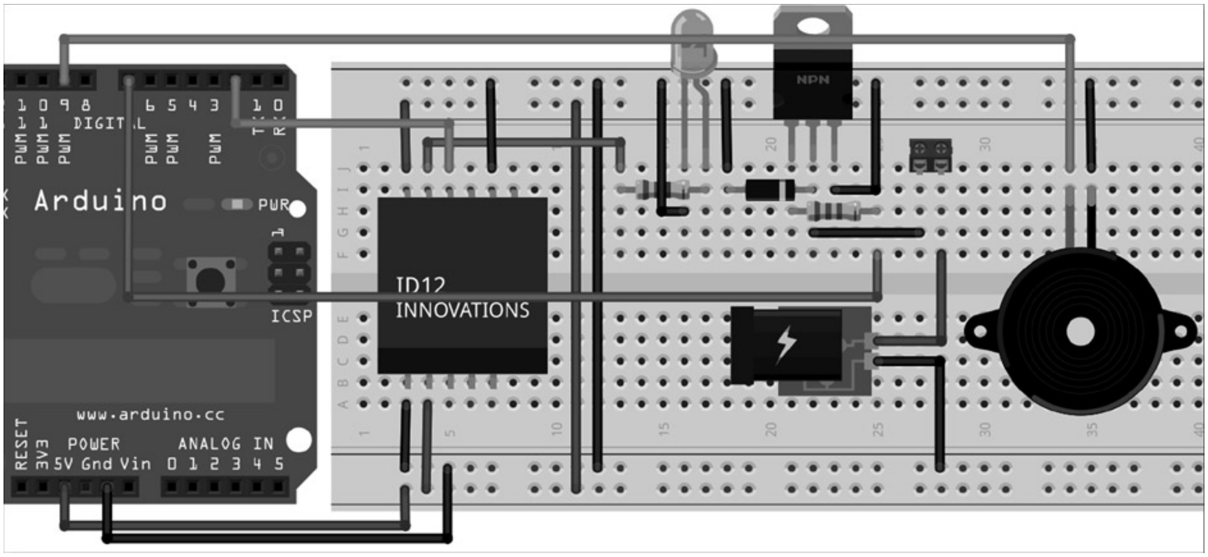


Figure 16-3. *The circuit for Project 45—Access Control System*

If you are not using the TIP120, make sure you read the datasheet for your transistor to ensure you have the pins wired correctly. From left to right on the TIP120, you have the base, collector, and emitter. The base goes to digital pin 7 via a 2.2k resistor, the collector goes to ground via the diode, and also to the negative terminal of the piezo or speaker. The emitter goes to ground. An 8 ohm speaker makes a nicer and louder sound than a piezo, if you can get hold of one. When power is removed from a coil, the collapsing magnetic field creates a current within the coil. To prevent the TIP120 from burning out, we'll add a diode to handle the surge created when the magnetic field collapses.

The power for the lock must come from an external 12V DC power supply with a rating of at least 500mA. Connect wires from the external power supply to the electric strike lock. The 12V supply goes to the 12V connection on the lock and the ground wire goes, via the centre pin of the TIP-120, to the ground connection on the lock.

Enter the Code

Enter the code in Listing 16-2.

Listing 16-2. Code for Project 45

```
// Project 45

#define lockPin 7
#define speakerPin 9
#define tx 3
#define rx 2
#define unlockLength 2000

#include <SoftwareSerial.h>

SoftwareSerial rfidReader = SoftwareSerial(rx, tx);

int users = 3;

char* cards[] = { // valid cards
  "3D00768B53",
  "3D00251C27",
  "3D0029E6BF",
};

char* names[] = { // cardholder names
  "Tom Smith",
  "Dick Jones",
  "Harry Roberts"
};

void setup() {
  pinMode (lockPin, OUTPUT);
  pinMode (speakerPin, OUTPUT);
  digitalWrite(lockPin, LOW);
  Serial.begin(9600);
  rfidReader.begin(9600);
}
```

```
void loop() {
  char cardNum[10];  // array to hold card number
  byte cardBytes[6]; // byte version of card number + checksum
  int index=0;       // current digit
  byte byteIn=0;     // byte read from RFID
  byte lastByte=0;   // the last byte read
  byte checksum = 0; // checksum result stored here

  if (rfidReader.read()==2) {         // read the RFID reader
      while(index<12) {               // 12 digits in unique serial number
          byteIn = rfidReader.read(); // store value in byteIn
          if ((byteIn==1) || (byteIn==2) || (byteIn==10) || (byteIn==13)) {return;}
 // if STX, ETX, CR or LF break
          if (index<10) {cardNum[index]=byteIn;} // store first 10 HEX digits only (last 2 are checksum)
          // convert ascii hex to integer hex value
          if ((byteIn>='0') && (byteIn<='9')) {
          byteIn -= '0';
          }
          else if ((byteIn>='A') && (byteIn<='F')) {
          byteIn = (byteIn+10)-'A';
          }
          if ((index & 1) == 1) { // if odd number merge 2 4 bit digits into 8 bit byte
                 // move the last digit 4 bits left and add new digit
                 cardBytes[index/2]= (byteIn | (lastByte<<4));
                if (index<10) {checksum ^= cardBytes[index/2];} // tot up the checksum value
          }
          lastByte=byteIn; // store the last byte read
          index++; // increment the index
          // if we have reached the end of all digits add a null terminator
          if (index==12) {cardNum[10] = '\0';}
    }

   Serial.println(cardNum);                          // print the card number
   int cardIndex =checkCard(cardNum);                // check if card is valid and return index number
   if(cardIndex>=0 && (cardBytes[5]==checksum)) { // if card number and checksum are valid
        Serial.println("Card Validated");
        Serial.print("User: ");
        Serial.println(names[cardIndex]);            // print the relevant name
        unlock();                                    // unlock the door
        Serial.println();
    }
    else {
        Serial.println("Card INVALID");
        tone(speakerPin, 250, 250);
        delay(250);
        tone(speakerPin, 150, 250);
        Serial.println();
    }
  }
}
```

```
// Check the detected card against all known to be valid card numbers
// Return the array index number of a matched card number
// or a negative value to indicate a non matching card number
int checkCard(char cardNum[10])
{
   for (int x=0; x<=users; x++)
   {
      if(strcmp(cardNum, cards[x])==0)
      {
         return (x);
      }
   }
   return (-1);
}

void unlock() {
    tone(speakerPin, 1000, 500);
    digitalWrite(lockPin, HIGH);
    delay(unlockLength);
    digitalWrite(lockPin, LOW);
}
```

Make sure that the code numbers for three of your tags are entered into the `cards` array at the start of the program. Use Project 44 to find out the code numbers, if necessary (run the code and open the serial monitor). Now, present your four cards to the reader. The reader will flash its LED to show it has read the card and you will then get an output similar to this:

```
3D00251C27
Card Validated
User: Dick Jones

3D0029E6BF
Card Validated
User: Harry Roberts

3D002A7C6C
Card INVALID

3D00768B53
Card Validated
User: Tom Smith
```

The card number will be displayed followed by either “Card INVALID” or “Card Validated,” followed by the name of the user. If the card is valid, a high-pitched tone will sound and the lock will open for two seconds. If the card is not valid, a low-pitched two-tone beep will sound and the door will not unlock. The 12V electric strike is powered using a transistor to power a higher voltage and current than the Arduino can provide. To prevent the TIP120 from burning out, we added a diode to handle the surge created when the magnetic field collapses. You used this same principle in Chapter 5 when you drove a DC Motor. Let’s see how this project works.

Project 45 – Access Control System – Code Overview

First, you have some definitions for the pins you will be using for the lock and the speaker. Also, you are using the SoftwareSerial library instead of the normal serial pins on digital pins 0 and 1, so you must define the rx and tx pins. You also have a length, in microseconds, that the lock will open for.

```
#define lockPin 7
#define speakerPin 9
#define tx 3
#define rx 2
#define unlockLength 2000
```

You are using the SoftwareSerial library (now a core part of the Arduino IDE) for convenience. If you were using the standard rx and tx pins, you would have to disconnect whatever was connected to those pins every time you wanted to upload any code to the Arduino. By using the SoftwareSerial library, you can use any pin you want.

```
#include <SoftwareSerial.h>
```

Next, you create an instance of a SoftwareSerial object and call it rfidReader. You pass to it the rx and tx pins you have defined.

```
SoftwareSerial rfidReader = SoftwareSerial(rx, tx);
```

Next comes a variable to hold the number of users in the database:

```
int users = 3;
```

Next are two arrays to hold the card ID numbers and the names of the cardholders. Change the card numbers to those of your own (first 10 digits only).

```
char* cards[] = { // valid cards
  "3D00768B53",
  "3D00251C27",
  "3D0029E6BF",
};

char* names[] = { // cardholder names
  "Tom Smith",
  "Dick Jones",
  "Harry Roberts"
};
```

The setup routine sets the lock and speaker pins as outputs

```
pinMode (lockPin, OUTPUT);
pinMode (speakerPin, OUTPUT);
```

Setup then sets the lock pin to LOW to ensure the lock does not unlock at the start.

```
digitalWrite(lockPin, LOW);
```

Then you begin serial communications on the serial port and the SoftwareSerial port:

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

Next comes the main loop. You start off by defining the variables that you will use in the loop:

```
char cardNum[10];  // array to hold card number
byte cardBytes[6]; // byte version of card number + checksum
int index=0;       // current digit
byte byteIn=0;     // byte read from RFID
byte lastByte=0;   // the last byte read
byte checksum = 0; // checksum result stored here
```

Next, you check if there is any data coming into the RFID readers' serial port. If so, and the first character is ASCII character 2, which is a transmission start code, then you know an ID string is about to be transmitted and you can start reading in the digits.

```
if (rfidReader.read()==2) {
```

Then comes a whole loop that will run while the index is less than 12:

```
while(index<12) {
```

The `index` variable will hold the value of your current place in the digit you are reading in. As you are reading in a digit of 12 characters in length, you will only read in the first 12 digits.

Next, the value from the serial port is read in and stored in `byteIn`:

```
byteIn = rfidReader.read();
```

Just in case some characters have been missed for any reason, you next check for an occurrence of a transmission start, transmission end, carriage return, or line feed codes. If they are detected, the loop is exited.

```
if ((byteIn==1) || (byteIn==2) || (byteIn==10) || (byteIn==13)) {return;} // if STX, ETX, CR or LF break
```

The last two digits of the 12 digit string are the checksum. You don't wish to store this in the cardNum array so you only store the first 10 digits:

```
if (index<10) {cardNum[index]=byteIn;}
```

Next, you need to convert the ASCII characters you are reading into their hexadecimal number equivalents, so you run an `if-else` statement to determine if the characters are between 0 and 9 and A and Z. If so, they are converted to their hexadecimal equivalents.

```
if ((byteIn>='0') && (byteIn<='9')) {
        byteIn -= '0';
}
else if ((byteIn>='A') && (byteIn<='F')) {
        byteIn = (byteIn+10)-'A';
}
```

Logical AND (&&) operators are used to ensure the characters fall between 0 and 9 or between A and Z. Next, you convert the two last hexadecimal digits into a byte. A hexadecimal digit is base sixteen. The number system we normally use is base ten, with digits ranging from 0 to 9. In hexadecimal, the digits go from 0 to 15. The letters A to F are used for numbers 10 to 15. So, the number FF in hex is 255 in decimal:

$$F = 15$$
$$(F * 16) + F = (15 * 16) + 15 = 255$$

You therefore need to convert the last two ASCII digits into a single byte and the decimal equivalent. You have already declared a variable called `lastByte` which stores the last digit you processed on the last run of the `while` loop. This is initially set to zero. You only need to do this for every second digit as each of the two HEX digits make up a single byte. So you check that the index is an odd number by carrying out a bitwise AND (&) operation on the value stored in index with 1 and seeing if the result is also 1. Remember that index starts off as zero, so the second digit has an index of 1.

```
if ((index & 1) == 1) {
```

Any odd number ANDed with 1 will result in 1 and any even number will result in 0:

```
       12 (even) & 1 = 0
       00001100
       00000001 &
=      00000000

       11 (odd) & 1 = 1
       00001011
       00000001 &
=      00000001
```

If the result determines you are on the second, fourth, sixth, eighth, or twelfth digit, then you store in cardBytes the result of the following calculation:

```
cardBytes[index/2]= (byteIn | (lastByte<<4)); // move the last digit 4 bits left and add new digit
```

You use index/2 to determine the index number. As index is an integer, only the value before the decimal point will be retained. So for every two digits that the index increments the index for, the cardBytes array will increase by one.

The calculation takes the last byte value and bitshifts it four places to the left. It then takes this number and carries out a bitwise OR (|) operation on it with the current value read. This has the effect of taking the first HEX value, which makes up the first four bits of the number and bit shifting it four places to the left. It then merges this number with the second HEX digit to give us the complete byte. So, if the first digit was 9 and the second was E, the calculation would do this:

```
Lastbyte = 9 =  00001001
00001001 << 4 = 10010000
E = 14 =        00001110
                10010000 OR
=               10011110
```

The checksum is a number you use to ensure that the entire string was read in correctly. Checksums are used a lot in data transmission; they're simply the result of each of the bytes of the entire data string XORed (exclusive OR) together.

```
if (index<10) {checksum ^= cardBytes[index/2];}
```

The ID number of your first card is:

```
3D00768B53
```

Therefore, its checksum will be:

```
3D XOR 00 XOR 76 XOR 8B XOR 53

3D = 00111101
00 = 00000000
76 = 01110110
8B = 10001011
53 = 01010011
```

If you XOR each of these digits to each other, you end up with 93. So 93 is the checksum for that ID number. If any of the digits were transmitted incorrectly due to interference, the checksum will come out as a value different than 93, so you know that the card was not read correctly and you discount it.

