Choose File→Examples→03.Analog→AnalogInOutSerial from the Arduino menu to load the sketch.

```
/*
   Analog input, analog output, serial output

   Reads an analog input pin, maps the result to a range from 0 to 255 and uses the result to set the pulsewidth modulation (PWM) of an output pin. Also prints the results to the serial monitor.

   The circuit:
   * potentiometer connected to analog pin 0.
     Center pin of the potentiometer goes to the analog pin.
     Side pins of the potentiometer go to +5V and ground
   * LED connected from digital pin 9 to ground

   created 29 Dec. 2008
   modified 9 Apr 2012
   by Tom Igoe

   This example code is in the public domain.
*/

// These constants won't change. They're used to give names
// to the pins used:
const int analogInPin = A0;  // Analog input pin that the potentiometer is attached to
const int analogOutPin = 9; // Analog output pin that the LED is attached to

int sensorValue = 0;       // value read from the pot
int outputValue = 0;       // value output to the PWM (analog out)

void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // read the analog in value:
  sensorValue = analogRead(analogInPin);
  // map it to the range of the analog out:
  outputValue = map(sensorValue, 0, 1023, 0, 255);
  // change the analog out value:
  analogWrite(analogOutPin, outputValue);

  // print the results to the serial monitor:
  Serial.print("sensor = ");
  Serial.print(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);
  delay(2000);
}
```

Press the Compile button to check your code. The compiler highlights any grammatical errors, which light up in red when they are discovered. If the sketch compiles correctly, click Upload to send the sketch to your board. When it is done uploading, mount your laser so that it hits the center of the light sensor. On the serial monitor, you should see analog values at the high end of the range (1024 max). When you obstruct the beam, that range should drop, and the change should be shown by the LED as well. Experiment with the values in the map function to determine the best range of values.

When the value drops below a certain threshold, you can trigger any of a variety of actions — you have a highly sensitive trip wire sensor.

If nothing happens, double-check your wiring:

✔ Make sure that you’re using the correct pin number.
✔ Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they do not work.

**Understanding the AnalogInOutSerial sketch**

For more details on the workings of this sketch, see the notes in AnalogInOutSerial in Chapter 7. You can also find suggestions for different sketches to provide smoothing and calibration in Chapter 11.

**Detecting Movement**

A Passive Infrared (PIR) sensor is a common sensor in some homes and most commercial buildings. You may have seen this sensor in the corner of a room, blinking red every once in a while. It registers heat given off by people, animals, or other heat sources as infrared radiation. Infrared radiation is invisible to the human eye but is easy for the sensor to distinguish. The sensor itself is similar to the sensor found in a digital camera, but without the complex lenses to capture a detailed picture. Essentially, a PIR sensor is somewhere between a high-resolution light sensor and a low-resolution camera. Simple lenses are usually fitted to PIR sensors to give them a wider viewing angle.
Most commonly, this type of sensor is used for motion detection in burglar alarms. Rather than detect motion, it actually detects changes in temperature. Temperature changes can trigger an alarm system or something more interesting (henchmen!), but the sensor is purely a way of monitoring changes in an environment.

You have two ways to get a PIR sensor. The first is to take apart a PIR burglar alarm, which is likely pre-packaged with a lens and a sensor, which may be difficult to identify. The second method is to buy one specifically intended for microcontroller projects. This most often comes with a basic, ping-pong-ball-styled lens and a bare circuit board underneath. The latter is easier to work with because all the details are known and it is detailed later in this section.

Consider the following during planning:

✔ **Complexity:** It can be tricky to hack an existing PIR sensor made for a specific system. Because it needs to communicate with that system, however, the sensor usually has clearly marked connections on the back. One of the benefits of using an existing sensor is that it is prepackaged, which reduces the amount of time you have to spend putting components together. Prepackaged systems are designed to be easy to install, so you may also be able to use manual calibration, by way of a potentiometer or screwdriver slot, which can be a huge benefit for on-the-fly calibration rather than having to re-upload.

If you’re using a PIR sensor that is not prepackaged, it should be a lot more straightforward on the hardware and software side but requires some careful thought concerning the housing. Some PIR sensors have their own on-board logic and operate like a switch, going **HIGH** when movement occurs over the threshold. This kind of sensor needs calibration to identify change from the norm.

✔ **Cost:** A household PIR sensor costs between $15 and $45 (£10 and £30). The main expense is the housing, usually designed to be discreet or look suitably high-tech. Bare PIR sensors cost a fraction of the price at around $10 (£6.50), but need a suitable housing to be of any real use.

✔ **Where:** Many housings allow you to neatly fit the sensor against a wall, or you might consider using mini tripods for direction, as suggested in the previous section of this chapter. Some of the tripod mounts also come with a suction-cup mount, which is perfect for fixing your sensor to glossy surfaces such as glass.

Most PIR sensors come ready for action, needing only power. They calibrate themselves based on what they can see and then send a **HIGH** or **LOW** value when they detect change. This makes them extremely easy to program because you are dealing with the same signals as with a pushbutton.
Implementing the DigitalReadSerial sketch

In this example, you learn how to use the SE-10, a PIR sensor available from all the major Arduino retailers. This particular PIR sensor has three wires: red, brown, and black. The red wire is the power source and should be connected to 5V. Oddly, the black wire is the signal wire and not the ground (see Figure 12-10; the black is the leftmost wire, brown is in the middle, and red is on the right). Brown should be wired to ground and black to pin 2.

The signal pin is known as an open-collector and needs to be pulled HIGH to start with. To do so, you use a 10k resistor to connect it to 5V as well. The pin, therefore, reads HIGH when no motion occurs and is pulled to LOW when there is motion.

You need:

- An Arduino Uno
- A breadboard
- An SE-10 PIR Motion Sensor
- A 10k ohm resistor
- Jump wires
Lay out the circuit as in the layout and circuit diagrams shown in Figures 12-11 and 12-12.

Figure 12-11: A PIR sensor circuit layout.

Figure 12-12: A PIR sensor circuit diagram.
Complete the circuit and choose File ➤ Examples ➤ 01.Basics ➤ DigitalReadSerial from the Arduino menu to load the sketch. This sketch is intended for a push-button but follows the same principles. If you want to make the sketch more specific, you can save it with a more appropriate name and variable names.

```cpp
/*
DigitalReadSerial
Reads a digital input on pin 2, prints the result to the serial monitor
This example code is in the public domain.
*/

// digital pin 2 has a pushbutton attached to it. Give it a name:
int pushButton = 2;

// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // read the input pin:
  int buttonState = digitalRead(pushButton);
  // print out the state of the button:
  Serial.println(buttonState);
  delay(1);        // delay in between reads for stability
}
```

Press the Compile button to check your code. Doing so highlights any grammatical errors and turns them red when they are discovered. If the sketch compiles correctly, click Upload to send the sketch to your board. When it is done uploading, affix the PIR sensor to a surface that is free of movement and open the serial monitor. Opening the serial monitor resets the sketch and the sensor calibrates itself in the first 1 to 2 seconds. When movement is detected, you should see the `buttonState` value change from 1 (no movement) to 0 (movement).

If nothing happens, double-check your wiring:

- Make sure that you’re using the correct pin number.
- 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.
- Try restarting the PIR sensor by disconnecting and reconnecting the GND wire, and be sure that it does not move during or after calibration.
Understanding the `DigitalReadSerial` sketch

The only variable to declare is pin 2, the `pushButton` pin, or in this case the `pirSensor` pin.

```cpp
// digital pin 2 has a pushbutton attached to it. Give it a name:
int pushButton = 2;
```

In setup, the serial port is opened and set to a baud rate of 9600; the input pin is set to an output.

```cpp
// the setup routine runs once when you press reset:
void setup() {
  // initialize serial communication at 9600 bits per second:
  Serial.begin(9600);
  // make the pushbutton's pin an input:
  pinMode(pushButton, INPUT);
}
```

In the `loop()` function, the input pin is read and its value stored in `buttonState`. This value reads `HIGH` when no movement is occurring because the pull-up resistor is providing a voltage from the 5V pin. When there is movement, the open-collector grounds the voltage and reads `LOW`.

```cpp
// the loop routine runs over and over again forever:
void loop() {
  // read the input pin:
  int buttonState = digitalRead(pushButton);

  The value of the input is then printed to the serial monitor.

```cpp
  Serial.println(buttonState);
  delay(1); // delay in between reads for stability
}
```

This is one example of how to use existing code for different hardware. From this point, it is possible to trigger different outputs based on the `HIGH` or `LOW` signal of the PIR sensor. For the ease of use and clarity of other people using your code, you should rename variables to more appropriate names, add your own comments, and save the sketch so that you can easily distinguish it.
Measuring Distance

Two sensors for measuring distance are extremely popular: the infrared proximity sensor and the ultrasonic range finder. They work in similar ways and achieve pretty much the same thing, but it’s important to pick the right sensor for the environment you’re in. An infrared proximity sensor has a light source and a sensor. The light source bounces infrared light off objects and back to the sensor, and the time it takes the light to return is measured to indicate how far away an object is.

An ultrasonic range finder fires out high frequency sound waves and listens for an echo when they hit a solid surface. By measuring the time that it takes a signal to bounce back, the ultrasonic range finder can determine the distance travelled.

Infrared proximity sensors are not as accurate and have a much shorter range than ultrasonic range finders.

Consider the following during planning:

- **Complexity:** Both of these sensors are designed to be extremely easy to integrate with Arduino projects. In the real world, they’re used for similar electronics applications, such as proximity meters on the back of cars that beep as you approach the curb. Again, the main complexity is housing them effectively. Infrared proximity sensors such as those made by Shape have useful screw holes on the outside of the body of the sensor. Maxbotix makes ultrasonic range finders that do not have these mounts, but their cylindrical shape makes them simple to mount in a surface by drilling a hole through.

- **Cost:** Infrared proximity sensors cost in the region of $15 (£10) and have a range up to about 59 inches (150 cm) or less. Ultrasonic range finders have a far greater possible range and accuracy but an equally great price, costing between $27 (£18) for a sensor that can read up to 254 inches (645 cm) and $100 (£65) for a more weather-resistant model that can read up to 301 inches (765 cm).

- **Where:** A common application for these sensors is monitoring presence of a person or an object in a particular floor space, especially when a pressure pad would be too obvious or easy to avoid, or when a PIR sensor would measure too widely. Using a proximity sensor lets you know where someone is in a straight line from that sensor, making it a very useful tool.
IR proximity sensors are okay in dark environments but perform terribly in direct sunlight. The MaxBotix Ultrasonic Range Finder is one of my favorite and most reliable sensors. When using ultrasonic range finders, you can also choose how wide or narrow a beam you want. A large, teardrop-shaped sensor is perfect for detecting large objects moving in a general direction, whereas narrow beams are great for precision measurement.

**Implementing the MaxSonar sketch**

In this example, you learn how to measure precise distances using a MaxBotix LV-EZ0. The EZ0, EZ1, EZ2, EZ3 and EZ4 all work the same way, but each has a slightly narrower beam, so choose the appropriate one for your project.

The range finder needs some minor assembly. To use the range finder in your circuit, you either need to solder on header pins to use it on a breadboard, or solder on lengths of wire.

You have three ways to connect your range finder: using analog, pulse width, or serial communication. In this example, you learn how to measure the pulse width and convert that to distance. The analog output can be read straight into your analog input pins but provide less accurate results than pulse width. This example does not cover serial communication.

You need:

- An Arduino Uno
- An LV-EZ0 Ultrasonic Range Finder
- Jump wires

Complete the circuit from the layout and circuit diagrams in Figures 12-13 and 12-14. The connections for the range finder are clearly marked on the underside of the PCB. The 5V and GND connections provide power for the sensor and should be connected to the 5V and GND supplies on your Arduino. The PW connection is the pulse width signal that will be read by pin 7 on your Arduino. Make sure that your distance sensor is affixed to some sort of base pointed in the direction that you want to measure.
Figure 12-13: An LV-EZ0 circuit layout.

Figure 12-14: An LV-EZ0 circuit diagram.
You can find the MaxSonar code by Bruce Allen in the Arduino playground at www.arduino.cc/playground/Main/MaxSonar, along with some additional notes and functions. Create a new sketch, copy or type the code into it, and save it with a memorable name, such as myMaxSonar.

```cpp
// Feel free to use this code.
// Please be respectful by acknowledging the author in the code if you use or modify it.
// Author: Bruce Allen
// Date: 23/07/09
// Digital pin 7 for reading in the pulse width from the MaxSonar device.
// This variable is a constant because the pin will not change throughout execution of this code.
const int pwPin = 7;
// variables needed to store values
long pulse, inches, cm;

void setup() {
  // This opens up a serial connection to shoot the results back to the PC console
  Serial.begin(9600);
}

void loop() {

  pinMode(pwPin, INPUT);

  // Used to read in the pulse that is being sent by the MaxSonar device.
  // Pulse Width representation with a scale factor of 147 uS per Inch.
  pulse = pulseIn(pwPin, HIGH);
  // 147uS per inch
  inches = pulse/147;
  // change inches to centimetres
  cm = inches * 2.54;

  Serial.print(inches);
  Serial.print("in, ");
  Serial.print(cm);
  Serial.print("cm");
  Serial.println();
  delay(500);
}
```

Press the Compile button to check your code. The compiler highlights any grammatical errors, turning them red when they are discovered. If the sketch compiles correctly, click Upload to send the sketch to your board. When it is done uploading, open the serial monitor and you should see the distance measured in inches and centimeters. If the value is fluctuating, try using an object with a bigger surface.
This sketch allows you to accurately measure distance in a straight line. Test this with a tape measure and make adjustments to the code if you find discrepancies.

If nothing happens, double-check your wiring:

✓ Make sure that you’re using the correct pin number.
✓ 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 MaxSonar sketch

In the declarations, pin 7 is defined as the `pwPin`.

```cpp
//This variable is a constant because the pin will not change throughout execution of this code.
const int pwPin = 7;
```

`long` variables are used to store the pulse width and distances in inches and centimeters. Notice that you can declare all three in a list if they have no value.

```cpp
//variables needed to store values
long pulse, inches, cm;
```

In setup, the serial connection is opened to print the results.

```cpp
void setup() {
  //This opens up a serial connection to shoot the results back to the PC console
  Serial.begin(9600);
}
```

In the main `loop()`, the `pwPin` is set as an input. You can set the input in the `loop()` but can also move it to `setup()`.

```cpp
void loop() {
  pinMode(pwPin, INPUT);
}
```

You use the function `pulseIn` to return the length of time it takes the pulse to return to the sensor in microseconds, or μS.

```cpp
//Used to read in the pulse that is being sent by the MaxSonar device.
//Pulse Width representation with a scale factor of 147 μS per Inch.
pulse = pulseIn(pwPin, HIGH);
```
A pulse travels 1 inch every 147 μS, so you can calculate the number of inches from the time. From this information, a simple conversion outputs the distance in different units.

```c
//147uS per inch
inches = pulse/147;
//change inches to centimetres
cm = inches * 2.54;
```

The results are printed to the serial monitor, with a `Serial.println` at the end to start a new line in between each reading.

```c
Serial.print(inches);
Serial.print("in, ");
Serial.print(cm);
Serial.print("cm");
Serial.println();
```

A delay is added to slow down the readings for legibility, but you can remove the delay if responsiveness is more important.

```c
delay(500);
```

This provides you with an accurate distance reading that you can incorporate into your own projects. A simple way to make use of this is with an `if` statement. For example:

