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.

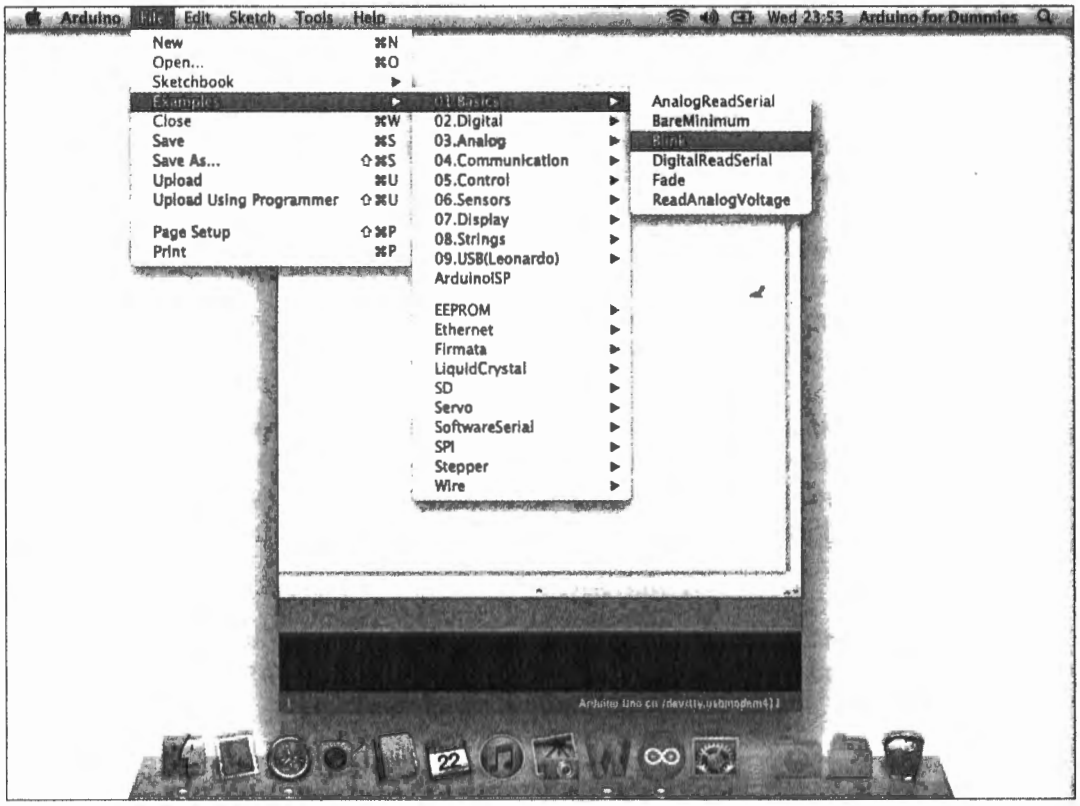

Figure 4-1: Find your way to the Blink sketch.

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.