Outside of that loop, you set `lastByte` to the current value so next time around you have a copy of it:

```
lastByte=byteIn;
```

The index number is incremented:

```
index++;
```

If you have reached the end of the string, you must make sure that the tenth digit in the `cardNum` array is set to the ASCII code for \0 or the null terminator. This is necessary for later on when you need to determine if the end of the string has been reached or not. The null terminator shows that you are at the end of the string.

```
if (index==12) {cardNum[10] = '\0';}
```

Then you print the card number you have read in from the RFID reader:

```
Serial.println(cardNum); // print the card number
```

Next, an integer called cardIndex is created and set to the value returned from the `checkCard()` function (explained shortly). The `checkCard()` function will return a positive value if the card number is a valid one from the database and a negative number if it is not.

```
int cardIndex =checkCard(cardNum); // check if card is valid and return index number
```

You then check that the number returned is positive and also that the checksum is correct. If so, the card was read correctly and is valid, so you can unlock the door.

```
if(cardIndex>=0 && (cardBytes[5]==checksum)) { // if card number and checksum are valid
        Serial.println("Card Validated");
        Serial.print("User: ");
        Serial.println(names[cardIndex]);      // print the relevant name
        unlock();                              // unlock the door
        Serial.println();
    }
```

If the card is not valid or the checksum is incorrect, the card is ascertained to be invalid and the user is informed:

```
else {
        Serial.println("Card INVALID");
        tone(speakerPin, 250, 250);
        delay(250);
        tone(speakerPin, 150, 250);
        Serial.println();
    }
```

Next comes the `checkCard()` function. It will be returning an integer so this is its type and its parameter is the card number you pass to it.

```
int checkCard(char cardNum[10]) {
```

Next, you cycle through each of the cards in the database to see if it matches the card number you have read in:

```
for (int x=0; x<=users; x++) { // check all valid cards
```

You use a `strcmp`, or String Compare function, to ascertain if the card number passed to the `checkCard()` function and the card number in the current location of the database match each other. This is why you need a null terminator at the end of your card number as the `strcmp` function requires it.

```
if(strcmp(cardNum, cards[x])==0) { // compare with last read card number
```

The `strcmp` function requires two parameters. These are the two strings you wish to compare. The number returned from the function is a zero if both strings are identical. A non-zero number indicates they don't match. If they do match, you return the value of x, which will be the index in the card and name database of the valid card.

```
return (x); // return index of card number
```

If the cards do not match, you return a -1.

```
return (-1); // negative value indicates no match
}
```

The final function is `unlock()` which plays a high pitched tone, unlocks the door, waits for a preset length of time, and then relocks the door:

```
void unlock() {
    tone(speakerPin, 1000, 500);
    digitalWrite(lockPin, HIGH);
    delay(unlockLength);
    digitalWrite(lockPin, LOW);
}
```

The next step up from this project would be to add more readers and locks in order to secure your entire home. Authorized users would carry a card or tag to allow them into the relevant rooms. Individual access rights could be given to each user so that they have different access to different parts of the building, only allowing valid users into separate areas.

Now onto the final chapter of this book where you connect your Arduino to the Internet!

Summary

In this chapter you have seen how easy it is to read data from an RFID card or tag and then to use that data to unlock an electric strike lock or to take another kind of action. I have seen projects where an RFID key chain is attached to a bunch of keys. The RFID reader is in a bowl and when the user gets home they throw their keys into the bowl. The house reacts to that person coming home, e.g., setting their chosen temperature and light levels, playing their favorite music, turning a shower on, etc. When it comes to using an RFID reader, you are only limited by your own imagination.

Subjects and Concepts covered in Chapter 16

- How RFID technology works
- How to connect an ID12 RFID reader to an Arduino
- Reading serial data from an RFID reader
- Using a transistor to control a higher powered device
- Using the SoftwareSerial library
- Converting ASCII to hexadecimal values
- Using bitwise AND to ascertain if a number is odd or even
- Merging two HEX digits using bitshifts and bitwise OR to create a byte
- Creating checksums using XOR (Exclusive OR)
- Using strcmp to compare two strings

CHAPTER 17

Communicating over Ethernet

For this final chapter, you are going to take a look at how to connect your Arduino to your router so that data can be sent over an Ethernet cable. By doing this, you can read it from elsewhere on your network. You can also send data out to the Internet so it's viewable via a web browser. You will be using the official Arduino Ethernet Shield or an Arduino Ethernet to accomplish this.

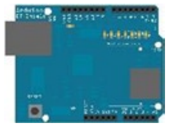
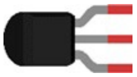
The ability to connect your Arduino to a network or the Internet opens up a whole new list of potential projects. You can send data to web sites, like posting Twitter updates. You can also control the Arduino over the Internet or use the Arduino as a web server to serve simple web pages containing sensor data, and so on. This chapter will give you the basic knowledge to create your own Ethernet or Internet Arduino-based projects.

Project 46 – Ethernet Shield

You'll now use the Ethernet Shield, or an Arduino Ethernet and a couple of temperature sensors to demonstrate accessing the Arduino over Ethernet.

Parts Required

Table 17-1. *Parts Required for Project 46*

Arduino Ethernet Shield Or Arduino Ethernet	
2 × DS18B20 Temperature Sensors	
4.7K ohm Resistor	

Connect It Up

Insert the Ethernet Shield on top of the Arduino, then connect everything as shown in Figure 17-1 with the wires going into the Ethernet Shield in the same place as they would on an Arduino.

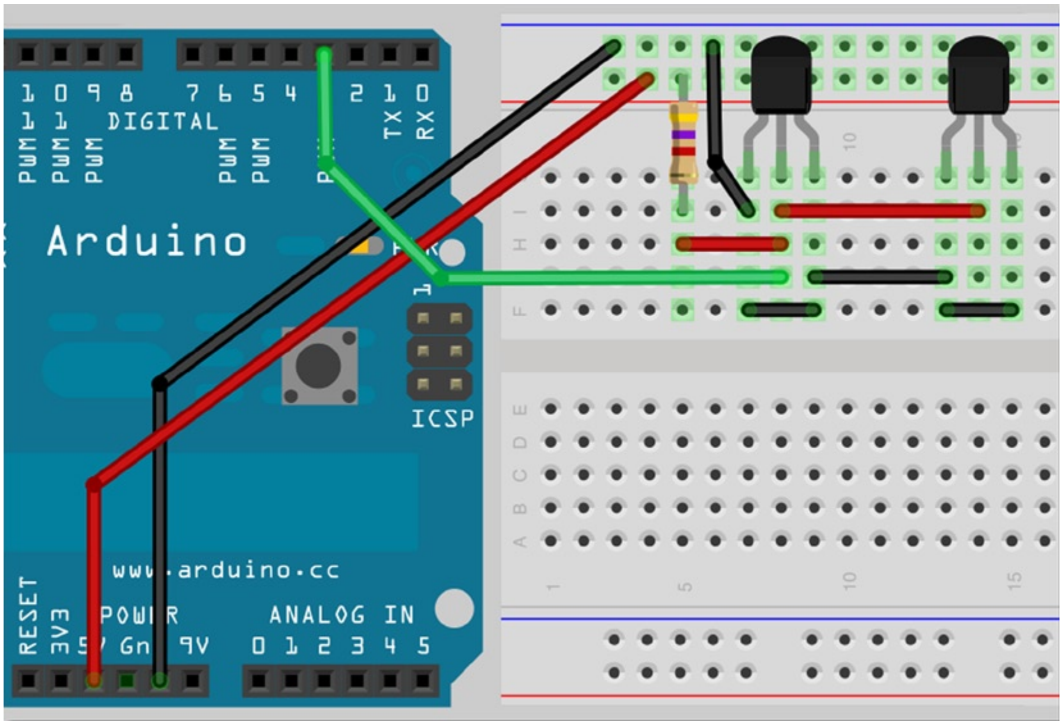


Figure 17-1. *The circuit for Project 46 – Ethernet Shield*

Enter the Code

Enter the code from Listing 17-1.

Listing 17-1. Code for Project 46

```
// Project 46 - Based on the Arduino Webserver example by David A. Mellis and Tom Igoe

#include <SPI.h>
#include <Ethernet.h>
#include <OneWire.h>
#include <DallasTemperature.h>

// Data wire is plugged into pin 3 on the Arduino
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12

float tempC, tempF;

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas
temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);
// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);
```

```
// arrays to hold device addresses
DeviceAddress insideThermometer = { 0x10, 0x7A, 0x3B, 0xA9, 0x01, 0x08, 0x00, 0xBF };
DeviceAddress outsideThermometer = { 0x10, 0xCD, 0x39, 0xA9, 0x01, 0x08, 0x00, 0xBE};

byte mac[] = { 0x48, 0xC2, 0xA1, 0xF3, 0x8D, 0xB7 };
byte ip[] = { 192,168,0, 104 };

// Start the server on port 80
EthernetServer server(80);

void setup()
{
  // Begin ethernet and server
  Ethernet.begin(mac, ip);
  server.begin();
    // Start up the sensors library
  sensors.begin();
    // set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
  sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
}

// function to get the temperature for a device
void getTemperature(DeviceAddress deviceAddress)
{
  tempC = sensors.getTempC(deviceAddress);
  tempF = DallasTemperature::toFahrenheit(tempC);
}
void loop()
{
  sensors.requestTemperatures();

  // listen for incoming clients
  EthernetClient client = server.available();
  if (client) {
    // an http request ends with a blank line
    boolean BlankLine = true;
    while (client.connected()) {
      if (client.available()) {
        char c = client.read();

        // If line is blank and end of line is newline character '\n' = end of HTTP request
        if (c == '\n' && BlankLine) {
          getTemperature(insideThermometer);
          getTemperature(outsideThermometer);
          // Display internal temp
          client.println("HTTP/1.1 200 OK\n"); // Standard HTTP response
          client.println("Content-Type: text/html\n");
          client.println("\n");
          client.println("<html>");
          client.println("<head>");
          client.println("<META HTTP-EQUIV=\"refresh\" CONTENT=\"5\">");
          client.println("<title>Arduino Web Server</title>");
          client.println("</head>");
          client.println("<body>");
          client.println("<h1>Arduino Web Server</h1>");
          client.println("<h3>Internal Temperature</h3>");
          client.println("Temp C:");
          client.println(tempC);
          client.println("<br/>");
          client.println("Temp F:");
          client.println(tempF);
          client.println("<br/>");
          // Display external temp
          client.println("<h3>External Temperature</h3>");
          client.println("Temp C:");
          client.println(tempC);
          client.println("<br/>");
          client.println("Temp F:");
          client.println(tempF);
          client.println("<br/>");
          client.println("</body>");
          client.println("</html>");

          break;
      }
        if (c == '\n') {
          // Starting a new line
          BlankLine = true;
        }
        elseif (c != '\r') {
          // Current line has a character in it
          BlankLine = false;
        }
      }
    }
    // Allow time for the browser to receive data
    delay(10);
    // Close connection
    client.stop();
  }
}
```

You will need to enter the two address numbers of the temperature sensors (See Project 37) in this line:

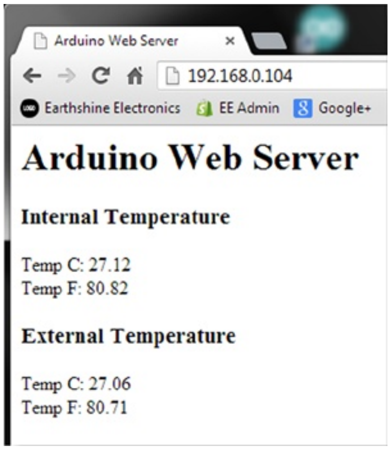
```
byte ip[] = { 192,168,0, 104 };
```

You will also need to change the IP address to one of your own. To do this, you will need to find out from your router what IP address range has been set aside for devices on your computer. Usually, the address will start off as 192.168.0 or 192.168.1—then you just add another number higher than about 100 to make sure it does not interfere with existing devices. You may also need to go into your router settings to ensure that any HTTP requests to port 80 are forwarded to the IP address of the Ethernet Shield. Look under ”Port Forwarding” in your router manual. It may also be necessary to open port 80 in your firewall.

Now, open up your web browser and type in the IP address and port, e.g.

```
192.168.0.104:80
```

If everything is working correctly, you will get the web page shown in Figure 17-2 in your browser.



Arduino Web Server

Internal Temperature

Temp C: 27.12
Temp F: 80.82

External Temperature

Temp C: 27.06
Temp F: 80.71

Figure 17-2. *The web browser output from the Arduino web server*

The page will auto refresh every five seconds to show any changes in the temperatures. If youhave set up the port forwarding and firewall correctly in your router, you will also be able to access the page from anywhere that has Internet access. You will need to know the IP address of the router, which can be found from the router's administration page. Type it, followed by the port number, into any web browser, e.g.

```
95.121.118.204:80
```

The above web page will now show up in the browser and you can check the temperature readings from anywhere you have Internet access.

Things You Need to Know about Networking

DHCP (Dynamic Host Configuration Protocol) is an autoconfiguration protocol used on IP networks. It allows the Ethernet Shield to automatically be assigned an IP address from one that is available from the router. Previously, you manually set the IP address; this time you use the Ethernet library to auto-assign one for you. The tradeoff is that your code is much longer.

The MAC (Media Access Control) address is a unique identifier for network interfaces. The network card in your PC or Mac will have had its MAC address set by the manufacturer. In your case, you are deciding what the MAC address is. It is simply a 48-bit number, so just put any six hexadecimal digits into the address, although leaving it as it is in the code will be fine. The IP address will need to be manually set and it must be one from the range allowed by your router.

Project 46 – Ethernet Shield – Code Overview

Some parts of this code are repeated from Project 37, so I will gloss over those sections and instead concentrate on the parts relating to the Ethernet Shield. First, you load in the libraries. Make sure you have the libraries for the temperature sensors in your libraries folder first (see Project 37). Note that as of Arduino IDE version 0019, it has been necessary to include the SPI.h library in any project that requires the Ethernet.h library.

```
#include <SPI.h>
#include <Ethernet.h>
#include <OneWire.h>
#include <DallasTemperature.h>
```

Next, the pin and precision for the sensors is set

```
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12
```

along with the two floats you will use to store the temperature in Celsius and Fahrenheit.

```
float tempC, tempF;
```

An instance of the oneWire object is created and you pass a reference to the Dallas Temperature library:

```
OneWire oneWire(ONE_WIRE_BUS);

DallasTemperature sensors(&oneWire);
```

The addresses for the two temperature sensors are set. Remember to find out what these are using the code in Project 37 if necessary.

```
DeviceAddress insideThermometer = { 0x10, 0x7A, 0x3B, 0xA9, 0x01, 0x08, 0x00, 0xBF };
DeviceAddress outsideThermometer = { 0x10, 0xCD, 0x39, 0xA9, 0x01, 0x08, 0x00, 0xBE};
```

Next, you need to define the MAC and IP address of the device:

```
byte mac[] = { 0x48, 0xC2, 0xA1, 0xF3, 0x8D, 0xB7 };
byte ip[] = { 192,168,0, 104 };
```

Next, an instance of an Ethernet server object is created along with the port number for the device:

```
EthernetServer server(80);
```

The server will listen for incoming connections on the specified port. A port number is simply a pathway for data. You only have one Ethernet cable going into your device but the port number decides where that data will go. Imagine the MAC address as being the building address of a large apartment block and the port number the individual number of the apartment.

