CHAPTER 6

USB and Serial Communication

Parts You'll Need for This Chapter

Arduino Uno
Arduino Leonardo
USB cable (A to B for Uno)
USB cable (A to Micro B for Leonardo)
LED
RGB LED (common cathode)
150Ω resistor
220Ω resistor (×3)
10kΩ resistor (×2)
Pushbutton
Photoresistor
TMP36 temperature sensor
Two-axis joystick (SparkFun, Parallax, or adafruit suggested)
Jumper wires
Breadboard
Potentiometer

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, video, and other digital content for this chapter can be found at `www.exploringarduino.com/content/ch6`.

In addition, all code can be found at `www.wiley.com/go/exploringarduino` on the Download Code tab. The code is in the chapter 06 download and individually named according to the names throughout the chapter.

Perhaps the most important part of any Arduino is its capability to be programmed directly via a USB serial port. This feature enables you to program the Arduino without any special hardware, such as an AVR ISP MKII. Ordinarily, microcontrollers rely on a dedicated piece of external hardware (such as the MKII) to serve as a *programmer* that connects between your computer and the microcontroller you are trying to program. In the case of the Arduino, this programmer is essentially built into the board, instead of being a piece of external hardware. What's more, this gives you a direct connection to the ATMega's integrated Universal Synchronous/Asynchronous Receiver and Transmitter (USART). Using this interface, you can send information between your host computer and the Arduino, or between the Arduino and other serial-enabled components (including other Arduinos).

This chapter covers just about everything you could want to know about connecting an Arduino to your computer via USB and transmitting data between the two. Different Arduinos have different serial capabilities, so this chapter covers each of them, and you build sample projects with each serial communication technology to get yourself acquainted with how to take advantage of them as best as possible. Note that, as a result of this, the parts list includes several types of Arduinos. Depending on which Arduino you are trying to learn about, you can pick and choose which sections to read, which examples to explore, and which parts from the parts list you actually need for your Arduino explorations.

Understanding the Arduino's Serial Communication Capabilities

As already alluded to in the introduction to this chapter, the different Arduino boards offer lots of different serial implementations, both in terms of how the hardware implements the USB-to-serial adapters and in terms of the software support for various features. First, in this section, you learn about the various serial communication hardware interfaces offered on different Arduino boards.

NOTE To learn all about serial communication, check out this tutorial: `www.jeremyblum.com/2011/02/07/arduino-tutorial-6-serial-communication-and-processing/`. You can also find this tutorial on the Wiley website shown at the beginning of this chapter.

To begin, you need to understand the differences between serial and USB. Depending on how old you are, you might not even remember serial (or technically, RS-232) ports, because they have been primarily replaced by USB. Figure 6-1 shows what a standard serial port looks like.

Figure 6-1: Serial port

The original Arduino boards came equipped with a serial port that you connected to your computer with a 9-pin serial cable. Nowadays, few computers still have these ports, although you can use adapters to make DB-9 (the type of 9-pin connector) serial ports from USB ports. Microcontrollers like the ATMega328P that you find on the Arduino Uno have one hardware serial port. It includes a transmit (TX) and receive (RX) pin that can be accessed on digital pins 0 and 1. As explained in the sidebar in Chapter 1, "Getting Up and Blinking with the Arduino," the Arduino is equipped with a bootloader that allows you to program it over this serial interface. To facilitate this, those pins are "multiplexed" (meaning that they are connected to more than one function); they connect, indirectly, to the transmit and receive lines of your USB cable. However, serial and USB are not directly compatible, so one of two methods is used to bridge

the two. Option one is to use a secondary integrated circuit (IC) to facilitate the conversion between the two (either on or off the Arduino board). This is the type of interface present on an Uno, where an intermediary IC facilitates USB-to-serial communication. Option two is to opt for a microcontroller that has a USB controller built in (such as the Arduino Leonardo's 32U4 MCU).

Arduino Boards with an Internal or External FTDI USB-to-Serial Converter

As just explained, many Arduino boards (and Arduino clones) use a secondary integrated circuit to facilitate the USB-to-serial conversion. The "FTDI" chip is a popular chip that has just one function: convert between serial and USB. When your computer connects to an FTDI chip, it shows up in your computer as a "Virtual Serial Port" that you can access as if it was a DB9 port wired right into your computer. Figure 6-2 shows the bottom of an Arduino Nano, which utilizes an integrated FTDI chip.