```c
if (cm < 50) {
// do something!
}
```

---

**Testing, Testing . . . Can Anybody Hear This?**

Sound is another way to detect presence, and the best way to do it is with an electret microphone. It’s common to think of sounds in their analog form as recognizable noises, but a lot of the sounds we hear every day have undergone or will undergo an analog to digital conversion. By converting sound into a digital signal, it’s possible to interpret it on a computer or an Arduino. An electret microphone is similar to the microphones found in computer headsets and is extremely sensitive, but needs an amplifier in order for the Arduino to register the readings.
Consider the following during your planning:

✓ **Complexity:** There are plenty of electret mics to choose from, but by far the easiest is Sparkfun’s Breakout Board for Electret Microphone. It comes preassembled with the mic mounted on a circuit board and with an amplifier, and can easily be wired up to an Arduino as an analog input. It is possible to use electret mics from other headsets or desktop microphones, but these need their own amplifier to be of any use. There is some work involved to make the correct housing for the mic to protect it from the environment or human contact, but this could simply be an enclosure with a hole.

✓ **Cost:** The microphone itself is extremely cheap at 90 cents (61p) from Sparkfun (or distributors of their products). The breakout board costs $7.50 (£5.10), which is not a huge expense for the amount of labor saved.

✓ **Where:** As an ambient sensor, the mic could be placed just about anywhere to map the noise levels of a room. If you’re listening for a specific noise like a door slamming, it may be best to place the microphone near that source to get a clear reading.

One unusual use for a mic that I have encountered is monitoring someone’s breath. As the mic measures the amplitude or volume of the sound, it’s perfect for this application. By placing the mic at the end of a tube it’s even possible to monitor the length and intensity of breaths as the air rushes past the mic.

Electret mics are great for measuring the amplitude or volume of noise. This can be a trigger for a variety of outputs.

**Implementing the AnalogInOutSerial sketch**

In this example, you monitor sound levels wherever you are using an electret mic. This simple sensor can be read as an analog input into your Arduino.

You need:

✓ An Arduino Uno
✓ A Breakout Board for Electret Microphone
✓ Jump wires

Complete the circuit from the layout and circuit diagrams in Figures 12-15 and 12-16 to connect the mic as your input and the LED as your output. The Electret Mic breakout board requires a small amount of soldering to use on a breadboard or connect to your Arduino. You can either solder on a set of three header pins or solder on a length of wire, depending on your situation.
Figure 12-15: An electret mic circuit layout.

Figure 12-16: An electret mic circuit diagram.
/*
   Analog input, analog output, serial output

Reads an analog input pin, maps the result to a range from 0 to 255
and uses the result to set the pulsewidth modulation (PWM) of an output pin.
Also prints the results to the serial monitor.

The circuit:
* potentiometer connected to analog pin 0.
   Center pin of the potentiometer goes to the analog pin.
   Side pins of the potentiometer go to +5V and ground
* LED connected from digital pin 9 to ground

created 29 Dec. 2008
modified 9 Apr 2012
by Tom Igoe

This example code is in the public domain.
*/

// These constants won't change. They're used to give names
// to the pins used:
const int analogInPin = A0;  // Analog input pin that the potentiometer is
                             // attached to
const int analogOutPin = 9; // Analog output pin that the LED is attached to

int sensorValue = 0;        // value read from the pot
int outputValue = 0;        // value output to the PWM (analog out)

void setup() {
  // initialize serial communications at 9600 bps:
  Serial.begin(9600);
}

void loop() {
  // read the analog in value:
  sensorValue = analogRead(analogInPin);
  // map it to the range of the analog out:
  outputValue = map(sensorValue, 0, 1023, 0, 255);
  // change the analog out value:
  analogWrite(analogOutPin, outputValue);

  // print the results to the serial monitor:
  Serial.print("sensor = ");
  Serial.println(sensorValue);
  Serial.print("\t output = ");
  Serial.println(outputValue);
}
Serial.println(outputValue);

// wait 2 milliseconds before the next loop
// for the analog-to-digital converter to settle
// after the last reading:
delay(2);
}

Press the Compile button to check your code. Doing so highlights any grammatical errors and turns them red when they are discovered. If the sketch compiles correctly, click Upload to send the sketch to your board. When it is done uploading, open the serial monitor to see analog values in the range of 0 to 1024.

If nothing happens, double-check your wiring:

☑ Make sure that you’re using the correct pin number.
☑ 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.

See what range of values you get from different noises in your environment and how sensitive or over sensitive the mic is. Another sketch to consider is the Smoothing sketch in Chapter 11. By using if statements, you can perform actions whenever the sound level crosses a threshold.

**Understanding the AnalogInOutSerial sketch**

For more details on the workings of this sketch, see the notes in AnalogInOutSerial in Chapter 7. You can also find suggestions for different sketches to provide smoothing and calibration in Chapter 11.
Part IV
Unlocking Your Arduino’s Potential

The 5th Wave
By Rich Tennant

“I’m kind of busy. I told my neighbor I’d fix his PS2 so he could play extra games on it.”
Part IV is all about how to do more with your Arduino project. You learn how to use the Arduino Mega 2560, which can control more outputs than a regular Uno. You also learn how additional hardware can allow your Uno board to do much more than you thought possible, with shift registers and PWM drivers. With this knowledge, you could animate huge numbers of LEDs to create your own version of the Las Vegas Strip (or the Blackpool Illuminations, for my British friends) and control an army of servo motors to create your very own robot walker to do your bidding!
Chapter 13
Becoming a Specialist with Shields and Libraries

In This Chapter
▶ Finding out about shields
▶ Looking at the range of shields available
▶ Understanding libraries

The further you progress in learning about Arduino, the more you want to do, and it’s natural to want to run before you can walk. The areas that interest you may be highly specialized in themselves and require a huge investment of time to understand. Perhaps the most important thing about Arduino is the Arduino community, which is where you can get help when you want go further.

The traditional viewpoint that is hammered into us in education is to protect our ideas for dear life. Thankfully, many people in the Arduino community have seen past that limitation and are kind enough to share their hard work. By sharing this knowledge, the Arduino community helps the hardware and software become available to more people, who find new and interesting uses for it. If these people in turn share their results, the community continues to grow and eventually makes even the most difficult projects achievable. In this chapter, you find out more about how powerful shared resources such as shields and libraries can be, even for beginners.

Looking at Shields
Shields are pieces of hardware that sit on top of your Arduino, often to give it a specific purpose. For example, you can use a shield to make it easier to connect and control motors or even to turn your Arduino into something as complex as a mobile phone. A shield may start out as an interesting bit of hardware that an enthusiast has been experimenting with and wants to share with the community. Or an enterprising individual (or company) may design a shield to make a specific application easier based on demand from the Arduino community.
Shields can be very simple or very complex. They are sold preassembled or as kits. Kits allow you more freedom to assemble the shield as you need it to be. Some kits require you to assemble the circuitry of the boards, although more complex shields may already be largely assembled, needing only header pins.

Shields allow you to use your Arduino for more than one purpose and to change that purpose easily. They neatly package the electronics for that circuit in the same footprint as an Arduino. They are stackable to combine different functionalities. But they all have to use the same pins on the Arduino, so if you stack shields, watch out for those that need to use the same pins. They always connect the GND pins, too, because any communication by your Arduino and another device needs a common GND.

**Considering combinations**

In theory, shields could be stacked on top of each other forever, but you should take some points into consideration before combining them:

- **Physical size**: Some shields just don’t fit on top of one another. Components that are higher than the header sockets may touch the underside of any board on top of it. This situation, which can cause short circuits if a connection is made that shouldn’t be, can seriously damage your boards.

- **Obstruction of inputs / outputs**: If an input or output is obstructed by another shield, it becomes redundant. For example, there’s no point having a Joystick Shield or an LCD Shield under another shield because no more than one can be used.

- **Power requirements**: Some hardware requires a lot of power. Although it is all right for shields to use the same power and ground pins, there is a limit to the amount of current that can flow through the other input/output (I/O) pins: 40mA per pin and 200mA max. between all I/O pins. Exceed this, and you run the risk of seriously damaging your board and any other attached shield. In most cases, you can easily remedy this problem by powering your Arduino and shields from an external power supply so that the current isn’t passed through the Arduino. Make sure to use a common GND if you’re communicating between a board using I2C, SPI, or serial.

- **Pins**: Some shields require the use of certain pins. It’s important to make sure that shields aren’t doubling up on the same pins. In the best case, the hardware will just be confused; in the worst case, you can send voltage to the wrong place and damage your board.

- **Software**: Some of these shields need specific libraries to work. There can be conflicts in libraries calling on the same functions, so make sure to read up on what’s required for your shield.
Interference with radio/WiFi/GPS/GSM: Wireless devices need space to work. Move antennas or aerials away from the board to get a clear signal. If an antenna is mounted on the board, it’s generally a bad idea to cover it. Always try to place wireless shields at the top of the stack.

Reviewing the field

To give you an idea of available shields, this section covers some of the most interesting and useful shields on the market shows you where to look for more information.

Note that all prices were current at the time this book was written and are liable to change, but I’ve included them to give you an idea of the cost. Links to products may also change, so always try searching for the product if the link is broken. Technical information is gathered from the manufacturers websites, but you should always check the details yourself to make sure that you are buying what you need. Boards are revised occasionally, so always keep an eye out for the latest versions.

Finally, a lot of feedback on all of these products is available online, so always read the comments and forums to get a good understanding of what you’re buying.

This range of shields covers a vast number of different uses and the huge potential of Arduino projects. For many projects, a shield is all you need, but a shield is also an excellent stepping stone for proving a concept before refining or miniaturizing your project.

Prices provided are from a range of distributors to show the approximate value of the items. If you’re a savvy shopper or looking to buy in bulk, you may also be able to reduce the cost further.

Proto Shield Kit Rev3

Made by: Arduino

Price: £8.95 from RS Components; $12.95 from Mouser

Pins used: None

The Proto Shield (shown in Figure 13-1) is a platform for building custom circuits on your Arduino. Many shields listed in this chapter add a specific function to your Arduino, but with a Proto Shield, you can decide how to use it. Take your existing breadboard layouts and solder them to the surface of the Proto Shield to make your project much more durable. Proto Shields also come in a larger size to match the Arduino Mega’s footprint. Another handy feature of these shields is the handy space to attach SMD parts, which can be difficult to do otherwise.
The Proto Shield is either sold fully assembled or as a kit that requires soldering.

You can find details about the shield on the Arduino product page (http://arduino.cc/en/Main/ArduinoProtoShield).

**Figure 13-1:** A fully assembled Proto Shield.

---

**ProtoScrew Shield**
Made by: WingShield Industries

Price: £9.94 from Proto-Pic; $14.95 from SparkFun

Pins used: None

The ProtoScrew Shield is extremely similar to the regular Proto Shield but has large screw terminals connected to the pins as well. This feature is great for applications that have lots of inputs that may need changing or swapping, or just for easier assembly and disassembly. Changing a piece of wire is much easier with screw terminals than with soldering, so bear this in mind when planning your next project.

The ProtoScrew Shield is sold as a kit and requires soldering.
You can find more details on the SparkFun products page (www.sparkfun.com/products/9729).

**Adafruit Wave Shield v1.1**
Made by: Adafruit

Price: £18.00 from Oomlout; $22.00 from Adafruit

Pins used: 13, 12, 11, 10, 5, 4, 3, 2 on the Uno R3

The Wave Shield (see Figure 13-2) is a relatively cheap kit that allows you to play sounds or music with your Arduino. The Wave Shield allows you to play .WAV files directly from an SD card, making it easy to upload and change the sound files from your computer. To use the shield, you need the WaveHC library available from the product page or Google Code (http://code.google.com/p/wavehc/).

The Wave Shield is sold as a kit and requires soldering. The SD card reader must use pins 13, 12, and 11 because they support high-speed serial peripheral interface (SPI), which is a protocol needed to transfer data quickly. Pin 10 is used to communicate with the SD card reader, and pins 5, 4, 3, and 2 are used to talk to the digital to analog converter (DAC), which converts a digital music signal into an analog voltage.

For more details, visit the product page on Adafruit’s website (www.adafruit.com/products/94), where you can find a thorough tutorial as well (www.ladyada.net/make/waveshield/).
**MP3 Player Shield**
Made by: SparkFun

Price: £23.95 from HobbyTronics; $39.95 from SparkFun

Pins used: 13, 12, 11, 9, 8, 7, 6, 5, 4, 3, 2 on the Uno R3

Turn your Arduino into an MP3 player with this easy-to-assemble shield! Not only can it decode MP3 files, it’s also capable of decoding Ogg Vorbis, AAC, WMA, and MIDI. The MP3 Shield also has a MicroSD card reader for ease of uploading files, and it has a 3.5mm mini jack that you can plug into most speaker systems.

The MP3 Player Shield (shown in Figure 13-3) is assembled, but requires minor soldering to attach the header pins or header sockets. The SD card reader uses pins 13, 12, and 11. You use pin 9 to talk with the SD card reader. Use pins 8, 7, 6, and 2 talk to the MP3 Audio Decoder VS1053B, and you use 4 and 3 for additional midi functionality.

---

**Figure 13-3:**
An MP3 shield kit.

For more details, visit the SparkFun products page ([www.sparkfun.com/products/10628](http://www.sparkfun.com/products/10628)). There is also a tutorial page to get you started ([www.sparkfun.com/tutorials/295](http://www.sparkfun.com/tutorials/295)), but this was written quite a while ago.
Chapter 13: Becoming a Specialist with Shields and Libraries

and is out of date. Thankfully, the comments beneath the tutorial address many of the issues with this guide, and one user has even written a library to make your life easier. This is a great example of the Arduino community supporting the products that already exist.

Always read the comments and forum entries on products and kits. These comments often contain a lot of detail on the ease (or lack of) of a specific product. This is also the place to voice your own problems. Just be sure that you’re not repeating something that’s solved further down the page; otherwise, you’ll be advised to read the manual!

**MIDI Shield**

Made by: SparkFun

Price: £13.90 from HobbyTronics; $19.95 from SparkFun

Pins used: Uses pins 7, 6, 4, 3, 2, A1, A0 on the Uno R3

MIDI is short for Musical Instrument Digital Interface, and it revolutionized the music industry in the 1980s. Despite being relatively old now, MIDI is still a good standard to connect instruments, computer, stage effects, and other hardware and so is still widely used. With the MIDI shield, you can interface with anything that can send or receive MIDI data and incorporate it into your Arduino project.

The MIDI Shield is sold as a kit and requires soldering.


**RGB LCD Shield w/16 x 2 character display**

Made by: Adafruit

Price: £15.36 from Adafruit; $24.95 from Adafruit

Pins used: Uses pins A4 and A5 on the Uno R3

This handy LCD (liquid crystal display) shield packages everything you need onto one board. LCDs are found in older mobile phones and Nintendo GameBoys (wow, that sounds old already). They use a film that sits over a solid-colored surface that is usually backlit. The pixels of this film can be turned on or off to make shapes, text, or graphics, and this is what you control with your Arduino. At the center of the shield is an RGB LCD display, so instead of being stuck with just one color, you can choose from any RGB
Part IV: Unlocking Your Arduino’s Potential

Color. The RGB backlight is controlled directly from your Arduino. The display is a 16 x 2 character display (no graphics), and you can write two rows of 16 characters. Depending on which display you choose, your display is either colored text on a dark background (negative) or dark text on a colored background (positive). You have a variety of LCD character displays with various backlighting and dimensions from which to choose, so be sure to shop around.

The RGB LCD Shield is sold as a kit and requires soldering. Instead of using nine pins or more, the LCD, backlight, and the buttons all use just two. By using the I2C to communicate with the shield, you can use use only an analog pin 4, which is the data (SDA) line, and analog pin 5, which is the clock (SCL) line. This protocol is used in many devices, so it is extremely useful to know about. For more details on I2C, check John Boxall’s excellent tutorial at http://tronixstuff.wordpress.com/2010/10/20/tutorial-arduino-and-the-i2c-bus/.

For more details, check out the Adafruit products page (http://adafruit.com/products/714) and tutorial (http://learn.adafruit.com/rgb-lcd-shield).

Shields are also available that use the same technology but don’t limit you to just letters and numbers. If you’re looking to display your own graphics, you might want to use the SparkFun Color LCD shield, which uses a Nokia 6100 screen, or the larger TFT Touch Shield.

**TFT Touch Shield**

Made by: Adafruit

Price: £56.74 from Proto-PIC; $59.00 from Adafruit

Pins used: 5, 6, 7, 8, 9, 10, 11, 12, 13, A0, A1, A2, A3

If an LCD display isn’t enough for you, try this TFT Touch Shield to add full color and touch input to your project. This display is a TFT LCD screen (a variation on a standard LCD screen that uses thin-film transistor (TFT) technology to improve the image quality) with a resolution of 240 x 320 pixels and 18-bit colors, giving you 262,144 different shades. The screen is also fitted with a resistive touch screen to register finger presses anywhere on the surface of the screen.

The TFT Touch Shield is sold fully assembled and requires no soldering, so it can simply be plugged on top of your Arduino. The Touch Shield needs a lot of pins to function and leaves you with only digital pins 2 and 3 and analog pins 4 and 5. Pin 12 is also available if you are not using the microSD reader.

Check out the products page at www.adafruit.com/products/376 and a full tutorial at http://learn.adafruit.com/2-8-tft-touch-shield. Adafruit has also very kindly written a complete library for the TFT to draw
pixels, shapes and text (https://github.com/adafruit/TFTLCD-Library) and a library for the touch screen that detects x, y, and z horizontal movement, vertical movement, and pressure (https://github.com/adafruit/Touch-Screen-Library).

**Joystick Shield**
Made by: SparkFun

Price: $8.94 from Proto-PIC; $12.95 from SparkFun

Pins used: 2, 3, 4, 5, 6, A0, A1

The Joystick Shield (shown in Figure 13-4) has all the functions of a modern game controller on a single Arduino-compatible board. Not only does it give you four pushbuttons to assign to various functions, it also has a hidden button in the control stick itself. With the ergonomic control stick, you can smoothly transition between x and y axes to perform movements with great accuracy.

The Joystick Shield is sold as a kit and requires soldering. The Joystick Shield uses only five digital pins and two analog pins, leaving many other Arduino pins free for other uses. It has five pushbuttons, using digital pins 2 to 6. The movement of the joystick is measured using two potentiometers: analog 0 maps the x or horizontal movement; analog 1 maps the y or vertical movement.
You can find more details on the SparkFun product page (www.sparkfun.com/products/9760). You can also see in-depth assembly tutorial (www.sparkfun.com/tutorials/161) and quick-start guide (www.sparkfun.com/tutorials/171).

**Gameduino**
Made by: James Bowman

Price: $41.99 from Cool Components; $52.95 from SparkFun

Pins used: 9, 11, 12, 13

The Atmel AVR processor in your Arduino is much more powerful than the 8-bit game consoles of the 1980s, so James Bowman decided to make his own shield to harness this power and created a game adapter for Arduino: the Gameduino (see Figure 13-5). With the Gameduino, you can output graphics to a monitor, projector, or any VGA-compatible display, using the VGA connector on the board. Output audio using the 3.5mm mini jack connector. You can pair this shield with the Joystick Shield described in the previous section to create an all-in-one console and controller.

![Figure 13-5: A Gameduino, which plays nicely with a Joystick Shield.](image)
The Gameduino is sold fully assembled and ready to use. The Gameduino is treated as an SPI (Serial Peripheral Interface) peripheral by the Arduino, and four pins are used for communication. For more details, on SPI peripherals go to (http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus). You can find Arduino-specific reading at http://arduino.cc/en/Reference/SPI. For reference, pin 9 is SEL or SS (slave select), 11 is MOSI (master output, slave input), 12 is MISO (master input, slave output), and 13 is SCK or SCLK: serial clock.

A wealth of resources is available on the Gameduino product page (http://excamera.com/sphinx/gameduino/). To get started, download the example sketches to see what’s possible from http://excamera.com/sphinx/gameduino/samples/. Many specifics of the Gameduino require a lot of reading to fully understand and should be considered advanced work. Good luck!

**Adafruit Motor/Stepper/Servo Shield Kit v1.0**

Made by: Adafruit

Price: £16.00 from Oomlout; $19.50 from Adafruit