Next comes the setup routine. You start by initializing the Ethernet communications and passing the MAC and IP address of the device to the instance:

```
Ethernet.begin(mac, ip);
```

Now you need to tell your server to start listening to incoming connections using the `begin()` command:

```
server.begin();
```

You also need to start your sensors and set their resolution:

```
sensors.begin();
sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
```

Next, you create the function to obtain the temperatures from the sensor (as was done in Project 37):

```
void getTemperature(DeviceAddress deviceAddress)
{
  tempC = sensors.getTempC(deviceAddress);
  tempF = DallasTemperature::toFahrenheit(tempC);
}
```

Next comes the main program loop. First, you request the temperatures from the two sensors:

```
sensors.requestTemperatures();
```

You need to listen for any incoming clients, i.e. web pages requesting to view the web page served by the Arduino. To do this, you create an instance of type EthernetClient and use it to check that there is data available for reading from the server. The client is the web browser that will connect to the Arduino. The server is the Arduino.

```
EthernetClient client = server.available();
```

Next, you check if a client has connected and if any data is available for it. If true, the code belonging to the `if()` statement is executed.

```
if (client) {
```

First, the `if` statement creates a Boolean variable called `BlankLine` and sets it to true:

```
boolean BlankLine = true;
```

A HTTP request from the client will end with a blank line, terminated with a newline character. So you use the `BlankLine` variable to determine if you have reached the end of the data or not.

Next, you check if the client is still connected or not, and if so, run the code within the `while` loop:

```
while (client.connected()) {
```

Next, you check if data is available for the client or not. If data is available, the code within the next `if` statement is executed. The `available()` command returns the number of bytes that have been written to the client by the server it is connected to. If this value is above zero, then the `if` statement runs.

```
if (client.available()) {
```

Then a variable of type char is created to store the next byte received from the server. Use the client.read() command to obtain the byte.

```
char c = client.read();
```

If the character read is a newline ('\n') character, you also need to check if `BlankLine` is true or not. If so, you have reached the end of the HTTP request and so can serve the HTML code to the client (the user's web browser).

```
if (c == '\n' && BlankLine) {
```

Next comes the data you will send out from your server. You start by obtaining the temperature from the internal and external sensors.

```
getTemperature(insideThermometer);
getTemperature(outsideThermometer);
```

Next comes the HTML code you have to issue to the client. Every page is made up of code called HTML (or HyperText Markup Language). Explaining HTML is beyond the scope of this book, so I will just give some basic information only. If you wish to learn more about HTML, check out the HTML entry on Wikipedia at `http://en.wikipedia.org/wiki/HTML`. There are also plenty of HTML tutorials available on the Internet. You use the client.println() command to issue data to the client. Basically, you send out the code to create a web page. If you right click in a web page in most browsers, you will be given the option to view the source code. Try this and you will see the HTML code that makes up the web page you have just been viewing. The code tells the browser what to display and how to display it.

First, you tell the client that you are using HTTP version 1.1, which is the standard protocol used for issuing web pages, and that the content you are about to send is HTML:

```
client.println("HTTP/1.1 200 OK\n"); // Standard HTTP response
client.println("Content-Type: text/html\n\n");
client.println("\n");
```

Next, you have the HTML tag to say that everything from now on will be HTML code and the head tag of the HTML code. The head contains any commands you wish to issue to the browser, scripts you want to run, etc. before the main body of the code. The first command tells the browser that you want the page to automatically refresh every five seconds.

```
client.println("<html>");
client.println("<head>");
client.println("<META HTTP-EQUIV=\"refresh\" CONTENT=\"5\">");
```

Then you give the page a title. It will appear at the top of the browser and in any tabs you have for that page.

```
client.println("<title>Arduino Web Server</title>");
client.println("</head>");
```

You end the head section by inserting a </head> tag. Next is the body of the HTML. This is the part that will be visible to the user.

```
client.println("<body>");
```

You display a <h1> heading saying "Arduino Web Server". H1 is the largest heading, followed by H2, H3, etc.

```
client.println("<h1>Arduino Web Server</h1>");
```

The code then has the title of the next section, which is “Internal Temperature” as an h3 heading

```
client.println("<h3>Internal Temperature</h3>");
```

Then, you print the temperature in C and F followed by line breaks <br/>:

```
client.println("Temp C:");
client.println(tempC);
client.println("<br/>");
client.println("Temp F:");
client.println(tempF);
client.println("<br/>");

client.println("<h3>External Temperature</h3>");
client.println("Temp C:");
client.println(tempC);
client.println("<br/>");
client.println("Temp F:");
client.println(tempF);
client.println("<br/>");
client.println("</body>");
client.println("</html>");
```

Then the `while` loop is exited with a break command:

```
break;
```

You now set `BlankLine` to true if a \n (newline) character is read and false if it is not a \r (Carriage Return), i.e. there are still characters to be read from the server.

```
if (c == '\n')
{
   // Starting a new line
   BlankLine = true;
}
elseif (c != '\r')
{
   // Current line has a character in it
   BlankLine = false;
}
```

You wait a short delay to allow the browser time to receive the data and then stop the client with a `stop()` command. This disconnects the client from the server.

```
delay(10);
client.stop();
```

This project is a basic introduction to serving a web page with sensor data embedded in it via the Ethernet Shield. There are nicer ways of presenting data over the Internet and you will look at one of those methods next.

Project 47 — Internet Weather Display

You are now going to use the same parts and circuit, but this time you'll send the temperature data for the two sensors to Xively (formally Cosm and then formally Pachube). Xively is an online database service that allows users to connect sensor data to the Web (See Figure 17-3). Data can be sent in a variety of formats and shows up in a feed on the website in the form of a graph and live data. The graphs are real-time and can be embedded into a website. You can also view historical data pulled from a feed as well as send alerts to control scripts, devices, etc. There is a series of tutorials on the website dedicated to the Arduino.

NEWS Xively and ARM announce strategic collaboration, release Jumpstart Kit.

The Internet of Things is Open for Business.

Build your connected product, here and now.

Xively makes building connected products easy

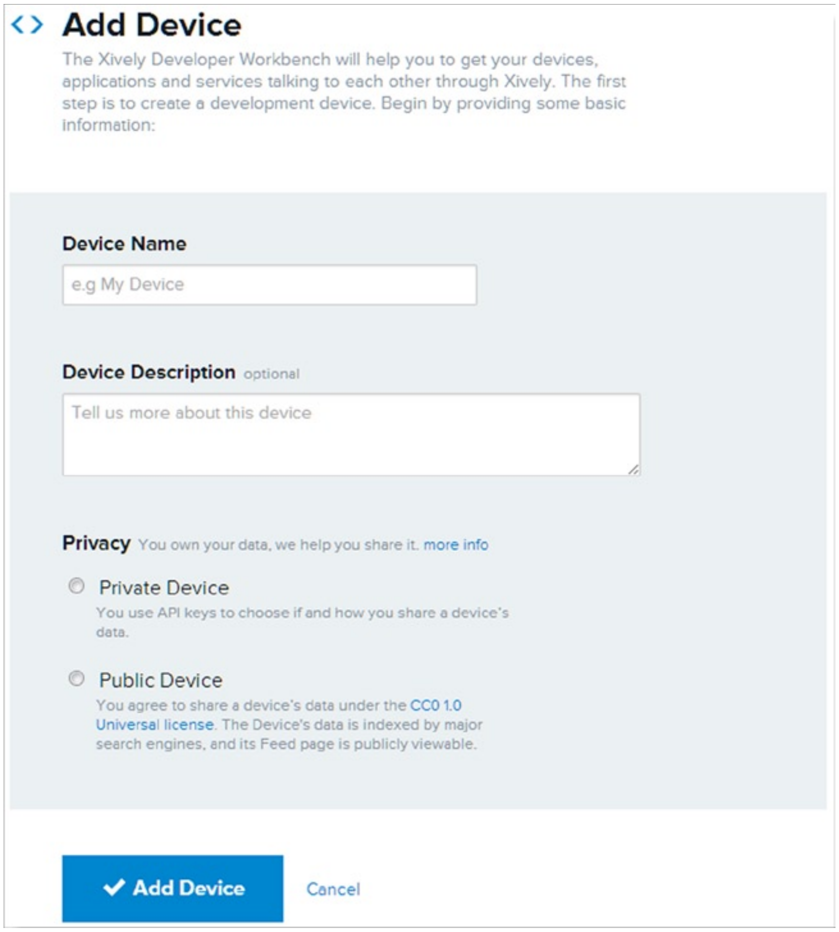
The Internet of Things promises to revolutionize the relationship between people and their world, but bringing transformative connected products to market takes more than a great idea. It requires tools, infrastructure and know-how that most businesses don't have the time, money or capacity to obtain. However, we have a solution! **Xively is a Platform as a Service** that provides everything you need to **simplify and accelerate** the creation of compelling **connected products and solutions**. With Xively, you can focus on innovation instead of infrastructure.

Figure 17-3. *The Xively website*

To use the service, you must sign up for a Xively developer account. It is an ideal way of displaying data from sensors, home energy monitors, building monitor systems, and so on. The service is completely free for a developer account and up to five connected devices. For commercial use of the service charges will apply.

A developer account must be created before you can upload data, so start by going to the website at `www.xively.com` and click the "SIGN UP" button at the top right. A box will appear to enter your username, email, and password. Then a box appears that says "Tell us a bit about yourself."

Once you have successfully signed up, you will be logged in and taken to the Development Devices page. Now you need to create a device and a feed. Click the + Add Device button. You will then be presented with a window that asks for a Device Name and Device Description; you then choose in the privacy section whether you want the device and its data to be Private or Public (See Figure 17-4). Name your device "Arduino" and enter a description if you wish. Then choose if you wish your data to be Private or Public.



Add Device

The Xively Developer Workbench will help you to get your devices, applications and services talking to each other through Xively. The first step is to create a development device. Begin by providing some basic information:

Device Name

e.g My Device

Device Description optional

Tell us more about this device

Privacy You own your data, we help you share it. more info

Private Device
You use API keys to choose if and how you share a device's data.

Public Device
You agree to share a device's data under the CC0 1.0 Universal license. The Device's data is indexed by major search engines, and its Feed page is publicly viewable.

Figure 17-4. *Choose your device type*

Once you have created a device, next you need to create a channel. A channel is simply a data feed from the device. In the case of this project we will create four channels, for the External Temperature in Celsius, also in Fahrenheit, and two more for the Internal Temperature in Celsius and Fahrenheit. Give the channel a name, and then add the units (Celsius or Fahrenheit) followed by the symbol C or F. Leave the current value box empty. (See Figure 17-5).

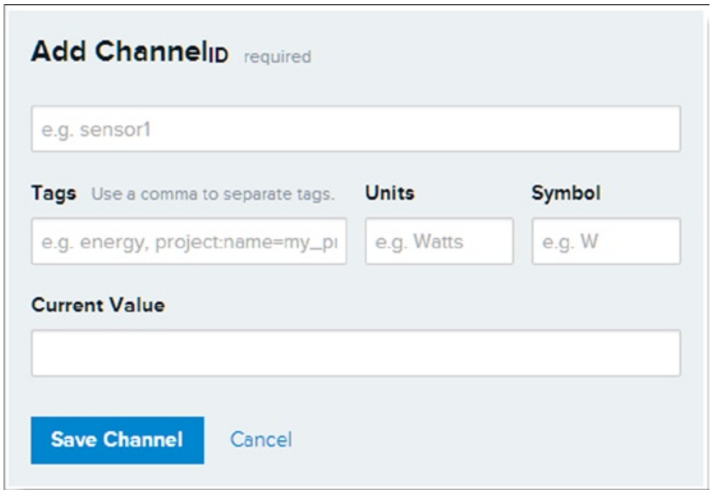
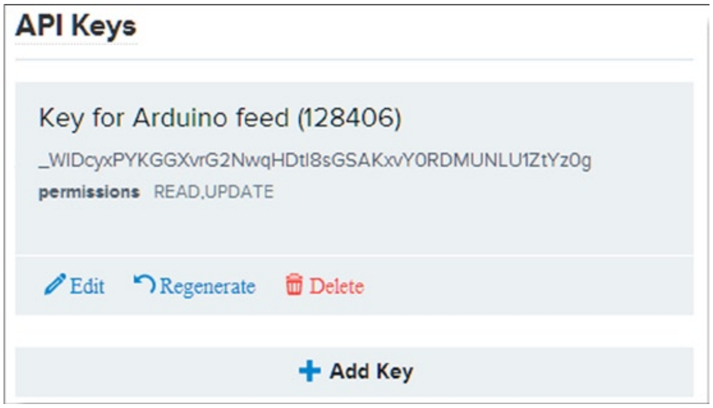


Figure 17-5. *The screen seen after successfully creating your channel*

Once you have created your channel it is ready to receive data from your Arduino. First, write down your Channel ID and also the API (Application Programming Interface) Key. You can get the Channel ID by clicking the name of the channel at the top of the Developer Center. The API Key is along the right hand side of the page. The API Keys page will list the keys for all the devices you have created (See Figure 17-6). Copy and paste the key into Notepad or some other text editor and save it for later use.


API Keys

Key for Arduino feed (128406)

_WIDcyxPYKGGXvrG2NwqHDtl8sGSAKxvY0RDMUNLU1ZtYz0g

permissions READ,UPDATE

Figure 17-6. *The API Key*

Now that you have created your Xively account, generated a device, copied your Channel ID number and API Key, you are ready to enter the code.

Enter the Code

Enter the code from Listing 17-2. You will need the Dallas Temperature library and also the HttpClient and Xively libraries which can be downloaded from the Xively website at https://github.com/xively/xively_arduino. The instructions show you how to install the HttpClient library and the Xively library.