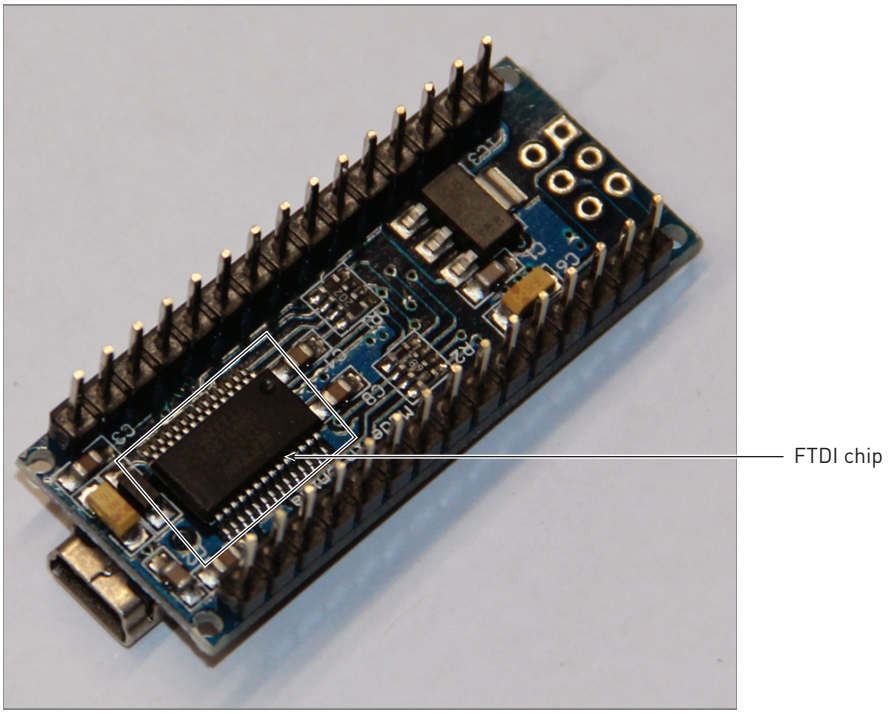

Figure 6-2: Arduino Nano with integrated FTDI chip shown

NOTE For your computer to communicate with a FTDI serial-to-USB adapter, you need to install drivers. You can find the most recent versions for Windows, OS X, and Linux at `www.ftdichip.com/Drivers/VCP.htm`. This is also linked from the Chapter 6 page on the Exploring Arduino website.

On some boards, usually to reduce board size, the FTDI chip is external to the main board, with a standardized 6-pin "FTDI connector" left for connecting to either an FTDI cable (A USB cable with an FTDI chip built in to the end of the cable) or a small FTDI breakout board. Figures 6-3 and 6-4 show these options.

Credit: adafruit Industries, www.adafruit.com.

Figure 6-3: FTDI cable

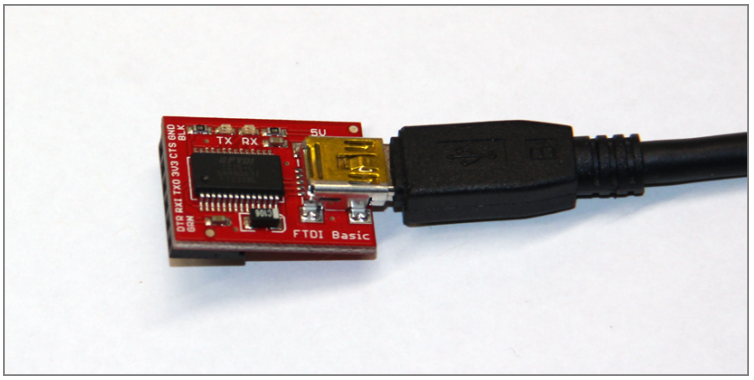

Figure 6-4: SparkFun FTDI adapter board

Using a board with a removable FTDI programmer is great if you are designing a project that will not need to be connected to a computer via USB to run. This will reduce cost if you are making several devices, and will reduce overall size of the finished product.

Following is a list of Arduino boards that use an onboard FTDI chip. Note, new Arduino boards no longer use an FTDI chip, so most of these have been discontinued. However, there are still many clones of these boards available for purchase, so they are listed here for completeness:

- Arduino Nano
- Arduino Extreme
- Arduino NG
- Arduino Diecimila
- Arduino Duemilanove
- Arduino Mega (original)

Following is a list of Arduino boards that use an external FTDI programmer:

- Arduino Pro
- Arduino Pro Mini
- LilyPad Arduino
- Arduino Fio
- Arduino Mini
- Arduino Ethernet

Arduino Boards with a Secondary USB-Capable ATMega MCU Emulating a Serial Converter

The Arduino Uno was the first board to introduce the use of an integrated circuit other than the FTDI chip to handle USB-to-serial conversion. Functionally, it works exactly the same way, with a few minor technical differences. Figure 6-5 shows the Uno's 8U2 serial converter (now a 16U2 on newer revisions).

