4/Really Getting Started with Arduino

Now you'll learn how to build and program an interactive device.

Anatomy of an Interactive Device

All of the objects we will build using Arduino follow a very simple pattern that we call the "Interactive Device". The Interactive Device is an electronic circuit that is able to sense the environment using sensors (electronic components that convert real-world measurements into electrical signals). The device processes the information it gets from the sensors with behaviour that's implemented as software. The device will then be able to interact with the world using actuators, electronic components that can convert an electric signal into a physical action.

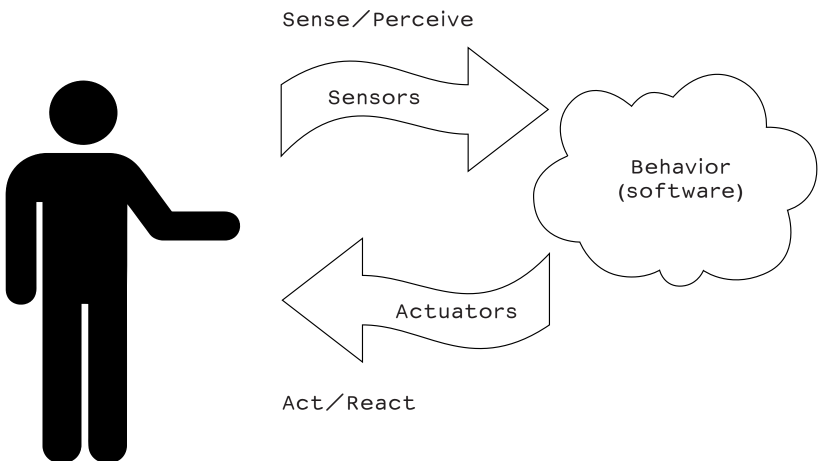

Figure 4-1.
The interactive device

Sensors and Actuators

Sensors and actuators are electronic components that allow a piece of electronics to interact with the world.

As the microcontroller is a very simple computer, it can process only electric signals (a bit like the electric pulses that are sent between neurons in our brains). For it to sense light, temperature, or other physical quantities, it needs something that can convert them into electricity. In our body, for example, the eye converts light into signals that get sent to the brain using nerves. In electronics, we can use a simple device called a light-dependent resistor (an LDR or photoresistor) that can measure the amount of light that hits it and report it as a signal that can be understood by the microcontroller.

Once the sensors have been read, the device has the information needed to decide how to react. The decision-making process is handled by the microcontroller, and the reaction is performed by actuators. In our bodies, for example, muscles receive electric signals from the brain and convert them into a movement. In the electronic world, these functions could be performed by a light or an electric motor.

In the following sections, you will learn how to read sensors of different types and control different kinds of actuators.

Blinking an LED

The LED blinking sketch is the first program that you should run to test whether your Arduino board is working and is configured correctly. It is also usually the very first programming exercise someone does when learning to program a microcontroller. A light-emitting diode (LED) is a small electronic component that's a bit like a light bulb, but is more efficient and requires lower voltages to operate.

Your Arduino board comes with an LED preinstalled. It's marked "L". You can also add your own LED—connect it as shown in Figure 4-2.

K indicates the cathode (negative), or shorter lead; A indicates the anode (positive), or longer lead.

Once the LED is connected, you need to tell Arduino what to do. This is done through code, that is, a list of instructions that we give the microcontroller to make it do what we want.