Listing 17-2. Code for Project 47

```
// Project 47 - Based on the Xively Arduino examples

#include <SPI.h>
#include <Ethernet.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <HttpClient.h>
#include <Xively.h>

#define SHARE_FEED_ID         128497     // this is your xively feed ID that you want to share to
#define xivelyKey  "_WlvrG2NwORxPYKGGDMUNLqHDtl8sDcyXGSAKxvYU1ZtYzOg" // fill in your API key

char sensorId1[] = "IntTempC";
char sensorId2[] = "IntTempF";
char sensorId3[] = "ExtTempC";
char sensorId4[] = "ExtTempF";
XivelyDatastream datastreams[] = {
  XivelyDatastream(sensorId1, strlen(sensorId1), DATASTREAM_FLOAT),
  XivelyDatastream(sensorId2, strlen(sensorId2), DATASTREAM_FLOAT),
  XivelyDatastream(sensorId3, strlen(sensorId3), DATASTREAM_FLOAT),
  XivelyDatastream(sensorId4, strlen(sensorId4), DATASTREAM_FLOAT)
};
// Finally, wrap the datastreams into a feed
XivelyFeed feed(SHARE_FEED_ID, datastreams, 4 /* number of datastreams */);

// Data wire is plugged into pin 3 on the Arduino
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas
temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device addresses
DeviceAddress insideThermometer = { 0x28, 0x44, 0x12, 0xC2, 0x03, 0x00, 0x00, 0x92 };
DeviceAddress outsideThermometer = { 0x28, 0xA5, 0x02, 0xC2, 0x03, 0x00, 0x00, 0xF0 };
```

```
unsigned int interval;
float itempC, itempF, etempC, etempF;

byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0xF7, 0x38 }; // make sure this is unique on your network

EthernetClient localClient;
XivelyClient xivelyclient(localClient);

void setup()
{
  Serial.begin(9600);
  Serial.println("Starting 4 stream data upload to Xively...");
  Serial.println();
  while (Ethernet.begin(mac) != 1)
  {
    Serial.println("Error getting IP address via DHCP, trying again...");
    delay(15000);
  }
      // Start up the sensors library
  sensors.begin();
    // set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
  sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
  delay(100);
}

void xively_in_out() {
  getTemperatures();
  datastreams[0].setFloat(itempC);
  datastreams[1].setFloat(itempF);
  datastreams[2].setFloat(etempC);
  datastreams[3].setFloat(etempF);
  xivelyclient.put(feed, xivelyKey);
  Serial.println("Read sensor value ");
  Serial.println(datastreams[0].getFloat());
  Serial.println(datastreams[1].getFloat());
  Serial.println(datastreams[2].getFloat());
  Serial.println(datastreams[3].getFloat());
  delay(10000);
}

// function to get the temperature for a device
void getTemperatures()
{
  sensors.requestTemperatures();
  itempC = sensors.getTempC(insideThermometer);
  itempF = DallasTemperature::toFahrenheit(itempC);
  etempC = sensors.getTempC(outsideThermometer);
  etempF = DallasTemperature::toFahrenheit(etempC);
}
```

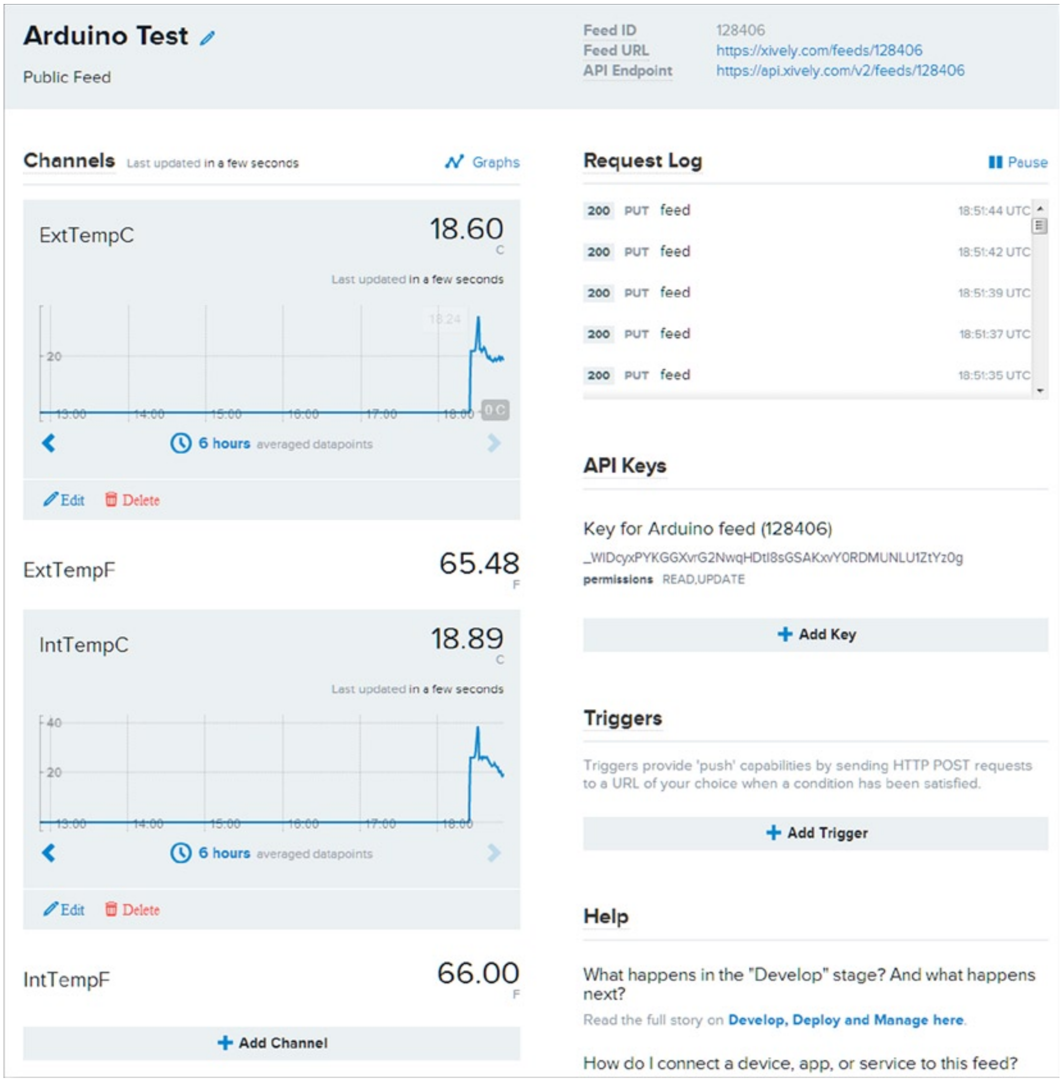
```
void loop()
{
  xively_in_out();
}
```

Upload the code and then open up the serial monitor. If everything is working correctly and you are successfully sending data to Xively, the output will look something like:

```
Starting 4 stream data upload to Xively...

Read sensor value
27.00
80.60
27.06
80.71
Read sensor value
26.94
80.49
27.00
80.60
```

Now open your web browser and go to `www.Xively.com`. Navigate to your feed in Developer Center and click the name of the device. You will now be shown the device page. The temperature readings from the sensors will be displayed over time. By clicking the clock symbol underneath each data stream you can change the time period from the default of 6 hours to whatever you want. The date and time the feed was last updated is shown to the right of the channels in the Request Log (See Figure 17-7). The graphs should also be showing the temperatures over time with the times along the bottom of the graph. If you leave this for a considerable length of time, you should see the temperature changes throughout the day.



Arduino Test

Public Feed

Feed ID 128406
Feed URL https://xively.com/feeds/128406
API Endpoint https://api.xively.com/v2/feeds/128406

Channels Last updated in a few seconds

Graphs

ExtTempC 18.60 C

Last updated in a few seconds

6 hours averaged datapoints

Edit Delete

ExtTempF 65.48 F

IntTempC 18.89 C

Last updated in a few seconds

6 hours averaged datapoints

Edit Delete

IntTempF 66.00 F

Add Channel

Request Log

Pause

```
200 PUT feed    18:51:44 UTC
200 PUT feed    18:51:42 UTC
200 PUT feed    18:51:39 UTC
200 PUT feed    18:51:37 UTC
200 PUT feed    18:51:35 UTC
```

API Keys

Key for Arduino feed (128406)

_WIDcyxPYKGGXvrG2NwqHDtI8sGSAKxvY0RDMUNLU1ZtYz0g

permissions READ,UPDATE

Add Key

Triggers

Triggers provide 'push' capabilities by sending HTTP POST requests to a URL of your choice when a condition has been satisfied.

Add Trigger

Help

What happens in the "Develop" stage? And what happens next?

Read the full story on **Develop, Deploy and Manage here.**

How do I connect a device, app, or service to this feed?

Figure 17-7. *A live Xively feed page*

Clicking the clock icon on each graph will allow you to choose to view the raw data points over 5 minutes, 30 minutes or 1 hour. You can also select averaged data points over 6 hours, 1 day, 7 days, 1 month and 3 months.

You can modify the code to add other temperature sensors such as the pressure sensor we used in Project 31, light sensors to measure ambient light, humidity sensors, and wind speed sensors to make a fully fledged weather station with your data logged and accessible over the Internet on Xively.

Now let's look at the code for this project to see how it works.

Project 47 – Internet Weather Display – Code Overview

The code starts off with the includes for the Ethernet Shield and the one wire temperature sensors:

```
#include <SPI.h>
#include <Ethernet.h>
#include <OneWire.h>
#include <DallasTemperature.h>
```

Also, the new libraries for using Xively are required. These are the Http Client library and the Xively library. The Xively library is at https://github.com/xively/xively_arduino Click the Download ZIP button on the right. The HTTP Client library can be found at https://github.com/amcewen/HttpClient. Again, click the Download ZIP button. Unzip the libraries and put them, still in their respective folders, inside the Arduino libraries folder. The pages also include instructions how to use the libraries.

```
#include <HttpClient.h>
#include <Xively.h>
```

Then we use four define statements to set up some manifest constants; the first one is to store the feed ID number and the second the API Key for the feed. Make sure you enter your own feed ID and API Key. These can be found in your Xively settings and Feed page.

```
#define SHARE_FEED_ID   128406
#define xivelyKey  "_WlDcyxPYKGGXvrG2NwqHDtl8sGSAKxvYORDMUNLU1ZtYzOg"
```

Next we need to create a data stream to be uploaded to Xively. We will have four data streams for the internal and external temperatures both in Celsius and Fahrenheit. We create some variables of type char to store an array of characters that will make up the names of the feed streams. These are the labels for each sensor that show up on the data graph on the Xively feed page.

```
char sensorId1[] = "IntTempC";
char sensorId2[] = "IntTempF";
char sensorId3[] = "ExtTempC";
char sensorId4[] = "ExtTempF";
```

Then we create an array of type XivelyDatastream to hold the streams. This data type is only relevant to the Xively library. Each stream must have three parameters and these are the name of the stream, the length of the name and the type of data being uploaded. In our case we are uploading temperature data as a float. We could also send the value as an int using DATASTREAM_INT or as a string using DATASTREAM_STRING and as a char buffer using DATASTREAM_BUFFER. Refer to the readme file for the Xively library for further information.

```
XivelyDatastream datastreams[] = {
  XivelyDatastream(sensorId1, strlen(sensorId1), DATASTREAM_FLOAT),
  XivelyDatastream(sensorId2, strlen(sensorId2), DATASTREAM_FLOAT),
  XivelyDatastream(sensorId3, strlen(sensorId3), DATASTREAM_FLOAT),
  XivelyDatastream(sensorId4, strlen(sensorId4), DATASTREAM_FLOAT)
};
```

Next, we need to be able to send this data to a remote system, so we wrap those streams into a feed using XivelyFeed. We give the feed a name and its three parameters are the ID number of the feed, the datastream array name and the number of datastreams in the array. The XivelyDatastream function gathers together the data to be uploaded to Xively and the XivelyFeed function creates a feed we will send the stream to.

```
XivelyFeed feed(SHARE_FEED_ID, datastreams, 4 /* number of datastreams */);
```

As in Chapter 15 we then define the pin we will use for the one wire bus to connect to the temperature sensors, and the precision used.

```
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12
```

We create an instance of One Wire to enable us to communicate via the Arduino's one wire bus.

```
OneWire oneWire(ONE_WIRE_BUS);
```

Then we send a reference to that One Wire instance to the Dallas Temperature library.

```
DallasTemperature sensors(&oneWire);
```

Next we create the two device addresses for the internal and external temperature sensors.

```
DeviceAddress insideThermometer = { 0x28, 0x44, 0x12, 0xC2, 0x03, 0x00, 0x00, 0x92 };
DeviceAddress outsideThermometer = { 0x28, 0xA5, 0x02, 0xC2, 0x03, 0x00, 0x00, 0xF0 };
```

Four variables of type float are created to store the internal and external temperatures in both degrees C and F.

```
float itempC, itempF, etempC, etempF;
```

To use the Ethernet library we need the MAC address (Media Access Control) of the Ethernet device we are going to use, in this case our Ethernet Shield. A MAC address is a unique identifier assigned to network devices. On the modern Ethernet Shields the address is on a sticker on the underside of the device. If the device doesn't have a predetermined MAC address then just make one up. The address is made up of 12 hexadecimal characters.

```
byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0xF7, 0x38 };
```

Next, we use the EthernetClient function to create a client that can connect to a specific Internet IP address and port (usually defined using the client.connect() function, but the Xively library takes care of that for us) and give the client a name.

```
EthernetClient localClient;
```

Then we instantiate a Xively client and pass it the name of the EthernetClient instance.

```
XivelyClient xivelyclient(localClient);
```

Now we come to the setup() function.

```
void setup()
{
```

We will use the serial monitor for feedback, so begin serial communications and set the baud rate.