Pins used: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Love motors? Want to try them all? Then this shield is the one for you. The Adafruit Motor/Stepper/Servo Shield is aptly named and allows you to run all those motors you love. You can connect up to two 5V hobby servos, two stepper motors, or four bidirectional DC motors. The screw terminals make attaching and switching motors as needed easy. When dealing with motors, it’s always important to make sure that you have enough current to drive them all, so a handy screw terminal is on the shield to allow you to power your motors independently of your Arduino.

The Adafruit Motor/Stepper/Servo Shield Kit is sold as a kit and requires soldering. If any DC or stepper motors are in use, pins 4, 7, 8 and 12 are needed to operate the chip that drives them all (74HC595). Pins 3, 5, 6, or 11 control the speed of each individual DC or stepper motor. Pins 9 and 10 control any servos that are connected. This leaves you with digital pins 2 and 13 as well as a full complement of analog input pins, which you can use as digital pins if needed.

You can find many details on the Adafruit product page (www.adafruit.com/products/81). A full, in-depth tutorial is on ladyada.net (www.ladyada.net/make/mshield/). Be aware of how much load is on the motor because the shield is designed to provide up to 600 mA per motor, with 1.2A peak current. If you’re approaching 1A, include a heat sink on the motor driver to dissipate the heat.
Also, the nice people at Adafruit provide an easy-to-use library for your motor project (www.ladyada.net/make/mshield/download.html). Happy motoring!

**Motor Shield R3**

Made by: Arduino

Price: £18.90 from RS Components; $36.25 from Arduino Store

Pins used: 3, 8, 9, 11, 12, 13, A0, A1

The Arduino Motor Shield is the official motor shield designed by the Arduino team. It can control a modest number of motors, either two DC motors or one unipolar stepper motor, but it can cope with high currents of up to 2A per channel, allowing some heavy lifting. It’s also compatible with the TinkerKit, which is ideal for situations in which wire stripping and soldering aren’t an option, or for quickly experimenting with using different sensors to control your motors.

The Arduino Motor Shield Rev3 is sold fully assembled and ready to use. The pins on the Arduino Motor Shield are divided into A and B channels. You can use each channel for two individual DC motors to provide four functions: direction, speed, brake and current sensing. Pins 12 and 13 control the direction of each channel, respectively; pins 3 and 11 control the speed of each motor using pwm; pins 9 and 8 are used to break the motor suddenly, rather than allowing it to slow down; and analog pins 0 and 1 can be used read the current draw of the motor on each channel, outputting 3.3V at the maximum rated current of 2A.

To power the motors, you need to use an external power supply that can be connected using the screw terminals on the shield. It’s also worth noting that headers on top of the board are for the TinkerKit modules and are not so easy to use without the correct connectors.


**LiPower Shield**

Made by: SparkFun

Price: £23.57 from Proto-PIC; $29.95 from SparkFun

Pins used: 3
If you want to make your Arduino project more mobile, batteries are the answer. Rather than use bulky AA or AAA battery packs, the LiPower Shield allows you to use rechargeable lithium batteries instead. Although lithium batteries are rated only as 3.7V, some clever hardware steps them up to 5V to make them sufficient to power your Arduino.

The LiPower Shield is assembled but requires minor soldering to attach the header pins or header sockets. Because the LiPower shield is there to provide power and not consume it, only one pin is in use. Pin 3 can be configured as an alert interrupt pin to signal whenever the battery drops to 32 percent or lower.

For more details, check out the SparkFun product page (www.sparkfun.com/products/10711). You find interesting notes on the hardware around the difficulties with charging lithium batteries, so make sure to read all the comments below the product description. Many other smaller lithium breakout boards are available that supply the standard 3.7V, such as the Lithium Polymer USB Charger and Battery by SparkFun (www.sparkfun.com/products/9876) and the USB/DC Lithium Polymer battery charger 5-12V - 3.7/4.2v by Adafruit (www.adafruit.com/products/280). These breakout boards are perfect when paired with an appropriate low-voltage Arduino, such as the Pro Micro - 3.3V/8Mhz (www.sparkfun.com/products/10999) or the Arduino Pro 328 - 3.3V/8MHz (www.sparkfun.com/products/10914). These are useful when trying to reduce the size of your Arduino project.

**GPS Shield Retail Kit**

Made by: SparkFun

Price: £60.59 from Cool Components; $79.95 from SparkFun

Pins used: 0, 1 or 2, 3 (default)

Using the GPS Shield (Figure 13-6), it’s quick and easy to incorporate location data into your project. Find your location to within a few meters, maybe to create GPS art or to map all your movements over the month. This shield is also great for giving you an extremely accurate time.

The GPS Shield comes assembled but requires minor soldering to attach the header pins or header sockets. Data from the GPS module can either be sent to UART to directly send location data to the Arduino’s hardware RX and TX pins, 0 and 1, or set to DLINE to send the data to digital pins 2 and 3 (default). Code can be sent to the Arduino only when it is set to DLINE.

Note that the retail kit is sold with a GPS module and header pins. This shield is designed for the EM-406a module (www.sparkfun.com/products/465).
You can use other modules, such as the EM-408 and EB-85A, but you need to buy the appropriate socket separately.

For more details, visit the SparkFun product page (www.sparkfun.com/products/10710). There is also a great GPS Shield Quickstart Guide to get you going in no time; go to (www.sparkfun.com/tutorials/173/). SparkFun also has an excellent GPS buying guide (www.sparkfun.com/pages/GPS_Guide).

**Figure 13-6:**
The Sparkfun GPS Shield with the EM-406a.

---

**GPS Logger Shield Kit v1.1**
Made by: Adafruit

Price: $20.53 from Proto-PIC; $19.50 from Adafruit

Pins used: 10, 11, 12, 13 and any two other pins

The GPS Logger Shield lets you track and store location information using the Global Positioning System. You can find your location within a few meters. Use it to create GPS art or to map all your movements over the month. It's also great for giving you an extremely accurate time. This data is stored on
an SD card as a .txt file which can then be overlaid onto Google Maps or visualized in some other way.

With the (ever increasing) size of SD cards, you are able to store much more data than your Arduino can on its own internal memory. This is especially useful because it keeps your data-logging device mobile without the need for a computer, meaning that you can leave that bulky laptop at home and send your GPS device out into the world!

The Adafruit GPS Logger Shield Kit is sold as a kit and requires soldering. Pins 10, 11, 12, and 13 are used for communication with the SD card. The GPS module actually requires only a few pins, which are connected by using jump wires to whichever pins you would prefer. RX and TX pins must be connected to two digital pins, such as 2 and 3. You can enable other optional functions, such as a signalling LED to indicate when data is logged, a pin to monitor the pulse from the GPS synchronization clock, and a pin to detect when an SD card is in the slot.

Note that you need to buy a GPS module and SD separately. This shield is designed for the EM-406a module (www.adafruit.com/products/99) but works with a variety of modules (listed at http://ladyada.net/make/gpsshield/modules.html).

Get the shield and more information on Adafruit (www.adafruit.com/products/98). You can view an in-depth tutorial on Ladyada’s site (www.ladyada.net/make/gpsshield/) that details everything from construction of the kit through to the Arduino code, along with uses for the GPS data.

**Wireless Proto Shield and Wireless SD Shield**

Made by: Arduino

Price: $12.22 and $15.92 from RS Components; $27 and $36 from Arduino Store

Pin requirements: 0 and 1 for the Wireless Proto Shield; 0, 1, 4, 11, 12, 13 for the Wireless Shield SD

The Arduino Wireless Proto Shield lets you create a network of wireless devices using XBee radio modules. These highly versatile, reliable, and affordable modules are used to link multiple Arduino boards to create your very own wireless sensor network. The Wireless Shield SD also gives you the ability to store any data you might be collecting from your network and visualize it later on a computer. Each shield also has ample space to add a number of sensors or actuators to your project.

The Wireless Shield and Wireless Shield SD are sold fully assembled and ready to use. The shield talks with either the sketch on the Arduino or through the Arduino to a computer using USB. Using the serial select switch, you can toggle between these two modes. If you have the Wireless Shield
SD, pins 4, 11, 12 and 13 are used to communicate with it and are therefore unavailable for any other function.

Note that you need to purchase the XBee module separately. The XBee 1mW Chip Antenna - Series 1 (802.15.4) is a good one to start with and is available from RS Components (http://uk.rs-online.com/web/p/zigbee-ieee-802154/0102715/) and SparkFun (www.sparkfun.com/products/8664). SparkFun has an excellent XBee buying guide that is extremely helpful if you’re starting in the world of wireless networks (www.sparkfun.com/pages/xbee_guide).

The Arduino product page has more detail on the shield (http://arduino.cc/en/Main/ArduinoWirelessShield), and there is a tutorial for simple communication using the XBee 1mW Chip Antenna - Series 1 (802.15.4) on the Arduino site (http://arduino.cc/en/Guide/ArduinoWirelessShield).

**Ethernet Shield R3 and Ethernet Shield R3 with PoE**

Made by: Arduino

Price: £25.42 and £36.82 from RS Components; $45.95 and $59.95 from SparkFun

Pin requirements: 4, 10, 11, 12, 13

The Ethernet Shield (shown in Figure 13-7) allows your Arduino to talk to the Internet without the need for a computer in the middle. Your Arduino can, therefore, either use or contribute to the wealth of data on the Internet. It may be monitoring Twitter for hashtags or uploading sensor data for the world to see. The shield even has a microSD card reader to allow you to save results or store files to distribute over the network. The R3 is the latest version of this shield and adds a few extra pins to make the shield compatible with the Uno R3.

The Ethernet Shield R3 with Power-over-Ethernet (PoE) includes an extra module that runs diagonally across the board. This is a PoE module and allows you to power your Arduino using the Ethernet cable. This requires a Category 5 (CAT5) Ethernet cable, conforming to the IEEE 802.3af PoE standard and a port that supports PoE. In many cases, a conventional house hub or router does not support PoE, but it’s a good feature to be aware of.

The Ethernet Shield R3 and Ethernet Shield R3 with PoE are sold fully assembled and ready to use. Pins 4, 11, 12, and 13 are used to communicate with the SD card on the shield, so they should not be used for any other function.

**WiFi Shield**

Made by: Arduino

Price: £74.39 from Proto-PIC; $84.95 from SparkFun

Pin requirements: 4, 7, 10, 11, 12, 13

The WiFi Shield (shown in Figure 13-8) allows you to connect wirelessly to a hub or hotspot. It's especially useful when you don’t have access to an Ethernet port or want a degree of mobility for your project, or to access or put information on the Internet. The shield even has a microSD card reader to allow you to save results or store files to distribute over the network.

The WiFi Shield is sold fully assembled and ready to use. The Arduino talks with both the WiFi Shield’s processor and the microSD card reader using the SPI bus, pins 11, 12, and 13. You use pin 10 to select the processor on the WiFi Shield (HDG104) and pin 4 to select the microSD card reader. Use pin 7 for handshaking — the process of setting up communication between two devices, in this case the Arduino and the WiFi Shield.

Find more details on the WiFi Shield on the Arduino product page (http://arduino.cc/en/Main/ArduinoWiFiShield). There is an excellent tutorial for getting started with the WiFi Shield on the Arduino site (http://arduino.cc/en/Guide/ArduinoWiFiShield) and details of the

![The Arduino WiFi Shield](image)

**Cellular Shield with SM5100B**
Made by: SparkFun

Price: $66.17 from Proto-PIC; $99.95 from SparkFun

Pin requirements: 0, 1 or 2, 3 (default)

The Cellular Shield (Figure 13-9) turns your modest Arduino into a functional mobile phone. With this shield, you can send and receive calls, text messages, and even data. All you need is a prepaid SIM card and an antenna, and you’re ready to communicate with the world. By using Serial.print, you can send the correct codes to talk to the SM5100B.

The Cellular Shield with SM5100B is sold fully assembled and ready to use. You need to purchase an antenna with an SMA connector; SparkFun offers the Quad-band Cellular Duck Antenna SMA (www.sparkfun.com/products/675). The Arduino talks with the SM5100B using either the RX and TX pins 0 and 1 or pins 2 and 3 using the SoftwareSerial library. By default, this is set to 2 and 3 but can be changed by removing the solder jumper on the board.
The SIM card should have enough credit to perform the actions you’re trying to do. Those offering unlimited text messages are especially useful. Other optional extras are a mic and speaker; without them, you can call and hang up but won’t be able to do any more than that.

Find more details on the SparkFun products page (www.sparkfun.com/products/9607). That page has an example sketch, but for a real introduction, I recommend heading over to John Boxall’s site (http://tronixstuff.wordpress.com/2011/01/19/tutorial-arduino-and-gsm-cellular-part-one/). He has some excellent resources available through his introduction to GSM, so make sure to have a good read.

**Geiger Counter – Radiation Sensor Board**

Made by: Liberium

Price: $110.80 from Cooking Hacks; $170 from MicroController Pros

Pin requirements: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

The Radiation Sensor Board is probably one of the most impressive Arduino shields. It allows you to monitor radiation levels in the environment. This board was made to help the people of Japan monitor radiation levels.
following the radiation leakages in Fukushima in March 2011. The Geiger
counter can use various Geiger tubes to detect different types and levels of
radiation. There is also an LCD display, LED, and piezo speaker for feedback.

This shield uses Geiger tubes that operate at dangerously high voltages (400V–
1000V), so it requires extreme care. It is best to keep the Radiation Sensor
Board in an enclosure to keep it out of human contact. Radiation is dangerous,
but so is electricity. If you don’t know what you’re doing, don’t mess around.

The piezo speaker and LED are connected to pin 2, which triggers an inter-
rupt with every pulse that the Geiger tube generates. Depending on the tube
used and the number of pulses or counts per minute (cpm), you can deter-
mine the actual radiation level in Sieverts per hour. Pins 3 to 8 are used for
the LCD display to generate the sensor readings in detail. Pins 9 to 13 are
used for the LED bar to give clear visual feedback of the radiation level. The
first three LEDs are green, and the last two are red, showing that a high and
potentially dangerous level of radiation is being approached.

More details on this project can be found on the Cooking Hacks product page
(www.cooking-hacks.com/index.php/documentation/tutorials/
gieger-counter-arduino-radiation-sensor-board).

Staying current

Many other great shields are available, and new and improved ones are being
released all the time. You can take some actions to keep up to date, though.

Check the stores regularly for the latest products. It’s a bit like browsing in a
DIY shop; you never know what you’ll find.

✓ Arduino Store (http://store.arduino.cc)
✓ Adafruit(www.adafruit.com)
✓ Maker Shed (www.makershed.com)
✓ Seeed Studio (www.seeedstudio.com)
✓ SparkFun(www.sparkfun.com)

Also, check Arduino-related blogs regularly. Blogs and news pages on Arduino-
related sites often show off new kits or new and interesting applications of
older hardware, so they’re always worth checking for a spot of inspiration.

✓ Arduino Blog (http://arduino.cc/blog)
✓ Adafruit (www.adafruit.com/blog)
✓ Hack A Day (http://hackaday.com)
Some people have made efforts to document the various shields. Check these sites:

- Arduino Playground ([www.arduino.cc/playground/Main/SimilarBoards#goShie](http://shieldlist.org))
- Arduino Shield List ([http://shieldlist.org](http://shieldlist.org))

**Browsing the Libraries**

Basic sketches can get you quite a long way, but when you get more advanced you need to know about libraries. Libraries provide extra functionality to your sketch, either to use specific hardware or to incorporate more complex functions in software. In the same way that you’d go to a physical library to learn something new, you include libraries in your code to teach your Arduino something new. By including a library in a sketch you can quickly and easily access functions to help you achieve your goals.

Getting started with complex hardware or software can be a difficult thing to do. Luckily a lot of people have taken the time to document their progress and have released libraries, often with examples, that can be easily integrated into your own sketches. From this it’s possible to get something working, get to grips with it, and hopefully gain a better understanding of it. This is the learn-by-doing approach of Arduino that allows you to make a lot of progress quickly and easily with hardware or software that would otherwise be a huge challenge.

**Reviewing the standard libraries**

This section covers is a selection of the libraries included in the current release of Arduino at the time of writing (1.0.1). The standard libraries cover a wide range of subject areas and are usually popular topics that have been heavily documented. You can find these libraries by choosing Sketch ➪ Import Library. Choosing a library includes one line at the top of your current sketch, such as `#include <EEPROM.h>`. Before you understand a library, you should try an example of it. You’ll find examples at the bottom of the menu that appears at File ➪ Examples.
Here is a brief description of what library does:

✓ **EEPROM** ([http://arduino.cc/en/Reference/EEPRO](http://arduino.cc/en/Reference/EEPRO)): Your Arduino has Electrically Erasable Programmable Read-Only Memory (EEPROM), which is permanent storage similar to the hard drive in a computer. Data stored in this location stays there even if your Arduino is powered down. Using the EEPROM library, you can read from and write to this memory.

✓ **Ethernet** ([http://arduino.cc/en/Reference/Ethernet](http://arduino.cc/en/Reference/Ethernet)): After you have your Ethernet Shield, the Ethernet library allows you to quickly and easily start talking to the Internet. When you use this library, your Arduino can either act as a server that is accessible to other devices or as a client that requests data.

✓ **Firmata** ([http://arduino.cc/en/Reference/Firmata](http://arduino.cc/en/Reference/Firmata)): Firmata is one way to control your Arduino from software on a computer. It is a standard communication protocol, which means that rather than write your own communication software, you can use the library to allow easy communication between hardware and software.

✓ **LiquidCrystal** ([http://arduino.cc/en/Reference/LiquidCrystal](http://arduino.cc/en/Reference/LiquidCrystal)): The LiquidCrystal library helps your Arduino talk to most liquid crystal displays. The library is based on the Hitachi HD44780 driver, and you can usually identify these displays by their 16-pin interface.

✓ **SD** ([http://arduino.cc/en/Reference/SD](http://arduino.cc/en/Reference/SD)): The SD library allows you to read from and write to SD and microSD cards connected to your Arduino. SD cards need to use SPI to transfer data quickly, which happens on pins 11, 12, and 13. You also need to have another pin to select the SD card when it’s needed.

✓ **Servo** ([http://arduino.cc/en/Reference/Servo](http://arduino.cc/en/Reference/Servo)): The Servo library allows you to control up to 12 servo motors on the Uno R3 (and up to 48 on the Mega). Most “hobby” servos turn 180 degrees, and using this library, you can specify the degree that you want your servo(s) to turn to.

✓ **SPI** ([http://arduino.cc/en/Reference/SPI](http://arduino.cc/en/Reference/SPI)): The Serial Peripheral Interface (SPI) is a method of communication allowing your Arduino to communicate very quickly with one or more devices over a short distance. This could be receiving data from sensors, talking to peripherals such as an SD card reader, or communicating with another microcontroller.

✓ **SoftwareSerial** ([http://arduino.cc/en/Reference/SoftwareSerial](http://arduino.cc/en/Reference/SoftwareSerial)): The Software Serial library allows you to use any digital pins to send and receive serial messages instead of, or in addition to, the usual hardware pins, 0 and 1. This is great if you want to keep the hardware pins free for communication to a computer, allowing you to have a permanent debug connection to your project while still being able to upload new sketches, or to send duplicate data to multiple serial devices.
 ✓ **Stepper** ([http://arduino.cc/en/Reference/Stepper](http://arduino.cc/en/Reference/Stepper)): The stepper library allows you to control stepper motors from your Arduino. This code also requires the appropriate hardware to work, so make sure to read Tom Igoe’s notes on the subject ([www.tigoe.net/pcomp/code/circuits/motors/stepper-motors/](http://www.tigoe.net/pcomp/code/circuits/motors/stepper-motors/)).

 ✓ **WiFi** ([http://arduino.cc/en/Reference/WiFi](http://arduino.cc/en/Reference/WiFi)): The WiFi library is based on the Ethernet library listed previously, but with alterations specific to the WiFi Shield to allow you to wirelessly connect to the Internet. The WiFi library also works well with the SD library, allowing you to store data on the shield.

 ✓ **Wire** ([http://arduino.cc/en/Reference/Wire](http://arduino.cc/en/Reference/Wire)): The Wire library allows your Arduino to communicate with I2C devices (also known as TWI, or two-wire interface). Such devices could be addressable LEDs or a Wii Nunchuk, for example.

### Installing additional libraries

Many libraries aren’t included in the Arduino software. Some libraries are for completely unique applications such as specific hardware or functions; others are refinements or adaptations of existing libraries. Luckily, Arduino makes including these incredibly easy, so you can quickly try them all to see which are right for your needs.

Libraries are usually be distributed as .zip files that have the same name as the library; for example, the capacitive sensing library CapSense should be distributed as CapSense.zip and should contain a folder of the same name when unzipped. It may also be distributed as a version, such as CapSense-1.0.1 or CapSense_20120930. Distributing it as a version allows you to keep track of the current version so that you can use the latest or a specific revision. Whichever version you are using, be sure to name the library folder with the library name CapSense.

Inside the folder there are files ending in .h and .cpp, such as CapPin.h and CapPin.cpp, and maybe even an Examples folder. If your .zip file contains only loose .h and .cpp files, you should place them in a folder with a library name. Sometimes you may find many .h and .cpp files that all perform different functions in the library, so make sure they’re all inside the folder.

In the latest release of Arduino (1.0.1 at time of writing), it’s easy to include libraries. Simply move the library folder into your Arduino sketch folder.

In Mac OSX, it will look like this:

```text
~/Documents/Arduino/libraries/CapSense/CapPin.h
~/Documents/Arduino/libraries/CapSense/CapPin.cpp
~/Documents/Arduino/libraries/CapSense/examples
```
In Windows, it will look like this:

```
My Documents /Arduino/libraries/CapSense/CapPin.h
My Documents /Arduino/libraries/CapSense/CapPin.cpp
My Documents /Arduino/libraries/CapSense/examples
```

After the library is installed, restart Arduino and choose Sketch ➤ Import Library to check that your library is in the list, as shown in Figure 13-10. The libraries does not function if the files are not in the correct folder structure or if the names are changed, so be sure to check this if you cannot see the library there.

![The Arduino menu shows the library in the Import Library drop-down list.](image)

If the library has an Examples folder, you should also be able to see the examples in File ➤ Examples under the name of the library, as in Figure 13-11.

That’s all there is to installing a library. Removing a library is as simple as taking the library folder in question out of the Arduino sketch folder.
Chapter 13: Becoming a Specialist with Shields and Libraries

Obtaining contributed libraries


CapSense and TimerOne are two commonly used and helpful libraries to start with as you get familiar with contributed libraries:

✓ **CapSense** (www.arduino.cc/playground/Main/CapSense):
  CapSense is a library that allows you to make one or many pins on your Arduino into capacitive sensors. This allows you to make simple touch, pressure, or presence detection sensors quickly and easily with very little hardware.

The Arduino Playground page has a lot of useful information, but a more recent version of the code can be found on GitHub (https://github.com/moderndevice/CapSense).
TimerOne (http://playground.arduino.cc//Code/Timer1): TimerOne or Timer1 uses a hardware timer on your Arduino to perform timed events at regular intervals. It’s a great library for reading sensor data regularly without interrupting what’s going on in the main loop. There is a page on the Arduino Playground and an up-to-date version of the library on Google Code (http://code.google.com/p/arduino-timerone/).

If you’re really keen to understand libraries more and maybe even write your own, check out this introduction to writing your own libraries on the Arduino page at http://arduino.cc/en/Hacking/LibraryTutorial.
In This Chapter
▶ Sending lots of signals using a Mega 2560 board
▶ Using a shift register
▶ Learning how to count in binary

An individual input and output is great, and sometimes that’s all that is necessary for your project. Often, however, you want to sense or control with a lot of inputs and outputs all at one time. Many of the most well-known digital art and installations are actually very simple at heart; the real complexity is in instructing those simple actions to occur hundreds, and maybe even thousands, of times.

In this chapter, you find out how to perform lots of activities simultaneously with your Arduino. You might accomplish this by using a bigger board, such as the Arduino Mega 2560, or by using additional hardware to allow your regular Arduino Uno to do more than it can on its own. You learn about the pros and cons of each of these methods to equip you with the right knowledge to help you build a monster of an Arduino project.

Controlling Multiple LEDs

One of the simplest ways to extend your Arduino project is by using a bigger board: the Arduino Mega 2560. This monster of a board, shown in Figure 14-1, gives you a many more pins to play with than a typical Arduino board does: 54 digital I/O pins, 15 PWM capable pins, and 16 analog input pins. With the Mega 2560, you have a lot more space for all kinds of inputs and outputs. But that’s not all. The Mega has four hardware serial ports, allowing it to communicate with multiple serial devices simultaneously. This capability is great for keeping the USB serial line free for communication with a computer without disturbing any connections to serial devices.
This communication is all possible because of the ATmega2560 microprocessor on the board. If you compare the Mega 2560 to an Uno, the first difference you notice is physical. If you look at the chip on an Uno R3, you see that ATmega328 is shaped like a millipede — a form known as dual in-line package (also known as DIP and DIL) because of the two parallel lines of pins. More specifically, this is a plastic dual in-line package (PDIP). If you look at the center of your Mega 2560, you see a diamond-shaped computer chip known as a quad flat package (QFP) because of the four sets of pins and the very flat profile of the chip. Specifically, this one is a thin quad flat package (TQFP), and it has many other variations. You see both chips in Figure 14-2.

The physical form isn’t the only difference; the names of the chips are also slightly different. The microcontroller names are printed on top of the chip: the Uno reads ATMEGA328P-PU and the Mega shows ATMEGA2560 16-AU.
(both names are visible in Figure 14-2). These are product numbers that relate to the amount of memory on the chip. The two important numbers to understand are 328 and 2560, which relate to the amount of flash memory that each microcontroller has — 32KB and 256KB, respectively. Flash memory is used to store your sketches, so you can see that the Mega 2560 has more than seven times the space of the Uno, which can be especially useful for programs that are storing a lot of data, such as strings.

To familiarize yourself with the Mega, take a look at the layout of the board with the pins labeled shown in Figure 14-3. Much of the board is similar to the Uno, but you have to pay special attention to the block of pins at the end to make sure that you don’t use the extra 5V and GND pins rather than the digital pins. Also, some shields need modification to work with the Mega 2560, so read the instructions carefully.
Implementing the AnalogWriteMega sketch

In this example, you learn how to make a Nightrider-style LED bar. Because the Mega 2560 has 15 possible PWM pins, it’s perfect for precise control of many analog outputs. The AnalogWriteMega sketch allows you to smoothly transition between LEDs.

For this project, you need:

- An Arduino Mega 2560
- A breadboard
- Twelve LEDs
- Twelve 220 ohm resistors
- Jump wires

This circuit is all about repetition, so it’s a good idea to find a pack of color-coded jump wires or carefully cut and strip lengths of equipment wire, preferably single core. When choosing LEDs and resistors, be sure to do your calculations. Standard 3mm or 5mm LEDs in most kits require a voltage of around 2.5V and a current of around 25mA, and the Arduino’s digital pins are capable of providing a maximum of 5V.
In an equation, this means:

\[
\frac{(5V - 2.5V)}{0.025A} = 100 \text{ ohms}
\]

A 100 ohm resistor is an exact match for running the LED up to the maximum recommended power, but by using a 220 ohm resistor, you keep the LED well within its limits, which allows it to operate longer. A general rule when choosing a resistor is to pick the nearest available resistor that is above the calculated resistance. If you feel that your LEDs just aren’t bright enough, you can find resistors that are closer to the optimum resistance, find higher brightness LEDs, or find a higher power LED that consumes closer to 5V.

After you have your LEDs, resistors, and jump wires, assemble the circuit as shown in Figures 14-4 and 14-5. The circuit is basically 12 of the same, smaller circuits, each of the Arduino Mega pins being connected to a 220 ohm resistor, then onto the anode (long leg) of the LED, with the cathode (short leg) being connected to ground.
After you’ve assembled your circuit, you want to find the appropriate sketch. From the Arduino menu, choose File ➪ Examples ➪ 03.Analog ➪ AnalogWriteMega to display a sketch for doing lots of analogWrite() functions on the Mega. The code for the sketch is as follows:

```c
/*
Mega analogWrite() test

This sketch fades LEDs up and down one at a time on digital pins 2 through 13. This sketch was written for the Arduino Mega, and will not work on previous boards.

The circuit:
* LEDs attached from pins 2 through 13 to ground.

created 8 Feb 2009
*/
```
This example code is in the public domain.

```c
/*
// These constants won't change. They're used to give names
// to the pins used:
const int lowestPin = 2;
const int highestPin = 13;

void setup() {
  // set pins 2 through 13 as outputs:
  for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
    pinMode(thisPin, OUTPUT);
  }
}

void loop() {
  // iterate over the pins:
  for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
    // fade the LED on thisPin from off to brightest:
    for (int brightness = 0; brightness < 255; brightness++) {
      analogWrite(thisPin, brightness);
      delay(2);
    }
    // fade the LED on thisPin from brightest to off:
    for (int brightness = 255; brightness >= 0; brightness--) {
      analogWrite(thisPin, brightness);
      delay(2);
    }
    // pause between LEDs:
    delay(100);
  }
}
```

If your sketch uploads correctly, you see each LED fade up and then down in sequence, and then return to the start again.

If you don’t see this happening, double-check your wiring:

- Make sure that you’re using the correct pin numbers.
- Check the connections on the breadboard. If the jump wires or components are not connected using the correct rows in the breadboard, they don’t work.
- Check that your LEDs are the right way around, with the digital pin going to a resistor to the long leg and the short leg going to ground.


**Understanding the AnalogWriteMega Sketch**

This sketch is similar to fading a single LED, as in the basic AnalogWrite sketch (see Chapter 7), but involves a lot of repetition. Using for loops it’s possible to perform a lot of repetitive tasks without writing the same code for each LED.

At the start of this sketch, you declare the constant integers. Rather than declare 12 separate values, you can declare the lowest and highest pin because the board has 12 PWM pins all in a row, from pin 2 to pin 13. The other three PWM pins are 44, 45, and 46, so they’re not quite as conveniently spaced for the sketch.

```cpp
const int lowestPin = 2;
const int highestPin = 13;
```

In setup, the pins are set to be outputs using a for loop. Because the pins are in one unbroken row, the for loop can simply count up through each pin from the first to the last and set each of them to OUTPUT. Inside the for loop, the local variable thisPin is declared as an integer equal to the lowestPin; this is your start point. The variable thisPin is then compared to highestPin. If thisPin is less than or equal to (<=) highestPin, the digital pin of the same number is set as an output and thisPin is incremented by one for the next loop. This process quickly and effectively sets all your pins to an output with two lines of code rather than 12.

```cpp
void setup() {
    // set pins 2 through 13 as outputs:
    for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
        pinMode(thisPin, OUTPUT);
    }
}
```

In the main loop, there are for loops within for loops! You can see where each loop sits by the indentation of the for loop code. To format your code automatically, choose Tools➪Auto Format (or press Ctrl+T on Windows or ⌘+T on a Mac). If you get confused, move the cursor to the right side of a {} curly brace to see the other corresponding curly brace highlighted. This is helpful but can be incorrect if you have too few curly braces, so make sure that you have them in the right places. Start with the first or highest level for loop, which contains the other two loops. The highest loop counts through the pins 2 to 13 in order, each time incrementing the value thisPin until it reaches 14.
The first loop gives you the current pin number, so it can be used again in
the next two for loops. The first of these loops creates another loop with the
local variable brightness. This brightness, too, is incremented, counting from
0 to 255, one value at a time. Every time the brightness is incremented, an
analogWrite is run with the current brightness value, with a short delay of
2mS before increasing the brightness again.

After the brightness has reached its maximum PWM value of 255, the next
for loop returns to 0. This is done exactly the same way, but instead counts
down one point of brightness at a time.

At the end of the second for loop, a 100mS delay occurs before the sketch
returns to the higher level for loop. The next loop round the LED (as held
in the variable thisPin) moves on one and repeats the same task until it
reaches pin 13. After pin 13, the criteria of the for loop is no longer true,
so it exits and starts the main loop from scratch, completing the sketch and
ready to start again.

This sketch fades up and down each LED from pin 2 to 13 and works well to
check that the circuit is functioning. This isn’t the most interesting of appli-
cations, though, so in the next section you learn how to do more with this
setup.
Tweaking the AnalogWriteMega sketch

Animating a string of LEDs can be great fun. There is a huge difference between blinking them on and off and animating them, which really has to be seen in the flesh (or LED) to be appreciated.

The first step is to change the sequence to make it more interesting. By adding another for loop and commenting out the delay, you can create an animation that loops nicely. Create a new sketch, type the code that follows, and save it with a memorable name, such as myLedAnimation. Alternatively, you can open the AnalogWriteMega sketch, make the changes (indicated by the arrows in the margin), and choose File ➤ Save As to save the sketch with a new name.

It’s also a good habit to update the comments of your sketch as you modify it, as shown in the code that follows.

```cpp
/*
  myLedAnimation
  This sketch fades LEDs up one at a time on digital pins 2 through 13 and then down one at a time on digital pins 13 through 2.

  This sketch was written for the Arduino Mega, and will not work on previous boards.

  The circuit:
  * LEDs attached from pins 2 through 13 to ground.

  Original code by Tom Igoe (2009) - Mega analogWrite() test
  Modified by [Your_Name] (20__) 
*/

const int lowestPin = 2;
const int highestPin = 13;

void setup() {
    for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
        pinMode(thisPin, OUTPUT);
    }
}

void loop() {
    for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
```
Chapter 14: Sensing More Inputs and Controlling More Outputs

When you run the changes, you get a row of LEDs that light up quickly one at a time. After the row is full, each LED dims until all are off; then the whole sequence repeats.

That’s fun, but you can do even more. This next bit of code transforms your LED display into something similar to KITT, David Hasselhoff’s companion from Knight Rider.

Create a new sketch, type the following code, and save it with a memorable name, such as myKnightRider. Alternatively, open the AnalogWriteMega sketch, make the changes (indicated by the arrows in the margin), and choose File➪Save As to save the sketch with a new name.

```cpp
// fade the LED on thisPin from off to brightest:
for (int brightness = 0; brightness < 255; brightness++) {
    analogWrite(thisPin, brightness);
    delay(2);
}

for (int thisPin = highestPin; thisPin >= lowestPin; thisPin--) {
    for (int brightness = 255; brightness >= 0; brightness--) {
        analogWrite(thisPin, brightness);
        delay(2);
    }
    // pause between LEDs:
    // delay(100);
}
```

When you run the changes, you get a row of LEDs that light up quickly one at a time. After the row is full, each LED dims until all are off; then the whole sequence repeats.

That’s fun, but you can do even more. This next bit of code transforms your LED display into something similar to KITT, David Hasselhoff’s companion from Knight Rider.

Create a new sketch, type the following code, and save it with a memorable name, such as myKnightRider. Alternatively, open the AnalogWriteMega sketch, make the changes (indicated by the arrows in the margin), and choose File➪Save As to save the sketch with a new name.

```cpp
/*
myKnightRider

This sketch quickly fades each LED up then down on digital pins 2 through 13
and returns on 13 through 2, like KITT from Knight Rider.
Unfortunately it won’t make your car talk.

This sketch was written for the Arduino Mega, and will not work on previous
boards.

The circuit:
* LEDs attached from pins 2 through 13 to ground.