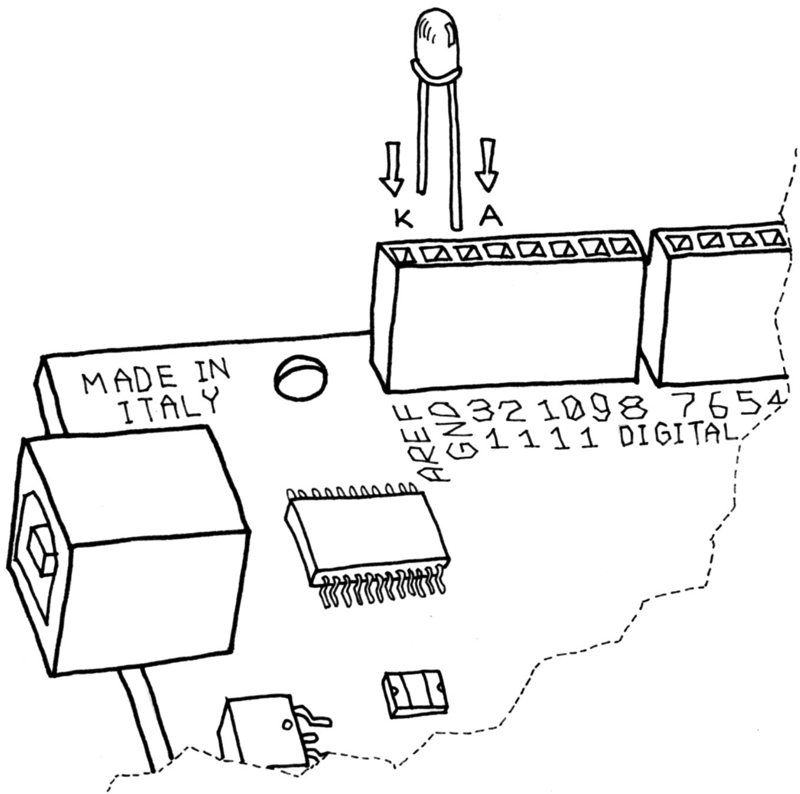

Figure 4-2.
Connecting an LED to Arduino

On your computer, go open the folder where you copied the Arduino IDE. Double-click the Arduino icon to start it. Select File > New and you'll be asked to choose a sketch folder name: this is where your Arduino sketch will be stored. Name it *Blinking_LED* and click OK. Then, type the following text (Example 01) into the Arduino sketch editor (the main window of the Arduino IDE). You can also download it from www.makezine.com/getstartedarduino. It should appear as shown in Figure 4-3.

```
// Example 01 : Blinking LED

#define LED 13   // LED connected to
                 // digital pin 13

void setup()
{
  pinMode(LED, OUTPUT);    // sets the digital
                           // pin as output
}

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

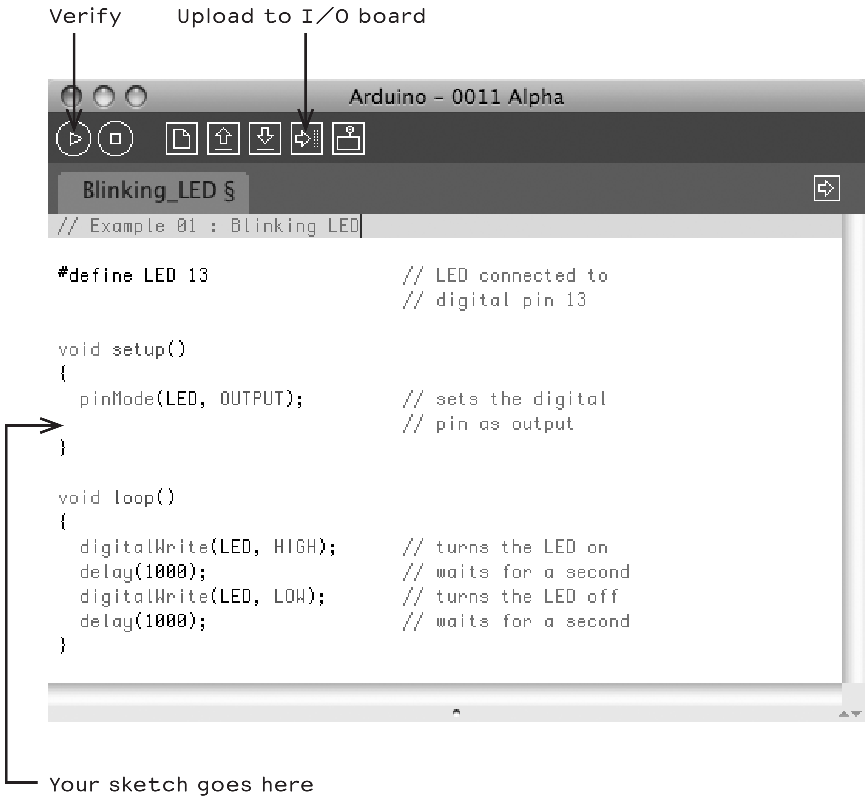

Figure 4-3.
The Arduino IDE with your first sketch loaded

Now that the code is in your IDE, you need to verify that it is correct. Press the "Verify" button (Figure 4-3 shows its location); if everything is correct, you'll see the message "Done compiling" appear at the bottom of the Arduino IDE. This message means that the Arduino IDE has translated your sketch into an executable program that can be run by the board, a bit like an .exe file in Windows or an .app file on a Mac.

At this point, you can upload it into the board: press the Upload to I/O Board button (see Figure 4-3). This will reset the board, forcing it to stop what it's doing and listen for instructions coming from the USB port. The Arduino IDE sends the current sketch to the board, which will store it in its memory and eventually run it.

You will see a few messages appear in the black area at the bottom of the window, and just above that area, you'll see the message "Done uploading" appear to let you know the process has completed correctly. There are two LEDs, marked RX and TX, on the board; these flash every time a byte is sent or received by the board. During the upload process, they keep flickering.

If you don't see the LEDs flicker, or if you get an error message instead of "Done uploading", then there is a communication problem between your computer and Arduino. Make sure you've selected the right serial port (see Chapter 3) in the Tools > Serial Port menu. Also, check the Tools > Board menu to confirm that the correct model of Arduino is selected there.

If you are still having problems, check Chapter 7, Troubleshooting.

Once the code is in your Arduino board, it will stay there until you put another sketch on it. The sketch will survive if the board is reset or turned off, a bit like the data on your computer's hard drive.

Assuming that the sketch has been uploaded correctly, you will see the LED "L" turn on for a second and then turn off for a second. If you installed a separate LED as shown back in Figure 4-2, that LED will blink, too. What you have just written and ran is a "computer program", or sketch, as Arduino programs are called. Arduino, as I've mentioned before, is a small computer, and it can be programmed to do what you want. This is done using a programming language to type a series of instructions in the Arduino IDE, which turns it into an executable for your Arduino board.

I'll next show you how to understand the sketch. First of all, the Arduino executes the code from top to bottom, so the first line at the top is the first one read; then it moves down, a bit like how the playhead of a video player like QuickTime Player or Windows Media Player moves from left to right showing where in the movie you are.

Pass Me the Parmesan

Notice the presence of curly brackets, which are used to group together lines of code. These are particularly useful when you want to give a name to a group of instructions. If you're at dinner and you ask somebody, "Please pass me the Parmesan cheese," this kicks off a series of actions that are summarised by the small phrase that you just said. As we're humans, it all comes naturally, but all the individual tiny actions required to do this must be spelled out to the Arduino, because it's not as powerful

as our brain. So to group together a number of instructions, you stick a *{* before your code and an *}* after.

You can see that there are two blocks of code that are defined in this way here. Before each one of them there is a strange command:

```
void setup()
```

This line gives a name to a block of code. If you were to write a list of instructions that teach Arduino how to pass the Parmesan, you would write *void passTheParmesan()* at the beginning of a block, and this block would become an instruction that you can call from anywhere in the Arduino code. These blocks are called **functions**. If after this, you write *passTheParmesan()* anywhere in your code, Arduino will execute those instructions and continue where it left off.

Arduino Is Not for Quitters

Arduino expects two functions to exists—one called *setup()* and one called *loop()*.

setup() is where you put all the code that you want to execute once at the beginning of your program and *loop()* contains the core of your program, which is executed over and over again. This is done because Arduino is not like your regular computer—it cannot run multiple programs at the same time and programs can't quit. When you power up the board, the code runs; when you want to stop, you just turn it off.

Real Tinkerers Write Comments

Any text beginning with *//* is ignored by Arduino. These lines are comments, which are notes that you leave in the program for yourself, so that you can remember what you did when you wrote it, or for somebody else, so that they can understand your code.

It is very common (I know this because I do it all the time) to write a piece of code, upload it onto the board, and say "Okay—I'm never going to have to touch this sucker again!" only to realise six months later that you need to update the code or fix a bug. At this point, you open up the program, and if you haven't included any comments in the original program, you'll think, "Wow—what a mess! Where do I start?" As we move along, you'll see some tricks for how to make your programs more readable and easier to maintain.

The Code, Step by Step

At first, you might consider this kind of explanation to unnecessary, a bit like when I was in school and I had to study Dante's *Divina Commedia* (every Italian student has to go through that, as well as another book called *I promessi sposi*, or *The Betrothed*—oh, the nightmares). For each line of the poems, there were a hundred lines of commentary! However, the explanation will be much more useful here as you move on to writing your own programs.

```
// Example 01 : Blinking LED
```

A comment is a useful way for us to write little notes. The preceding title comment just reminds us that this program, Example 01, blinks an LED.

```
#define LED 13  // LED connected to
                // digital pin 13
```

#define is like an automatic search and replace for your code; in this case, it's telling Arduino to write the number 13 every time the word *LED* appears. The replacement is the first thing done when you click Verify or Upload to I/O Board (you never see the results of the replacement as it's done behind the scenes). We are using this command to specify that the LED we're blinking is connected to the Arduino pin 13.

```
void setup()
```

This line tells Arduino that the next block of code will be called *setup()*.

```
{
```

With this opening curly bracket, a block of code begins.

```
pinMode(LED, OUTPUT); // sets the digital
                      // pin as output
```

Finally, a really interesting instruction. *pinMode* tells Arduino how to configure a certain pin. Digital pins can be used either as INPUT or OUTPUT. In this case, we need an output pin to control our LED, so we place the number of the pin and its mode inside the parentheses. *pinMode* is a function, and the words (or numbers) specified inside the parentheses are **arguments**. INPUT and OUTPUT are constants in the Arduino language. (Like variables, constants are assigned values, except that constant values are predefined and never change.)