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

Then inform the user what we are about to do.

```
Serial.println("Starting 4 stream data upload to Xively...");
Serial.println();
```

The Ethernet.begin() function initializes the Ethernet library and network settings. It will return a true or false flag if successful. We use a While function to check if communication via Ethernet is successful or not and inform the user accordingly.

```
while (Ethernet.begin(mac) != 1)
{
  Serial.println("Error getting IP address via DHCP, trying again...");
  delay(15000);
}
```

As in Chapter 15 we use the Dallas Temperature library to begin communications with the sensors and set their resolution.

```
  sensors.begin();
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
  sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
  delay(100);
}
```

After the setup() function we have our own function called xively_in_out which will send our datastreams to Xively.

```
void xively_in_out() {
```

First, the `getTemperatures()` function is called to read and store the temperature readings.

```
getTemperatures();
```

Then, in the first element of the datastream array variable, we store the first temperature and make sure it is set to a floating point number using the setFloat() function of the Xively library. We could also set it to an int, string or char buffer using setInt() setString() or setBuffer() accordingly.

```
datastreams[0].setFloat(itempC);
datastreams[1].setFloat(itempF);
datastreams[2].setFloat(etempC);
datastreams[3].setFloat(etempF);
```

Now the stream is ready to send to Xively. We use the put command to send the stream. The put command is added to the end of the Xively client name with the . notation followed by the two required parameters or the feed ID and the API key.

```
xivelyclient.put(feed, xivelkyKey);
```

Finally the user is informed of what data has been uploaded using the getFloat() function followed by a 10 seconds delay.

```
  Serial.println("Read sensor value ");
  Serial.println(datastreams[0].getFloat());
  Serial.println(datastreams[1].getFloat());
  Serial.println(datastreams[2].getFloat());
  Serial.println(datastreams[3].getFloat());
  delay(10000);
}
```

The next function is the same one as in Chapter 15 that reads the sensors and assigns the values to the respective variables.

```
void getTemperatures()
{
  sensors.requestTemperatures();
  itempC = sensors.getTempC(insideThermometer);
  itempF = DallasTemperature::toFahrenheit(itempC);
  etempC = sensors.getTempC(outsideThermometer);
  etempF = DallasTemperature::toFahrenheit(etempC);
}
```

Finally, our main program function simply repeatedly calls the `xively_in_out()` function.

```
void loop()
{
  xively_in_out();
}
```

Now that you know the basic methods of sending sensor data to Xively to be stored and viewed, it will be a relatively easy task for you to modify the code to add further sensors or other data.

So far, you have learned how to send data over Ethernet to a web browser on the network, to a web browser over the Internet, and to the Xively data storage and graphing service. Next, you will learn how to make the Arduino send an e-mail to alert you when the temperatures get too hot or too cold.

Project 48 — Email Alert System

You are now going to look at a different method of sending data. In this project, you will get the Arduino with the Ethernet Shield to send an e-mail when a temperature is either too cold or too hot. This project is designed to show you the basics of sending an e-mail via the Ethernet Shield. You'll use the same circuit, but with just one of the temperature sensors.

Enter the Code

Enter the code from Listing 17-3. You will need to obtain the IP address of your SMTP email server. To do this, open up a terminal window (command window, console, whatever you know it as on your system) and type in ping, followed by the web address you wish to obtain the IP address of. In other words, if you wanted to know the IP address of the SMTP server at `smtp.livehotgmail.com`, you would type

```
ping smtp.livehotgmail.com
```

and you will get back a reply similar to (note: this is not a real web address)

```
PING smtp.hot.glbdns.livehotgmail.com (10.75.161.202): 56 data bytes
```

This shows you that the IP address is 10.75.161.202. Do this for the SMTP server of your e-mail service and enter this into the server section of the code. If your SMTP server requires authentication, you will need to obtain the Base-64 version of your username and password. There are many websites that will do this for you, such as

www.motobit.com/util/base64-decoder-encoder.asp

Enter your username and encrypt it to Base-64 and then do the same with your password. Copy and paste the results into the relevant section in the code. Also, change the FROM and TO sections of the code to your own e-mail address and the e-mail address of the recipient.

Listing 17-3. Code for Project 48

```
// Project 48 - Email Alert System

#include <Ethernet.h>
#include <SPI.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define time 1000
#define emailInterval 60
#define HighThreshold 28
#define LowThreshold 10

// Data wire is plugged into pin 3 on the Arduino
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12

float tempC, tempF;
char message1[35], message2[35];
char subject[] = "ARDUINO: TEMPERATURE ALERT!!\0";
unsigned long lastMessage;

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas
temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device addresses
DeviceAddress insideThermometer = { 0x28, 0x44, 0x12, 0xC2, 0x03, 0x00, 0x00, 0x92 };

byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0xF7, 0x38 };
byte ip[] = { 192,168,0, 104 };
byte server[] = { 10, 254, 30, 60 };  // Mail server address. Change this to your own mail servers IP.
```

```
EthernetClient client;

void sendEmail(char subject[], char message1[], char message2[], float temp) {
  Serial.println("connecting...");

 if (client.connect(server, 25)) {
   Serial.println("connected");
   client.println("EHLO MYSERVER");  delay(time);
   client.println("AUTH LOGIN"); delay(time);
   client.println("lcbWNybbWl2JttnRzLmNvrZSbQ==");  delay(time);
   client.println("GV0yVGbjLlnZ2VEw");  delay(time);
   client.println("MAIL FROM:<sales@earthshineelectronics.com>");      delay(time);
   client.println("RCPT TO:<fred@crunchytoad.com>");      delay(time);
   client.println("DATA");       delay(time);
   client.println("From: < sales@earthshineelectronics.com >");       delay(time);
   client.println("To: < fred@crunchytoad.com >");       delay(time);
   client.print("SUBJECT: ");
   client.println(subject);       delay(time);
   client.println();      delay(time);
   client.println(message1);      delay(time);
   client.println(message2);      delay(time);
   client.print("Temperature: ");
   client.println(temp);   delay(time);
   client.println(".");      delay(time);
   client.println("QUIT");      delay(time);
   Serial.println("Email sent.");
   lastMessage=millis();
 } else {
   Serial.println("connection failed");
 }

}

void checkEmail() {
  while (client.available()) {
   char c = client.read();
   Serial.print(c);
 }

 if (!client.connected()) {
   Serial.println();
   Serial.println("disconnecting.");
   client.stop();
 }
}

// function to get the temperature for a device
void getTemperature(DeviceAddress deviceAddress)
{
  tempC = sensors.getTempC(deviceAddress);
  tempF = DallasTemperature::toFahrenheit(tempC);
}
```

```
void setup()
{
 lastMessage = 0;
 Ethernet.begin(mac, ip);
 Serial.begin(9600);

     // Start up the sensors library
  sensors.begin();
    // set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);

 delay(1000);
}

void loop()
{
  sensors.requestTemperatures();
  getTemperature(insideThermometer);
  Serial.println(tempC);
  if (tempC >= HighThreshold && (millis()>(lastMessage+(emailInterval*1000)))) {
    Serial.println("High Threshhold Exceeded");
    char message1[] = "Temperature Sensor\0";
    char message2[] = "High Threshold Exceeded\0";
    sendEmail(subject, message1, message2, tempC);
  }
  else if (tempC<= LowThreshold && (millis()>(lastMessage+(emailInterval*1000)))) {
    Serial.println("Low Threshhold Exceeded");
    char message1[] = "Temperature Sensor\0";
    char message2[] = "Low Threshold Exceeded\0";
    sendEmail(subject, message1, message2, tempC);
  }

    if (client.available()) {checkEmail();}
}
```

Upload the code and then open up the serial monitor window. The serial monitor will display the temperature from the first sensor repeatedly. If the temperature drops below the LowThreshold value, the serial monitor will display “Low Threshold Exceeded” and then send the relevant e-mail alert. If the temperature goes above the HighThreshold, it will display the message “High Threshold Exceeded” and send the appropriate alert for a high temperature situation.

You can test this by setting the high threshold to be just above the ambient temperature and then holding the sensor until the temperature rises above the threshold. This will set the alert system into action.

Note that for the first 60 seconds the system will ignore any temperature alert situations. It will only start sending alerts once 60 seconds have passed. If the thresholds have been breached, the alert system will keep sending e-mails until the temperature drops to within acceptable levels. E-mails will be sent every `emailInterval` seconds when the thresholds have been breached. You can adjust this interval to your own settings.

After an e-mail is sent, the system will wait until a successful receipt has been received back from the client, and then it will display the response. You can use this data to debug the system if things do not work as planned.

Project 48 – Email Alert System — Code Overview

First, the libraries are included:

```
#include <Ethernet.h>
#include <SPI.h>
#include <OneWire.h>
#include <DallasTemperature.h>
```

Next, you define the delay, in milliseconds, when sending data to the server

```
#define time 1000
```

This define is followed by a time, in seconds, in-between each e-mail being sent.

```
#define emailInterval 60
```

Then you need to set the temperature high and low levels that will cause an alert:

```
#define HighThreshold 40 // Highest temperature allowed
#define LowThreshold 10  // Lowest temperature
```

Next, you set the pin and precision for the sensors

```
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12
```

Declare the floats to store the temperatures,

```
float tempC, tempF;
```

then a pair of character arrays that will store the message in the e-mail

```
char message1[35], message2[35];
```

Create another character array to store the subject of the e-mail. This is declared and initialized:

```
char subject[] = "ARDUINO: TEMPERATURE ALERT!!\0";
```

As you don't want to bombard the user with e-mail messages once the thresholds have been breached, you need to store the time the last e-mail was sent. This will be stored in an unsigned integer called lastMessage:

```
unsigned long lastMessage;
```

The instances for the sensor are set up along with the sensor address:

```
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

DeviceAddress insideThermometer = { 0x10, 0x7A, 0x3B, 0xA9, 0x01, 0x08, 0x00, 0xBF };
```

The MAC and IP address of the Ethernet Shield is defined. Note that the MAC address needs to be a unique number to differentiate the device from the billions of other Internet-connected devices in the world. Generating a random address is usually enough. Note that it isn't possible to connect to the device from outside your own home network without a more advanced network setup:

```
byte mac[] = { 0x64, 0xB9, 0xE8, 0xC3, 0xC7, 0xE2 };
byte ip[] = { 192,168,0, 105 };
```

Then you set the IP address of your e-mail SMTP server. This must be changed to your own IP address or the code will not work.

```
byte server[] = { 10, 234, 219, 95 };
```

A client instance is created and given a name.

```
EthernetClient client
```

Next comes the first of your own functions. This one does the job of sending the e-mail to the server. The function requires four parameters: the e-mail subject, the first line of the message, the second line of the message, and finally, the temperature.

```
void sendEmail(char subject[], char message1[], char message2[], float temp) {
```

The user is advised that you are attempting to connect:

```
Serial.println("connecting...");
```

Then you check if the client has connected. If so, the code within the `if`-block is executed. The connect() function also needs the server IP address and port. This is the address and port of your email SMTP server. Change the IP address and port to your own specifications.

```
if (client.connect(server, 25)) {
```

First, the user is informed that you have connected to the client. The client in this case is your e-mail SMTP server.

```
Serial.println("connected");
```

You now send commands to the server in pretty much the same way that you did in Project 46. First, you must introduce yourselves to the SMTP server. This is done with an EHLO command and the server details. After each command, you must wait a while to allow the command to be processed. I found 1,000 milliseconds was required for my server; you may need to increase or decrease this number.

```
client.println("EHLO MYSERVER");  delay(time); // log in
```

This is like a “shake-hands” procedure between the server and the client in which they introduce themselves to each other. Next, you need to authorize the connection. If your SMTP server does not require authorization, you can comment out this line and the username and password lines.

```
client.println("AUTH LOGIN"); delay(time); // authorize
```

Sometimes the server requires an unencrypted login, in which case you would send AUTH PLAIN and the username and password in plain text.

Next, the Base-64 encrypted username and password must be sent to the server:

```
client.println("lcbWNybbWl2JttnRzLmNvrZSbQ==");  delay(time);
client.println("GVOyVGbjLlnZ2VEw");  delay(time);
```

Then you need to tell the server who the mail is coming from and who the mail is going to:

```
client.println("MAIL FROM:<sales@earthshineelectronics.com>");      delay(time);
client.println("RCPT TO:<fred@crunchytoad.com>");      delay(time);
```

These must be changed to your own e-mail address and the address of the recipient. Most SMTP servers will only allow you to send e-mail using an e-mail address from its own domain (e.g. you cannot send an e-mail from a Hotmail account using a Yahoo server.)

Next is the DATA command to tell the server that what comes next is the e-mail data, i.e. the stuff that will be visible to the recipient.

```
client.println("DATA");       delay(time);
```

You want the recipient to see whom the e-mail is to and from, so these are included again for the recipient's benefit.

```
client.println("From: < sales@earthshineelectronics.com >");       delay(time);
client.println("To: < fred@crunchytoad.com >");       delay(time);
```

Next, you send the e-mail subject. This is the word "SUBJECT:" followed by the subject passed to the function:

```
client.print("SUBJECT: ");
client.println(subject);       delay(time);
```

Before the body of the e-mail, you must send a blank line to separate it from the headers.

```
client.println();      delay(time);
```

This is followed by the two lines of the message passed to the function.

```
client.println(message1);      delay(time);
client.println(message2);      delay(time);
```

Then you include the temperature:

```
client.print("Temperature: ");
client.println(temp);   delay(time);
```

All e-mails must end with a . on a line of its own to tell the server you have finished:

```
client.println(".");       delay(time);
```

Then you send a QUIT command to disconnect from the server:

```
client.println("QUIT");      delay(time);
```

Finally, the user is informed that the e-mail has been sent:

```
Serial.println("Email sent.");
```

Next, you store the current value of millis() in lastMessage,as you will use that later to see if the specified interval has passed or not in-between message sends.

```
lastMessage=millis();
```

If the connection to the client was not successful, the e-mail is not sent and the user informed:

```
 } else {
   Serial.println("connection failed");
 }
```

Next comes the function to read the response back from the client:

```
void checkEmail() { // see if any data is available from client
```

While data is available to be read back from the client

```
while (client.available()) {
```

The code stores that byte in the c variable

```
char c = client.read();
```

then prints the content of the c variable to the serial monitor window.

```
Serial.print(c);
```

If the client is NOT connected:

```
if (!client.connected()) {
```

then the user is informed, the system disconnects, and the client connected is stopped.

```
Serial.println();
Serial.println("disconnecting.");
client.stop();
```

Next is the function you have used before to obtain the temperature from the one-wire sensor

```
void getTemperature(DeviceAddress deviceAddress)
{
  tempC = sensors.getTempC(deviceAddress);
  tempF = DallasTemperature::toFahrenheit(tempC);
}
```

followed by the setup routine that simply sets up the Ethernet and sensors.

```
void setup()
{
Ethernet.begin(mac, ip);
 Serial.begin(9600);

     // Start up the sensors library
  sensors.begin();
    // set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);

 delay(1000);
}
```

Finally, there's the main program loop:

```
void loop()
```

You start off by requesting the temperatures from the DallasTemperature library

```
sensors.requestTemperatures();
```

then call your `getTemperature` function, passing it the address of the sensor

```
getTemperature(insideThermometer);
```

that is then displayed in the serial monitor window.

```
Serial.println(tempC);
```

Next you check that temperature to see if it has reached or exceeded your high threshold. If it has, then the appropriate e-mail is sent. However, you only want to send one e-mail every (`emailInterval*1000`) seconds so check also that `millis()` is greater than the last time the e-mail message was sent (`lastMessage`) plus the interval time. If true, the code is executed.

```
if (tempC >= HighThreshold && (millis()>(lastMessage+(emailInterval*1000)))) {
```

The user is informed and then the two lines that make up the e-mail message are sent:

```
Serial.println("High Threshhold Exceeded");
char message1[] = "Temperature Sensor\0";
char message2[] = "High Threshold Exceeded\0";
```

The `sendEmail` function is then called, passing it the parameters that make up the subject, message line one and two, and the current temperature:

```
sendEmail(subject, message1, message2, tempC);
```

If the high temperature threshold has not been reached, you check if it has dropped below the low temperature threshold. If so, carry out the same procedure with the appropriate message.

```
else if (tempC<= LowThreshold && (millis()>(lastMessage+(emailInterval*1000))))
    Serial.println("Low Threshhold Exceeded");
    char message1[] = "Temperature Sensor\0";
    char message2[] = "Low Threshold Exceeded\0";
    sendEmail(subject, message1, message2, tempC);
  }
```

Finally, you check if there is any data ready to be received back from the client (after an e-mail has been sent) and display the results:

```
if (client.available()) {checkEmail();}
```

This data is useful for debugging purposes.

