Chapter 3

Downloading and Installing Arduino

In This Chapter

- Obtaining and installing the Arduino software
- Getting a feel for the Arduino environment

Before you can start working with the Arduino board, you need to install the software for it. This is similar to the software on your home computer, laptop, tablet, or phone: It's required to make use of your hardware.

The Arduino software is a type of an Integrated Development Environment (IDE). This is a tool that is common in software development and allows you to write, test, and upload programs. Versions of Arduino software are available for Windows, Macintosh OS X, and Linux.

In this chapter, you find out where to obtain the software for the platform you're using, and I walk you through the steps for downloading and installing it. Also in this chapter is a brief tour of the environment in which you develop your Arduino programs.

Installing Arduino

This section talks you through installing the Arduino environment on your platform of choice. These instructions are specifically for installation using an Arduino Uno R3, but work just as well for previous boards, such as the Mega2560, Duemilanove, Mega, or Diecimila. The only difference may be the drivers needed for the Windows installations.

Installing Arduino for Windows

The instructions and screenshots in this section describe the installation of the Arduino software and Arduino Uno drivers on Windows 7, but the same instructions work just as well for Windows Vista and Windows XP.

The only hurdle to jump is in Windows 8, which for the time being, at least, requires a few tricks to install the drivers. You can find a discussion on the Arduino forum titled "Missing digital signature for driver on Windows 8" that details a workaround (go to `http://arduino.cc/forum/index.php?topic=94651.15`).

With your Arduino Uno and a USB A-B cable (shown in Figure 3-1) at hand, follow these steps to obtain and install the latest version of Arduino on your version of Windows:

1. **Open the Arduino downloads page at** `http://arduino.cc/en/Main/Software`, **and click the Windows link to download the .zip file containing a copy of the Arduino application for Windows.**

   At the time of writing, the zipped file was 90.7MB. That's quite a large file, so it may take a while to download. When downloading is complete, unzip the file and place the Arduino folder in an appropriate location, such as

   ```
   C:/Program Files/Arduino/
   ```

2. **Plug the square end of the USB cable into the Arduino and the flat end into an available port on your PC to connect the Arduino to your computer.**

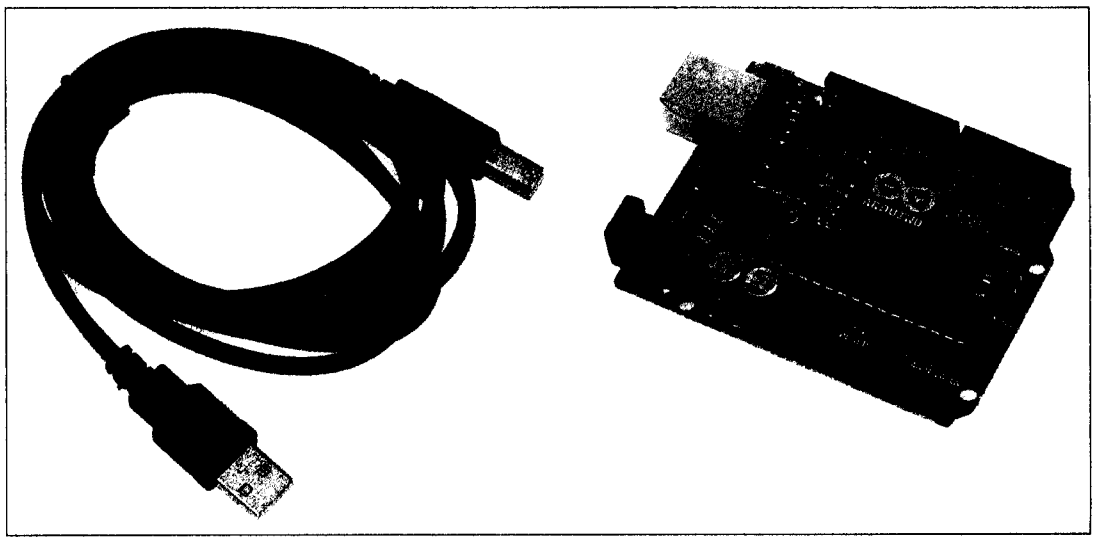

Figure 3-1: An A-B USB cable and Arduino Uno.

As soon as the board is connected, the green LED labeled ON indicates that your Arduino is receiving power. Windows then makes a valiant effort to find drivers, but it will likely fail, as indicated in Figure 3-2. It's best to close the wizard and install the driver yourself, as described in the following steps.

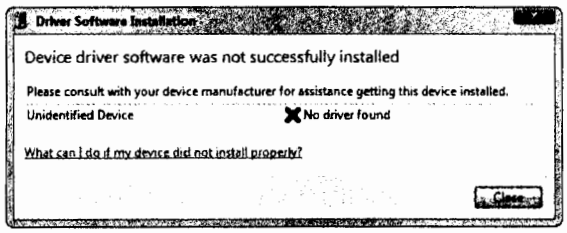

Figure 3-2: New hardware found — or not, as the case may be.

3. **Open the Start Menu and type** devmgmt.msc **in the Search Programs and Files box; then press Enter.**

   The Device Manager window opens. Device Manager shows you all the different hardware and connected peripherals in your computer, such as your Arduino board.

   If you look down the list, you should see Arduino Uno with an exclamation mark next to it. The exclamation mark indicates that it is not yet recognized.

4. **Right-click Arduino Uno and select Update Driver Software in the list that appears; then click the Browse My Computer for Driver Software option (see Figure 3-3).**

   The window advances to the next page.

5. **Click Browse to find your Arduino folder.**

   You should find this folder in the location you saved it to in Step 1 of these steps.

6. **Within your Arduino folder, click the Drivers folder and then click the Arduino UNO file.**

   Note that if you're in the FTDI USB Drivers subfolder, you have gone too far.

7. **Click Next, and Windows completes the installation.**

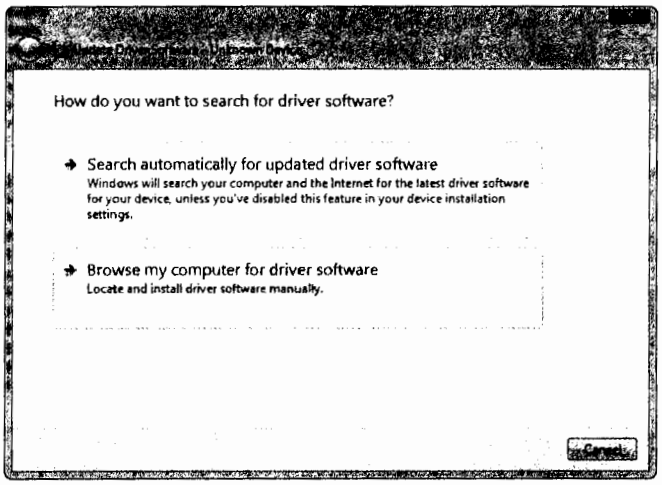

Figure 3-3: Installing drivers in Device Manager.

After you've taken care of the software installation, an easy way to launch the program is to place a shortcut on your desktop or your computer's Start menu, whichever you prefer. Just go to your main Arduino folder, find the Arduino.exe file, right-click and click Create Shortcut to make a shortcut. Double-click the shortcut icon whenever you want to launch the Arduino application. This opens a new sketch window, as shown in Figure 3-4.

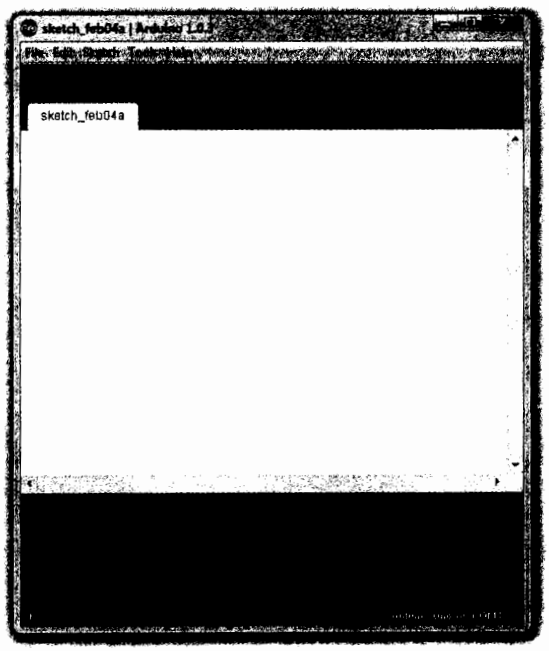

Figure 3-4: A beautiful turquoise Arduino window in Windows 7.

Installing Arduino for Linux

Installation on Linux is more involved and varies depending on the distribution you use, so it is not covered in this book. If you use Linux, you are probably already more than competent at installing the software and will relish the challenge. All the details on installing Arduino for Linux can be found in the Arduino Playground:

```
http://arduino.cc/playground/Learning/Linux
```

Surveying the Arduino Environment

Programs written for Arduino are known as *sketches*. This is a naming convention that was passed down from Processing, which allowed users to create programs quickly, in the same way that you would scribble out an idea in your sketchbook.

Before you look at your first sketch, I encourage you to stop and take a look around the Arduino software, and to that end, this section offers a brief tour. The Arduino software is an integrated development environment, or IDE, and this environment is presented to you as a graphical user interface, or GUI (pronounced *goo-ey*).

A GUI provides a visual way of interacting with a computer. Without it, you would need to read and write lines of text, similar to what you may have seen in the DOS prompt in Windows, Terminal in Mac OS X, or that bit about the white rabbit at the start of the Matrix.

The turquoise window is Arduino's GUI. It's divided into the following four main areas (labeled in Figure 3-8):

- **Menu bar:** Similar to the menu bar in other programs you're familiar with, the Arduino menu bar contains drop-down menus to all the tools, settings, and information that are relevant to the program. In Mac OS, the menu bar is at the top of your screen; in Windows and Linux, the menu bar is at the top of the active Arduino window.
- **Toolbar:** The toolbar contains several buttons that are commonly needed when writing sketches for Arduino. These buttons, which are also available on the menu bar, perform the following functions:
  - *Verify:* Checks that your code makes sense to the Arduino software. Known as *compiling*, this process is a bit like a spelling and grammar checker. Be aware, however, that although the compiler checks that your code has no obvious mistakes, it does not guarantee that your sketch works correctly.