Original code by Tom Igoe (2009) – Mega analogWrite() test
Modified by [Your_Name] (20__)
*/
```
// These constants won't change. They're used to give names
// to the pins used:
const int lowestPin = 2;
const int highestPin = 13;

void setup() {
// set pins 2 through 13 as outputs:
for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
    pinMode(thisPin, OUTPUT);
}
}

void loop() {
// iterate over the pins:
for (int thisPin = lowestPin; thisPin <= highestPin; thisPin++) {
// fade the LED on thisPin from off to brightest:
    for (int brightness = 0; brightness < 255; brightness++) {
        analogWrite(thisPin, brightness);
        delay(2);
    }
    for (int brightness = 255; brightness >= 0; brightness--) {
        analogWrite(thisPin, brightness);
        delay(2);
    }
}

for (int thisPin = highestPin; thisPin >= lowestPin; thisPin--) {
// fade the LED on thisPin from brightest to off:
    for (int brightness = 0; brightness < 255; brightness++) {
        analogWrite(thisPin, brightness);
        delay(2);
    }
    for (int brightness = 255; brightness >= 0; brightness--) {
        analogWrite(thisPin, brightness);
        delay(2);
    }
    // pause between LEDs:
    delay(100);
}

This sketch gives you an awesome LED animation, with the row of LEDs light-
ing up one at a time, moving from left to right, and then repeating. You can
tweak it further by adjusting the delay() times and brightness values to get
the timing just right. For more inspiration give “KITT from Knight Rider” a
Google and you’ll find a lot of videos of people doing similar.
Controlling Lots of LEDs by Shifting Out

Sometimes even the Mega 2560 with its 70 pins isn’t enough, and you need options that allow you to have even more inputs and outputs. Luckily, you can obtain chips that allow you to increase the number of outputs your Arduino can control. One such chip is a shift register.

Many types of shift registers are available. A popular one is the 74HC595, which is described as an “8-bit serial-in, serial or parallel-out shift register with output latches; 3 state” on its datasheet (http://www.nxp.com/documents/data_sheet/74HC_HCT595.pdf). The “8-bit” part refers to the number of outputs that can be controlled, so to understand how the shift register works, you must first look at binary, bits, and bytes, which I explain in the “Making sense of binary, bits, and bytes” sidebar. The 74HC595 is shown in Figure 14-6.
Making sense of binary, bits, and bytes

The binary number system uses only two values: 0 or 1. Because it uses only two values, it is also known as base-2. Decimal numbers that you use are usually referred to as base-10 and use 0 to 9; hexadecimal numbers are base-16, and use 0 to 9 and A to F.

But how is binary useful when you’re trying to talk to lots of things and have only two options? The answer is that you use a lot of binary values.

If you take a base-2 binary number such as 10101101, you can determine its value in base-10 using a simple lookup table. Binary is typically read from right to left. Because binary is base-2, each value is the binary value multiplied by 2 to the power of \(2^x\), where \(x\) is equal to the order of the bit, starting at 0 on the right. For example, as shown below the fourth binary value is equal to \(1 \times (2 \times 2^2) = 8\).

<table>
<thead>
<tr>
<th>Binary</th>
<th>Calculation</th>
<th>1(\times 2^7)</th>
<th>0(\times 2^6)</th>
<th>1(\times 2^5)</th>
<th>0(\times 2^4)</th>
<th>1(\times 2^3)</th>
<th>1(\times 2^2)</th>
<th>0(\times 2^1)</th>
<th>1(\times 2^0)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal</td>
<td></td>
<td>128</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>8</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>173</td>
</tr>
</tbody>
</table>

As you can see, extremely large numbers can be formed using only zeros and ones. In this case, you have eight binary values with a total decimal value of 255. When talking about memory, each binary value takes one bit of memory, and each group of eight bits is referred to as a byte. To give you an idea of scale, a blank Arduino sketch uses 466 bytes; an Uno can store a maximum of 32,256 bytes, and a Mega can store a maximum of 258,048 bytes.

Figure 14-7 shows a diagram of the pins on the 74HC595. The pin names are explained in Table 14-1.

In the case of the 74HC595, there are eight output pins and, conveniently, there are also eight bits in a byte. The values are sent one bit at a time to the shift register. When the clock pin (SH_CP) is set HIGH, all the bits shift one place forward, the last pin “shifts out” and a new pin takes its value from the
serial data input (DS). The bits are all stored in the register until the latch pin (ST_CP) is pulled \textbf{HIGH} and the values are sent to the outputs.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q0–Q7</td>
<td>Output pins</td>
<td>These are linked to your LEDs.</td>
</tr>
<tr>
<td>GND</td>
<td>Ground</td>
<td>This is linked to your Arduino’s ground.</td>
</tr>
<tr>
<td>Q7'</td>
<td>Serial out</td>
<td>Serial out is used to shift data to another 74HC595.</td>
</tr>
<tr>
<td>MR</td>
<td>Master Reclear, active low</td>
<td>This clears the shift register if pulled LOW.</td>
</tr>
<tr>
<td>SH_CP</td>
<td>Shift register clock pin</td>
<td>If pulled HIGH, this shifts all the values forward one.</td>
</tr>
<tr>
<td>ST_CP</td>
<td>Storage register lock pin (latch pin)</td>
<td>When pulled HIGH, it outputs the new shift register values. This must be pulled HIGH straight after SH_CP goes low again.</td>
</tr>
<tr>
<td>OE</td>
<td>Output enable, active low</td>
<td>This enables the output when grounded and disables it when HIGH.</td>
</tr>
<tr>
<td>DS</td>
<td>Serial data input</td>
<td>This is the input pin for the new serial data.</td>
</tr>
<tr>
<td>Vcc</td>
<td>Positive voltage supply</td>
<td>This is the voltage supply for the LEDs.</td>
</tr>
</tbody>
</table>

The total value in eight bits (or one byte or a decimal number from 0 to 255) represents every combination of the register, and that is how the shift register communicates its outputs. If you send 11111111, all output pins go \textbf{HIGH}; if you send 10101101, pins 0, 2, 3, 5 and 7 go \textbf{HIGH}. You can also cascade the 74HC595s, meaning that you can add multiple chips to extend the number of outputs further by using the serial out pin (Q7). Doing so gives you more registers to shift bits of data into. If you send 16 bits (or two bytes or a decimal number from 0 to 511), the bits flow through to the first register and into the second as they are shifted.

**Implementing the shiftOutCode, Hello World sketch**

In this example, you look at shifting out using a single 74HC595 to control eight LEDs. The LEDs count up in binary from 0 to 255.

You need:
Part IV: Unlocking Your Arduino’s Potential

✓ An Arduino Uno
✓ A big breadboard
✓ A 74HC595
✓ Eight LEDs
✓ Eight 220 ohm resistors
✓ Jump wires

The 74HC595 is placed across the gap in the center of the breadboard and should be an exact fit. That’s because your breadboard was designed with this placement in mind. The gap in the center allows you to easily connect wires to both sides of the chip, keeping each side of pins separate from one another. Moreso for this example than some others, it is a good idea to have a power and ground rail on each side of the board. To that end, you want to link the Arduino’s 5V and ground pins to each of the two tracks on both sides of the board, as shown in the Figure 14-8 circuit layout.

The layout of the pins is quite straightforward, except that the first pin — pin 0 — is on the opposite side of the board from the others. For this example, it’s a good idea to use lots of color-coded jump wires or equipment wire to keep track of where the connections are going. Complete the circuit as shown in Figures 14-8 and 14-9.
After your circuit is assembled, you need the appropriate software to use it. Create a new sketch by typing the code that follows and saving it with a memorable name, such as myShiftOut. Alternatively, you can find this example at http://arduino.cc/en/Tutorial/ShiftOut1.

```
//**************************************************************//
//  Name    : shiftOutCode, Hello World
//  Author  : Carlyn Maw,Tom Igoe, David A. Mellis
//  Date    : 25 Oct, 2006
//  Modified: 23 Mar 2010
//  Version : 2.0
//  Notes   : Code for using a 74HC595 Shift Register           //
//            : to count from 0 to 255
//****************************************************************

//Pin connected to ST_CP of 74HC595
int latchPin = 8;
//Pin connected to SH_CP of 74HC595
int clockPin = 12;
//Pin connected to DS of 74HC595
int dataPin = 11;

void setup() {
    //set pins to output so you can control the shift register
    pinMode(latchPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    pinMode(dataPin, OUTPUT);
    digitalWrite(latchPin, LOW);
}
```
```cpp
pinMode(clockPin, OUTPUT);
pinMode(dataPin, OUTPUT);
}

void loop() {
  // count from 0 to 255 and display the number
  // on the LEDs
  for (int numberToDisplay = 0; numberToDisplay < 256; numberToDisplay++) {
    // take the latchPin low so
    // the LEDs don't change while you're sending in bits:
    digitalWrite(latchPin, LOW);
    // shift out the bits:
    shiftOut(dataPin, clockPin, MSBFIRST, numberToDisplay);
    // take the latch pin high so the LEDs will light up:
    digitalWrite(latchPin, HIGH);
    // pause before next value:
    delay(500);
  }
}
```

You should see your row of LEDs counting up in binary from 0 to 255. Usually, binary is read from right to left, with the least significant bit (LSB), 1, on the right and the most significant bit, in this case 128, on the left. Work out the first few numbers to see whether the pattern looks right. Table 14-2 shows you numbers 0 to 9.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
</tr>
<tr>
<td>3</td>
<td>00000011</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>5</td>
<td>00000101</td>
</tr>
<tr>
<td>6</td>
<td>00000110</td>
</tr>
<tr>
<td>7</td>
<td>00000111</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>9</td>
<td>00001001</td>
</tr>
</tbody>
</table>
If you don’t see this happening, double-check your wiring:

✓ 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.
✓ Make sure that your LEDs are oriented correctly, in the right order, and that the Arduino pins are wired to the correct 74HC595 pins.

**Understanding the shiftOutCode, Hello World sketch**

At the start of the sketch, the three pins needed to control the shift register are declared. Pin 8 is the latch pin, used to output the values from the register; pin 12 is the clock pin, used to shift the bits along by one pin; pin 11 is used to send new data into the register.

```cpp
// Pin connected to ST_CP of 74HC595
int latchPin = 8;
// Pin connected to SH_CP of 74HC595
int clockPin = 12;
/// Pin connected to DS of 74HC595
int dataPin = 11;
```

In setup, you simply set the pinMode of each pin to OUTPUT.

```cpp
void setup() {
  // set pins to output so you can control the shift register
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}
```

You immediately notice that the loop() is quite small for the seemingly complicated task explained earlier. A for() loop is used to count from 0 to 255. A local variable called numberToDisplay is the current number.

```cpp
void loop() {
  // count from 0 to 255 and display the number
  // on the LEDs
  for (int numberToDisplay = 0; numberToDisplay < 256; numberToDisplay++) {
    // The latch pin is set LOW so that it doesn’t output the value of the register while you are changing it.
  }
```
Part IV: Unlocking Your Arduino’s Potential

The `shiftOut` function is included in Arduino for this exact purpose and requires four parameters. The first is `dataPin`; the second is `clockPin`; the third is the order of the bits, either most significant bit (`MSBFIRST`) or least significant bit (`LSBFIRST`); the fourth parameter is the value to display, which is the local variable `numberToDisplay`. This function takes the value and handles all the pulses to convert it to a combination of pins that fills the buffer.

All that is left to do is to set the latch pin `HIGH` again to send the updated values to the LEDs.

There is a brief pause of half a second by using the `delay` function, before the program returns to the start of the `for` loop and incrementing the number by one.

This cycle continues until the `for` loop reaches 255, which sets all the LEDs `HIGH`, then returns to 0 and repeats the count. This is great for counting in binary, but not much else. In the next section, I refine the code to let you address the LEDs individually.

**Tweaking the `shiftOutCode`, **
**Hello World sketch**

The previous binary method is great for understanding how the shift register works, but what if you want to turn on a specific pin without converting it to binary? This code uses the same circuit as described previously but allows you to select LEDs individually from the serial port.

Start by creating a new sketch by typing out the code that follows and saving it with a memorable name, such as `mySerialShiftOut`. Alternatively, find this example at [http://arduino.cc/en/Tutorial/ShftOut12](http://arduino.cc/en/Tutorial/ShftOut12).
/*
   Shift Register Example
   for 74HC595 shift register

   This sketch turns reads serial input and uses it to set the pins
   of a 74HC595 shift register.

   Hardware:
   * 74HC595 shift register attached to pins 2, 3, and 4 of the Arduino,
     as detailed below.
   * LEDs attached to each of the outputs of the shift register

   Created 22 May 2009
   Created 23 Mar 2010
   by Tom Igoe

   */

//Pin connected to latch pin (ST_CP) of 74HC595
const int latchPin = 8;
//Pin connected to clock pin (SH_CP) of 74HC595
const int clockPin = 12;
//Pin connected to Data in (DS) of 74HC595
const int dataPin = 11;

void setup() {
  //set pins to output because they are addressed in the main loop
  pinMode(latchPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  Serial.begin(9600);
  Serial.println("reset");
}

void loop() {
  if (Serial.available() > 0) {
    // ASCII '0' through '9' characters are
    // represented by the values 48 through 57.
    // so if the user types a number from 0 through 9 in ASCII,
    // you can subtract 48 to get the actual value:
    int bitToSet = Serial.read() - 48;

    // write to the shift register with the correct bit set high:
    registerWrite(bitToSet, HIGH);
  }
}

// This method sends bits to the shift register:
void registerWrite(int whichPin, int whichState) {
  //...
After the sketch is done uploading, click the serial monitor button to open the window. The first word should be “reset,” which is displayed in setup when the sketch starts. Enter numbers between 0 and 7 and click Send (or press Return) to turn on different LEDs in the row.

In this piece of code, you convert the binary addressing system to a decimal one, and instead of having to enter all the binary combinations, you can just pick out individual addresses of each LED by entering a decimal number from 0 to 7.

In the main loop, the if statement checks to see whether there is any data to read. Serial.available is the buffer or storage for incoming bytes, so the if statement is true and then progress only if data has been sent.

Characters sent through the serial port are sent as ASCII characters, and when they are read using Serial.read, they are interpreted as their equivalent integer value on the ASCII chart. This puts 0 to 9 in the range of 48 to 57, so to get them to the correct range, you simply subtract 48 from the reading.

A custom function called registerWrite makes conversion of 0 to 9 to the relevant byte value simpler. The function is outside the main loop at the bottom of the sketch.
The two values that can be entered into `registerWrite` are declared at the top of the sketch as `whichPin` and `whichState`. These variables and any within the function are local and cannot be referenced in the main loop.

```c
void registerWrite(int whichPin, int whichState) {

A byte variable is declared and set to 0.

```c
  byte bitsToSend = 0;
```

As in the previous example, the latch pin is set LOW before the bits are shifted.

```c
  digitalWrite(latchPin, LOW);
```

This time, a new function called `bitWrite` is used. It has three parameters: the variable that you are writing to, in this case `bitsToSend`; which bit of the byte to write to, from 0 (least significant bit, rightmost bit) to 7 (most significant bit, leftmost bit); and the state, either HIGH or LOW.

```c
  bitWrite(bitsToSend, whichPin, whichState);
```

For example, if byte `bitsToSend` is equal to 0, it equals 00000000 in binary. When you use the `bitWrite` function with `whichPin` equal to 4 and `whichState` equal to HIGH (or 1), `bitsToSend` equals 00010000.

After `bitWrite` updates the value of the byte `bitsToSend`, `bitsToSend` is put through the `shiftOut()` function to update the register.

```c
  shiftOut(dataPin, clockPin, MSBFIRST, bitsToSend);
```

Finally, the latch pin is set HIGH and the results are updated on the LEDs themselves.

```c
  digitalWrite(latchPin, HIGH);
}
```

**Doing more with the same circuit**

This last example allows you to communicate values over serial, but you could just as easily automate this process using a for loop to count from 0 to 7 and back again. The value could also respond to a sensor, such as a
potentiometer, using the map function to give you a power bar as you turn it. The possibilities are endless. There are many other examples on the Arduino site for increasingly complex functionality (go to http://arduino.cc/en/Tutorial/ShiftOut) including cascading multiple shift registers.

After you go past two shift registers, you’re well-advised to power the LEDs externally to reduce the amount of current passing through the Arduino. When you’re using lots of LEDs, you can easily go over the 200mA max current draw, so use your multimeter to check the current draw of the circuit and stay safe.

This chapter’s example described just one chip that is available. You can find many other chips, such as the TLC5940, which can control 16 PWM outputs and works on a very similar principle. The Google Code page contains many details as well as a library contributed by Alex Leone (http://code.google.com/p/tlc5940arduino/). You can also reverse the principle of shifting out with a different chip and read lots of inputs. More details on shifting in are available on the Arduino site at http://arduino.cc/en/Tutorial/ShiftIn.
Chapter 15

Multiplying Your Outputs with I²C

In This Chapter
▶ Discovering I²C
▶ Driving lots of servos
▶ Finding the right power supply

This chapter tells you all about working with I²C (pronounced eye-squared-see or eye-two-see), giving you another way to control lots of outputs. As an example of this great communications protocol, instead of just controlling LEDs (described in Chapter 14), you learn how to control an army of servo motors. With the potential to control hundreds of servos comes the need to power them all so this chapter also talks you through the various options when choosing a power supply.

What Is I²C?

I²C is a great communication protocol that is ideal for getting signals to lots of outputs (I also talk about other ways to control multiple outputs in Chapter 14). Luckily, I'm not the only one who thinks so. There's an excellent product that uses the I²C controlled PCA9685 chip, a PWM driver that lets you drive up to 16 servo motors with a single board. The 16-channel 12-bit PWM/Servo Driver (see Figure 15-1), I²C interface (http://www.adafruit.com/products/815) from Adafruit Industries is a partially assembled kit that makes it easy to control lots of motors, LEDs, or even other circuits.

The board is designed around controlling servos, which have three pin connections (ground, 5V, and the signal wire), so there are 16 groups of three to allow you to easily plug your servos in using the standard three-pin socket. Physically, these pins are in groups of four, so the header pin is three deep and four wide (3 x 4). On either end of the board is space for a six-header pin to communicate with the PCA9685 chip.
A servo, as mentioned in Chapter 8, is a DC motor packaged with an encoder that allows it to keep track of its position. Standard DC motors are work-horses that are great at speeding up and continuously rotating in one direction, as they do on remote control cars and planes. Servos, on the other hand, are for precision movement; some servos continuously rotate (or can be hacked to do so), but generally they move to a specific degree within their range and are excellent for all sorts of applications, from walking robots to electronic sail winches on boats.

The pins from top to bottom are

- **GND**: Ground
- **OE**: Output eEnable
- **SCL**: Serial clock
- **SDA**: Serial data
- **VCC**: Power line for the PCA9685
- **V+**: Power line for the servos

Of these pins, all you need to use are VCC and GND to power the PCA9685 chip, and SCL and SDA to let I²C talk with the Arduino and other boards. Because the board has the same pins on each side, you may rightly assume
that these boards have the ability to be daisy chained together. Each board is given a physical address using the little golden terminals to the top right of the board, known as solder jumpers. These allow you to set the order of the boards by connecting the relevant jumpers to give each board a binary value. Board 0 has no terminals connected, so boards are addressed from 1 onward by soldering the terminals together, using the same principles of counting up in binary that Chapter 14 covers. This arrangement is semi-permanent but can be undone with use of a solder sucker or solder braid.