This project has given you the basic knowledge for sending an e-mail from an Arduino with Ethernet Shield. You can use this to send alerts or report whenever an action has occurred, such as a person has been detected entering a room or a box has been opened. The system can also take other actions, i.e. to open a window if the temperature in a room gets too hot or to top up a fish tank if the water level drops too low.

Next, you will learn how to send data from an Arduino to Twitter.

Project 49 — Twitterbot

Again you will use the circuit with the two temperature sensors. This time you will send regular updates about the status of the two sensors to Twitter. This will give you a simple system for checking on the status of any sensors you have connected to the Arduino.

Twitter is a micro blogging service that allows you to send miniature blog posts or "tweets" of up to 140 characters in length. The tweets are publically accessible to anyone who does a search or to those persons who have chosen to subscribe to (or *follow*) your blog. Twitter is incredibly popular and can be accessed from any web browser or from one of the many Twitter clients that are available, including mobile phone apps. This makes it ideal for sending simple short pieces of information that you can check while on the move.

You will need to go to `Twitter.com` and create a new account. I recommend creating an account just for tweeting from your Arduino.

As of August 31,2010, Twitter changed its policy regarding third party apps accessing the website. An authentication method known as OAuth is now used that makes it very difficult to tweet directly from an Arduino; prior to this change it was an easy process. Tweeting, at the moment, can only be done via a third party. In other words, you sending the tweet to a website, or proxy, that will tweet on your behalf using the OAuth token (authorization code). The current Twitter library uses this method.

Once you have your Twitter account set up (or use an existing one), enter the code below.

Enter the Code

Before you upload the code, you will need a token for the Twitter account. The library you are using has been created by NeoCat and uses his website as a proxy for sending the tweet. This means you must first obtain a token, which is an encrypted version of your username and password, to access the Twitter website. To do this visit NeoCat's website at `http://arduino-tweet.appspot.com` and click on the "Step 1" link to obtain the token. Copy and paste this into the token section of the code.

Note that because you are using a proxy and have to give your Twitter username and password over to obtain the token, it is advisable to create a new twitter account and keep it anonymous (i.e. don't add any names or e-mail addresses into the Twitter profile of that account). I believe it is safe to use the library with your own account if you wish, but it is better to be very safe than sorry.

Next, click the "Step 2" link and obtain the two sets of libraries that the code relies on <SPI.h> and <Ethernet.h>. Install these in the libraries folder of the Arduino IDE. You will need to restart the IDE before you can use these. The Twitter library also comes with a few examples you can try out. If you wish to read up about the Twitter library you can find it on the Arduino playground at `www.arduino.cc/playground/Code/TwitterLibrary`.

Once you have your token and libraries installed, enter and upload the code in Listing 17-4.

Listing 17-4. Code for Project 49

```
// Project 49 - Twitterbot

#include <Ethernet.h>
#include <Twitter.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <SPI.h>

// Data wire is plugged into pin 3 on the Arduino
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12

float itempC, itempF, etempC, etempF;
boolean firstTweet = true;

// Setup a oneWire instance to communicate with any OneWire devices (not just Maxim/Dallas
temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

// arrays to hold device addresses

DeviceAddress insideThermometer = { 0x28, 0x44, 0x12, 0xC2, 0x03, 0x00, 0x00, 0x92 };
DeviceAddress outsideThermometer = { 0x28, 0xA5, 0x02, 0xC2, 0x03, 0x00, 0x00, 0xF0 };

byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0xF7, 0x38 };

// Your Token to Tweet (get it from http://arduino-tweet.appspot.com/)
Twitter twitter("19735326-neITsUBnLTZHgN9UGaRkcGvAe9vYuaRP7E55K26J"); // DuinoBot

unsigned long interval = 600000; // 10 minutes
unsigned long lastTime; // time since last tweet

// Message to post
char message[140], serialString[60];
```

```
// function to get the temperature for a device
void getTemperatures()
{
  itempC = sensors.getTempC(insideThermometer);
  itempF = DallasTemperature::toFahrenheit(itempC);
  etempC = sensors.getTempC(outsideThermometer);
  etempF = DallasTemperature::toFahrenheit(etempC);
}

void tweet(char msg[]) {
   Serial.println("connecting ...");
  if (twitter.post(msg)) {
    int status = twitter.wait();
    if (status == 200) {
      Serial.println("OK. Tweet sent.");
      Serial.println();
      lastTime = millis();
      firstTweet = false;
    } else {
      Serial.print("failed : code ");
      Serial.println(status);
    }
  } else {
    Serial.println("connection failed.");
  }
}

void setup()
{
  Ethernet.begin(mac);
  Serial.begin(9600);
   sensors.begin();
    // set the resolution
  sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
  sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
  sensors.requestTemperatures();

  getTemperatures();

  while (firstTweet) {
    sprintf(message, "Int. Temp: %d C (%d F) Ext. Temp: %d C (%d F). Tweeted from Arduino. %ld",
int(itempC), int(itempF), int(etempC), int(etempF), millis());
    tweet(message);
  }
}

void loop()
{
  Ethernet.maintain();
  sensors.requestTemperatures();
  sprintf(serialString, "Internal Temp: %d C  %d F. External Temp: %d C %d F", int(itempC),
int(itempF), int(etempC), int(etempF));
  delay(500);
  Serial.println(serialString);
  Serial.println();

  if (millis() >= (lastTime + interval)) {
    sprintf(message, "Int. Temp: %d C (%d F) Ext. Temp: %d C (%d F). Tweeted from Arduino. %ld",
int(itempC), int(itempF), int(etempC), int(etempF), millis());
       delay(500);
      tweet(message);
    }

  delay(10000); // 10 seconds
}
```

After you have uploaded the code to your Arduino, open the serial monitor window. The Arduino will attempt to connect to Twitter (actually NeoCat's website) and send the tweet. If the first tweet is successful, the output in the serial monitor window will be a bit like this:

```
connecting ...
OK. Tweet sent.

Internal Temp: 26 C  79 F. External Temp: 26 C 79 F

Internal Temp: 26 C  79 F. External Temp: 26 C 79 F

Internal Temp: 26 C  79 F. External Temp: 26 C 79 F
```

When the program first runs, it will obtain the temperature and then keep attempting to connect to Twitter in the setup routine before it moves onto the main loop. It will not stop until it successfully connects. If the program fails to connect, you will get a "`failed : code 403`" or "`connection failed`" message. If the tweet is successful, it will not tweet again until the interval period has passed. By default, this is set to 10 minutes, though you can change it. Twitter limits you to a maximum of 350 requests per hour, so don't overdo it. You can now access the Twitter website and view the account from anywhere to check up in the temperature readings.

Let's see how this code works.

Project 49 – Twitterbot – Code Overview

The program starts off by including the relevant libraries:

```
#include <Ethernet.h>
#include <Twitter.h>
#include <OneWire.h>
#include <DallasTemperature.h>
#include <SPI.h>
```

Next, the `defines` for the sensors are set:

```
#define ONE_WIRE_BUS 3
#define TEMPERATURE_PRECISION 12
```

You create four floats for the temperatures, this time for internal and external temperatures in both C and F:

```
float itempC, itempF, etempC, etempF;
```

The first time the program attempts to make a tweet, you want it to keep on trying until it successfully connects and sends the message. Therefore, a boolean is created and set to true, so you know if you have yet to make your first tweet or not:

```
boolean firstTweet = true;
```

As before, you create instances for the one-wire and temperature sensors as well as the addresses for the two sensors:

```
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

DeviceAddress insideThermometer = { 0x10, 0x7A, 0x3B, 0xA9, 0x01, 0x08, 0x00, 0xBF };
DeviceAddress outsideThermometer = { 0x10, 0xCD, 0x39, 0xA9, 0x01, 0x08, 0x00, 0xBE};
```

You give the Ethernet shield a MAC address:

```
byte mac[] = { 0x64, 0xB9, 0xE8, 0xC3, 0xC7, 0xE2 };
```

Next, you create an instance of the Twitter library and pass it the token for your account:

```
Twitter twitter("19735326-neITsUBnLTZHgN9UGaRkcGvAe9vYuaRP7E55K26J");
```

The interval in-between tweets is set

```
unsigned long interval = 600000; // 10 minutes
```

as is a variable to store the time you last tweeted.

```
unsigned long lastTime; // time since last tweet
```

Two character arrays are created. These will store the message to be tweeted and the message you will output to the serial monitor window.

```
char message[140], serialString[60];
```

Now you create some functions. The first one is the function to obtain the temperatures from the two sensors and store them in your variables.

```
void getTemperatures()
{
  itempC = sensors.getTempC(insideThermometer);
  itempF = DallasTemperature::toFahrenheit(itempC);
  etempC = sensors.getTempC(outsideThermometer);
  etempF = DallasTemperature::toFahrenheit(etempC);
}
```

Next is the function that will do the tweeting for you. It requires one parameter, which is the character array that has your message in it.

```
void tweet(char msg[]) {
```

The user is informed that you are attempting to connect:

```
Serial.println("connecting ...");
```

Next, you use the `post()` method of the Twitter object to send the message. If the post is successful, the function returns true. If it fails to connect, it returns false.

```
  if (twitter.post(msg)) {
```

If you connect successfully, then you check the status of the post using the `wait()` method. This returns the HTTP status code in the response from Twitter.

```
int status = twitter.wait();
```

If the status code is 200, this is the HTTP code's way of saying everything is OK. In other words, if the tweet was successfully sent, then the code within the block will execute.

```
if (status == 200) {
```

If successful, you inform the user:

```
Serial.println("OK. Tweet sent.");
Serial.println();
```

Then set `lastTime` to the current value in millis(). This is so you can determine how long has passed since the last tweet.

```
lastTime = millis();
```

The first time you carry out a successful tweet, you want the program to jump out of the `while` loop in the setup routine and move onto the main loop, so you set the `firstTweet` flag to false.

```
firstTweet = false;
```

If the status is not 200, i.e. the post failed, then the user is informed and the code passed back for debugging purposes

```
} else {
      Serial.print("failed : code ");
      Serial.println(status);
}
```

and if you were not even able to connect in the first place, the user is informed of that instead.

```
} else {
    Serial.println("connection failed.");
}
```

The user functions out of the way, you now come to the setup routine:

```
void setup()
```

First, you begin the Ethernet library and pass it the MAC address:

```
Ethernet.begin(mac);
```

Next, you begin serial communications at 9600 baud and set up the sensors as before:

```
Serial.begin(9600);
sensors.begin();
sensors.setResolution(insideThermometer, TEMPERATURE_PRECISION);
sensors.setResolution(outsideThermometer, TEMPERATURE_PRECISION);
```

The temperatures are requested, as you are about to use them:

```
sensors.requestTemperatures()
getTemperatures();
```

You now attempt to send your first tweet. The `while` loop to do this will keep running as long as `firstTweet` is set to true:

```
while (firstTweet) {
```

Next, you use a `sprintf` command to compile the tweet into the message[] array. You pass it the four sets of temperatures as well as the value of millis(). As millis is an unsigned long number, you use the %ld specifier in `sprintf` to print a long integer. The sprintf (string print formatted) command is an excellent way of packing lots of different bits of information into one string

```
sprintf(message, "Int. Temp: %d C (%d F) Ext. Temp: %d C (%d F). Tweeted from Arduino. %ld",
int(itempC), int(itempF), int(etempC), int(etempF), millis());
```

The `sprintf()` function takes three parameters: a variable in which you will store the formatted data (in this case, the message to be tweeted), the contents of the string with specifiers, then the variables. What this does is insert the first variable into the string where the first %d appears, the second variable where the next %d appears, and so on. The four specifiers are separated by commas. Therefore, the numbers will be separated by commas in the final string. So, if the values of the variables were

```
itempC          25
itempF          77
etempC          14
tempF           52
millis()        4576
```

then, after running the sprintf command, the contents of `message` will be

```
"Int. Temp: 25 C (77 F) Ext. Temp: 14 C (52 F). Tweeted from Arduino. 4576"
```

As you can see, the sprintf command is a powerful tool for converting longs mixes of strings and numbers into one string.

The reason you add the value of millis() onto the end of the tweet is that Twitter will not post a message that is the same as the last one sent. If the temperatures have not changed since the last tweet, the message will be the same and Twitter will return an error code instead. As you want regular updates every interval period, by adding the value of millis() to the end you will ensure that the message differs from the last one sent. Make sure that your tweet length does not go over 140 characters in total; otherwise, you will end up with weird messages appearing in your Twitter timeline.