- *Upload:* Sends your sketch to a connected Arduino board. It automatically compiles your sketch before uploading it.
- *New:* Creates a new sketch.
- *Open:* Opens an existing sketch.
- *Save:* Saves the current sketch.
- *Serial monitor:* Allows you to view data that is being sent to or received by your Arduino board.

- **Text editor:** This area displays your sketch is displayed as text. It is almost identical to a regular text editor but has a couple of added features. Some text is color coded if it is recognized by the Arduino software. You also have the option to auto format the text so that it is easier to read.
- **Message area:** Even after years of using Arduino, you'll still make mistakes (everybody does), and this message area is one of the first ways for you to find out that something is wrong. (***Note:*** The second way is the smell of burning plastic.)

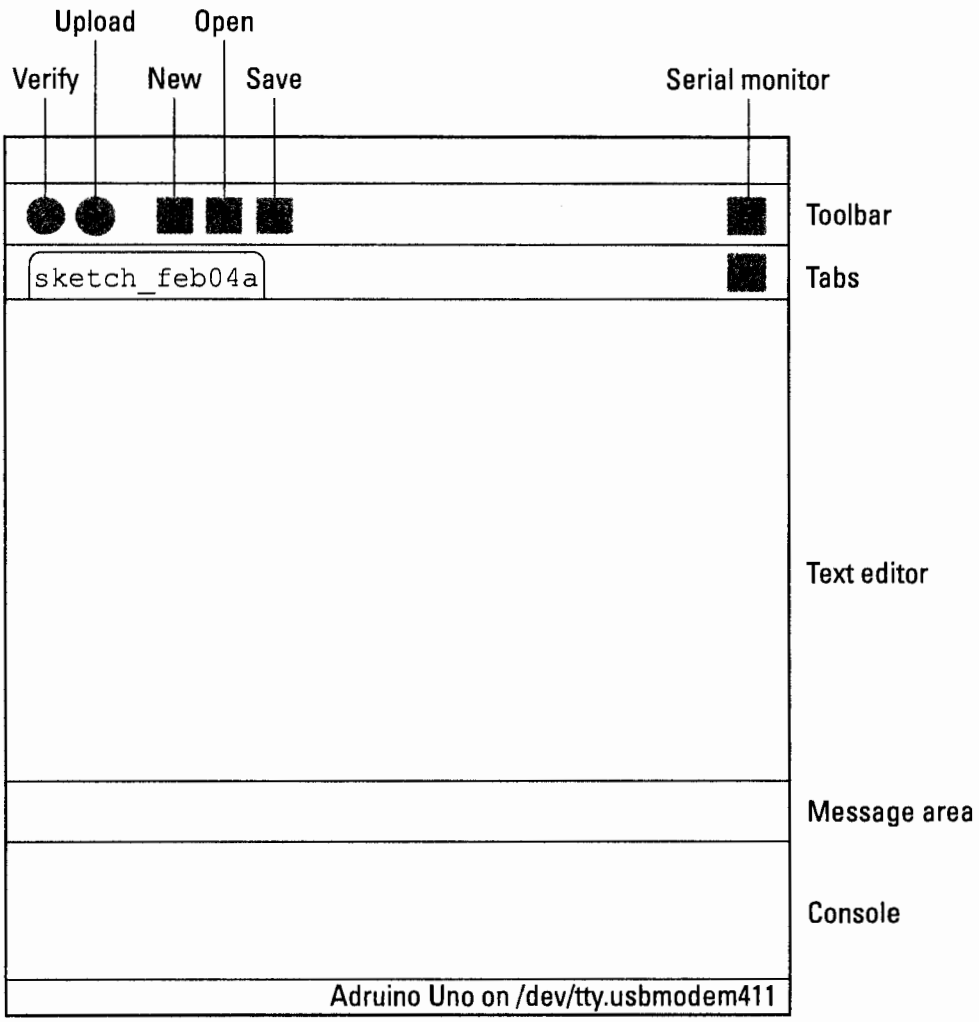

Figure 3-8: The areas of the GUI.

Chapter 4

Blinking an LED

In This Chapter

- ▶ Finding the Blink sketch
- ▶ Identifying your board
- ▶ Setting the software
- ▶ Uploading Blink
- ▶ Completing your first Arduino sketch
- ▶ Explaining the Sketch
- ▶ More Blink

Brace yourself. You are about to take your first real step into the world of Arduino! You've bought a board, maybe an Arduino starter kit (possibly from one of the suppliers I recommended), and you're ready to go.

It's always a good idea to have a clear work surface or desk to use when you're tinkering. It's not uncommon to drop or misplace some of the many tiny components you work with, so make sure your workspace is clear, well lit, and accompanied by a comfortable chair.

By its nature, Arduino is a device intended for performing practical tasks. The best way to learn about Arduino, then, is in practice — by working with the device and *doing* something. That is exactly the way I write about it throughout this book. In this chapter, I take you through some simple steps to get you on your way to making something.

I also walk you through uploading your first Arduino sketch. After that, you examine how it works and see how to change it to do your bidding.

Working with Your First Arduino Sketch

In front of you now should be an Arduino Uno R3, a USB cable, and a computer running your choice of operating system (Windows, Mac OS, or Linux). The next section shows what you can do with this little device.

Finding the Blink Sketch

To make sure that the Arduino software is talking to the hardware, you upload a *sketch*. What is a sketch, you ask? Arduino was created as a device that allows people to quickly prototype and test ideas using little bits of code that demonstrate the idea — kind of like how you might sketch out an idea on paper. For this reason, programs written for Arduino are referred to as sketches. Although a device for quick prototyping was its starting point, Arduino devices are being used for increasingly complex operations. So don't infer from the name *sketch* that an Arduino program is trivial in any way.

The specific sketch you want to use here is called Blink. It's about the most basic sketch you can write, a sort of "Hello, world!" for Arduino. Click in the Arduino window. From the menu bar, choose File➪Examples➪ 01.Basics➪Blink (see Figure 4-1).

A new window opens in front of your blank sketch and looks similar to Figure 4-2.

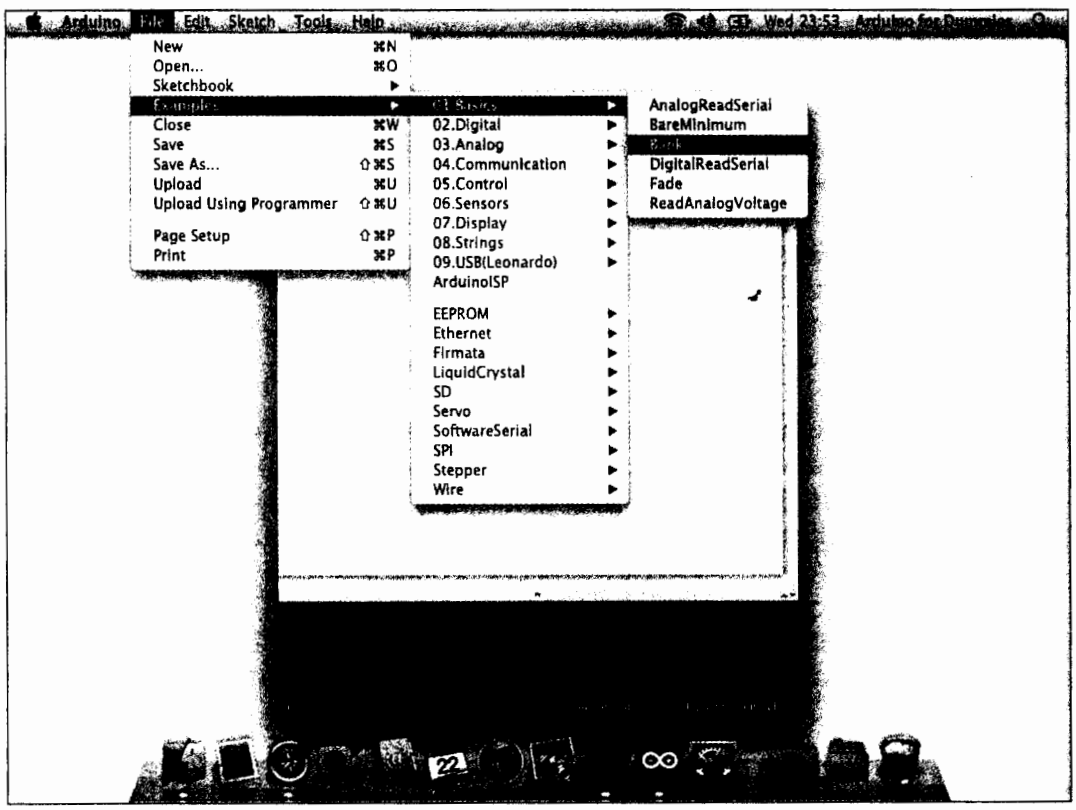

Figure 4-1: Find your way to the Blink sketch.