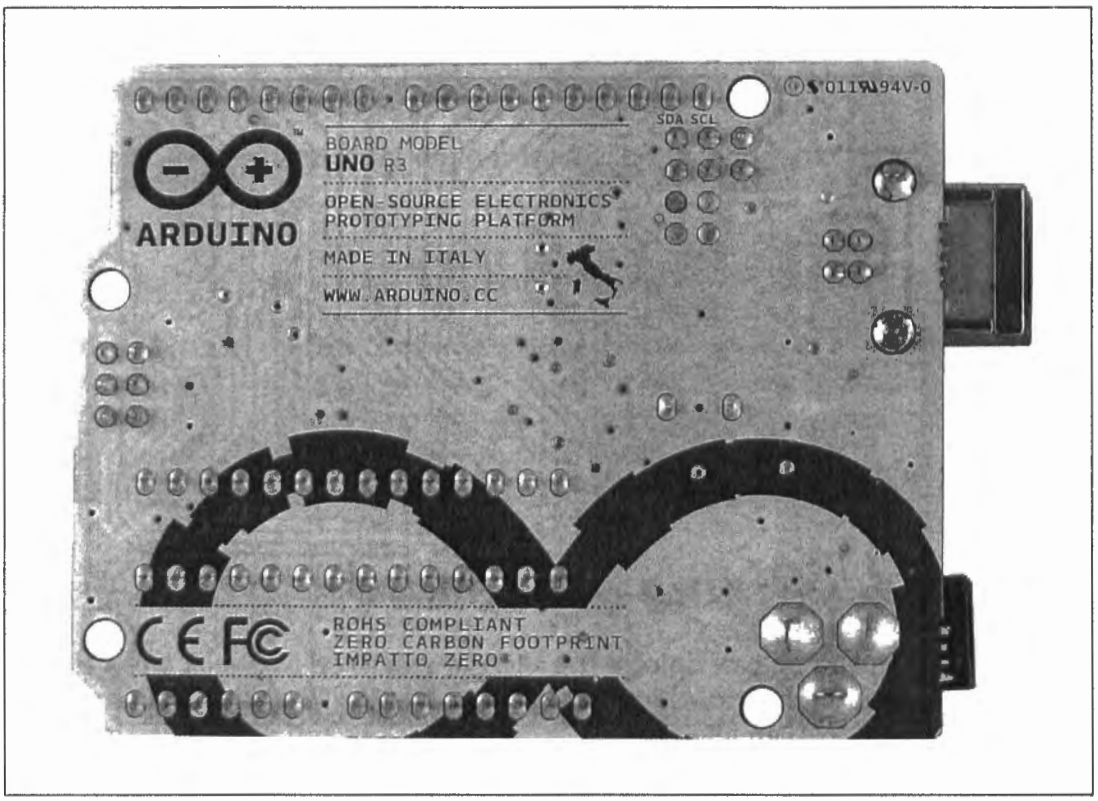

Figure 4-3: Back side of Arduino Uno.

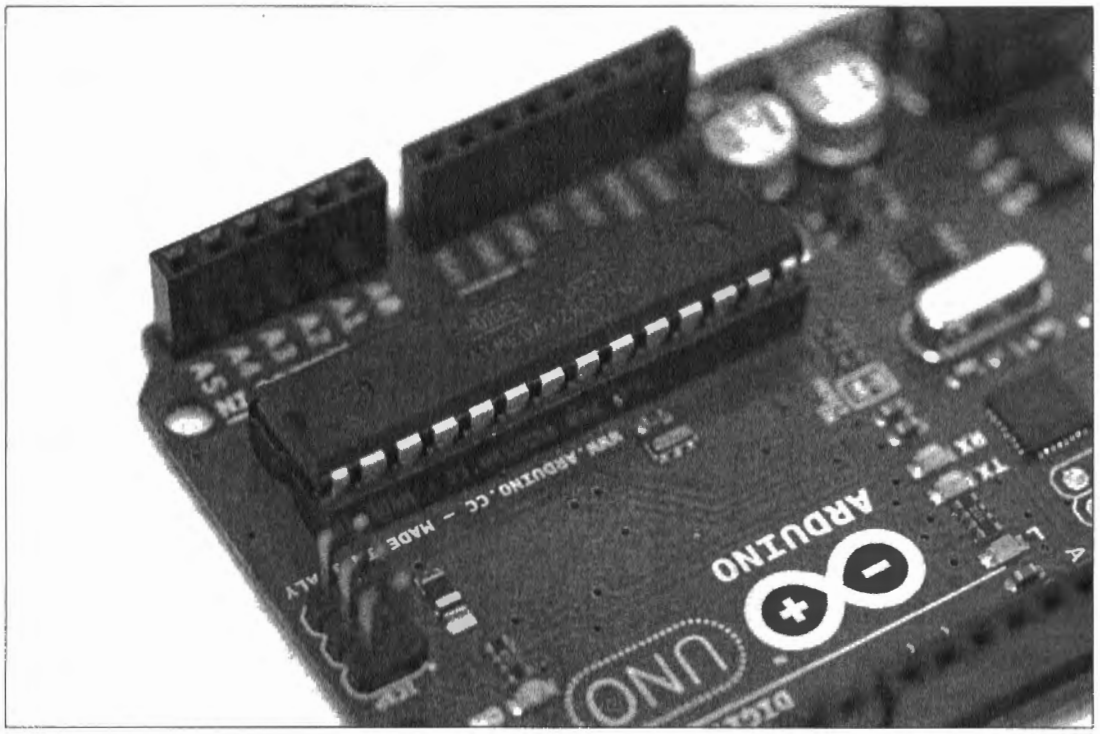

Figure 4-4: Close-up of the ATmega-328P-PU chip.