Now that you have compiled your message, you pass it to the `tweet()` function:

```
tweet(message);
```

Next comes the main loop, which you will only reach if the first tweet in the setup routine is successful:

```
void loop()
```

First, you run a maintain command on the Ethernet library. This keeps the auto-assigned IP address live and valid.

```
Ethernet.maintain();
```

The temperatures are updated.

```
sensors.requestTemperatures();
```

Then you use a `sprintf` command to compile the output for the serial monitor. It's more convenient than a whole list of Serial.print() commands so you may as well use it, though it does increase the size of your code.

```
sprintf(serialString, "Internal Temp: %d C  %d F. External Temp: %d C %d F", int(itempC),
int(itempF), int(etempC), int(etempF));
```

Then the string is output to the serial monitor after a short delay:

```
delay(500);
Serial.println(serialString);
Serial.println();
```

Next you ascertain if the interval time has passed since the last tweet, and if so, send another one. You calculate the value of `lastTime` + interval and see if the current value in millis() is greater than it (i.e. the interval period has passed since the last tweet). If so, you compile the new message and tweet again.

```
if (millis() >= (lastTime + interval)) {
          sprintf(message, "Int. Temp: %d C (%d F) Ext. Temp: %d C (%d F). Tweeted from
          Arduino. %ld", int(itempC), int(itempF), int(etempC), int(etempF), millis());
          tweet(message);
}
```

Finally, you have a 10-second delay between the updates to the serial monitor so that you don't bombard the user with information:

```
delay(10000); // 10 seconds
```

Now that you know how to send tweets from your Arduino, you can use it for all kinds of purposes. How about a potted plant that tweets to let you know it needs watering? Or sensors around a house to tweet whenever anyone enters a room, a doorbell that tweets when someone is at the door, or a cat flap that tells you when your cat has left or entered the house? The possibilities are endless.

Now you've reached the final project in your journey. In this last project, you will use the Ethernet Shield to read some data from the Internet instead of sending data out.

Project 50 – RSS Weather Reader

The final project in this book will use the Ethernet Shield again, but instead of transmitting data out to a web service, you will use the Arduino and Ethernet Shield to fetch data from the Internet and then display it in the serial monitor window. The data you are going to use is an RSS (Really Simple Syndication) feed from the `www.weather.gov` website to obtain weather data for an area of your choosing in the United States. This code will easily adapt to read an RSS weather feed from any other source if you are outside of the United States.

RSS is a web format for publishing frequently updated information, such as weather, news, etc. The data is in XML (Extensible Markup Language) format, which is a set of rules for encoding documents in a machine-readable form. XML is a simple format and it's not really necessary to understand how it works. The Arduino will simply look for tags within the XML code where the temperature, humidity, and pressure data is stored and strip out that information for displaying.

You'll be using the XML feed for Edwards Air Force Base in California. If you wish to use a different feed, go to `http://www.weather.gov/xml/current_obs/` and choose your area, then look for the full address of the XML data for that feed. Adjust the code accordingly to show the weather for that area.

As for hardware, this time you are using nothing more than an Ethernet Shield plugged into an Arduino.

Enter the Code

Plug the Ethernet shield into the Arduino (if it is not already there) and enter the code from Listing 17-5. Thanks to Bob S. (Xtalker) from the Arduino forums for the code.

Listing 17-5. Code for Project 50

```
// Project 50
// Thanks to  Bob S. for original code
// Get current weather observation for Edwards AFB from weather.gov in XML format

// Include description files for other libraries used (if any)
//#include <string.h>
#include <Ethernet.h>
#include <SPI.h>

// Max string length may have to be adjusted depending on data to be extracted
#define MAX_STRING_LEN  20

// Setup vars
char tagStr[MAX_STRING_LEN] = "";
char dataStr[MAX_STRING_LEN] = "";
char tmpStr[MAX_STRING_LEN] = "";
char endTag[3] = {'<', '/', '\0'};
int len;
```

```
// Flags to differentiate XML tags from document elements (ie. data)
boolean tagFlag = false;
boolean dataFlag = false;

// Ethernet vars
byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0xF7, 0x38 };
byte ip[] = {192, 168, 0, 35};
byte server[] = { 140, 90, 113, 200 }; // www.weather.gov

// Start ethernet client
EthernetClient client;

void setup()
{
  Serial.begin(9600);
  Serial.println("Starting Weather RSS Reader");
  Serial.println("connecting...");
  Ethernet.begin(mac, ip);
  delay(1000);

  if (client.connect(server, 80)) {
    Serial.println("connected");
    Serial.println("Current weather at Edwards AFB:");
    client.println("GET /xml/current_obs/KEDW.xml HTTP/1.0");
    client.println();
    delay(2000);
  } else {
    Serial.println("connection failed");
  }
}

void loop() {

  // Read serial data in from web:
  while (client.available()) {
    serialEvent();
  }

  if (!client.connected()) {

    client.stop();

      for (int t=0; t<15; t++) {
      delay(60000); // 1 minute
      }

    if (client.connect(server, 80)) {
      client.println("GET /xml/current_obs/KEDW.xml HTTP/1.0");
      client.println();
      delay(2000);
    } else {
      Serial.println("Reconnection failed");
    }
  }
}

// Process each char from web
void serialEvent() {
   // Read a char
        char inChar = client.read();

   if (inChar == '<') {
      addChar(inChar, tmpStr);
      tagFlag = true;
      dataFlag = false;

   } else if (inChar == '>') {
      addChar(inChar, tmpStr);

      if (tagFlag) {
         strncpy(tagStr, tmpStr, strlen(tmpStr)+1);
      }

      // Clear tmp
      clearStr(tmpStr);

      tagFlag = false;
      dataFlag = true;

   } else if (inChar != 10) {
      if (tagFlag) {
         // Add tag char to string
         addChar(inChar, tmpStr);

         // Check for </XML> end tag, ignore it
         if ( tagFlag && strcmp(tmpStr, endTag) == 0 ) {
            clearStr(tmpStr);
            tagFlag = false;
            dataFlag = false;
         }
      }

      if (dataFlag) {
         // Add data char to string
         addChar(inChar, dataStr);
      }
   }
```

```
   // If a LF, process the line
   if (inChar == 10 ) {

      // Find specific tags and print data
      if (matchTag("<temp_f>")) {
              Serial.print("TempF: ");
         Serial.print(dataStr);
      }
          if (matchTag("<temp_c>")) {
              Serial.print(", TempC: ");
         Serial.print(dataStr);
      }
      if (matchTag("<relative_humidity>")) {
              Serial.print(", Humidity: ");
         Serial.print(dataStr);
      }
      if (matchTag("<pressure_in>")) {
              Serial.print(", Pressure: ");
         Serial.print(dataStr);
         Serial.println("");
      }

      // Clear all strings
      clearStr(tmpStr);
      clearStr(tagStr);
      clearStr(dataStr);

      // Clear Flags
      tagFlag = false;
      dataFlag = false;
   }
}

// Function to clear a string
void clearStr (char* str) {
   int len = strlen(str);
   for (int c = 0; c < len; c++) {
      str[c] = 0;
   }
}

//Function to add a char to a string and check its length
void addChar (char ch, char* str) {
   char *tagMsg  = "<TRUNCATED_TAG>";
   char *dataMsg = "-TRUNCATED_DATA-";

   // Check the max size of the string to make sure it doesn't grow too
   // big.  If string is beyond MAX_STRING_LEN assume it is unimportant
   // and replace it with a warning message.
   if (strlen(str) > MAX_STRING_LEN - 2) {
      if (tagFlag) {
         clearStr(tagStr);
         strcpy(tagStr,tagMsg);
      }
      if (dataFlag) {
         clearStr(dataStr);
         strcpy(dataStr,dataMsg);
      }

      // Clear the temp buffer and flags to stop current processing
      clearStr(tmpStr);
      tagFlag = false;
      dataFlag = false;

   } else {
      // Add char to string
      str[strlen(str)] = ch;
   }
}

// Function to check the current tag for a specific string
boolean matchTag (char* searchTag) {
   if ( strcmp(tagStr, searchTag) == 0 ) {
      return true;
   } else {
      return false;
   }
}
```

Upload the code and open up the serial monitor. If everything is working correctly, you will have an output similar to this:

```
Starting Weather RSS Reader
connecting...
connected
Current weather from Edwards AFB:
TempF: 60.0, TempC: 15.4, Humidity: 100, Pressure: 29.96
```

Every sixty seconds the display will update again with the latest data. Let's see how this code works.

Project 50 – RSS Weather Reader – Code Overview

The program starts off by including the relevant Ethernet libraries you will need:

```
#include <Ethernet.h>
#include <SPI.h>
```

Then you define the maximum length of the data string:

```
#define MAX_STRING_LEN  20
```

You may need to increase this if you are requesting further information from the feed. Next, you create three arrays that will store the various strings you will be processing (these all have the just the length defined).

```
char tagStr[MAX_STRING_LEN] = "";
char dataStr[MAX_STRING_LEN] = "";
char tmpStr[MAX_STRING_LEN] = "";
```

Then you create another array to store the possible end tags you will encounter in the XML feed:

```
char endTag[3] = {'<', '/', '\0'};
```

Then you create a variable that will store the length of the string you will be processing at the relevant section of the code:

```
int len;
```

Then you create two flags. These will be used to differentiate between the XML tags and the information after the tags that you wish to strip out of the XML code.

```
boolean tagFlag = false;
boolean dataFlag = false;
```

Next, you set up the MAC and IP address of the Ethernet Shield:

```
byte mac[] = { 0x90, 0xA2, 0xDA, 0x00, 0xF7, 0x38 };
byte ip[] = {192, 168, 0, 35};
```

Then the IP address of the `www.weather.gov` website:

```
byte server[] = { 140, 90, 113, 200 }; // www.weather.gov
```

If you are using a different website for your weather feed, change this IP address to the URL you are using. Next, you create a client object and name it:

```
EthernetClient client();
```

Next comes the setup routine

```
void setup()
```

which starts off with beginning serial communications at 9600 baud so you can print data to the serial monitor.

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

You inform the use of the name of the program and that you are attempting to connect:

```
Serial.println("Starting Weather RSS Reader");
Serial.println("connecting...");
```

Ethernet communications are started, passing the MAC and IP address of your device, followed by a short delay to allow it to connect:

```
Ethernet.begin(mac, ip);
delay(1000);
```

Next, you check if you have connected to your client (the `www.weather.gov` website) successfully. We use the connect() function and pass it the server IP address and the port:

```
if (client.connect(server, 80)) {
```

If so, you inform the user

```
Serial.println("connected");
Serial.println("Current weather at Edwards AFB:");
```

Then carry out a HTML GET command to access the XML data from the sub-directory that stores the relevant feed, followed by a delay to allow successful communications.

```
client.println("GET /xml/current_obs/KEDW.xml HTTP/1.0");
client.println();
delay(2000);
```

If the connection was not made, you inform the user of a failed connection:

```
  } else {
    Serial.println("connection failed");
  }
}
```

Next comes the main loop:

```
void loop() {
```

As you performed a GET command in the setup loop, the serial buffer should contain the contents of the XML feed returned from the server. So, while you have data available

```
while (client.available()) {
```

the `serialEvent()` function is called.

```
serialEvent();
```

This function will be explained shortly. If a connection has not been made

```
if (!client.connected()) {
```

```
the connection to the client is stopped.
```

```
client.stop();
```

Then you wait 15 minutes before you attempt another connection. The data feed is updated once every 15 minutes at most, so it is pointless updating the information any earlier than this:

```
if (int t=0; t<15; t++) { // the feed is updated once every 15 mins
      delay(60000); // 1 minute
      }
```

If you have made a successful connection to the client

```
if (client.connect(server, 80)) {
```

then you perform another GET command to obtain the latest XML feed data

```
client.println("GET /xml/current_obs/KEDW.xml HTTP/1.0");
client.println();
delay(2000);
```

and if a connection fails, the user is informed.

```
} else {
      Serial.println("Reconnection failed");
    }
```

Next comes the `serialEvent()` function. The purpose of this function is to read the data from the XML feed and process it according to what it finds

```
void serialEvent() {
```

The function starts by reading in the first character and storing it in `inChar`:

```
char inChar = client.read();
```

Now you need to take a look at that character and decide if it is a tag or if it is data. If it is a tag, then we set the `tagFlag` to true. If it is data, we set the `dataFlag` to true. The other flag is set to false each time.

The raw data for the feed looks like:

```
<current_observation version="1.0"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="http://www.weather.gov/view/current_observation.xsd">
        <credit>NOAA's National Weather Service</credit>
        <credit_URL>http://weather.gov/</credit_URL>
        <image><url>http://weather.gov/images/xml_logo.gif</url><title>NOAA's National
Weather Service</title><link>http://weather.gov</link></image>
        <suggested_pickup>15 minutes after the hour</suggested_pickup>
        <suggested_pickup_period>60</suggested_pickup_period>
        <location>Edwards AFB, CA</location>
        <station_id>KEDW</station_id>
        <latitude>34.91</latitude>
        <longitude>-117.87</longitude>
        <observation_time>Last Updated on Apr 30 2013, 9:55 am PDT</observation_time>
        <observation_time_rfc822>Tue, 30 Apr 2013 09:55:00 -0700</observation_time_rfc822>
        <weather>Mostly Cloudy</weather>
        <temperature_string>62.0 F (16.4 C)</temperature_string>
        <temp_f>62.0</temp_f>
        <temp_c>16.4</temp_c>
        <relative_humidity>100</relative_humidity>
```

As you can see, each piece of information is embedded inside a tag. For example, the temperature in Fahrenheit has the `<temp_f>` tag to start it off and a `</temp_f>` to end it. Everything in-between the tags are data.

First, you check if the character is a < character. If so, this is the start of a tag.

```
if (inChar == '<') {
```

If so, you call the `addChar` function, which will check if the string length is within the limits of MAX_STRING_LEN and if so, add the character to your tmpStr string. You will examine this function later on.

```
addChar(inChar, tmpStr);
```

As you have found a tag, the `tagFlag` is set to true and the `dataFlag` set to false:

```
tagFlag = true;
dataFlag = false;
```

If you reach the end of the tag by finding the > character

```
} else if (inChar == '>') {
```

then the character is added to the tmpStr string.

```
addChar(inChar, tmpStr);
```

If you are currently processing a tag and have reached the end of the tag, you can copy the entire tag from the tmpStr (temporary string) in the tag string (tgrStr). You use the strncpy command to do this.

```
if (tagFlag) {
         strncpy(tagStr, tmpStr, strlen(tmpStr)+1);
}
```

The strncpy command copies part of one string into another string. It requires three parameters: the string you are copying the data into, the string you are copying the data from, and the amount of characters to copy. For example, if you had

```
strncpy(firstString, secondString, 10);
```

then the first 10 characters of secondString are copied into firstString. In this case, you copy the entire contents by finding the length of the temporary string (tmpStr)+1 and copying that number of characters into the tag string.