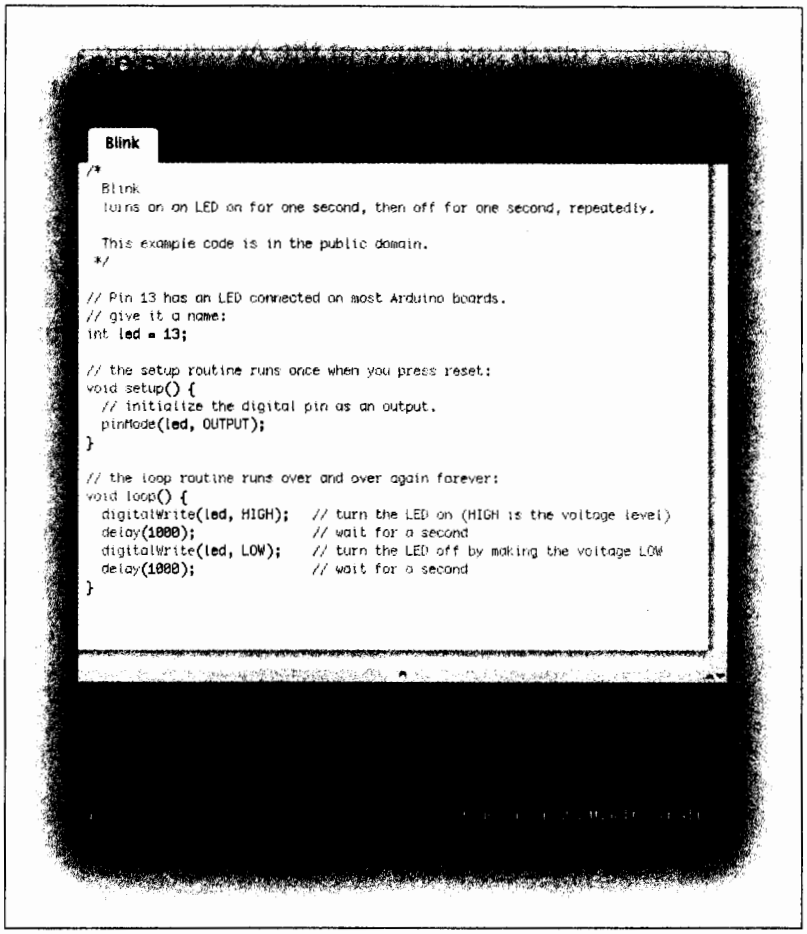

Figure 4-2: The Arduino Blink sketch.

Identifying your board

Before you can upload the sketch, you need to check a few things. First you should confirm which board you have. As I mention in Chapter 2, you can choose from a variety of Arduino devices and several variations on the USB board. The latest generation of USB boards is the Uno R3. If you bought your device new, you can be fairly certain that this is the type of board you have. To make doubly sure, check the back of the board. You should see details about the board's model, and it should look something like Figure 4-3.

Also worth checking is the ATMEL chip on the Arduino. As I mention in Chapter 2, the ATMEL chip is the brains of the Arduino and is similar to the processor in your computer. Because the Uno and earlier boards allow you to replace the chip, there is always a chance, especially with a used board, that the chip has been replaced with a different one.

Although the ATMEL chip looks quite distinctive on an individual board, if you compare it to an older Arduino, telling them apart at first glance would be difficult. The important distinguishing feature is written on the surface of the chip. In this case, you are looking for ATmega328P-PU. Figure 4-4 shows a close-up of the chip.

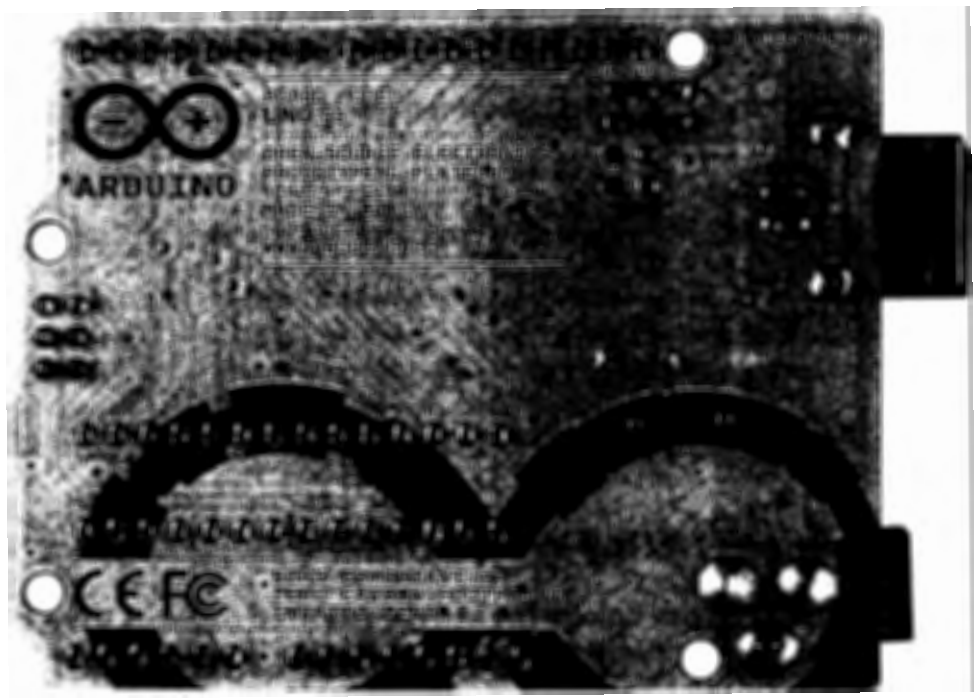

Figure 4-3: Back side of Arduino Uno.

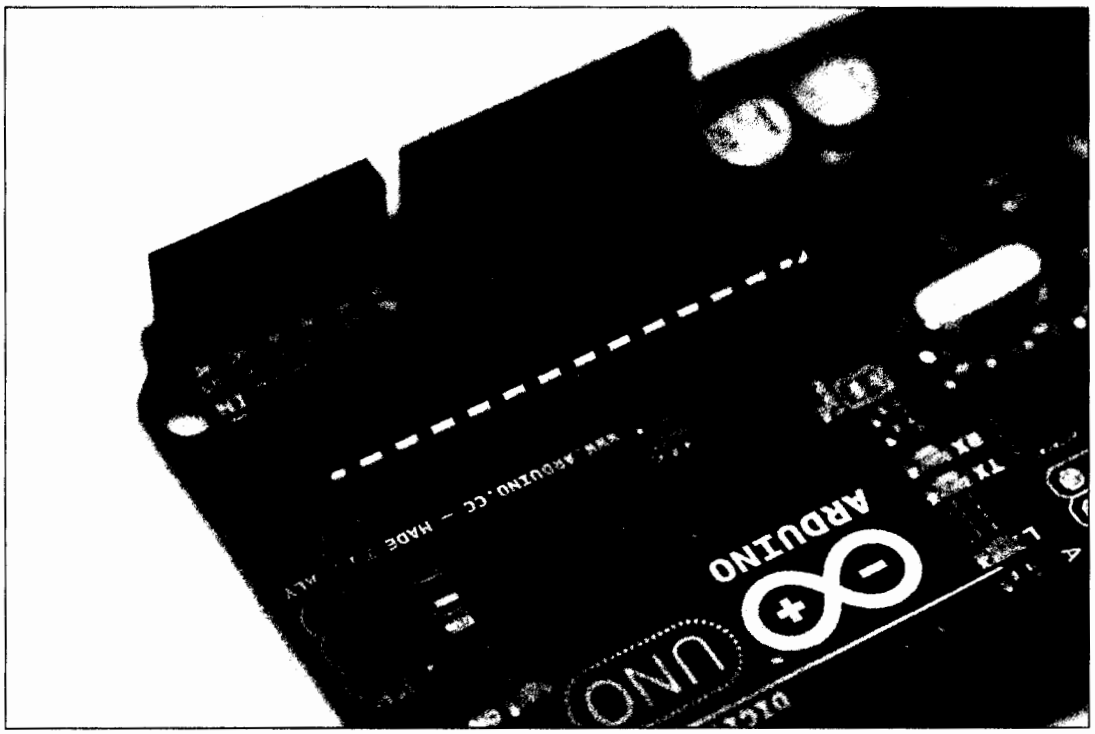

Figure 4-4: Close-up of the ATmega-328P-PU chip.

Configuring the software

After you confirm the type of board you are using, you have to provide that information to the software. From the Arduino main menu bar (at the top of the Arduino window on Windows and at the top of the screen on Mac OS X), choose Tools➪Board. You should see a list of the different kinds of boards supported by the Arduino software. Select your board from the list, as shown in Figure 4-5.

Next, you we need to select the serial port. The serial port is the connection that enables your computer and the Arduino device to communicate. *Serial* describes the way that data is sent, one bit of data (0 or 1) at a time. The *port* the physical interface, in this case a USB socket. I talk more about serial communication in Chapter 7.

To determine the serial port, choose Tools➪Serial Port. A list displays of devices connected to your computer (see Figure 4-6). This list contains any device that can talk in serial, but for the moment, you're only interested in finding the Arduino. If you've just installed Arduino and plugged it in, it should be at the top of the list. For OS X users, this is shown as `/dev/tty.usbmodemXXXXXX` (where XXXXXX is a randomly signed number). On Windows, the same is true, but the serial ports are named `COM1`, `COM2`, `COM3`, and so on. The highest number is usually the most recent device.

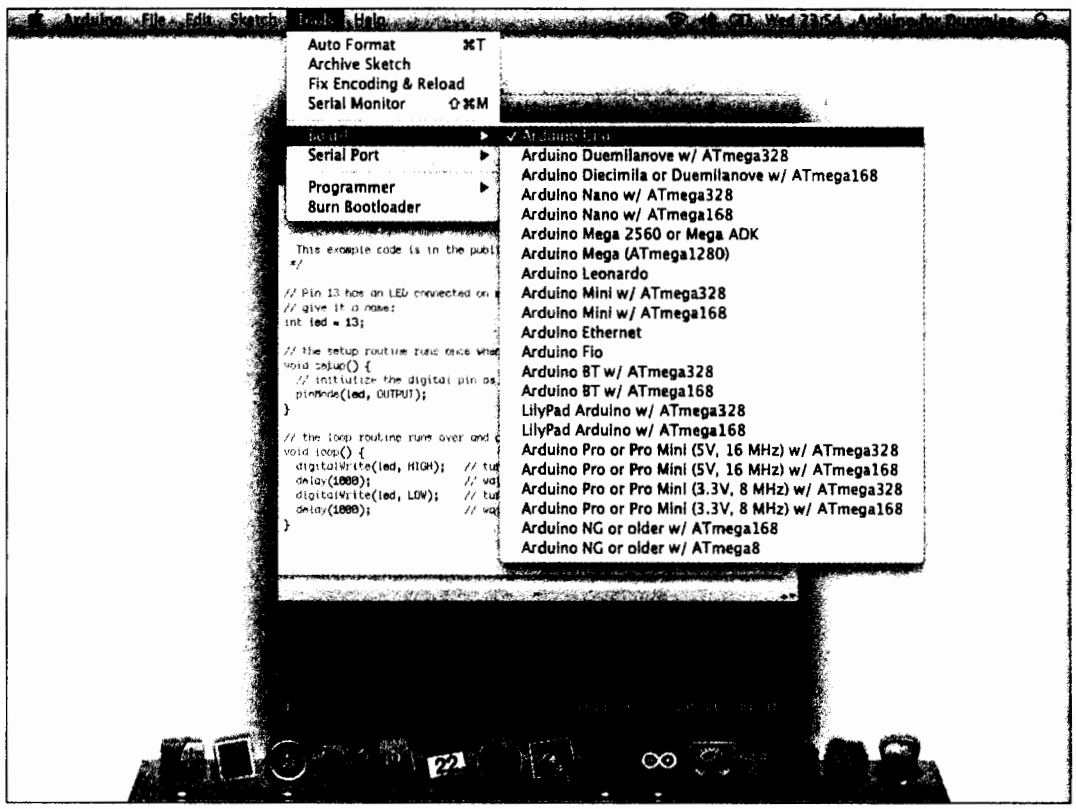

Figure 4-5: Select Arduino Uno from the Board menu.