```
}
```

This closing curly bracket signifies the end of the *setup()* function.

```
void loop()
{
```
loop() is where you specify the main behaviour of your interactive device. It will be repeated over and over again until you switch the board off.

```
digitalWrite(LED, HIGH);   // turns the LED on
```
As the comment says, *digitalWrite()* is able to turn on (or off) any pin that has been configured as an OUTPUT. The first argument (in this case, *LED*) specifies which pin should be turned on or off (remember that *LED* is a constant value that refers to pin 13, so this is the pin that's switched). The second argument can turn the pin on (HIGH) or off (LOW).

Imagine that every output pin is a tiny power socket, like the ones you have on the walls of your apartment. European ones are 230 V, American ones are 110 V, and Arduino works at a more modest 5 V. The magic here is when software becomes hardware. When you write *digitalWrite(LED, HIGH)*, it turns the output pin to 5 V, and if you connect an LED, it will light up. So at this point in your code, an instruction in software makes something happen in the physical world by controlling the flow of electricity to the pin. Turning on and off the pin at will now let us translate these into something more visible for a human being; the LED is our actuator.

```
delay(1000);      // waits for a second
```
Arduino has a very basic structure. Therefore, if you want things to happen with a certain regularity, you tell it to sit quietly and do nothing until it is time to go to the next step. *delay()* basically makes the processor sit there and do nothing for the amount of milliseconds that you pass as an argument. Milliseconds are thousands of seconds; therefore, 1000 milliseconds equals 1 second. So the LED stays on for one second here.

```
digitalWrite(LED, LOW);     // turns the LED off
```
This instruction now turns off the LED that we previously turned on. Why do we use HIGH and LOW? Well, it's an old convention in digital electronics. HIGH means that the pin is on, and in the case of Arduino, it will be set at 5 V. LOW means 0 V. You can also replace these arguments mentally with ON and OFF.

```
delay(1000); // waits for a second
```
Here, we delay for another second. The LED will be off for one second.