Once the temporary string has been copied, you need to clear it so it's ready for the next piece of data. To do this, you call the `clearStr` function and pass it the string you wish to clear.

```
clearStr(tmpStr);
```

The two flags are set to false, ready for the next piece of information:

```
tagFlag = false;
dataFlag = true;
```

If the character read is a linefeed (ASCII 10)

```
} else if (inChar != 10) {
```

then you add the character to the string if you are currently processing a tag.

```
if (tagFlag) {
       addChar(inChar, tmpStr);
```

You want to ignore the end tags, so you check if you are currently processing a tag and have reached the end of the tag (by comparing with the endTag characters).

```
if ( tagFlag && strcmp(tmpStr, endTag) == 0 ) {
```

then the tag is ignored, the string is cleared, and the tags set to their defaults.

```
clearStr(tmpStr);
tagFlag = false;
dataFlag = false;
```

The strcmp command compares two strings. In your case, it compares the temporary string (tmpStr) with the characters in the endTag array:

```
strcmp(tmpStr, endTag)
```

The result will be 0 if the strings match and another value if they don't. By comparing it with the endTag array, you are checking that any of the three end tag characters are present.

If the current string is data

```
if (dataFlag) {
```

then you add the current character to the data string (dataStr).

```
addChar(inChar, dataStr);
```

The above code has basically decided if you are processing a tag, and if so, stores the characters in the tag string (tagStr); if it is data, it stores it in the data string (dataStr). You will end up with the tag and the data stored separately.

If you have reached a linefeed character, you are clearly at the end of the current string. So you now need to check the tags to see if they are the temperature, humidity, or pressure data that you want.

```
if (inChar == 10 ) {
```

To do this, you use the `matchTag` function (that you will come to shortly) which checks if the specified tag is within the tag string, and if, so returns a true value. You start by looking for the temperature in the Fahrenheit tag

```
<temp_f>
```

```
if (matchTag("<temp_f>")) {
```

and if it is found, prints out the data string, which if the tag is <temp_f> will contain the temperature in Fahrenheit.

```
Serial.print("Temp: ");
Serial.print(dataStr);
```

Next you check for the temperature in Celsius,

```
if (matchTag("<temp_c>")) {
        Serial.print(", TempC: ");
        Serial.print(dataStr);
}
```

the humidity,

```
if (matchTag("<relative_humidity>")) {
        Serial.print(", Humidity: ");
        Serial.print(dataStr);
      }
```

and the pressure.

```
if (matchTag("<pressure_in>")) {
         Serial.print(", Pressure: ");
         Serial.print(dataStr);
         Serial.println("");
}
```

Then all of the strings are cleared ready for the next line

```
clearStr(tmpStr);
clearStr(tagStr);
clearStr(dataStr);
```

and the tags are cleared, too.

```
tagFlag = false;
dataFlag = false;
```

Next, you have your user functions, starting with the clear string (clearStr) function

```
void clearStr (char* str) {
```

that simply finds the length of the string passed to the function using the strLen() command

```
int len = strlen(str);
```

then uses a for loop to fill each element of the array with an ASCII 0 (null) character.

```
for (int c = 0; c < len; c++) {
      str[c] = 0;
}
```

The next function is the `addChar` function. You pass the character currently read and the current string to it as parameters.

```
void addChar (char ch, char* str) {
```

You define two new character arrays and store error messages in them:

```
char *tagMsg  = "<TRUNCATED_TAG>";
char *dataMsg = "-TRUNCATED_DATA-";
```

If you find that the strings are over the length of MAX_STRING_LEN then you will replace them with these error messages.

You now check the length of the string to see if it has reached the maximum length:

```
if (strlen(str) > MAX_STRING_LEN - 2) {
```

If it has and you are currently processing a tag

```
if (tagFlag) {
```

then the tag string is cleared and you copy the error message into the tag string.

```
clearStr(tagStr);
strcpy(tagStr,tagMsg);
```

If you are processing data, then the data string is cleared and you copy the data error message into the data string.

```
if (dataFlag) {
         clearStr(dataStr);
         strcpy(dataStr,dataMsg);
}
```

The temporary string and tags are cleared

```
clearStr(tmpStr);
tagFlag = false;
dataFlag = false;
```

and if the length of the string has not exceeded the maximum length, you add the current character that has been read into the string. You use the length of the string to find out the last character, i.e., the next place you can add a character to.

```
} else {
      // Add char to string
      str[strlen(str)] = ch;
}
```

Finally, you come to the matchTag function that is used to check that the search tag passed to it as a parameter has been found or not, and if so, returns a true or false accordingly:

```
boolean matchTag (char* searchTag) {
```

The function is of type boolean as it returns a boolean value and requires a character array as a parameter:

```
if ( strcmp(tagStr, searchTag) == 0 ) {
      return true;
   } else {
      return false;
   }
}
```

By changing the XML feed URL and the tags found within that feed, you can use this code to look for pieces of data in any RSS feed you wish. For example, you could use the Yahoo weather feeds at http://weather.yahoo.com and navigate to the region you wish to view and click the RSS button. The URL of that feed can then be entered into the code. You can view the raw source of the feed by right clicking and choosing the right click menu option to view the source. You can then view the tags and modify the code to find the relevant piece of information.

This last project has shown you how to use your Ethernet Shield to obtain information from the Internet. Previously, you sent data out from the Shield to external sources. In this project, you read data back from the Internet instead. Rather than displaying the weather data in the serial monitor window, you can use the skills you have learned in the previous projects to display it on an LCD screen or on an LED dot matrix display.

Summary

This final chapter has shown you how to connect your Arduino to the Internet, either for the purpose of sending data out in the form of a served web page, a tweet to Twitter, an e-mail, sensor data sent to Xively, or for requesting a web page and stripping data from that web page for your own use. Having the ability to connect your Arduino to a LAN or the Internet opens up a whole new list of potential projects. Data can be sent anywhere around your house or office where an Ethernet port is available, or data can be read from the Internet for the Arduino to process, display, or act upon.

For example, you could use a current weather feed to determine if it is about to rain and warn you to bring your washing in from the clothesline or to close a skylight window. What you do with your Arduino once it's connected is only limited by your imagination.

I hope you enjoyed the 50 projects presented to you in *Beginning Arduino* and that you have great fun in using the knowledge gained to create your own fantastic Arduino creations. Please get in touch on Twitter or G+ if you need any further help. All the best, Mike.

Subjects and Concepts covered in Chapter 17:

- How to manually assign a MAC and IP address to your device
- The concept of client and server
- How to listen for client connections with the client.connected() command
- Sending HTML code by printing to a client
- Using your Arduino as a web server
- Connecting to the Arduino from a web browser
- Checking data is available with the client.available() command.
- Reading data with the client.read() command

- Sending data to Xively, viewing the data as graphs, etc.
- Sending tweets to `Twitter.com` from the Arduino via a proxy
- Sending e-mails from the Arduino
- Fetching data from an RSS feed and parsing it for display
- Copying strings with the strcpy and strncpy commands
- Comparing strings with the strcmp command
- Filling memory locations with the memset command
- Finding the size of arrays with the sizeof command
- Processing strings with the sprintf command
- Finding lengths of strings with the strlen command
- Searching for sub-strings with the strstr command
- Finding IP addresses with the ping command
- Checking if a client has connected with the client.connect() command
- Creating an Ethernet connection with the Ethernet.begin(mac, ip) command
- Encrypting usernames and passwords to Base-64 with website utilities
- Using the Ethernet.h library to automatically assign an IP address
- Using the Twitter library post and wait commands
- Looking for tags in an XML RSS feed

Index

Beginning Arduino

Michael McRoberts

Apress®

Beginning Arduino

ISBN-13 (pbk): 978-1-4302-5016-6

ISBN-13 (electronic): 978-1-4302-5017-3

President and Publisher: Paul Manning
Lead Editor: Michelle Lowman
Development Editor: Douglas Pundick
Technical Reviewers: Brad Levy (Chs. 1–14) and Cliff Wootton
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Kevin Shea
Copy Editor: Larissa Shmailo
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary material referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code`.

I would like to dedicate this book to my mother for her encouragement throughout the writing process and for being the best Mum anyone could ask for, and to my grandfather, Reginald Godfrey, for igniting the spark of the love of science and electronics in me at a young age. Without all those kits from Radio Shack at Christmas I may never have reached the point where I ended up writing a book about microcontrollers and electronics. Finally, to my love Petra, for always being there for me. Thank you all.

—Mike McRoberts

Contents

About the Author

Mike McRoberts discovered the Arduino in 2008 while looking for ways to connect a temperature sensor to a PC to make a cloud detector for his other hobby of astrophotography. After a bit of research, the Arduino seemed like the obvious choice, and the cloud detector was successfully made, quickly and cheaply. Mike's fascination with the Arduino had begun. Since then, he has gone on to make countless projects using the Arduino. He regularly runs Arduino workshops across the United Kingdom for hackspaces, businesses, and other organisations. He has also founded an Arduino starter kit and component online business called Earthshine Electronics. His next project is to use an Arduino-based circuit to send a high altitude balloon up to the edge of space to take stills and video for the heck of it, with the help of the guys from UKHAS and CUSF.

Mike's hobby of electronics began when he was a child and the 100-in-1 electronics kits from Radio Shack comprised his Christmas present list. He started programming as a hobby when he obtained a Sinclair ZX81 computer as a teenager. Since then, he's never been without a computer.

He is a member of London Hackspace and the Observing Officer for the Orpington Astronomical Society and can regularly be found contributing to the Arduino Forum. He also likes to lurk on IRC in the Arduino, high altitude, and london-hack-space channels (as "earthshine"), on Twitter as @TheArduinoGuy, and also on Google+ under "Mike McRoberts"(look for the avatar with the "Do you Arduino?" T-shirt). When he is not messing around with Arduinos, he likes to indulge in astronomy, astrophotography, motorcycling, and sailing.

About the Technical Reviewers

Brad Levy is a practitioner with over thirty years of experience in software and hardware design. He has developed embedded systems for energy management, solar, and avionics test equipment. He developed weather stations and live weather presentation software for Weather Metrics, a company he co-founded. He has worked on office automation, graphics software, system libraries and device drivers for multiple platforms, as well as large scientific simulations. Brad has experience with many computer languages old and new, including C, C++, APL, Fortran, Pascal, JavaScript, PHP, Python, and assembler language for many different processors, large and small. He has done compiler, linker, and run-time library development, and written a multitasking operating system for an embedded system. He has developed systems communicating over direct links, landline, and cellular phone networks, satellite links, and the Internet.

Brad is currently working on Arduino- and PIC-based projects for energy management and model railroad control, and on an interactive art piece utilizing the Raspberry Pi platform. Brad's technical interests also include user-interface design, library sciences, the future of books and information, web design, photography, graphics design, font design, and audio equipment design. He is a long-time member of the Association for Computing Machinery.

Brad is also an active artist, exhibiting photography on a regular basis.

You can find more about Brad at `www.bradlevy.com`.

Cliff Wootton is a former interactive TV systems architect at the BBC. The "Interactive News Loops" service developed there was nominated for a BAFTA and won a Royal Television Society Award for Technical Innovation. He was invited as a guest speaker on pre-processing for video compression at the Apple WWDC developer conference in San Francisco and was a speaker on interactive TV systems at the National Association of Broadcasters conference.

Cliff taught postgraduate MA students about real-world computing, multimedia, video compression, metadata and how to leverage artistic creativity through the use of engineering and developer tools.

He is currently working as a consultant engineer on R&D projects studying the deployment of next generation interactive TV systems based on open standards. His pro-bono projects are in MPEG, epublishing metadata, and Cloud storage standards working groups. This work generates useful knowledge input for consulting, writing, teaching, mentoring, and speaking opportunities. The learning outcomes from this research are occasionally published in the Interactive TV blog (`http://blog.newsinteractive.tv/`).

Acknowledgments

First of all, I'd like to thank my editors, Michelle Lowman, Douglas Pundick, and Kevin Shea from Apress, as without them this book would never have gotten off the ground; my technical reviewers, Josh Adams (first edition), Brad Levy, and Cliff Wotton (second edition)for patiently checking my code and circuit diagrams to get them right and suggesting numerous technical improvements.

A huge thank you to all those people from Flickr and Wikimedia Commons who chose to put their images under a Creative Commons license and who gave me permission to use those images: Bruno Soares, Richard V. Gilbank, Inductiveload, Snorpey, Iain Fergusson, Patrick H. Lauke, cultured_society2nd, Cyril Buttay, Tony Jewell, Tod E. Kurt, Adam Grieg, David Stokes, Mike Prevette, David Mitchell, Aki Korhonen, Alex43223, Sparkfun, DFRobot, Adafruit Industries, Colin M.L. Burnett, David Batley, Jan-Piet Mens, Mercury13, Georg Wiora, and Timo Arnall.

Thanks to everyone who let me use or modify their code or Arduino libraries to create the projects and who gave technical assistance or advice: Michael Margolis, Usman Haque, and the guys from Xively, Georg Kaindl, Tom Pollard, Jim Studt, Miles Burton, Robin James, Paul Stoffregen, Conor, Tom Igoe, Tim Newsome, James Whiddon, Bill Greiman, Matt Joyce, D. Sjunnesson, David A. Mellis, Bob S. (Xtalker), Ian Baker, Henry Lahr, Henning Karlsen, and NeoCat.

Thanks to Sparkfun and Adafruit Industries for providing me with parts and for letting me use their images. Thanks also to the Arduino core team, without whom the fantastic Arduino and its community would not exist: Massimo Banzi, Tom Igoe, David Cuartielles, Gianluca Martino, David Mellis, and Nicholas Zambetti.

Finally, thanks to all those people on the Arduino Forum, Arduino IRC channel, Google+, and Twitter for your help, advice, and encouragement throughout the book process and to London Hackspace for giving me a place to try out some experiments.

If I have missed anyone, my apologies, and thanks to you, too.

—Mike McRoberts