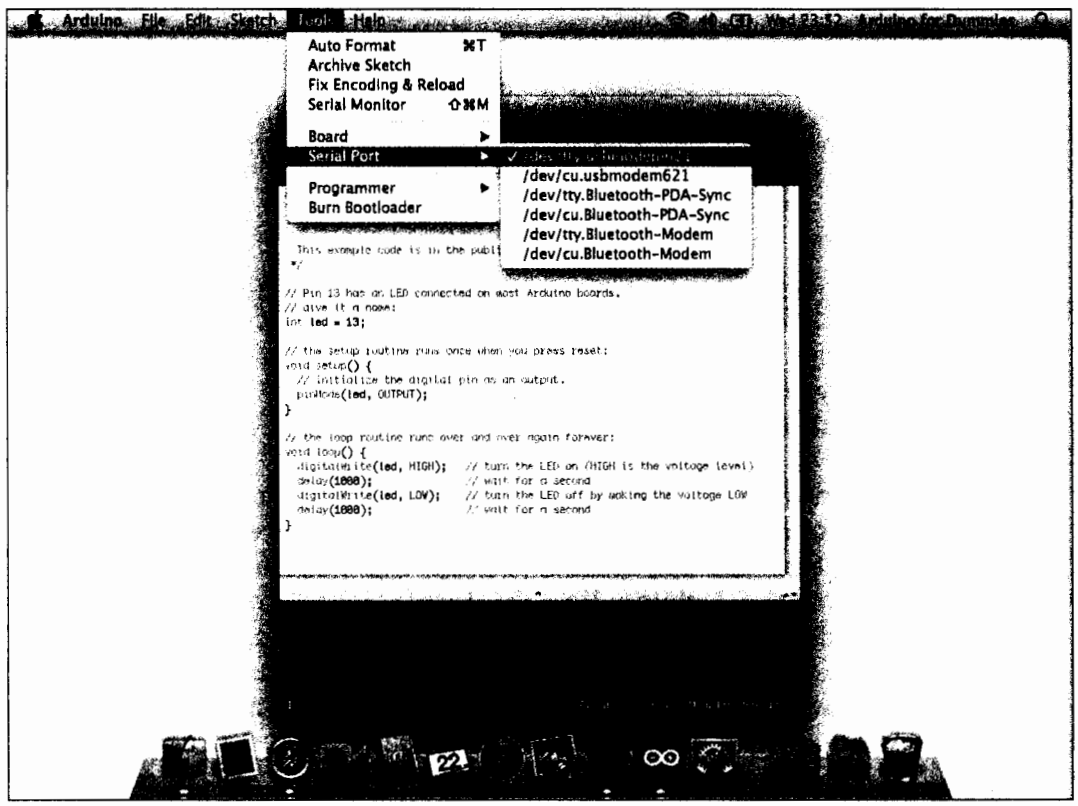

Figure 4-6: A list of serial connections available to the Arduino environment.

After you find your serial port, select it. It should appear in the bottom right of the Arduino GUI, along with the board you selected (see Figure 4-7).

Figure 4-7: Arduino GUI board and port.

Uploading the sketch

Now that you have told the Arduino software what kind of board you are communicating with and which serial port connection it is using, you can upload the Blink sketch you found earlier in this chapter.

First click the Verify button. Verify checks the code to make sure it makes sense. This doesn't necessarily mean your code will do what you are anticipating, but it verifies that the syntax is written in a way Arduino can understand (see Chapter 2). You should see a progress bar and the text `Compiling Sketch` (see Figure 4-8) for a few seconds, followed by the text `Done compiling` after the process has finished.

Figure 4-8: The progress bar shows that the sketch is compiling.

If the sketch compiled successfully, you can now click the Upload button next to the verify button. A progress bar appears, and you see a flurry of activity on your board from the two LEDs marked RX and TX (that I mentioned in Chapter 2). These show that the Arduino is sending and receiving data. After a few seconds, the RX and TX LEDs stop blinking, and a `Done Uploading` (see Figure 4-9) message appears at the bottom of the Arduino window.

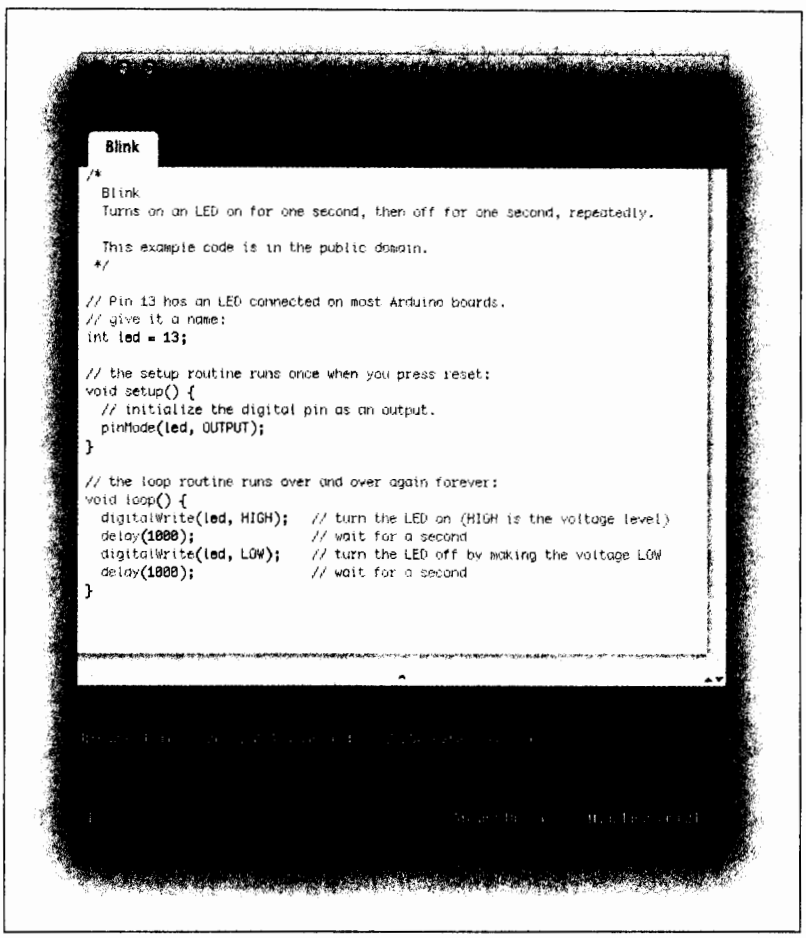

Figure 4-9: The Arduino GUI is done uploading.

Congratulate yourself!

You should see the LED marked L blinking away reassuringly: on for a second, off for a second. If that is the case, give yourself a pat on the back. You've just uploaded your first piece of Arduino code and entered the world of physical computing!

If you don't see the blinking L, go back through all the preceding sections. Make sure you have installed Arduino properly and then give it one more go. If you still don't see the blinking L, check out the excellent troubleshooting page on the official Arduino site: `http://arduino.cc/en/Guide/troubleshooting`.

What just happened?

Without breaking a sweat you've just uploaded your first sketch to an Arduino. Well done (or good job, if you're from the States)!

Just to recap, you have now

- Plugged your Arduino into your computer
- Opened the Arduino software
- Set the board and serial port
- Opened the Blink sketch from the Examples folder and uploaded it to the board

In the following section, I walk you through the various sections of the first sketch you just uploaded.

Looking Closer at the Sketch

In this section, I show you the Blink sketch in a bit more detail so that you can see what's actually going on. When the Arduino software reads a sketch, it very quickly works through it one line at a time, in order. So the best way to understand the code is to work through it the same way, very slowly.

Arduino uses the programming language C, which is one of the most widely used languages of all time. It is an extremely powerful and versatile language, but it takes some getting used to.

If you followed the previous section, you should already have the Blink sketch on your screen. If not, you can find it by choosing File➪Examples➪01.Basics➪Blink (refer to Figure 4-1).

When the sketch is open, you should see something like this:

```
/*
  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
}
```

The sketch is made up of lines of code. When looking at the code as a whole, you can identify four distinct sections:

- Comments
- Declarations
- void loop
- void setup

Read on for more details about each of these sections.