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.

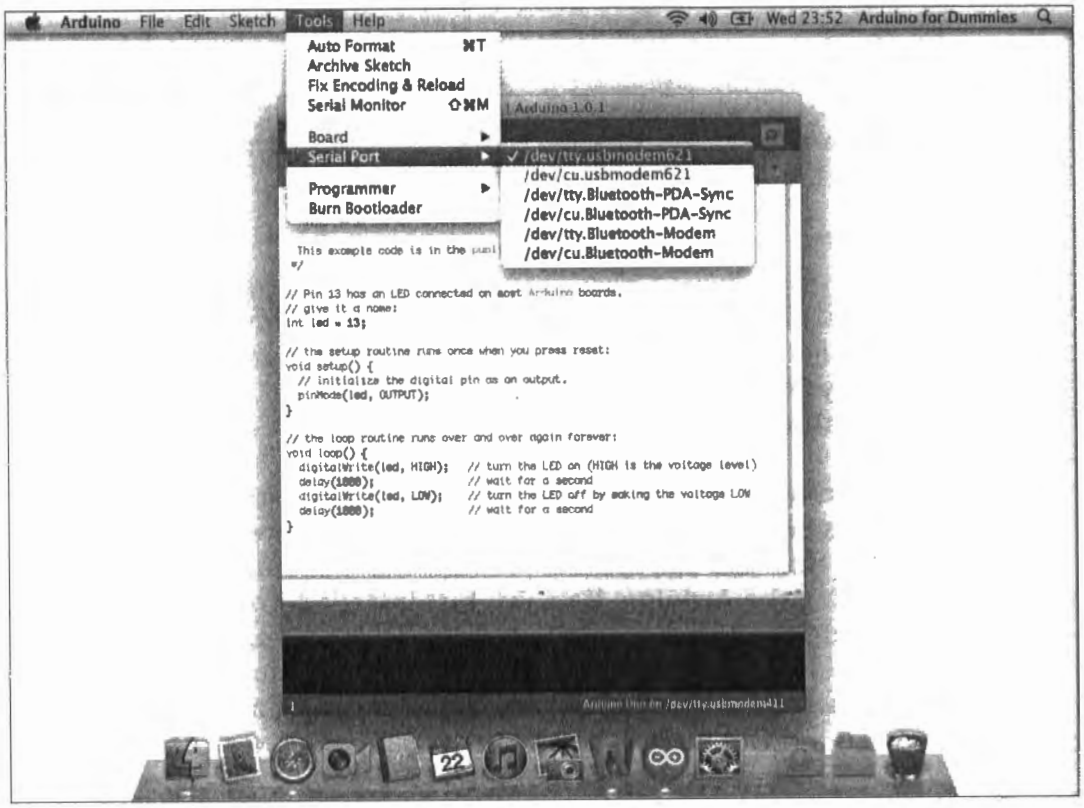

Figure 4-6: A list of serial connections available to the Arduino environment.

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).