```
}
```
This closing curly bracket marks end of the loop function.

To sum up, this program does this:

- Turns pin 13 into an output (just once at the beginning)
- Enters a loop
- Switches on the LED connected to pin 13
- Waits for a second
- Switches off the LED connected to pin 13
- Waits for a second
- Goes back to beginning of the loop

I hope that wasn't too painful. You'll learn more about how to program as you go through the later examples.

Before we move on to the next section, I want you to play with the code. For example, reduce the amount of delay, using different numbers for the on and off pulses so that you can see different blinking patterns. In particular, you should see what happens when you make the delays very small, but use different delays for on and off . . . there is a moment when something strange happens; this "something" will be very useful when you learn about pulse-width modulation later in this book.

What We Will Be Building

I have always been fascinated by light and the ability to control different light sources through technology. I have been lucky enough to work on some interesting projects that involve controlling light and making it interact with people. Arduino is really good at this. Throughout this book, we will be working on how to design "interactive lamps", using Arduino as a way to learn the basics of how interactive devices are built.

In the next section, I'll try to explain the basics of electricity in a way that would bore an engineer, but won't scare a new Arduino programmer.

What Is Electricity?

If you have done any plumbing at home, electronics won't be a problem for you to understand. To understand how electricity and electric circuits work, the best way is to use something called the "water analogy". Let's take a simple device, like the battery-powered portable fan shown in Figure 4-4.

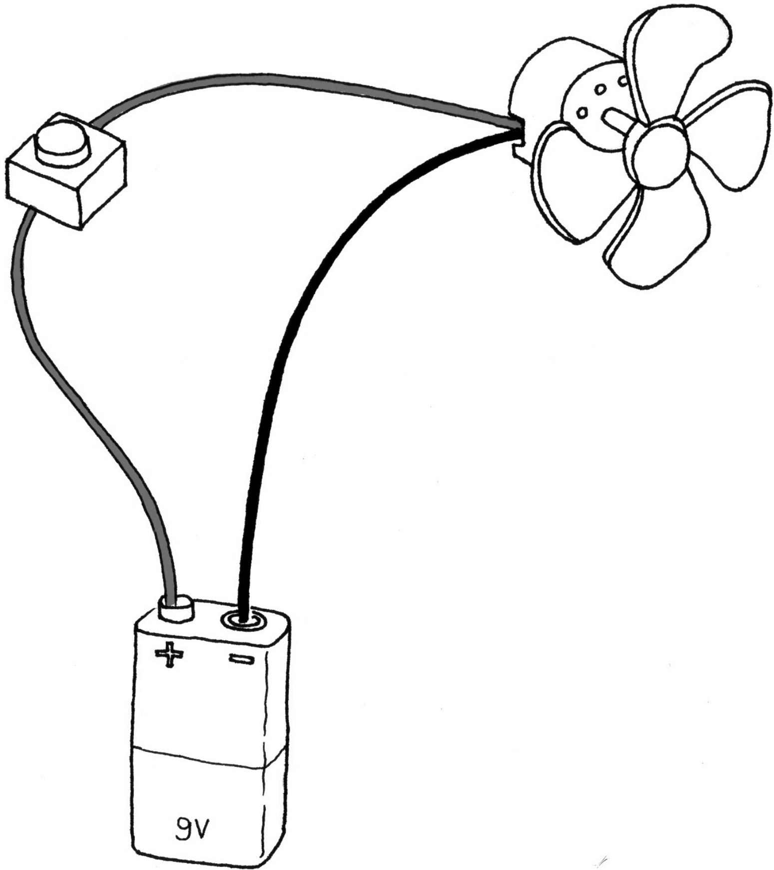

Figure 4-4.
A portable fan

If you take a fan apart, you will see that it contains a small battery, a couple of wires, and an electric motor, and that one of the wires going to the motor is interrupted by a switch. If you have a fresh battery in it and turn the switch on, the motor will start to spin, providing the necessary

chill. How does this work? Well, imagine that the battery is both a water reservoir and a pump, the switch is a tap, and the motor is one of those wheels that you see in watermills. When you open the tap, water flows from the pump and pushes the wheel into motion.

In this simple hydraulic system, shown in Figure 4-5, two factors are important: the pressure of the water (this is determined by the power of pump) and the amount of water that will flow in the pipes (this depends on the size of the pipes and the resistance that the wheel will provide to the stream of water hitting it).

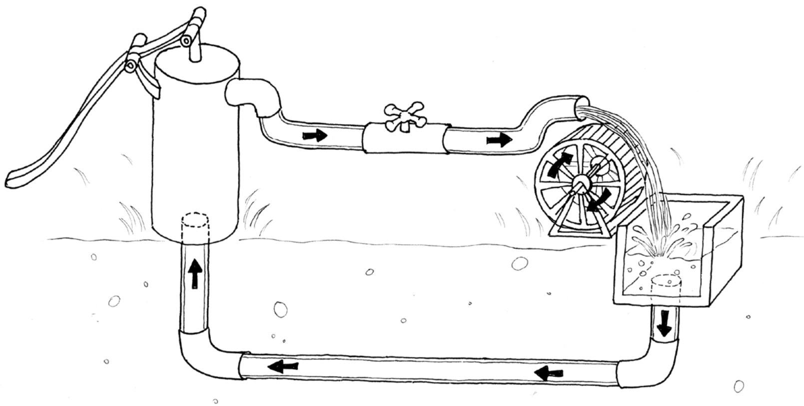

Figure 4-5.
A hydraulic system

You'll quickly realise that if you want the wheel to spin faster, you need to increase the size of the pipes (but this works only up to a point) and increase the pressure that the pump can achieve. Increasing the size of the pipes allows a greater flow of water to go through them; by making them bigger, we have effectively reduced the pipes' resistance to the flow of water. This approach works up to a certain point, at which the wheel won't spin any faster, because the pressure of the water is not strong enough. When we reach this point, we need the pump to be stronger. This method of speeding up the watermill can go on until the point when the wheel falls apart because the water flow is too strong for it and it is destroyed. Another thing you will notice is that as the wheel spins, the axle will heat up a little bit, because no matter how well we have mounted the wheel,

the friction between the axle and the holes in which it is mounted in will generate heat. It is important to understand that in a system like this, not all the energy you pump into the system will be converted into movement; some will be lost in a number of inefficiencies and will generally show up as heat emanating from some parts of the system.

So what are the important parts of the system? The pressure produced by the pump is one; the resistance that the pipes and wheel offer to the flow of water, and the actual flow of water (let's say that this is represented by the number of litres of water that flow in one second) are the others. Electricity works a bit like water. You have a kind of pump (any source of electricity, like a battery or a wall plug) that pushes electric charges (imagine them as "drops" of electricity) down pipes, which are represented by the wires—some devices are able to use these to produce heat (your grandma's thermal blanket), light (your bedroom lamp), sound (your stereo), movement (your fan), and much more.

So when you read that a battery's voltage is 9 V, think of this voltage like the water pressure that can potentially be produced by this little "pump". Voltage is measured in volts, named after Alessandro Volta, the inventor of the first battery.

Just as water pressure has an electric equivalent, the flow rate of water does, too. This is called current, and is measured in amperes (after André-Marie Ampère, electromagnetism pioneer). The relationship between voltage and current can be illustrated by returning to the water wheel: a higher voltage (pressure) lets you spin a wheel faster; a higher flow rate (current) lets you spin a larger wheel.

Finally, the resistance opposing the flow of current over any path that it travels is called—you guessed it—resistance, and is measured in ohms (after the German physicist Georg Ohm). Herr Ohm was also responsible for formulating the most important law in electricity—and the only formula that you really need to remember. He was able to demonstrate that in a circuit the voltage, the current, and the resistance are all related to each other, and in particular that the resistance of a circuit determines the amount of current that will flow through it, given a certain voltage supply.

It's very intuitive, if you think about it. Take a 9 V battery and plug it into a simple circuit. While measuring current, you will find that the more resistors you add to the circuit, the less current will travel through it. Going back to the analogy of water flowing in pipes, given a certain pump, if I install a valve (which we can relate to a **variable resistor** in electricity), the more

I close the valve—increasing resistance to water flow—the less water will flow through the pipes. Ohm summarised his law in these formulae:

```
R (resistance) = V (voltage) / I (current)
V = R * I
I = V / R
```

This is the only rule that you really have to memorise and learn to use, because in most of your work, this is the only one that you will really need.

Using a Pushbutton to Control the LED

Blinking an LED was easy, but I don't think you would stay sane if your desk lamp were to continuously blink while you were trying to read a book. Therefore, you need to learn how to control it. In our previous example, the LED was our actuator, and our Arduino was controlling it. What is missing to complete the picture is a sensor.

In this case, we're going to use the simplest form of sensor available: a pushbutton.

If you were to take apart a pushbutton, you would see that it is a very simple device: two bits of metal kept apart by a spring, and a plastic cap that when pressed brings the two bits of metal into contact. When the bits of metal are apart, there is no circulation of current in the pushbutton (a bit like when a water valve is closed); when we press it, we make a connection.

To monitor the state of a switch, there's a new Arduino instruction that you're going to learn: the *digitalRead()* function.

digitalRead() checks to see whether there is any voltage applied to the pin that you specify between parentheses, and returns a value of HIGH or LOW, depending on its findings. The other instructions that we've used so far haven't returned any information—they just executed what we asked them to do. But that kind of function is a bit limited, because it will force us to stick with very predictable sequences of instructions, with no input from the outside world. With *digitalRead()*, we can "ask a question" of Arduino and receive an answer that can be stored in memory somewhere and used to make decisions immediately or later.

Build the circuit shown in Figure 4-6. To build this, you'll need to obtain some parts (these will come in handy as you work on other projects as well):

- Solderless breadboard: RadioShack (www.radioshack.com) part number 276-002, Maker Shed (www.makershed.com) part number MKKN3. Appendix A is an introduction to the solderless breadboard.

- Pre-cut jumper wire kit: RadioShack 276-173, Maker Shed MKKN4

- One 10K Ohm resistor: RadioShack 271-1335 (5-pack), SparkFun (www.sparkfun.com) COM-08374

- Momentary tactile pushbutton switch: SparkFun COM-00097

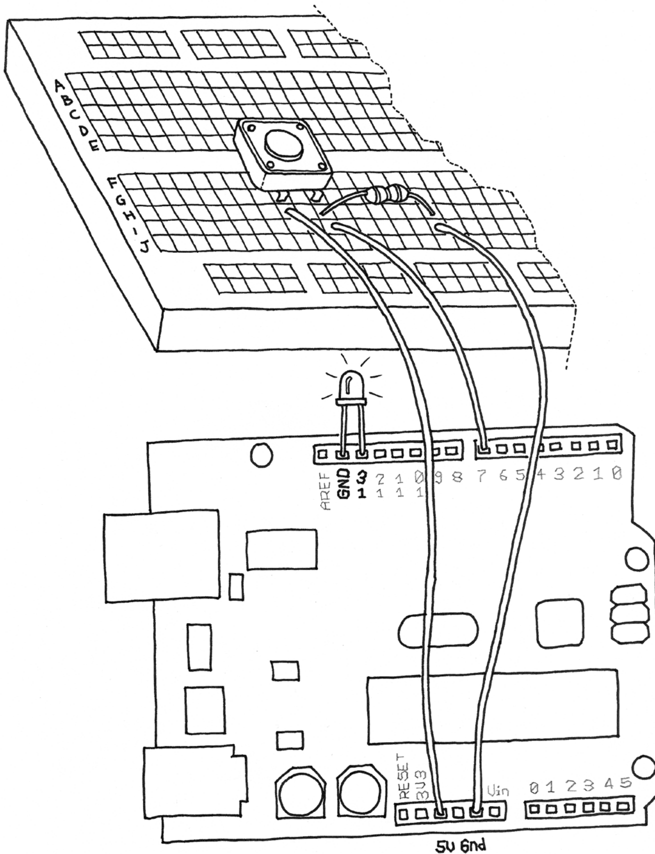

Figure 4-6.
Hooking up a pushbutton

NOTE: instead of buying precut jumper wire, you can also buy 22 AWG solid-core hookup wire in small spools and cut and strip it using wire cutters and wire strippers.

Let's have a look at the code that we'll be using to control the LED with our pushbutton:

```
// Example 02: Turn on LED while the button is pressed

#define LED 13   // the pin for the LED
#define BUTTON 7 // the input pin where the
                 // pushbutton is connected
int val = 0;     // val will be used to store the state
                 // of the input pin

void setup() {
  pinMode(LED, OUTPUT);   // tell Arduino LED is an output
  pinMode(BUTTON, INPUT); // and BUTTON is an input
}

void loop(){
  val = digitalRead(BUTTON); // read input value and store it

  // check whether the input is HIGH (button pressed)
  if (val == HIGH) {
    digitalWrite(LED, HIGH); // turn LED ON
  } else {
    digitalWrite(LED, LOW);
  }
}
```

In Arduino, select File > New (if you have another sketch open, you may want to save it first). When Arduino asks you to name your new sketch folder, type PushButtonControl. Type the Example 02 code into Arduino (or download it from www.makezine.com/getstartedarduino and paste it into the Arduino IDE). If everything is correct, the LED will light up when you press the button.

How Does This Work?

I have introduced two new concepts with this example program: functions that return the result of their work and the *if* statement.

The *if* statement is possibly the most important instruction in a programming language, because it allows the computer (and remember, the Arduino is a small computer) to make decisions. After the *if* keyword, you have to write a "question" inside parentheses, and if the "answer", or result, is true, the first block of code will be executed; otherwise, the block of code after else will be executed. Notice that I have used the == symbol instead of =. The former is used when two entities are compared, and returns TRUE or FALSE; the latter assigns a value to a variable. Make sure that you use the correct one, because it is very easy to make that mistake and use just =, in which case your program will never work. I know, because after 25 years of programming, I still make that mistake.

Holding your finger on the button for as long as you need light is not practical. Although it would make you think about how much energy you're wasting when you walk away from a lamp that you left on, we need to figure out how to make the on button "stick".

One Circuit, A Thousand Behaviours

The great advantage of digital, programmable electronics over classic electronics now becomes evident: I will show you how to implement many different "behaviours" using the same electronic circuit as in the previous section, just by changing the software.

As I've mentioned before, it's not very practical to have to hold your finger on the button to have the light on. We therefore must implement some form of "memory", in the form of a software mechanism that will remember when we have pressed the button and will keep the light on even after we have released it.

To do this, we're going to use what is called a **variable**. (We have used one already, but I haven't explained it.) A variable is a place in the Arduino memory where you can store data. Think of it like one of those sticky notes you use to remind yourself about something, such as a phone number: you take one, you write "Luisa 02 555 1212" on it, and you stick it to your computer monitor or your fridge. In the Arduino language, it's equally simple: you just decide what type of data you want to store (a number or some text, for example), give it a name, and when you want to, you can store the data or retrieve it. For example:

```
int val = 0;
```

int means that your variable will store an integer number, *val* is the name of the variable, and = *0* assigns it an initial value of zero.

A variable, as the name intimates, can be modified anywhere in your code, so that later on in your program, you could write:

```
val = 112;
```

which reassigns a new value, 112, to your variable.

NOTE: Have you noticed that in Arduino, every instruction, with one exception (*#define*), ends with a semicolon? This is done so that the compiler (the part of Arduino that turns your sketch into a program that the microcontroller can run) knows that your statement is finished and a new one is beginning. Remember to use it all the time, excluding any line that begins with *#define*. The *#defines* are replaced by the compiler before the code is translated into an Arduino executable.

In the following program, *val* is used to store the result of *digitalRead()*; whatever Arduino gets from the input ends up in the variable and will stay there until another line of code changes it. Notice that variables use a type of memory called RAM. It is quite fast, but when you turn off your board, all data stored in RAM is lost (which means that each variable is reset to its initial value when the board is powered up again). Your programs themselves are stored in flash memory—this is the same type used by your mobile phone to store phone numbers—which retains its content even when the board is off.

Let's now use another variable to remember whether the LED has to stay on or off after we release the button. Example 03A is a first attempt at achieving that:

```
// Example 03A: Turn on LED when the button is pressed
// and keep it on after it is released