Comments

Here's what you see in the first section of the code:

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

  This example code is in the public domain.
 */
```

Multiline comment

Notice that the code lines are enclosed within the symbols `/*` and `*/`. These symbols mark the beginning and end of a *multi-line* or *block comment.* Comments are written in plain English and, as the name suggests, provide an explanation or comment on the code. Comments are completely ignored by the software when the sketch is compiled and uploaded. Consequently, comments can contain useful information about the code without interfering with how the code runs.

In this example, the comment simply tells you the name of the sketch, and what it does, and provides a note explaining that this example code is in the public domain. Comments often include other details such as the name of the author or editor, the date the code was written or edited, a short description of what the code does, a project URL, and sometimes even contact information for the author.

Single-line comment

Further down the sketch, inside the `setup` and `loop` functions, text on your screen appears with the same shade of gray as the comments above. This text is also a comment. The symbols `//` signify a single-line comment as opposed to a multiline comment. Any code written after these lines will be ignored for that line. In this case, the comment is describing the piece of code that comes after it:

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

This single line of code is in the declarations section of the sketch, but "what is a declaration?" I hear you ask. Read on to find out.

Declarations

Declarations (which aren't something you put up at Christmas, ho ho ho) are values that are stored for later use by the program. In this case, a single variable is being declared, but you could declare many other variables or even include libraries of code in your sketch. For now, all that is important to remember is that variables can be declared before the setup function.

Variables

Variables are values that can change depending on what the program does with them. In C, you can declare the type, name, and value of the variable before the main body of code, much as ingredients are listed at the start of a recipe.

```
int led = 13;
```

The first part sets the type of the variable, creating an integer (`int`). An integer is any whole number, positive or negative, so no decimal places are required. It's worth noting that for Arduino, there are lower and upper limits for the `int` type of variable: -32,768 to 32,767. Beyond those limits, a different

type of variable must be used, known as a `long` (you learn more about these in Chapter 11). But for now, an `int` will do just fine. The name of the variable is `led` and is purely for reference; it can be any single word that's useful for figuring out what the variable applies to. Finally, the value of the variable is set to 13. In this case, that is the number of the pin that is being used.

Variables are especially useful when you refer to a value repeatedly. In this case, the variable is called `led` because it refers to the pin that the physical LED is attached to. Now, every time you want to refer to pin 13, you can write `led` instead. Although this approach may seem like extra work initially, it means that if you decided to change the pin to pin 11, you would need only to change the variable at the start; every subsequent mention of `led` would automatically be updated. That's a big timesaver over having to trawl through the code to update every occurrence of 13.

With the declaration made, the code enters the `setup` function.

Functions

The next two sections are functions and begin with the word `void`: `void setup` and `void loop`. A *function* is a bit of code that performs a specific task, and that task is often repetitive. Rather than writing the same code out again and again, you can use a function to tell the code to perform this task again.

Consider the general process you follow to assemble IKEA furniture. If you were to write these general instructions in code, using a function, they would look something like this:

```
void buildFlatpackFurniture() {
        buy a flatpack;
        open the box;
        read the instructions;
        put the pieces together;
        admire your handiwork;
        vow never to do it again;
}
```

The next time you want to use these same instructions, rather than writing out the individual steps, you can simply call the procedure named `buildFlatpackFurniture()`.

Although not compulsory, there is a naming convention for function or variable names containing multiple words. Because these names cannot have spaces, you need a way to distinguish where all the words start and end; otherwise, it takes a lot longer to scan over them. The convention is to

capitalize the first letter of each word after the first. This greatly improves the readability of your code when scanning through it, so I highly recommend that you adhere to this rule in all your sketches for your benefit and the benefit of those reading your code!

The word *void* is used when the function returns no value, and the word that follows is the name of that function. In some circumstances, you might either put a value(s) into a function or expect a value(s) back from it, the same way you might put numbers into a calculation and expect a total back, for example.

`void setup` and `void loop` must be included in every Arduino sketch; they are the bare minimum required to upload. But it is also possible to write your own custom functions for whatever task you need to do. For now, you just need to remember that you have to include `void setup` and `void loop` in every Arduino sketch you create. Without these functions, the sketch will not compile.

Setup

Setup is the first function an Arduino program reads, and it runs only once. Its purpose, as hinted in the name, is to set up the Arduino device, assigning values and properties to the board that do not change during its operation. The `setup` function looks like this:

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

Notice on your screen that the text `void setup` is orange. This color indicates that the Arduino software recognizes it as a *core* function, as opposed to a function you have written yourself. If you change the case of the words to `Void Setup`, you see that they turn black, which illustrates that the Arduino code is *case sensitive*. This is an important point to remember, especially when it's late at night and the code doesn't seem to be working.

The contents of the `setup` function are contained within the curly brackets, `{` and `}`. Each function needs a matching set of curly brackets. If you have too many of either bracket, the code does not compile, and you are presented with an error message that looks like the one shown in Figure 4-10.

Figure 4-10: The Arduino software is telling you that a bracket is missing.

PinMode

The `pinMode` function configures a specified pin either for input or output: to either receive or send data. The function includes two parameters:

- `pin`: The number of the pin whose mode you want to set
- `mode`: Either `INPUT` or `OUTPUT`

In the Blink sketch, after the two lines of comments, you see this line of code:

```
pinMode(led, OUTPUT);
```

The word `pinMode` is highlighted in orange. As I mentioned earlier in this chapter, orange indicates that Arduino recognizes the word as a core function. `OUTPUT` is also coloured blue so that it can be identified as a constant, which is a predefined variable in the Arduino language. In this case, that constant sets the mode of this pin. You can find more about constants in Chapter 7.

That's all you need for `setup`. The next section moves on to the `loop` section.

Loop

The next section you see in the Blink sketch is the `void loop` function. This is also highlighted in orange so the Arduino software recognizes it as a core function. `loop` is a function, but instead of running one time, it runs continuously until you until you press the reset button on the Arduino board or you remove the power. Here is the `loop` code:

```
void loop() {
  digitalWrite(led, HIGH);   // set the LED on
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // set the LED off
  delay(1000);               // wait for a second
}
```

DigitalWrite

Within the `loop` function, you again see curly brackets and two different orange functions: `digitalWrite` and `delay`.

First is digitalWrite:

```
digitalWrite(led, HIGH);   // set the LED on
```

The comment says `set LED on`, but what exactly does that mean? The function `digitalWrite` sends a digital value to a pin. As mentioned in Chapter 2, digital pins have only two states: on or off. In electrical terms, these can be referred to as either a `HIGH` or `LOW` value, which is relative to the voltage of the board.

An Arduino Uno requires 5V to run, which is provided by either a USB or a higher external power supply, which the Arduino board reduces to 5V. This means that a `HIGH` value is equal to 5V and `LOW` is equal to 0V.

The function includes two parameters:

- `pin`: The number of the pin whose mode you want to set
- `value`: Either `HIGH` or `LOW`

So `digitalWrite(led, HIGH);` in plain English would be "send 5V to pin 13 on the Arduino," which is enough voltage to turn on an LED.

Delay

In the middle of the `loop` code, you see this line:

```
delay(1000);          // wait for a second
```

This function does just what it says: It stops the program for an amount of time in milliseconds. In this case, the value is 1000 milliseconds, which is equal to one second. During this time, nothing happens. Your Arduino is chilling out, waiting for the delay to finish.

The next line of the sketch provides another `digitalWrite` function, to the same pin, but this time writing it low:

```
digitalWrite(led, LOW);    // set the LED off
```

This tells Arduino to send 0V (ground) to pin 13, which turns the LED off. This is followed by another delay that pauses the program for one second:

```
delay(1000);               // wait for a second
```

At this point, the program returns to the start of the loop and repeats itself, ad infinitum.

So the loop is doing this:

- Sending 5v to pin 13, turning on the LED
- Waiting a second
- Sending 0v to pin 13, turning off the LED
- Waiting a second

As you can see, this gives you the blink!

Blinking Brighter

I have mentioned pin 13 a few times in this chapter. Why does that pin blink the LED on the Arduino board? The LED marked L is actually connected just before it reaches pin 13.On early boards, it was necessary to provide your own LED. Because the LED proved so useful for debugging and signaling, there is now one in permanent residence to help you out.

For this next bit, you need a loose LED from your kit. LEDs come in a variety of shapes, colors, and sizes but should look something like the one shown in Figure 4-11.

Take a look at your LED and notice that one leg is longer than the other. Place the long leg (anode or +) of the LED in pin 13 and the short leg (cathode or -) in GND (ground). You see the same blink, but it is (hopefully) bigger and brighter depending on the LED you use. Insert the LED as shown in Figure 4-12.

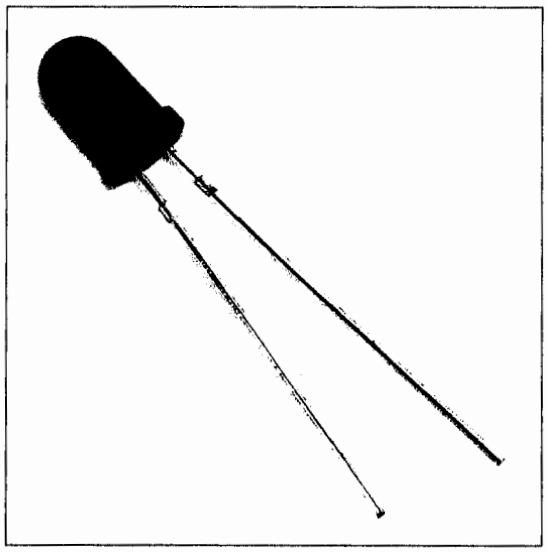

Figure 4-11: A lone LED, ready to be put to work.

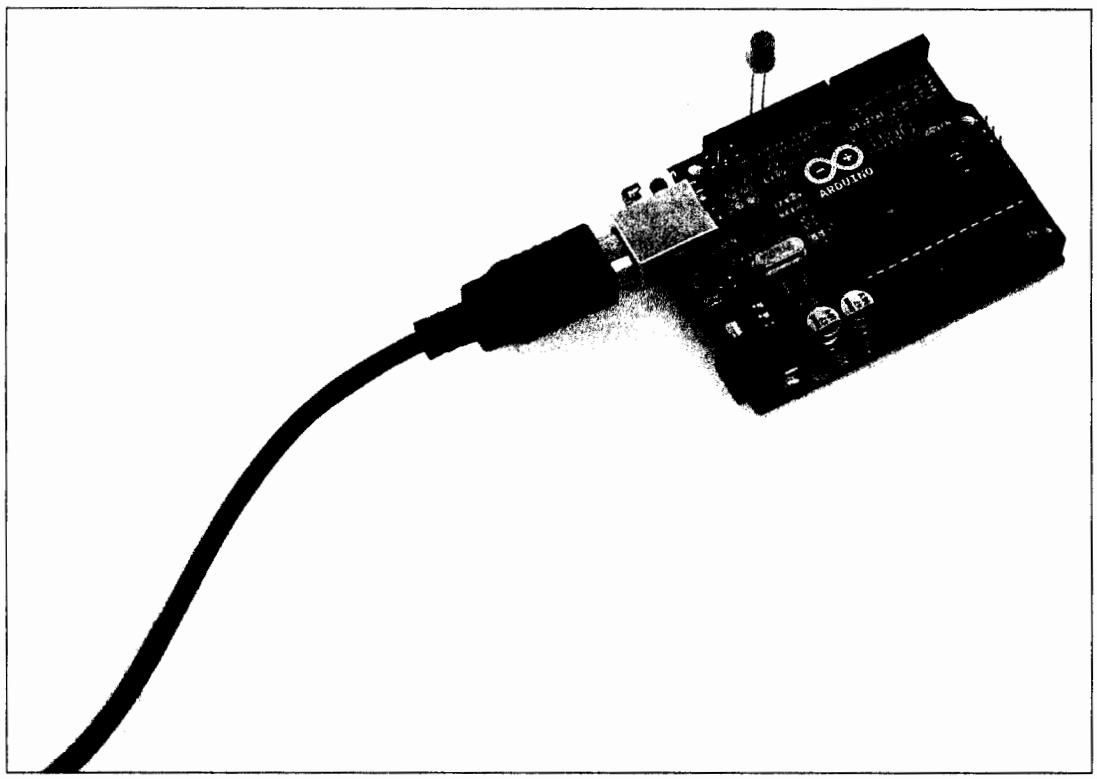

Figure 4-12: Arduino LED Pin 13.

From the description of the `digitalWrite` function in the preceding section, you know that your Arduino is sending 5V to pin 13 when it is HIGH. This can be too much voltage for most LEDs. Fortunately, another feature of pin 13 is a built-in pull-down resistor. This resistor keeps your LED at a comfortable voltage and ensures it has a long and happy life.

Tweaking the Sketch

I've gone over this sketch in great detail, and I hope everything is making sense. The best way to understand what is going on, however, is to experiment! Try changing the delay times to see what results you get. Here are a couple of things you can try:

- Make the LED blink the SOS signal.
- See how fast you can make the LED blink before it appears to be on all the time.

Chapter 7

Basic Sketches: Inputs, Outputs, and Communication

In This Chapter

- Fading like a pro
- Coming to grips with inputs
- Varying resistances with potentiometers
- Showing off your stats with the serial monitor

In this chapter, I discuss some of the basic sketches that you need to get you on your Arduino feet. This chapter covers a broad range of inputs and outputs using the sensors in your kit. If you don't yet have a kit, I suggest reading through Chapter 2 to find one of the recommended ones.

The Blink sketch (described in Chapter 4) gives you the basis of an Arduino sketch, but in this chapter, you expand on it by adding circuits to your Arduino. This chapter walks you through building circuits using a breadboard, as mentioned in Chapter 5, and additional components from your kit to build a variety of circuits.

I detail uploading the appropriate code to your Arduino, walk you through each sketch line by line, and suggest tweaking the code yourself to gain a better understanding of it.

Uploading a Sketch

Throughout this chapter and much of the book, you learn about a variety of circuits, each with their respective sketches. The content of the circuits and sketches can vary greatly and are detailed in each of the examples in this book. Before you get started, there is one simple process for uploading a sketch to an Arduino board that you can refer back to.

Follow these steps to upload your sketch:

1. **Connect your Arduino using the USB cable.**

   The square end of the USB cable connects to your Arduino and the flat end connects to a USB port on your computer.

2. **Choose Tools➪Board➪Arduino Uno to find your board in the Arduino menu.**

   In most of the examples in this book, the board is Arduino Uno, but you can also find many other boards through this menu as well, such as the Arduino MEGA 2560 and Arduino Leonardo.

3. **Choose the correct serial port for your board.**

   You find a list of all the available serial ports by choosing Tools➪Serial Port➪ comX or /dev/tty.usbmodemXXXXX. X marks a sequentially or randomly assigned number. ln Windows, if you have just connected your Arduino, the COM port will normally be the highest number, such as com 3 or com 15. Many devices can be listed on the COM port list, and if you plug in multiple Arduinos, each one will be assigned a new number. On Mac OS X, the /dev/tty.usbmodem number will be randomly assigned and can vary in length, such as /dev/tty.usbmodem1421 or /dev/tty.usb modem262471. Unless you have another Arduino connected, it should be the only one visible.

4. **Click the Upload button.**

   This is the button that points to the right in the Arduino environment, as detailed in Chapter 3. You can also use the keyboard shortcut Ctrl+U for Windows or Cmd+U for Mac OS X.

Now that you know how to upload a sketch, you should be suitably hungry for some more Arduino sketches. To help you understand the first sketch in this chapter, I first tell you about a technique called Pulse Width Modulation (PWM). The next section briefly describes PWM and prepares you for fading an LED.

Using Pulse Width Modulation (PWM)

When I tell you about the board in Chapter 2, I mention that sending an analog value uses something called Pulse Width Modulation (PWM). This is a technique that allows your Arduino, a digital device, to act like an analog device. In the following example, this allows you to fade an LED rather than just turn it on or off.

Here's how it works: A digital output is either on or off. But it can be turned on and off extremely quickly thanks in part to the miracle of silicon chips. If

the output is on half the time and off half the time, it is described as having a 50 percent duty cycle. The *duty cycle* is the period of time during which the output is active, so that could be any percentage — 20 percent, 30 percent, 40 percent, and so on.

When you're using LEDs as an output, the duty cycle has a special effect. Because it is blinking faster than the human eye can perceive, an LED with a 50 percent duty cycle looks as though it is at half brightness. This is the same effect that allows you to perceive still images shown at 24 frames per second (or above) as a moving image.

With a DC motor as an output, a 50 percent duty cycle has the effect of moving the motor at half speed. So in this case PWM allows you to control the speed of a motor by pulsing it at an extremely fast rate.

So despite PWM's being a digital function, it is referred to as an `analogWrite` because of the perceived effect it has on components.

The LED Fade Sketch

In this sketch, you make an LED fade on and off. In contrast to the sketch that resulted in a blinking LED in Chapter 4, you need some extra hardware to make the LED fade on and off.

For this project you need:

- An Arduino Uno
- A breadboard
- An LED
- A resistor (greater than 120 ohm)
- Jump wires

It's always important to make sure that your circuit is not powered while you're making changes to it. You can easily make incorrect connections, potentially damaging the components. So before you begin, make sure that the Arduino is unplugged from your computer or any external power supply.

Lay out the circuit as shown in Figure 7-1. This makes a simple circuit like the one used for the Blink sketch in Chapter 4, using pin 9 instead of pin 13. The reason for using pin 9 instead of 13 is that 9 is capable of Pulse Width Modulation (PWM), which is necessary to fade the LED. However, note that pin 9 requires a resistor to limit the amount of current supplied to the LED. On pin 13, this resistor is already included on the Arduino board itself, so you didn't need to worry about this.

Putting up resistance

As you learn from Chapter 6, calculating the correct resistance is important for a safe and long lasting circuit. In this case you are potentially supplying your LED with a source 5V (volts), the maximum that a digital pin can supply. A typical LED such as those in your kit has an approximate maximum forward voltage of 2.1V (volts), so a resistor is needed to protect it. It draws a maximum current of approximately 25mA (milliamps). Using these figures, you can calculate the resistance (ohms):

$R = (V_S - V_L) / I$

$R = (5-2.1) / 0.025 = 116$ ohms

The nearest fixed resistor above this calculation that you can buy is 120 ohms (brown, red, brown), so if you have one of those you are in luck. If not you can apply the rule of using the nearest resistor above this value. This resists more voltage than the optimum, but your LED is safe and you can always switch out the resistor later when you are looking to make your project more permanent. Suitable values from various kits would include 220Ω, 330Ω, and 560Ω.

You can always refer to Chapter 6 to find your resistor value on the color chart or use a multimeter to measure the value of your resistors. There are even apps for your smartphone that have resistor color charts (although this may be a source of great embarrassment and ridicule among friends).

Figure 7-2 shows the schematic of the circuit. This schematic shows you the simple circuit connection. Your digital pin, pin 9, is connected to the long leg of the LED; the short leg connects to the resistor and that goes on to ground, GND. In this circuit, the resistor can be either before or after the LED, as long as it is in the circuit.

It's always a good idea to color code your circuits — that is, use various colors to distinguish one type of circuit from another. Doing so greatly helps keep things clear and can make problem solving much easier. There are a few good standards to keep to. The most important areas to color code are power and ground. These are nearly always colored red and black, respectively, but you might occasionally see them as white and black as well, as mentioned in Chapter 6.

The other type of connection is usually referred to as a signal wire, which is a wire that sends or receives an electrical signal between the Arduino and a component. Signal wires can be any color that is not the same as the power or ground color.

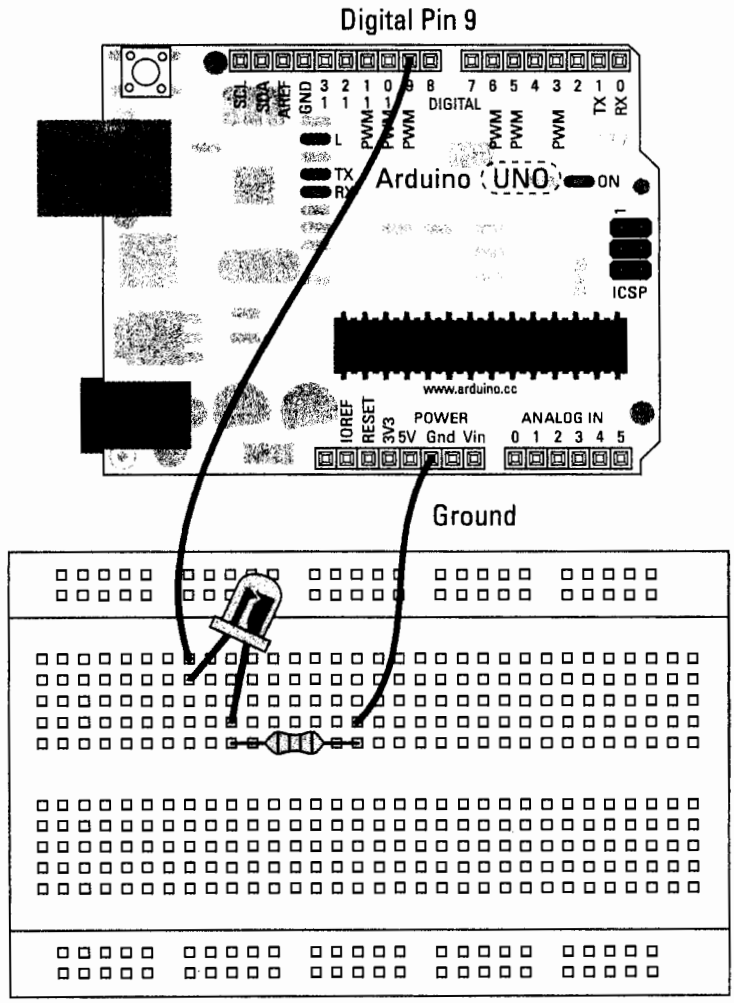

Figure 7-1: Pin 9 is connected to a resistor and an LED and then goes back to ground.

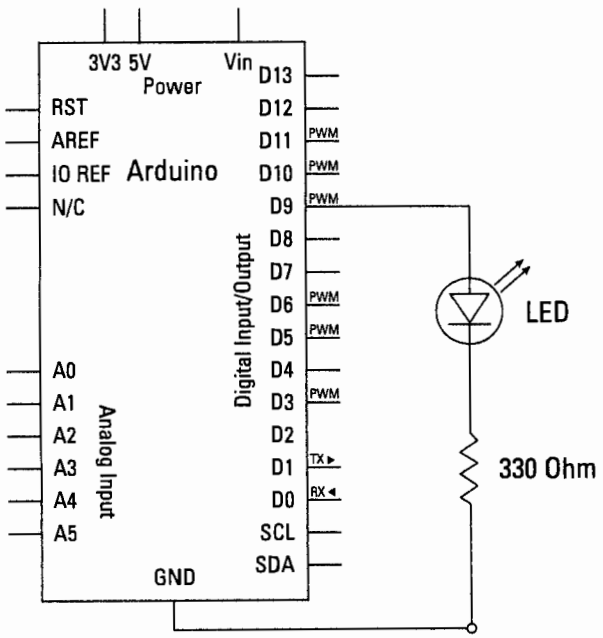

Figure 7-2: A schematic of the circuit to fade an LED.

After you assemble your circuit, you need the appropriate software to use it. From the Arduino menu, choose File⇨Examples⇨01.Basics⇨Fade to call up the Fade sketch. The complete code for the Fade sketch is as follows:

```
/*
 Fade

 This example shows how to fade an LED on pin 9
 using the analogWrite() function.

 This example code is in the public domain.
 */

int led = 9;           // the pin that the LED is attached to
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup()  {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop()  {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

Upload this sketch to your board following the instructions at the start of the chapter. If everything has uploaded successfully, the LED fades from off to full brightness and then back off again.

If you don't see any fading, double-check your wiring:

- Make sure that you're using the correct pin number.
- Check that your LED is correctly situated, with the long leg connected by a wire to pin 9 and the short leg connected via the resistor and a wire to GND (ground).
- Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

Understanding the fade sketch

By the light of your fading LED, take a look at how this sketch works.

The comments at the top of the sketch reveal exactly what's happening in this sketch: Using pin 9, a new function called `analogWrite()` causes the LED to fade off and on. After the comments, three declarations appear:

```
int led = 9;           // the pin that the LED is attached to
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by
```

Declarations, as mentioned in Chapter 4, are declared before the `setup` or `loop` functions. The Fade sketch has three variables: `led`, `brightness` and `fadeAmount`. These are integer variables and are capable of the same range of values, but are all used for different parts of the process of fading an LED.

With declarations made, the code enters the setup function. The comments are reminders that setup runs only once and that just one pin is set as an output. Here you can see the first variable at work. Instead of writing `pinMode(9, OUTPUT)`, you have `pinMode(led, OUTPUT)`. Both work exactly the same, but the latter uses the `led` variable.

```
// the setup routine runs once when you press reset:
void setup()  {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}
```

The loop starts to get a bit more complicated:

```
// the loop routine runs over and over again forever:
void loop()  {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

Instead of just on and off values, a fade needs a range of values. `analogWrite` allows you to send a value of 0 to 255 to a PWM pin on the Arduino. 0 is equal to 0v and 255 is equal to 5v, and any value in between gives a proportional voltage, thus fading the LED.

The loop begins by writing the brightness value to pin 9. A `brightness` value of `0` means that the LED is currently off.

```
// set the brightness of pin 9:
  analogWrite(led, brightness);
```

Next you add the fade amount to the `brightness` variable, making it equal to 5. This won't be written to pin 9 until the next loop.

```
// change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;
```

The brightness must stay within the range that the LED can understand. This is done using an `if` statement, which essentially tests variables to determine what to do next.

The word `if` starts the statement. The conditions are in the brackets that follow, so in this case you have two: the first being, is brightness equal to 0? The second is, is brightness equal to 255? In this case == is used rather than =. The double equal sign indicates that the code is comparing two values (if a is equal to b) rather than assigning a value (a equals b). In between the two conditional statements is the symbol `||`, which is the symbol for `OR`.

```
if (brightness == 0 || brightness == 255) {
  fadeAmount = -fadeAmount ;
}
```

So the complete statement is, "If the variable named brightness is equal to 0 or equal to 255, then do whatever is inside the curly brackets." When this eventually becomes true, the line of code inside the curly brackets is read. This is a basic mathematical statement that inverts the variable named `fadeAmount`. During the fade up to full brightness, 5 is added to the brightness with every loop. When 255 is reached, the `if` statement becomes true and `fadeAmount` changes from 5 to -5. Then every loop updates to "add minus 5" to the brightness until 0 is reached, when the `if` statement becomes true again. This inverts the `fadeAmount` of -5 back to 5 to bring everything back to where it started.

```
  fadeAmount = -fadeAmount ;
```

These conditions give us a number that is continually counting up and then down that an Arduino can use to continually fade your LED on and then off again.

Tweaking the fade sketch

There are many ways to get something done, but I don't cover them all in this book; I can, however, show you one different way to fade an LED using

the circuit that you created in the previous section. The following code is the Fading code from a previous release of Arduino, and in some ways I prefer it to the current example. Upload it and you will see that no visible difference exists between this and the previous example.

Some areas of the code appear colored on your screen, most often either orange or blue. This marks a function or a statement that is recognized by the Arduino environment (can be extremely handy for spotting typos). Color can be difficult to recreate in a black-and-white book, so any colored code appears in **bold.**

```
/*
 Fading

 This example shows how to fade an LED using the analogWrite() function.

 The circuit:
 * LED attached from digital pin 9 to ground.

 Created 1 Nov 2008
 By David A. Mellis
 modified 30 Aug 2011
 By Tom Igoe

 http://arduino.cc/en/Tutorial/Fading

 This example code is in the public domain.

 */


int ledPin = 9;    // LED connected to digital pin 9

void setup()  {
  // nothing happens in setup
}

void loop()  {
  // fade in from min to max in increments of 5 points:
  for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }

  // fade out from max to min in increments of 5 points:
  for(int fadeValue = 255 ; fadeValue >= 0; fadeValue -=5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
}
```

The default example is very efficient and does a simple fade very well, but it relies on the `loop` function to update the LED value. This version uses `for` loops, which operate within the main Arduino `loop` function.

Using for loops

After a sketch enters a `for` loop, it sets up the criteria for exiting the loop and cannot move out of it until the criteria are met. `for` loops are often used for repetitive operations; in this case, `for` loops are used for increasing or decreasing a number at a set rate to create the repeating fade.

The first line of the `for` loop defines the initialization, the test, and the amount of increment or decrement:

```
for(int fadeValue = 0;fadeValue <= 255;fadeValue +=5)
```

In plain English, this would read: "Make a variable called `fadeValue` (that is local to this `for` loop) equal to a value of 0; check to see whether it is less than or equal to 255; if it is, set `fadeValue` to be equal to `fadeValue` plus 5." `fadeValue` is equal to 0 only when it is created; after that, it is increased by 5 every time the `for` loop cycles.

Within the loop, the code updates the `analogWrite` value of the LED and waits 30 milliseconds (ms) before attempting the loop one more time.