The V+ pin is linked to the last component, the screw terminal that sits at the top of the board. This is an important feature of this board. Although it is possible to power motors and their controllers using the same supply, doing so is rarely advisable. Motors often need very large amounts of current and voltage to rotate and move loads. Sometimes, current spikes or drops depending on the number of motors and the load on them. Such spikes can be extremely hazardous for the low-voltage PCA9685 control chip, so on this board, the voltage supply has been separated, meaning that the servo motors (or LEDs) can be run from a separate high-voltage, high-current power 12V supply, and the PWM driver chip can operate from an Arduino’s low-voltage, low-current, 5V power supply.

Unlike the 74HC595 shift register and the TLC5940 PWM driver, the PCA9685 has a built-in clock, so your Arduino doesn’t need to constantly send messages to update it, leaving it free to do other things.

In the following example, you learn how to assemble and control an I²C PWM/Servo Driver to make your own mass motor project.

Assembling the I²C PWM/Servo Driver

All the tricky components on the I²C PWM/Servo Driver board have been assembled for you, but you need to know a trick for doing the final bits of soldering. Doing a dry run and laying out the components is a good idea, so do that first.

In the kit, you have:

- One I²C PWM/Servo Driver board
- Four 3 x 4 header pins
- One length of header pins
- One screw terminal

Follow these steps to perform your dry run of the assembly:
1. Using a pair of wire cutters, cut the length of header pins so that you have two lengths of 1 x 6 (see Figure 15-2).

You use these at either end to connect your first board to the Arduino and any daisy chained boards.

2. Place all the parts loosely in the board to make sure they all fit.

The header pins should have the long side pointing up, and the screw terminal should be facing away from the board, to make it easier to connect cables. When you’re happy with where the pins are, it’s time to get soldering.

3. Ensure that you have a clear workspace (as described in Chapter 5), power up your soldering iron, wet your sponge, and find your solder.

Making a cup of tea or coffee is optional, but always advised.

Before you start soldering, note that the data pins on the board are the most sensitive to heat, and if they get too hot, they can be damaged. These are the single rows of headers at either end and the signal pin on the servo headers. I recommend using a high heat on your iron to allow the solder to melt quickly. This means that the solder melts around the joint before it can be conducted and spread all the way down to the chip, by which time you have finished the joint and it is already cooling down. If you are uncomfortable with this level of soldering, practice first on a piece of stripboard with some spare header pins.

The 3 x 4 header pins are the most difficult, so you do best to start with them. The aim is to get the plastic bit level on the board. You can do so by pressing adhesive putty onto the component and board or by using a third hand clamp. I find third hands and other clamps useful for holding boards,
but more trouble than they are worth for holding components. They can also
sometimes scratch the delicate circuitry, so be careful with them and avoid
dragging boards out of their teeth. Using Blue Tack, you can safely secure the
component until you have a couple of solder joints. Then you should secure
the header pins. I recommend connecting the GND and V+ first, because
these can handle more heat than the data line can.

When soldering the inside connectors of the 3 x 4 headers, it can be awkward
to get the soldering iron in position. Try holding your iron at an angle across
the rows and working from one side to the other, such as left to right if you
are right handed. Doing so avoids working over your previous solder joints
and lessens the chance of the individual joints becoming connected. If they
do connect, simply use a solder sucker to remove the excess and reapply
solder if necessary.

After you have the 3 x 4 header pins soldered in place, move on to the header
pins at each end. These should be a lot simpler in comparison, and with luck,
your technique improves after the repetition of the previous task.

Finally, connect the screw terminal. Notice that the holes and pins are a lot
wider than the header pins. This is because this connection is used for sup-
plying power to every motor connected, so a thicker piece of metal is needed
to cope with the amount of current. If the connection is too small, it builds up
heat until it melts, so it’s important to always have thick wires and connec-
tors that are rated for more than the current that you are expecting. Because
of this thicker connection, you may notice that the screw terminal takes a
little longer to heat up and melt the solder. If it takes too long, the plastic part
may melt, so be sure to turn up the heat on your iron if it’s taking a while.

That’s it. Simple, isn’t it? You now have an I²C PWM/Servo Driver board ready
for use. Make sure to check over your connections for any short circuits, and
use your multimeter’s continuity checker when the board isn’t attached to a
power source to check connections if you are unsure. The V+ and GND lines
for the servos should be connected in two rows because they all draw power
from the same source. For now, you should leave the addressable solder
jumpers unsoldered because you are starting with the first board, board 0.

---

**Using the I²C PWM/Servo Driver**

In the previous section, you find out how to assemble an I²C PWM/Servo
Driver. In this section, you see how to make it perform. The example in this
section shows you how to send a signal to the board to control servos on
each pin. You can test this functionality with a single servo, which is included
in most Arduino kits, but after you have it working, you want to order other
servos to get the full effect.

For this project, you need:
✓ An Arduino Uno
✓ A servo motor
✓ Jump wires or pre-crimped wires
✓ An external power supply
✓ A precision screwdriver

The shield has header pins to connect to, but these pins don’t slot directly into an Arduino. A good way to connect them is to use jump wires with sockets on one side and pins on the other (male to female), such as those from Cool Components (http://www.coolcomponents.co.uk/catalog/jumper-wires-male-female-p-355.html) or from SparkFun (https://www.sparkfun.com/products/9140), shown in Figure 15-3.

![Figure 15-3: Jump wires with sockets.](image)

If you’re a real connoisseur, you can make your own custom cables using pre-crimped wires and header sockets, such as these from Technobots (http://www.technobotsonline.com/cable-and-accessories/cable/pre-crimped-wires.html and http://www.technobotsonline.com/connectors-and-headers/cable-assembly-housings.html) or Pololu (http://www.pololu.com/catalog/category/71 and http://www.pololu.com/catalog/category/70). These leave you with one connector at each end and look super professional when bound up using spiral cable wrap, available from RS Components (http://uk.rs-online.com/mobile/p/cable-spiral-wrapping/6826842/).
For the power supply, ensure that you have the correct voltage and current for the number of servos you are using. Some small hobby servos use 5V, but you can find much more variety after you get onto 12V ones. You have two easy options for a power supply: a bench-top power supply to quickly connect your board, or a modified external power supply like those used for charging phones, drills, or laptops. See the “Bench-top power supply” and “External power supplies” sidebars in this chapter for more information.
External power supplies

When shopping for external power supplies (shown in the following figure), you should always buy those that are regulated. Unregulated supplies don’t have a definite voltage and so are cheaper, but more risky. Look on the box and the supply to find out which is regulated and which isn’t. They can also be either fixed or multi-voltage. Fixed gives you the specified voltage and current; multi-voltage gives you a selection of voltages (such as 3, 4.5, 6, 9, 12) and, often, the current varies, allowing more current on the lower voltages.

A good example from Maplin in the United Kingdom is the High Power Multi-Voltage Desktop Power Supply at [http://www.maplin.co.uk/high-power-multi-voltage-desktop-power-supply-48517](http://www.maplin.co.uk/high-power-multi-voltage-desktop-power-supply-48517). It’s expensive but can operate on a very useful range of voltages (5, 6, 9, 12, 13.5, and 15) and supply a maximum current of up to 4A, which is more than enough for most applications. You can find similar power supplies from RadioShack, such as the Enercell Universal 1000mA AC Adapter ([http://www.radioshack.com/product/index.jsp?productId=3875403](http://www.radioshack.com/product/index.jsp?productId=3875403)).

These normally come with a variety of common connectors, such as the 2.1mm power jack connector on the Arduino. This is useful for lower amounts of current (up to 500mA), because it can be plugged into the Arduino to allow you to power things from the Vin pin. However, running lots of servos requires a significant current that could be over 1A, so I don’t advise running this through your Arduino.

Instead, you can strip and tin the wires to allow them to be easily connected directly to the screw terminal on your servo driver board or to a choc block to extend the length of wire. First make sure that the power supply is unplugged. Cut the wire 10cm or so from the base of the connector (that way, you can always connect it back again). The wire on the power supply is most likely either biaxial — two multicore wire side by side — or coaxial — one multicore wire in the center and another surrounding it.

Biaxial is easy to bare using cable strippers and can then be tinned using a bit of solder. The positive is usually be marked in some discreet way, such as faint white or red dashes, but not always, so be sure to test with a multimeter.

Coaxial is a bit more difficult to work with. If you strip back the top layer of insulation on the wire with a cable stripper, you expose a mesh of wire. This can be straightened out and pulled to one side to reveal the central wire. Twist the now-straightened mesh of wire to one side of the central wire and then strip back the central wire as well, leaving enough insulation so that the two wires do not touch. You can then solder these two wires to keep the multicore wire together and connect it to a screw terminal. The ground wire is usually on the outside mesh of wire, but not always, so test it with a multimeter. When you’re tinning, be careful not to melt the insulation around the center core because doing so could create a short circuit.

To test your power supply’s polarity, secure the loose ends to your work surface so that they cannot touch; electrical tape is good for this. Plug in your power supply. Turn your multimeter to the DC voltage setting and place the probes on the ends of each wire. The probes are usually red and black for clarity. If you’re reading a negative voltage, you are reading the power supply the wrong way around — switch your probes to find the correct positive and negative lines. Unplug the supply until the ends of each wire are secured on a screw terminal or tape them up with electrical tape when not in use to avoid accidents.

Always test the voltage of your external power supply with a multimeter before using it to power your project (as shown in Chapter 5). It can be easy to forget the voltage that your supply is supplying, and it is also good to practice to test older power supplies to ensure that they’re up to scratch and supplying the full voltage for which they’re rated.
If you have an external power supply or a bench-top power supply without a current display, link your multimeter in series with either the power or ground line to see how much current you’re drawing as explained in Chapter 5. When you are sure of your positive and negative wires, turn the power supply off and attach them to the screw terminal using a miniature screwdriver. The polarity is marked on the top of the board, so make sure that the +5V and GND are the right way around.

The circuit diagram (Figure 15-4) and schematic (Figure 15-5) detail the wiring for the PWM driver board. The connections can be broken into three areas. First, to power the I²C chip on the board, the VCC and GND pins on the board must be connected to the 5V and GND pins on your Arduino. To send data to the chip, the SCL and SDA pins on the board connect to the SCL and SDA pins on your Arduino. If you are using an older board than the Uno R3, you won’t be able to see these pins because they’re a recent addition. Instead, use analog pins 4 (SDA) and 5 (SCL). The last bit of the wiring is separate from the others and is to supply the power for the servo motors. Connect your power supply to the positive and negative terminals using a precision screwdriver.

![Figure 15-4: Diagram of a PWM Driver board wiring.](image)

After your circuit is assembled, you need the appropriate software to use it. To get it, follow these steps:


2. **From the project page, download the .zip file and unzip it to display the library folder.**
3. Rename the library folder to Adafruit_PWMServoDriver, and place it in your libraries folder in your Arduino sketch directory.

4. Restart Arduino and click File ➤ Examples to find the Adafruit_PWMServoDriver sketch.

Figure 15-5:
A schematic of a PWM Driver board.

Following is the code for this sketch:

```plaintext
/* **************************************************
This is an example for our Adafruit 16-channel PWM & Servo driver
Servo test - this will drive 16 servos, one after the other

Pick one up today in the adafruit shop!
------> http://www.adafruit.com/products/815

These displays use I2C to communicate, 2 pins are required to interface. For Arduino UNOs, thats SCL -> Analog 5, SDA -> Analog 4

Adafruit invests time and resources providing this open source code, please support Adafruit and open-source hardware by purchasing products from Adafruit!

Written by Limor Fried/Ladyada for Adafruit Industries.
BSD license, all text above must be included in any redistribution
**********************************************************/
```
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

// called this way, it uses the default address 0x40
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();
// you can also call it with a different address you want
//Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x41);

// Depending on your servo make, the pulse width min and max may vary, you
// want these to be as small/large as possible without hitting the hard stop
// for max range. You'll have to tweak them as necessary to match the servos you
// have!
#define SERVOMIN 150 // this is the 'minimum' pulse length count (out of 4096)
#define SERVOMAX 600 // this is the 'maximum' pulse length count (out of 4096)

// our servo # counter
uint8_t servonum = 0;

void setup() {
  Serial.begin(9600);
  Serial.println("16 channel Servo test!");
  pwm.begin();
  pwm.setPWMFreq(60);  // Analog servos run at ~60 Hz updates
}

// you can use this function if you'd like to set the pulse length in seconds
// e.g. setServoPulse(0, 0.001) is a ~1 millisecond pulse width. its not
// precise!
void setServoPulse(uint8_t n, double pulse) {
  double pulselength;
  pulselength = 1000000; // 1,000,000 us per second
  pulselength /= 60; // 60 Hz
  Serial.print(pulselength); Serial.println(" us per period");
  pulselength /= 4096; // 12 bits of resolution
  Serial.print(pulselength); Serial.println(" us per bit");
  pulse *= 1000;
  pulse /= pulselength;
  Serial.println(pulse);
  pwm.setPWM(n, 0, pulse);
}

void loop() {
  // Drive each servo one at a time
  Serial.println(servonum);
  for (uint16_t pulselen = SERVOMIN; pulselen < SERVOMAX; pulselen++) {
    pwm.setPWM(servonum, 0, pulselen);
  }
}
Part IV: Unlocking Your Arduino’s Potential

```c
} 
  delay(500); 
  for (uint16_t pulselen = SERVOMAX; pulselen > SERVOMIN; pulselen--) { 
    pwm.setPWM(servonum, 0, pulselen); 
  } 
  delay(500); 
  servonum ++; 
  if (servonum > 15) servonum = 0; 
}
```

Upload the sketch, and turn on your power supply. The servo moves to its maximum degree value and then back to its 0 position. If you have only one, there is a delay while it works through all 16 outputs. Monitor the current draw of your circuit, and if you are within the maximum current of the power supply, you can add more servos to the board to see the full effect. If you want to monitor the progression of the program, open the Serial Monitor to see the current motor as it works through each of them from 0 to 15.

If you don’t see any movement or see erratic behavior, double-check your wiring:

- Make sure you’re using the correct pin numbers.
- If you see a jerking movement from the servo(s), you are most likely not providing enough current. Monitor the current draw for any peaks in current and compare that to your supply.
- If you hear nasty grating noises from your servo, power it down immediately; you may have to adjust the SERVOMAX and SERVOMIN values. See how in the next section.

## Understanding the I²C PWM/Servo Driver Sketch

Before setup, two libraries are included that are essential to use this hardware. `Wire.h` is included to allow you to talk I²C with the board, and `Adafruit_PWMServoDriver.h` is included to perform more specific functions relating to the design of this board.

```c
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>
```

A new object, named `pwm`, is declared using a custom function from `Adafruit_PWMServoDriver.h`. This sets the address of the board, which
defaults to 0x40 if omitted. This is used as the zero value for choosing a board. If you wanted to choose a board with id 1, you’d use 0x41, as shown.

```c
// called this way, it uses the default address 0x40
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();
// you can also call it with a different address you want
//Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x41);
```

The next two statements use `#define` to set the minimum and maximum pulse length. Practically, this sets the degrees of rotation of the servos, so if you find that your servo is rotating too far or not far enough, you can adjust this value to fine-tune it.

```c
// Depending on your servo make, the pulse width min and max may vary, you // want these to be as small/large as possible without hitting the hard stop // for max range. You'll have to tweak them as necessary to match the servos you // have!
#define SERVOMIN 150 // this is the 'minimum' pulse length count (out of 4096)
#define SERVOMAX 600 // this is the 'maximum' pulse length count (out of 4096)
```

The term `uint8_t` is a C datatype, which is an unsigned integer 8 bits in length. Unsigned means it uses only positive values, and 8 bits means it holds values between 0 and 255. In this case, it is used to declare `servonum`, the variable that stores the current servo as the program counts through them. A standard `int` in Arduino code is 2 bytes of data (-32,768 to 32,767), which can also be called an `int16_t`. You can find more details about `int` here: http://arduino.cc/en/Reference/Int.

```c
// our servo # counter
uint8_t servonum = 0;
```

In `setup`, the serial port is opened with a baud rate of 9600, and the opening message “16 channel Servo test!” is sent to mark the start of the program. The object `pwm` (or board 0, as you know it) is initialized using `pwm.begin`, and the frequency of the servos is set to 60Hz.

```c
void setup() {
    Serial.begin(9600);
    Serial.println("16 channel Servo test!");

    pwm.begin();

    pwm.setPWMFreq(60); // Analog servos run at ~60 Hz updates
}
```

Next is the custom function `setServoPulse`, which sets the pulse length in seconds rather than hertz, but this function is not used in this example.
// you can use this function if you’d like to set the pulse length in seconds
// e.g. setServoPulse(0, 0.001) is a ~1 millisecond pulse width. it’s not
// precise!

void setServoPulse(uint8_t n, double pulse) {
    double pulselength;

    pulselength = 1000000;   // 1,000,000 us per second
    pulselength /= 60;   // 60 Hz
    Serial.print(pulselength);
    Serial.println(" us per period");

    pulselength /= 4096;  // 12 bits of resolution
    Serial.print(pulselength);
    Serial.println(" us per bit");

    pulse *= 1000;
    pulse /= pulselength;
    Serial.println(pulse);
    pwm.setPWM(n, 0, pulse);
}

In loop the current servo number is printed to the Serial Monitor for you
to see.

void loop() {
    // Drive each servo one at a time
    Serial.println(servonum);
}

Now that you know which servo you’re on, it’s time to move it. A for loop
is used to move the current servo from its minimum value to its maximum.
When that servo has reached its maximum value (SERVOMAX), it waits for
half a second and then a second for loop returns its value back from maxi-
mum value to minimum (SERVOMIN). When at its minimum, there is another
delay for a half second. The for loop increments the pulse one value at a
time to give a smooth movement. Notice that uint16_t is used to declare
the local variable pulselen; this is equivalent to an unsigned int in regular
Arduino code. You can find out more about unsigned int values at

for (uint16_t pulselen = SERVOMIN; pulselen < SERVOMAX; pulselen++) {
    pwm.setPWM(servonum, 0, pulselen);
}
    delay(500);
for (uint16_t pulselen = SERVOMAX; pulselen > SERVOMIN; pulselen--) {
    pwm.setPWM(servonum, 0, pulselen);
}
    delay(500);

After one servo has completed its movement, servonum is incremented
by one. If servonum goes higher than 16, it is sent back to 0 because it has
reached the last servo. There is another bit of C shorthand here, which is
why the if statement can be written all on one line without curly braces({ }).
This is a great sketch for testing communication with the board and allows you to control your servos in any way you want by using `pwm`. `setPWM(servonum, 0, pulselen)`. As with most libraries, this one is a work in progress, and you may find many questions without answers, as well see that many improvements and refinements could be made to it. The best place to ask further questions about this board and the library is on the Adafruit forum, where you can find other people asking the same questions and very helpful people trying to answer them. Head over to [http://forums.adafruit.com](http://forums.adafruit.com) and look in the Other Arduino products from Adafruit section to find out more.