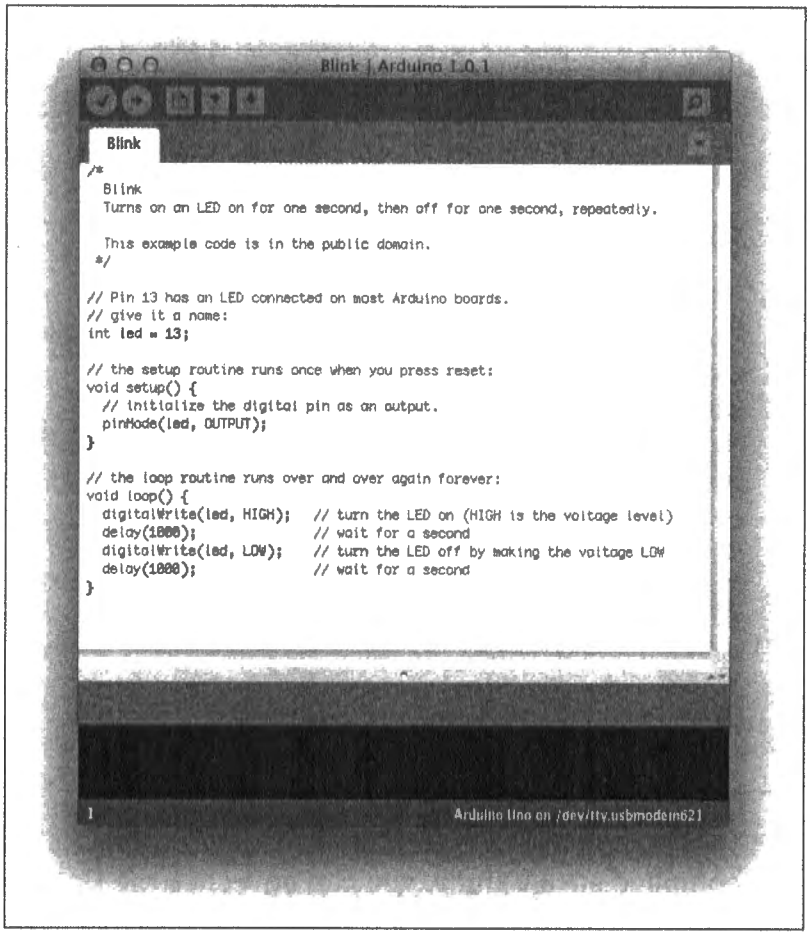

Figure 4-7: Arduino GUI board and port.

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.

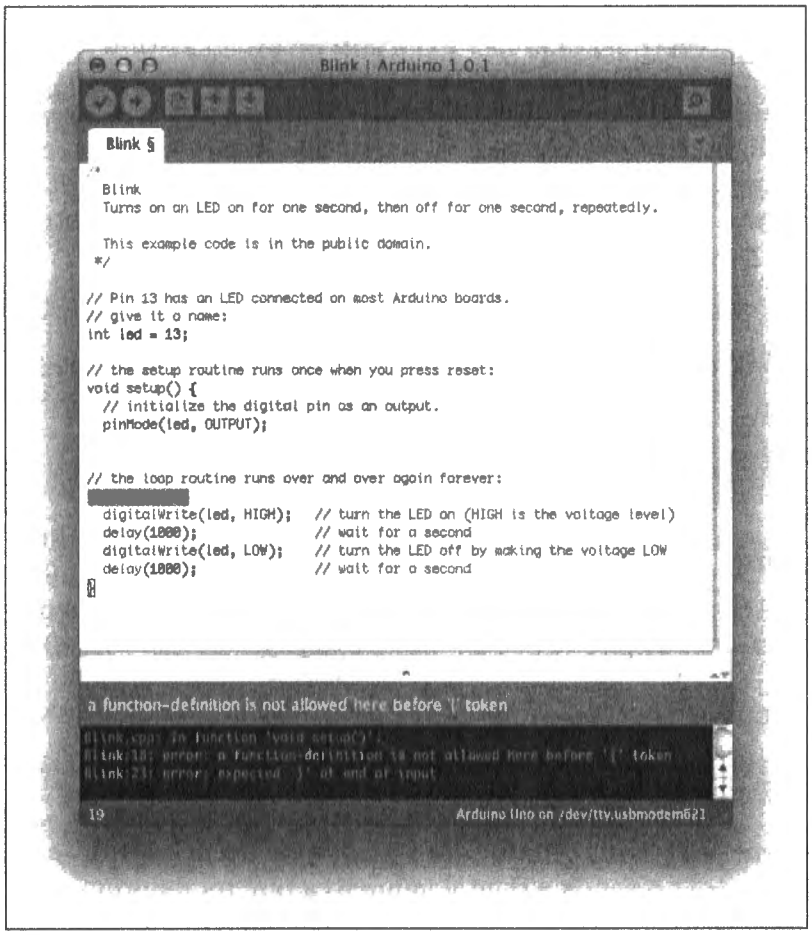

Figure 4-10: The Arduino software is telling you that a bracket is missing.