#define LED  13  // the pin for the LED
#define BUTTON 7 // the input pin where the
                 // pushbutton is connected
int val = 0;     // val will be used to store the state
                 // of the input pin
int state = 0;   // 0 = LED off while 1 = LED on

void setup() {
  pinMode(LED, OUTPUT);   // tell Arduino LED is an output
  pinMode(BUTTON, INPUT); // and BUTTON is an input
}


void loop() {
  val = digitalRead(BUTTON); // read input value and store it

  // check if the input is HIGH (button pressed)
  // and change the state
  if (val == HIGH) {
    state = 1 - state;
  }


  if (state == 1) {
    digitalWrite(LED, HIGH); // turn LED ON
  } else {
    digitalWrite(LED, LOW);
  }
}
```

Now go test this code. You will notice that it works . . . somewhat. You'll find that the light changes so rapidly that you can't reliably set it on or off with a button press.

Let's look at the interesting parts of the code: *state* is a variable that stores either 0 or 1 to remember whether the LED is on or off. After the button is released, we initialise it to 0 (LED off).

Later, we read the current state of the button, and if it's pressed (*val == HIGH*), we change state from 0 to 1, or vice versa. We do this using a small trick, as state can be only either 1 or 0. The trick I use involves a small mathematical expression based on the idea that 1 – 0 is 1 and 1 – 1 is 0:

```
state = 1 – state;
```

The line may not make much sense in mathematics, but it does in programming. The symbol = means "assign the result of what's after me to the variable name before me"—in this case, the new value of state is assigned the value of 1 minus the old value of state.

Later in the program, you can see that we use *state* to figure out whether the LED has to be on or off. As I mentioned, this leads to somewhat flaky results.

The results are flaky because of the way we read the button. Arduino is really fast; it executes its own internal instructions at a rate of 16 million per second—it could well be executing a few million lines of code per second. So this means that while your finger is pressing the button, Arduino might be reading the button's position a few thousand times and changing *state* accordingly. So the results end up being unpredictable; it might be off when you wanted it on, or vice versa. As even a broken clock is right twice a day, the program might show the correct behaviour every once in a while, but much of the time it will be wrong.

How do we fix this? Well, we need to detect the exact moment when the button is pressed—that is the only moment that we have to change state. The way I like to do it is to store the value of *val* before I read a new one; this allows me to compare the current position of the button with the previous one and change state only when the button becomes HIGH after being LOW.

Example 03B contains the code to do so:

```
// Example 03B: Turn on LED when the button is pressed
// and keep it on after it is released
// Now with a new and improved formula!


#define LED  13   // the pin for the LED
#define BUTTON 7  // the input pin where the
                  // pushbutton is connected
int val = 0;      // val will be used to store the state
                  // of the input pin
int old_val = 0; // this variable stores the previous
                  // value of "val"
int state = 0;    // 0 = LED off and 1 = LED on

void setup() {
  pinMode(LED, OUTPUT);    // tell Arduino LED is an output
  pinMode(BUTTON, INPUT);  // and BUTTON is an input
}
void loop(){
  val = digitalRead(BUTTON); // read input value and store it
                             // yum, fresh

  // check if there was a transition
  if ((val == HIGH) && (old_val == LOW)){
    state = 1 - state;
  }

  old_val = val;  // val is now old, let's store it

  if (state == 1) {
    digitalWrite(LED, HIGH); // turn LED ON
  } else {
    digitalWrite(LED, LOW);
  }
}
```

Test it: we're almost there!

You may have noticed that this approach is not entirely perfect, due to another issue with mechanical switches. Pushbuttons are very simple devices: two bits of metal kept apart by a spring. When you press the

button, the two contacts come together and electricity can flow. This sounds fine and simple, but in real life the connection is not that perfect, especially when the button is not completely pressed, and it generates some spurious signals called **bouncing**.

When the pushbutton is bouncing, the Arduino sees a very rapid sequence of on and off signals. There are many techniques developed to do de-bouncing, but in this simple piece of code I've noticed that it's usually enough to add a 10- to 50-millisecond delay when the code detects a transition.

Example 03C is the final code:

```
// Example 03C: Turn on LED when the button is pressed
// and keep it on after it is released
// including simple de-bouncing
// Now with another new and improved formula!!

#define LED  13   // the pin for the LED
#define BUTTON 7  // the input pin where the
                  // pushbutton is connected
int val = 0;      // val will be used to store the state
                  // of the input pin
int old_val = 0; // this variable stores the previous
                  // value of "val"
int state = 0;    // 0 = LED off and 1 = LED on

void setup() {
  pinMode(LED, OUTPUT);   // tell Arduino LED is an output
  pinMode(BUTTON, INPUT); // and BUTTON is an input
}

void loop(){
  val = digitalRead(BUTTON); // read input value and store it
                             // yum, fresh

  // check if there was a transition
  if ((val == HIGH) && (old_val == LOW)){
    state = 1 - state;
    delay(10);
  }

  old_val = val; // val is now old, let's store it

  if (state == 1) {
    digitalWrite(LED, HIGH); // turn LED ON
  } else {
    digitalWrite(LED, LOW);
  }
}
```