**Buying Servo Motors**

Now that you have a way to control lots of servos, you need to know where to shop for them and what to look for. There are a variety of sites that sell servo motors online, so a quick Google returns lots of options. The two main applications for servos are in robotics and remote controlled aircraft, so any shop or site that supplies these is likely to have a range of servos that are specific to the application.

The strength of the servos, or the twisting force, is called *torque* and is usually measured in kilograms per centimetre (kg/cm) or pounds per inch (lb/in). Servos can range in torque from miniature servos designed to move the ailerons on a plane that provide around 0.2 kg/cm, to monster servos that can be used as sail winches on boats (9.8 kg/cm). Just for comparison, the torque created by an adult male hand is approximately 8 kg/cm (as NASA’s human performance charts show at [http://msis.jsc.nasa.gov/sections/section04.htm](http://msis.jsc.nasa.gov/sections/section04.htm)).

Obviously, the amount of torque created depends on the strength of the servo motor and the quality of the parts. A wide variety of servo motors operate from 4V to 12V and cater to most needs. The main difference is the amount of current that is required when the servo is pulling or pushing, which is referred to as being “under load.” When the servo is exerting itself, it can require much more voltage than when it has no load, so it is worth testing individual servos with a multimeter connected to the power line to monitor the current draw (as described in Chapter 5) and leaving a reasonable margin for any additional current. It’s also worth noting that the average power supply is only 70–80 percent efficient, meaning that a 1A power supply
may only really be supplying 700–800mA. Although it may be possible to get 1A out of it, it likely deteriorates quickly and so should be seen as the maximum and not the recommended supply under typical use.

Power doesn’t always dictate price, and with servos, you find a great variety of seemingly less powerful servos for a high price. This is often because of the physical build of the servo itself. The gears inside the servo are of great importance, and depending on the material used, they can make a huge difference to the price and performance. Most hobby servos are for relatively small loads and use nylon gears. This is great because nylon exerts almost no wear on other nylon parts, even after thousands of uses. Nylon is self-lubricating, requiring no additional grease. More serious servos use metal gears. These can take much greater loads because they are not as flexible as nylon and are less likely to break if they are under an excessive load.

Servos can also either be digital or analog, which relates to the way the motor is controlled. Both types of servo have the same parts and the same three wires, but digital motors have a microprocessor that can analyze the signals sent to the servo and send pulses to the servo many times faster than a traditional servo with analog electronics. With more signals per second, the servo has a much faster response time. Because each pulse is a voltage, the servo consumes a lot more current per second and, therefore, also is capable of providing more torque and supporting a greater load. The drawback of digital servos is the price and power consumption, but if you can justify it, you won’t be disappointed.

There are a variety of brands out there to choose from, so the best thing to do first is to research them. An excellent site that’s aptly named www.servodatabase.com gives you a good idea of the cost versus performance of most servos, as well as a good indication of the most popular ones.

If you’re looking for an affordable, basic servo that’s a bit bigger than the hobby servos in most kits, you can’t go far wrong with a Futaba S3003 found at http://www.gpdealera.com/cgi-bin/wgainf100pgm?I=FUTM0031. In this case, you cannot buy the motor directly from the manufacturer, so you have to shop around. Two such suppliers are Servo Shop in the United Kingdom (http://www.servoshop.co.uk/index.php?pid=FUTS3003) and Servo City in the United States (http://www.servocity.com/html/s3003_servo_standard.html). Happy shopping!
Other Uses for I\textsuperscript{2}C

Servo motors aren’t the only kind of object that you can control using \textsuperscript{I}\textsuperscript{2}C. Other products are ready to be hooked up to an Arduino to help you build a huge installation. One such product is the Addressable LED Ribbon (https://www.sparkfun.com/products/11272), shown in Figure 15-6. LED ribbon is a flexible tape of LEDs that can be stuck to a surface to allow lights to cover a huge area. The ribbon itself is a type of flexible PCB that has surface-mounted LEDs, resistors, and contacts, and has only recently become widely available for an affordable price. You may have seen it in venues that have mood lighting around the edges of the room.

Just like individual LEDs, this can either come as a single strip of an individual color or RGB, allowing you to tailor the color to the use. Most LED ribbon is a single circuit, meaning that you can change the brightness or color of the whole strip, but not affect the individual LEDs. Addressable LED Ribbon, however, is different because it allows you to control each LED individually, allowing full control of the brightness and color and creating a potential for very cool animation.

There are many great online tutorials for integrating addressable LED ribbon in your project. Check out the SparkFun (https://www.sparkfun.com/products/11272) and Adafruit (http://learn.adafruit.com/digital-led-strip) product pages for more details.
Part V

Sussing Out Software

The 5th Wave
By Rich Tennant

“Roger! Check the sewing machine’s connection to the PC. I’m getting e-mails stitched across my curtains again.”
In this part . . .

So, by this stage of the book, you must feel that you have a pretty good grip for the physical world, wiring, soldering, and uploading code. But what if I told you there was a whole other virtual world out there as well? Interested? It’s possible to combine your (now copious) knowledge of electronics with software on your computer to create stunning interactive visuals in the virtual world that exists on your computer, or to use data from software, your computer, or the Internet to create light, sound, or motion in the real world.
In the previous chapters, you learn all about using Arduino as a stand-alone device. A program is uploaded onto the Arduino and it carries out its task ad infinitum until it is told to stop or powered down. You are affecting the Arduino by simple, clear, electrical signals, and as long as there are no outside influences or coding errors, and if the components last, the Arduino reliably repeats its function. This simplicity is extremely useful for many applications and allows the Arduino to not only serve as a great prototyping platform but also work as a reliable tool for interactive products and installations for many years, as it already does in many museums.

Although this simplicity is something to admire, many applications are outside the scope of an Arduino’s capabilities. One of these (at least for now) is running computer software. Although the Arduino is basically a computer, it’s not capable of running comparably large and complex computer programs in the same way as your desktop or laptop. Many of these programs are highly specialized depending on the task that you’re doing. You could benefit hugely if only you could link this software to the physical world in the same way your Arduino can.

Because the Arduino can connect to your computer and be monitored over the serial port, other programs may also be able to do this, in the same way that your computer talks to printers, scanners, or cameras. So by combining the physical world interaction capabilities of your Arduino with the data crunching software capabilities of your computer, you can create projects with an enormous variety of inputs, outputs, and processes.

Many specific programs are made for specific tasks, but until you want to specify, it’s best to find software that you can experiment with — that is, be a jack-of-all-trades in the same way that your Arduino is for the physical world. Processing is a great place to start.
In this chapter, you learn about Processing, the sister project that was in the first stages of development around the same time as Arduino. Processing is a software environment that, in the same way that an Arduino is used to test circuits quickly, can be used to “sketch” programs quickly. Processing is a great piece of open source software to learn about, and because of its similarities to Arduino, learning it is even easier.

**Looking Under the Hood**

An Arduino can communicate over its serial port as a serial device, which can be read by any program that can talk serial. Many programs are available, but Processing is one of the most popular.

Processing has an enormous breadth of applications—ranging visualizing data to creating generative artwork to performing motion capture using your webcam for digital performances. These are just a few niches, but you can find a wealth of examples on the Processing exhibition to wet your digital whistle. Head over to [http://processing.org/exhibition/](http://processing.org/exhibition/) to check them out.

Processing is a Java-based language that looks very similar to C (on which Arduino code is based) and C++. It is available for Windows, Mac OS, and Linux. Ben Fry and Casey Reas developed Processing to allow artists, designers, or anyone to experiment with code, rather than just developers and engineers. In the same way that ideas are sketched out, Processing is designed to sketch software. Programs can be quickly developed and adapted without a huge investment of time.

Processing is a text-based IDE very similar to that of Arduino (in fact, it was “borrowed” by the Arduino team when the Arduino integrated development environment [IDE] was in development). A window (see Figure 16-1) displays the Java applet that the code creates. As with Arduino, the strength of Processing is the vast community that shares and comments on sketches, allowing the many participants to benefit from a diverse array of creative applications. Processing is open source and allows users to modify the software as well as use it.

In this chapter, you learn how to get started with Processing, but for more information, head over to the Processing site at [http://processing.org/](http://processing.org/).

Many other programming languages exist that can interface with Arduino. I describe Max/Pure Data and OpenFrameworks in sidebars in this chapter. For a list of even the most obscure, check the Arduino Playground at [http://arduino.cc/playground/Main/Interfacing](http://arduino.cc/playground/Main/Interfacing).
Max/PureData

Max (also previously known as Max/MSP) is a visual programming language with a vast variety of uses but is most commonly used for audio, music and sound synthesis applications. It is available for Windows, Mac OS, Linux, and other, more obscure operating systems.

Unlike traditional text-based programming languages, Max uses a graphical user interface to connect visual objects to one another in the same way that traditional synthesizers could be “patched” using wires to connect the various functions of the instrument. Software company Cycling 74 released the commercial software Max in 1990 based on earlier work by Miller Puckette to create a system for interactive computer music. Although the software is not open source, the application programming interface (API) allows third parties to make their own extensions to the software for specific uses. Miller Puckette also developed a free and open source, but completely redesigned, version of Max called PureData.

You can find more information about Max and PureData on their respective web pages: http://cycling74.com/products/max and http://puredata.info. To start communicating between Max and Arduino, check out the aptly named Maxuino (http://www.maxuino.org/archives/category/updates), and for PureData and Arduino, check out Pduino (http://at.or.at/hans/pd/objects.html).

(continued)
Part V: Sussing Out Software

362

Installing Processing

Processing is free to download from http://processing.org/. To download Processing, go to the Download page and select your platform. At the time of writing, Processing was version 2.0 Beta 5 and supported Mac OS X, Windows 32-bit and 64-bit, and Linux 32-bit and 64-bit. Remember that things may change between when I put these words down and when you get started.

To install Processing:

- **On a Mac:** The .zip file unzips automatically and can be placed in your application folder at ~/Applications/Processing. Or you can place the application on the desktop. From there you can drag Processing to the dock for easy access or create a desktop alias.

- **On Windows:** Unzip the .zip file and place the Processing folder on your desktop or in a sensible location such as your Program Files folder: C:/Program Files/Processing/. Create a shortcut to Processing.exe and place it somewhere convenient, such as on your desktop or in the Start menu.

Other helpful links are on the Arduino playground at (http://www.arduino.cc/playground/interfacing/MaxMSP) and http://www.arduino.cc/playground/Interfacing/PD).
OpenFrameworks, an open source C++ toolkit for experimenting with code, is actively developed by Zachary Lieberman, Theo Watson, and Arturo Castro, as well as other members of the openFrameworks community. OpenFrameworks runs on Windows, Mac OS, Linux, iOS, and Android. OpenFrameworks is not based on Java as Processing is; it is a C++ library that is designed to be the bare bones for getting started with audio-visual applications.

OpenFrameworks is especially powerful with graphics, allowing you to use OpenGL easily for intensive rendering or video applications. In contrast to Processing, Max, and PureData, OpenFrameworks is not its own language; it is, in fact, a collection of open source libraries, known as a software framework — hence the name.

Because OpenFrameworks does not have its own IDE, the software used to write and compile the code depends on the platform. This feature can make getting started difficult because there is no centrally controlled IDE for continuity. The benefit is that the C++ is highly versatile and can be used on almost any platform you can think of, including mobile operating systems.

Taking a look at Processing

When you have Processing installed, run the application. Processing opens with a blank sketch (see Figure 16-2) similar to the Arduino window and is divided into five main areas:

- Toolbar with buttons
- Tabs
- Text editor
- Message area
- Console

The blank sketch also contains a menu bar for the main Processing application, which gives you drop-down menus to access the preferences of the processing application, load recent sketches and import libraries, and perform many other functions.

Figure 16-2: The Processing application is similar to but different from the Arduino one.
Here’s an overview of the Processing toolbar:

- **Run**: Executes or runs the code in the text editor as an applet (small application) in a new window. The keyboard shortcuts for this is Ctrl+R for Windows and Cmd+R for Mac OS.

- **Stop**: Stops the code from running and closes the applet window.

- **New**: Opens a new, blank sketch with a dated name and a character to differentiate between sketches, such as sketch_121030a.

- **Open**: Lets you select from a directory, recent sketches, or the Examples folder.

- **Save**: Saves the current sketch. When saving, assigning a descriptive name rather than the preassigned name is best.

- **Export Application**: Allows you to export your Processing sketch as a self-contained application. This is useful when you need to run the application at start-up or if you need to distribute the code to people who don’t have Processing installed.

- **Mode**: Allows you to change mode between Java (standard), Android (mobile and tables), and JavaScript (online applications). This capability is a new development in the latest release. You can find more details on these modes here: [http://wiki.processing.org/w/Android](http://wiki.processing.org/w/Android) and [http://wiki.processing.org/w/JavaScript](http://wiki.processing.org/w/JavaScript).

- **Tabs**: Organizes multiple files in a Processing sketch. Use tabs in larger programs to separate objects from the main sketch or to incorporate look-up tables of data into a sketch.

- **Text editor**: Enters code into the sketch. Recognized terms or functions are highlighted in appropriate colors for clarity. The text editor is the same as that in the Arduino IDE.

- **Message area**: Displays errors, feedback, or information about the current task. You might see a notification that the sketch saved successfully, but more often than not, the message shows where errors are flagged.

- **Console**: Displays more details on your sketch. You can use the `println()` function here to display the values in your sketch; additional detail on errors is also shown.

### Trying Your First Processing Sketch

Unlike with Arduino, you don’t need an extra kit to get going with Processing, which makes Processing extremely useful for learning about coding because you can enter a line or two of code, click Run, and see what you’ve done.
Part V: Sussing Out Software

Start your first sketch with these steps:

1. Press Ctrl+N (in Windows) or Cmd+N (on a Mac) to open a new sketch.
2. Click in the text editor and enter this line of code:
   
   ```
   ellipse(50,50,10,10);
   ```

3. Click the Run button.

   A new applet window opens, showing a white circle in the middle of a gray box, as in Figure 16-3.

   Well done! You’ve just written your first Processing program.

   ![Figure 16-3: A Processing sketch that draws an ellipse with equal dimensions, also known as a circle.](image)

   Done admiring your circle? That line of code draws an ellipse. An ellipse normally is not circular, but you gave it the parameters to make a circle. The word `ellipse` is highlighted in orange in the text editor, indicating that it is a recognized function. The first two numbers are the coordinates of the ellipse, which in this case are 50, 50. The unit of the numbers is in pixels. Because the default window is 100 x 100 pixels, coordinates of 50, 50 put the ellipse in the center. The 10, 10 values indicate the width and height of the ellipse, giving you a circle. You could write the function as

   ```
   ellipse(x, y, width, height)
   ```

   The coordinates for the ellipse (or any shape or point, for that matter) are written as x and y. These indicate a point in two-dimensional (2D) space, which in this case is a point measured in pixels on your screen. Horizontal positions are referred to as the x coordinate; vertical positions are the y
coordinate. Depth used in 3D space is referred to as z. Add the following line of code, just above `ellipse()` statement:

```java
size(300,200);
```

Click the Run button and you get a rectangular window with the ellipse in the top left of the window, as shown in Figure 16-4. The `size()` function is used to define the size of the applet window in pixels, which in this case is 300 pixels wide and 200 pixels high. If your screen isn’t like Figure 16-4, you may have put the statements in the wrong order. The lines of code are read in order, so if the ellipse code is first, the blank window is drawn over the ellipse. And with a rectangular window, you see that the coordinates are measured from the top left.

Coordinates are measured on an invisible a grid with the center point at 0, 0 for 2D (or 0, 0, 0 for 3D), which is referred to as the origin. This is based on the Cartesian coordinate system, which you may have studied in school. Numbers can be positive or negative, depending on which side of the origin they are on. On computer screens, origin is at the top left because pixels are drawn from top left to bottom right, one row at a time (check out Figure 16-5). This means that the statement `size(300,200)` draws a window 300 pixels from left to right on the screen and then 200 pixels from top to bottom.
Drawing shapes

To gain a better understanding of the possibilities you have in drawing shapes, look at a few basic shapes:

✓ **point()**

A single point is the most basic shape and is useful for making up more complex shapes. Write this code and then click the Run button. Look closely and you see a single black pixel in the center of the display window (see Figure 16-6). That is the point that your code drew.

```javascript
size(300,200);
point(150,100);
```

Point can also be written as:

```javascript
point(x,y);
```

✓ **line()**

A line is made by connecting two points, which is done by defining the start and end points. Write the code to generate a screen like the one in Figure 16-7:

```javascript
size(300,200);
line(50,50,250,150);
```

You can also write a line written as

```javascript
line(x1,y1,x2,y2);
```
Figure 16-6: If you look closely, you can see the point.