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.

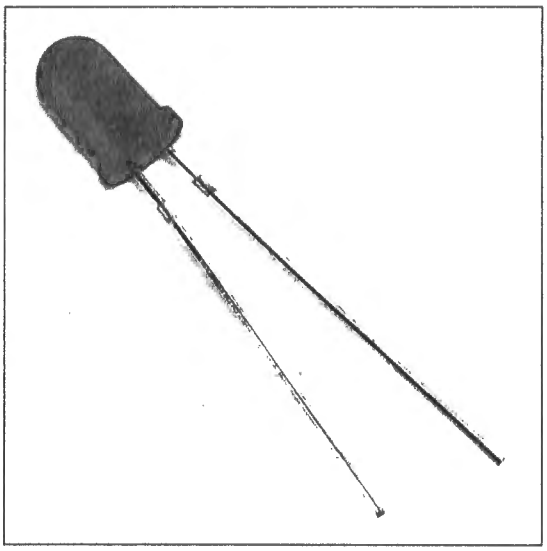

Figure 4-11: A lone LED, ready to be put to work.

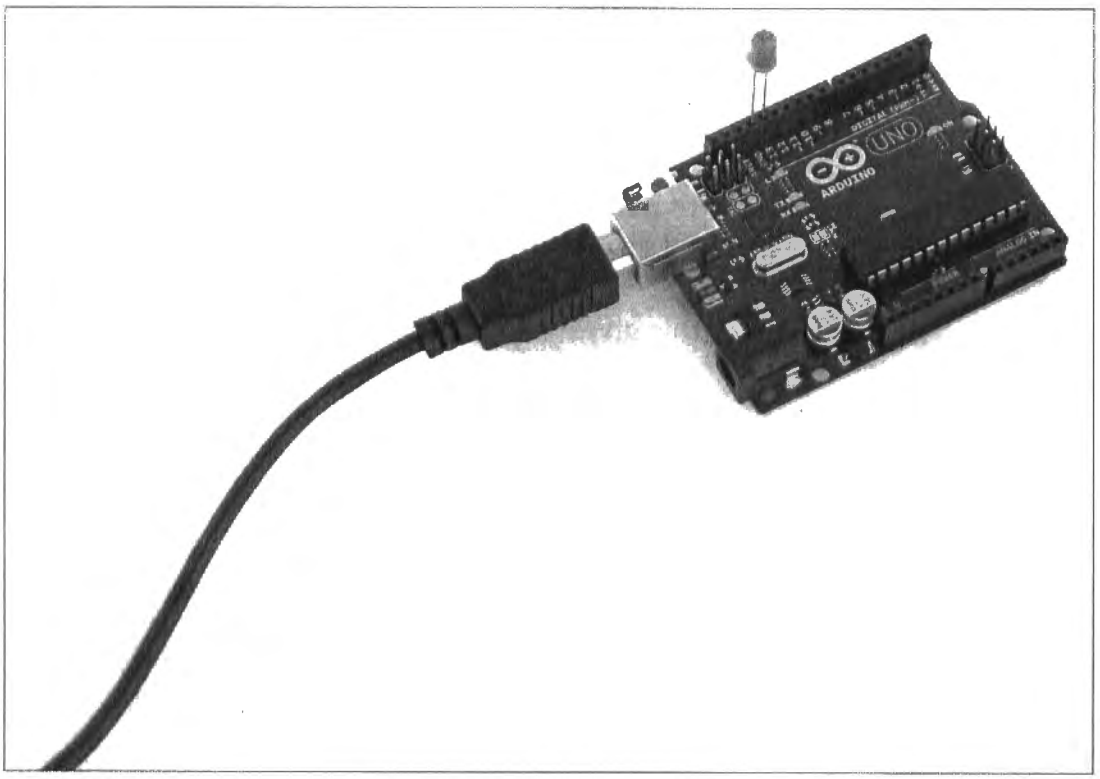

Figure 4-12: Arduino LED Pin 13.

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.