```
for(int fadeValue = 0 ; fadeValue <= 255; fadeValue +=5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
```

This `for` loop behaves the same as the main `loop` in the default Fade example, but because the `fadeValue` is contained in its own loop, and broken into fade up and fade down loops, it is a lot easier for to start experimenting with fading patterns in a more controlled way. For example, try changing `+=5` and `-=5` to different values (that divide into 255 neatly) and you can have some interesting asymmetrical fading.

You could also copy and paste the same `for` loops to create further fading animations. Bear in mind, however, that while it's in a `for` loop, your Arduino can do nothing else.

The Button Sketch

This is the first and perhaps most basic of inputs that you can and should learn for your Arduino projects: the modest pushbutton.

For this project, you need:

- An Arduino Uno
- A breadboard
- A 10k ohm resistor
- A pushbutton
- An LED
- Jump wires

Figure 7-3 shows the breadboard layout for the Button circuit. It's important to note which legs of the pushbutton are connected. In most cases, these small pushbuttons are made to bridge the gap over the center of your breadboard exactly. If they do bridge the gap, the legs are usually split at 90 degrees to the gap (left to right on this diagram).

You can test the legs of a pushbutton with a continuity tester if your multimeter has that function (as detailed in Chapter 5).

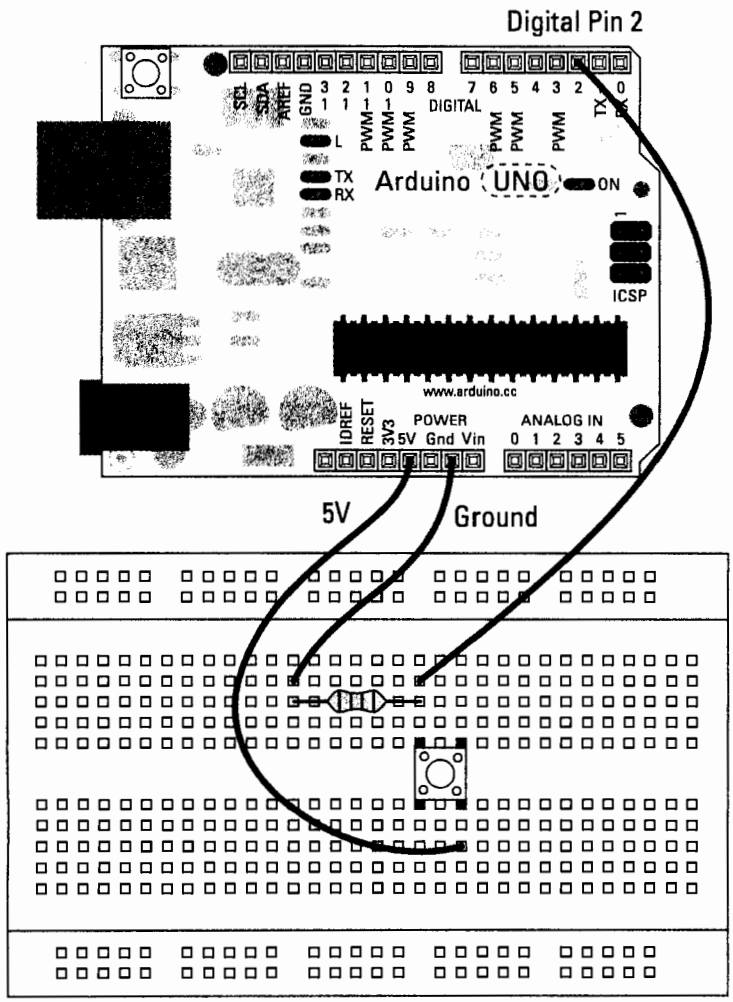

Figure 7-3: Pin 2 is reading the pushbutton.

From the schematic in Figure 7-4, you can see that the resistor leading to ground should be connected to the same side as pin 2, and that when the button is pressed, it connects those to the 5V pin. This setup is used to compare ground (0V) to a voltage (5V) so that you can tell whether the switch is open or closed.

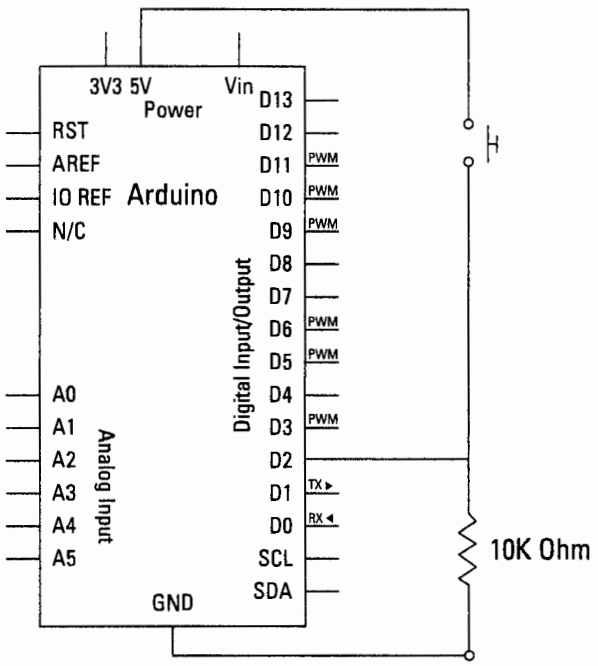

Figure 7-4: A schematic of the pushbutton circuit.

Build the circuit and upload the code from File➪Examples➪02.Digital➪Button.

```
/*
  Button

 Turns on and off a light emitting diode(LED) connected to digital
 pin 13, when pressing a pushbutton attached to pin 2.


 The circuit:
 * LED attached from pin 13 to ground
 * pushbutton attached to pin 2 from +5V
 * 10K resistor attached to pin 2 from ground

 * Note: on most Arduinos there is already an LED on the board
 attached to pin 13.


 created 2005
 by DojoDave <http://www.0j0.org>
 modified 30 Aug 2011
```

```
 by Tom Igoe

 This example code is in the public domain.

 http://www.arduino.cc/en/Tutorial/Button
 */

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

After you upload the sketch, give your button a press and you should see the pin 13 LED light up. You can add a bigger LED to your Arduino board between pin 13 and GND to make it easier to see.

If you don't see anything lighting up, you should double-check your wiring:

- Make sure that your button is connected to the correct pin number.
- If you are using an additional LED, check that it is correctly situated, with the long leg in pin 13 and the short leg in GND. You can also remove it and monitor the LED mounted on the board (marked L) instead.
- Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they will not work.

Understanding the Button sketch

This is your first interactive Arduino project. The previous sketches were all about outputs, but now you are able to affect those outputs by providing your own real-world, human input!

While pressed, your button turns on a light. When released, the light turns off. Take a look at the sketch from the top to see how this happens.

As before, the first step is to declare variables. In this case, there are a couple of differences. The word `const` is short for constant, so instead of changing these two values, they are fixed for the duration of the program. This approach is best used for values that aren't supposed to change; this way, you can make doubly sure that they won't. In this case, pin numbers are being assigned because you won't change the pin number physically.

The variable `buttonState` is set to 0. This is used to monitor changes to the button.

```
const int buttonPin = 2;     // the number of the pushbutton pin
const int ledPin =  13;      // the number of the LED pin

// variables will change:
int buttonState = 0;         // variable for reading the pushbutton status
```

Setup establishes `pinMode`, with `ledPin` (pin 13) as the output and `buttonPin`,(pin 2) as the input.

```
void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}
```

In the main loop, you can see the order of things quite clearly. First, the `digitalRead` function is used on pin 2. Just as `digitalWrite` can write a `HIGH` or `LOW` (1 or 0) value to a pin, `digitalRead` can read a value from a pin. That value is then stored in the variable `buttonState`.

```
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);
```

With the button state established, a test is used to determine what happens next using an `if` statement. The statement reads: "If there is a `HIGH` value (voltage connected to the circuit), then send a `HIGH` value to `ledPin` (pin 13) to turn the LED on; if there is a `LOW` value (the pin is grounded), then send a `LOW` value to `ledPin` to turn the LED off; repeat."

```
  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
  else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

Tweaking the Button sketch

It's often necessary to invert the output of a switch or sensor, and you have two ways to do this. The easiest is to change one word in the code.

By changing the line of code in the above sketch from

```
if (buttonState == HIGH)
```

to

```
if (buttonState == LOW)
```

the output is reversed.

This means that the LED is on until the button is pressed. If you have a computer, this is the easiest option. Simply upload the code.

However, there are often occasions (such as when your laptop battery is dead) when you don't have the means to upload the edited code. Often, the easiest way to flip the logic is to flip the polarity of the circuit.

Instead of connecting pin 2 to a resistor and then GND, connect that resistor to 5V and move the GND wire to the other side of the button, as shown in Figure 7-5.

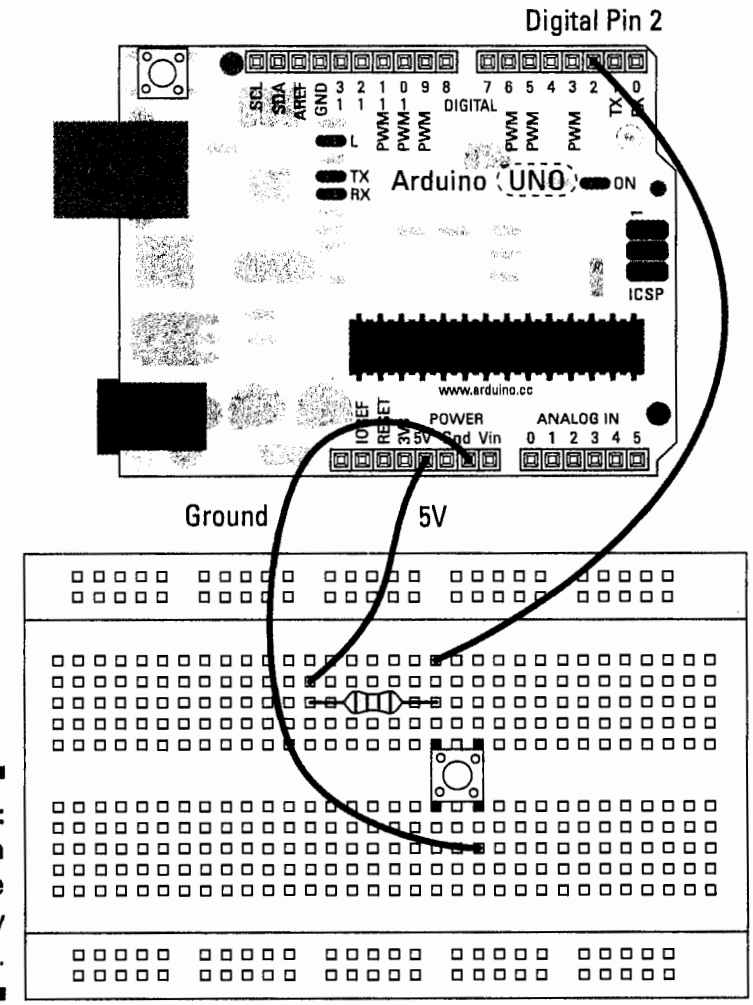

Figure 7-5: A button with the polarity flipped.

The AnalogInput Sketch

The previous sketch showed you how to use a `digitalRead` to read either on or off, but what if you want to handle an analog value such as a dimmer switch or volume control knob?

For this project, you need

- An Arduino Uno
- A breadboard
- A 10k ohm variable resistor
- An LED
- Jump wires