Figure 16-7: A line between two points.
✓ **rect()**

You can draw a rectangle a number of different ways. In this first example, a rectangle is drawn by identifying the starting point and then the width and height of the rectangle. Write the following code to draw a rectangle in the center of your display window:

```plaintext
size(300, 200);
rect(150, 100, 50, 50);
```

In this case, you have a rectangle that starts in at point 150,100 in the center of the display window. That is the top-left corner of the rectangle, and from there it has a width of 50, which extends the rectangle to the right of the window, and a height of 50, which extends to the bottom of the window. This function is particularly useful if you want the size of the rectangle to remain constant but change the position of the rectangle. You could also write this as

```plaintext
rect(x, y, width, height);
```

When drawing rectangles, you can choose among different modes (see Figure 16-8). If the mode is set to center, the rectangle is drawn centered around a point instead of being drawn from that point. Write the following code and you see that the same values display a different position when `rectMode` is changed to CENTER.

```plaintext
rectMode(CENTER);
size(300, 200);
rect(150, 100, 50, 50);
```

You can see that the rectangle is now centered in the display window. The shape extends equally from the center point both left to right and top to bottom. You can also write this as

```plaintext
rect(x, y, width, height);
```

You can also draw a rectangle by declaring two diagonally opposite corners. Write the following code, this time with `rectMode` set to CORNERS.

```plaintext
rectMode(CORNERS);
size(300, 200);
rect(150, 100, 50, 50);
```

You see a rectangle that is quite different from the others because it starts at the same point in the center, 150,100, but ends at point 50,50, effectively doubling back on itself. You can also write this as

```plaintext
rect(x1, y1, x2, y2);
```

✓ **ellipse()**

The first item covered in this chapter was `ellipse`, which can be used to simply draw an ellipse. Write out the following code to draw an ellipse in the center of the display window.

```plaintext
ellipse(150, 100, 50, 50);
```
It is clear that the default mode for `ellipse` is CENTER, unlike `rect`, which is CORNER. This can also be written as:

```java
ellipse(x, y, width, height);
```

As with `rectMode()` it’s possible to set different modes (see Figure 16-9) for drawing ellipses using `ellipseMode()`. Write out the following code to draw an ellipse from its corner instead of its center.

```java
ellipseMode(CORNER);
size(300, 200);
ellipse(150, 100, 50, 50);
```

This draws an ellipse from starting from its top-left corner with a width of 50 and a height of 50. This can also be written as:

```java
ellipse(x, y, width, height);
```

It is also possible to draw an ellipse by specifying multiple corners. Write out the following code to change the `ellipseMode` to CORNERS.

```java
ellipseMode(CORNERS);
size(300, 200);
ellipse(150, 100, 50, 50);
```

Similarly to `rectMode(CORNERS)` you see that the ellipse doubled back on itself. The first corner is the center point of the sketch and the second is at point 50,50.
Changing color and opacity

Now that you have an understanding of shapes, it’s time to affect their appearance. The simplest way to do this is with color and opacity. In fact, with this it is possible to see through each of the shapes and mix colors by overlapping shapes.

Here are the details:

 ✓ **background(0)**

   This function changes the background of your sketch. You can choose grayscale values, or color.

   - Grayscale

     Open a new sketch, and simply type the following code to change the default gray window to black.

     ```
     background(0);
     ```

     Change 0 to 255 to change the color to white.

     ```
     background(255);
     ```

     Any value between 0 (black) and 255 (white) is a grayscale value. The reason that this range is 0 to 255 is that there are 8 bits of data in a byte (see Chapter 14), meaning that you need one byte to store a grayscale color value.
• Color

To liven things up a bit you can add color to your sketch background. Instead of 8-bit grayscale you can use 24-bit color, which is 8-bit red, 8-bit green, and 8-bit blue. The color of the background is defined with three values instead of one.

```
background(200,100,0);
```

This gives you an orange background. The orange background is composed of a red value of 200, a green value of 100, and a blue value of 0. There are several color modes, but in this case this line of code can be interpreted as:

```
background(red,green,blue);
```

✓ `fill()`

Want to change the color of the shapes you draw? Use `fill` to both set color and control the shape’s opacity:

• Color

`fill` sets the color for any shape that is drawn after it. By calling `fill` multiple times it is possible to change the color of several different shapes. Write out the following code to draw three ellipses with different colors, as in Figure 16-10.

```
background(255);
noStroke();

// Bright red
fill(255,0,0);
ellipse(50,35,40,40);

// Bright green
fill(0,255,0);
ellipse(38,55,40,40);

// Bright blue
fill(0,0,255);
ellipse(62,55,40,40);
```

The background is set to white (255), and the function `noStroke` removes border lines from the shapes (you can comment it out to see the effect). The first circle to be drawn is red. You can see this in the applet because the other two circles overlap it. The red value is the highest possible (255), as is the second for green and the third for blue. If another shape was drawn at the end of the code it would be the same strong blue as that is the last `fill` value.
• Opacity

It’s also possible to affect the opacity of the colors, creating semi-transparent shapes. By adding a fourth value to the `fill` function you can set the opacity from 0 (fully transparent) to 255 (solid color). Update the previous code with the following circles to give the circles transparency.

```cpp
background(255);
noStroke();

// Bright red
fill(255,0,0,100);
ellipse(50,35,40,40);

// Bright green
fill(0,255,0,100);
ellipse(38,55,40,40);

// Bright blue
fill(0,0,255,100);
ellipse(62,55,40,40);
```

### Playing with interaction

All of this is fun, but it’s very static. In this example you learn how to quickly inject some life into your sketches using your mouse as an input. To do this you need to constantly update the sketch by looping through it over and over again, sending new values in each loop. Write out the following code to create an interactive sketch.
This code draws an ellipse centered on your mouse pointer coordinates, so when you move your mouse you leave a trail of ellipses behind, as shown in Figure 16-11. The functions `mouseX` and `mouseY` are highlighted blue in the text editor and take the coordinates of your mouse pointer within the display window. The values are the number of pixels horizontally and vertically.

With luck, this code looks familiar to you. Instead of Arduino’s `void setup` and `void loop`, Processing uses `void setup` and `void draw`. These work in almost exactly the same way: `setup` runs once at the start of the sketch; `loop` and `draw` run forever or until they are told to stop.

Change the sketch slightly, and you can cover up all those previous ellipses to only show the most recent (Figure 16-12).
There is much more to Processing that I can’t cover in this book, but these few points should be enough to gain a basic comprehension of how code relates to visuals onscreen. You can find a wealth of examples, both on the Processing site and included in the Processing software. The best approach is always to run these examples and then tweak the values to see what happens. By experimenting, you learn what’s going on much more quickly, and with no electronics, you are at much less risk of breaking things.
In the previous chapter, you learn the basics of Processing and its similarities to and differences from Arduino. This chapter is all about combining both of these tools to integrate the virtual and physical worlds. These few exercises teach you the basics about sending and receiving data in both Processing and Arduino. You can build on this knowledge to create your own projects, maybe to generate some awesome onscreen visuals from your sensors or to turn on a light every time someone mentions you on Twitter.

Making a Virtual Button

In this example, you learn how to make an onscreen button in Processing that affects a physical LED on your Arduino. This is a great sketch to get started with interactions between computers and the real world, and between an Arduino and Processing.

You need:

✓ An Arduino Uno
✓ An LED

The setup is simple for this introduction to Arduino and Processing, requiring only a single LED.

As shown in Figures 17-1 and 17-2, insert the long leg of the LED into pin 13 and the short leg into GND. If you don’t have an LED, you can simply monitor the onboard LED marked L.
**Setting up the Arduino code**

After your circuit is assembled, you need the appropriate software to use it. From the Arduino menu, choose File ➪ Examples ➪ 04.Communication ➪ PhysicalPixel to find the sketch.

This sketch contains both Arduino code and the relevant Processing code for the sketch to work (it also has a variation in Max 5). The code beneath the Arduino code is commented out to avoid interfering with the Arduino code.
In older versions of Arduino, the sketch files ended with .pde, which is the Processing suffix. This caused confusion, so now the Arduino suffix is .ino. Different suffixes make it possible to have the Arduino sketch and the Processing sketch in the same place. If you try to open a .pde in Arduino, the application assumes that it is an old Arduino sketch and asks whether you want to change the suffix to .ino.

```c
/*
 Physical Pixel

An example of using the Arduino board to receive data from the computer. In this case, the Arduino board turns on an LED when it receives the character 'H', and turns off the LED when it receives the character 'L'.

The data can be sent from the Arduino serial monitor, or another program like Processing (see code below), Flash (via a serial-net proxy), PD, or Max/MSP.

The circuit:
* LED connected from digital pin 13 to ground

created 2006
by David A. Mellis
modified 30 Aug 2011
by Tom Igoe and Scott Fitzgerald

This example code is in the public domain.

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

const int ledPin = 13;  // the pin that the LED is attached to
int incomingByte;    // a variable to read incoming serial data into

void setup() {
  // initialize serial communication:
  Serial.begin(9600);
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // see if there's incoming serial data:
  if (Serial.available() > 0) {
    incomingByte = Serial.read();
    // if it's a capital H (ASCII 72), turn on the LED:
    if (incomingByte == 'H') {
      digitalWrite(ledPin, HIGH);
    }
    // if it's an L (ASCII 76) turn off the LED:
  }
```

Now go through the steps to upload your sketch.

With the Arduino set up to receive a message from Processing, you need to set up the Processing sketch to send a signal message over the same serial port to your Arduino.

Setting up the Processing code

This code is available within multiline comment markers (/* */) at the bottom of the Arduino PhysicalPixel sketch. Copy the code within the comment markers, paste it into a new Processing sketch, and save it with an appropriate name, such as PhysicalPixel.

```java
// mouseover serial
// Demonstrates how to send data to the Arduino I/O board, in order to turn ON a light if the mouse is over a square and turn it off if the mouse is not.

// created 2003-4
// based on examples by Casey Reas and Hernando Barragan
// modified 30 Aug 2011
// by Tom Igoe
// This example code is in the public domain.

import processing.serial.*;

float boxX;
float boxY;
int boxSize = 20;
boolean mouseOverBox = false;

Serial port;

void setup() {
    size(200, 200);
    boxX = width/2.0;
    boxY = height/2.0;
    rectMode(RADIUS);

    // List all the available serial ports in the output pane.
    // You will need to choose the port that the Arduino board is
```
Chapter 17: Processing the Physical World

// connected to from this list. The first port in the list is
// port #0 and the third port in the list is port #2.
println(Serial.list());

// Open the port that the Arduino board is connected to (in this case #0)
// Make sure to open the port at the same speed Arduino is using (9600bps)
port = new Serial(this, Serial.list()[0], 9600);

}

void draw()
{
background(0);

// Test if the cursor is over the box
if (mouseX > boxX-boxSize && mouseX < boxX+boxSize &&
mouseY > boxY-boxSize && mouseY < boxY+boxSize) {
mouseOverBox = true;
// draw a line around the box and change its color:
stroke(255);
fill(153);
// send an 'H' to indicate mouse is over square:
port.write('H');
}
else {
// return the box to its inactive state:
stroke(153);
fill(153);
// send an 'L' to turn the LED off:
port.write('L');
mouseOverBox = false;
}

// Draw the box
rect(boxX, boxY, boxSize, boxSize);
}

Click the Run button to execute the Processing sketch, and an applet appears. The applet shows a black background with a gray square in the middle, representing your virtual button (shown in Figure 17-3). If you move your mouse over the gray square (or pixel), you can see that its edges turn white. If you then look at your Arduino, you see that whenever your mouse hovers over the gray square, the LED on your board illuminates, giving you a physical representation of your pixel.

If your LED doesn’t light, double-check your wiring:

✓ Make sure that you’re using the correct pin number.
✓ Make sure that the LED legs are the correct way around.
Check that your Arduino code uploaded correctly and that your Processing code has no errors. Note that you cannot upload while the Processing sketch is communicating with your Arduino, so you need to stop the sketch before uploading.

Figure 17-3: Your Processing applet displaying the virtual pixel.

Understanding the Processing PhysicalPixel sketch

Wherever possible, dividing projects into their elements is a good idea. You may have many inputs and outputs, but if you deal with them one at a time, they are easier to understand and easier to troubleshoot. Because the Processing side is the input, you’re wise to start with that.

The structure of a Processing sketch is similar to Arduino. You include libraries and declare variables at the start of the sketch and set fixed values or initializations in setup. The draw function then repeats its process until told otherwise.

Processing uses libraries to add functionality in the same way as Arduino does. In this case, a serial communication library is needed to talk to the Arduino. In Arduino, this library is included by using #include <libraryName.h>, but in Processing, you use the import keyword, followed by the name and the syntax * to load all the related parts of that library.

```java
import processing.serial.*;
```

A float is a floating-point number, one with a decimal place, such as 0.5, 10.9, and so on. In this case, two floating-point numbers are declared, boxX and boxY. These are the coordinates for the location of the box.

```java
float boxX;
float boxY;
```

Next, boxSize defines the size of the box as an integer, or whole number. Because it is square, only one value is needed.

```java
int boxSize = 20;
```
A Boolean (which can be only true or false) is used to communicate that the mouse is over the box. This is set to start as false.

```java
boolean mouseOverBox = false;
```

The last thing to do is create a new serial port object. Many serial connections could be in use by your computer, so it’s important that each one be named so that it can be used as needed. In this case, you are using only one port. The word `serial` is specific to the serial library to indicate that you want to create a new serial object (connection), and the word `port` is the name of the object (connection) used to refer to the port from this point on. Think of it as giving your cat a collar. If too many cats are in a room, they all look fairly similar and follow the same general rules, but they are still all individual. If you put a colored collar on each cat with its name printed on it, you can easily identify which one is which.

```java
Serial port;
```

In `setup`, the first item to define is the size of the display window. This is set to 200 pixels square.

```java
void setup() {
    size(200, 200);
}
```

The variables for `boxX` and `boxY` are set to be proportional to the width and height of the display window. They are always equal to half the width and height, respectively. Next, `rectMode` is set to `RADIUS`, which is similar to `CENTER`, but instead of specifying the overall width and height of the rectangle, `RADIUS` specifies half the height and width. (`CENTER` could be interpreted as diameter in that respect.) Because the coordinates of the box are centered and are aligned to the center point of the display window, the box is also perfectly centered.

```java
boxX = width/2.0;
boxY = height/2.0;
rectMode(RADIUS);
```

Your computer may have a lot of serial connections, so it’s best to print a list of them to locate your Arduino.

```java
println(Serial.list());
```

The most recent port usually appears at the top of this list in position 0, so if you’ve just plugged in your Arduino, the first item is likely the one you want. If you are not using the `Serial.list` function, you could replace `Serial.list()[0]` with another number in the list, which will be printed in the console. You can also replace `Serial.list()[0]` with the exact name of the port, such as `/dev/tty.usbmodem26221` or `COM5`. Specifying the exact name is also useful if you have multiple Arduinos connected to the same computer. The number 9600 refers to the baud rate, the rate at which you are communicating with the Arduino.
If the baud rate number is not the same on both ends (the sending and receiving end), the data will not be received.

```java
port = new Serial(this, Serial.list()[0], 9600);
}
```

In `draw`, the first task is to draw a black background.

```java
void draw()
{
    background(0);
}
```

Processing uses the same (or similar) conditionals as Arduino. This `if` statement tests the mouse value to see whether it is over the box area. If `mouseX` is greater than the box coordinate (center), minus the size of the box (half the box width), and less than the box coordinate (center), plus the size of the box (half the box width), the horizontal position is over the box. This statement is used again with the vertical position, using `AND` statements (`&&`) to add to the conditions of the `if` statement. Only if all these are true can the Boolean `mouseOverBox` be declared true.

```java
// Test if the cursor is over the box
if (mouseX > boxX-boxSize && mouseX < boxX+boxSize &&
    mouseY > boxY-boxSize && mouseY < boxY+boxSize) {
    mouseOverBox = true;
}
```

To indicate that `mouseOverBox` is true, the code draws a white line around the box. Rather than requiring that another box be drawn, the white line appears simply by changing the stroke or outline value (`stroke` is a term common in most graphics software). The stroke is set to 255, which outlines the box in white.

```java
// draw a line around the box and change its color:
stroke(255);
```

Fill is set to 153, a mid-gray, which colors the next object that is drawn.

```java
fill(153);
```

Then the all important communication is sent. The statement `port.write` is similar to `Serial.print` but is used for writing to a serial port in Processing. The character sent is `H` for high.

```java
// send an 'H' to indicate mouse is over square:
port.write('H');
}
```

The `else` statement tells Processing what to do if the mouse is not over the box.
The stroke value is set to the same mid-gray as the box. The box fill color remains the same whether active or inactive.

```
// return the box to its inactive state:
stroke(153);
fill(153);
```

The character L is sent to the serial port to signify that the LED should be set low.

```
// send an 'L' to turn the LED off:
port.write('L');
```

The Boolean `mouseOverBox` is set to false.

```
mouseOverBox = false;
}
```

Finally, the box (technically a rectangle) itself is drawn. Its coordinates are always centered, and its size remains the same; the only difference is that the color applied by the if statement. If the mouse is over the box, the stroke value is changed to white (active), and if not, the stroke value is set to the same gray as the box and appears to not be there (inactive).

```
// Draw the box
rect(boxX, boxY, boxSize, boxSize);
```

# Understanding the Arduino

**Physical Pixel sketch**

In the preceding section, you find out how the Processing side works, providing a signal. The signal is sent over the serial connection to your Arduino, so in this section I explain what the Arduino code does with it. The Arduino code for this example is relatively simple compared to other examples in this book and is great for understanding how a serial connection is made. I always recommend starting with this sketch for any Processing-to-Arduino communication. It’s great as a foundation to make sure that your hardware and software are working, and you can then build on it or adapt it as needed.

First, the constant and variable values are declared. The LED pin — pin 13 — is the LED output and does not change, so it is marked as a constant. The `incomingByte` value does change and is declared as an integer. Note that it is declared as an integer (int), not a character (char). I explain why a bit later.

```
const int ledPin = 13; // the pin that the LED is attached to
int incomingByte;      // a variable to read incoming serial data into
```