Following is a brief list of the differences:

- First, in Windows, boards with this new USB-to-serial conversion solution require a custom driver to be installed. This driver comes bundled with the Arduino IDE when you download it. (Drivers are not needed for OS X or Linux.)

- Second, the use of this second microcontroller unit (MCU) for the conversion allowed a custom Arduino vendor ID and product ID to be reported to the host computer when the board is connected. When an FTDI-based board was connected to a computer, it just showed up as generic USB-serial device. When an Arduino using a non-FTDI converter IC (an ATMega 8U2 in the case of early Arduino Unos, now a 16U2) is connected, it is identified to the computer as an Arduino.

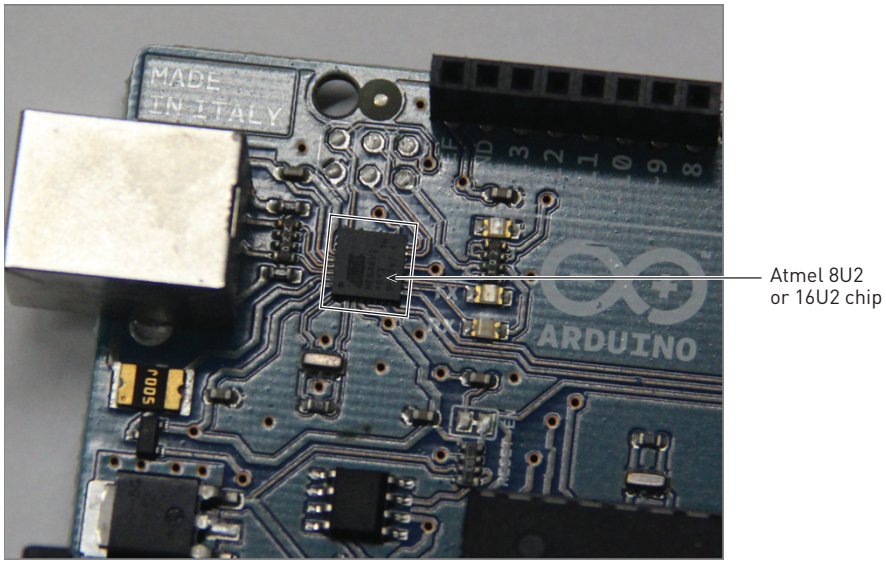

Figure 6-5: View of the Arduino Uno's 8U2 serial converter chip

- Lastly, because the secondary MCU is fully programmable (it's running a firmware stack called LUFA that emulates a USB-to-serial converter), you can change its firmware to make the Arduino show up as something different from a virtual serial port, such as a joystick, keyboard, or MIDI device. If you were to make this sort of change, the USB-to-serial LUFA firmware would not be loaded, and you would have to program the Arduino directly using the in-circuit serial programmer with a device like the AVR ISP MKII.

Following is a list of Arduino boards that use an onboard secondary MCU to handle USB-to-serial conversion:

- Arduino Uno
- Arduino Mega 2560
- Arduino Mega ADK (based on 2560)
- Arduino Due (can also be programmed directly)

Arduino Boards with a Single USB-Capable MCU

The Arduino Leonardo was the first board to have only one chip that acts both as the user-programmable MCU and as the USB interface. The Leonardo (and similar Arduino boards) employs the ATMega 32U4 microcontroller, a chip that has direct USB communication built in. This feature results in several new features and improvements.

First, board cost is reduced because fewer parts are required, and because one less factory programming step is needed to produce the boards. Second, the board can more easily be used to emulate USB devices other than a serial port (such as a keyboard, mouse, or joystick). Third, the single ordinary USART port on the ATMega does not have be multiplexed with the USB programmer, so communication with the host computer and a secondary serial device (such as a GPS unit) can happen simultaneously.

Following is a list of Arduino boards that use a single USB-capable MCU:

- Arduino Due (can also be programmed via secondary MCU)
- LilyPad Arduino USB
- Arduino Esplora
- Arduino Leonardo
- Arduino Micro

Arduino Boards with USB-Host Capabilities

Some Arduino boards can connect to USB devices as a host, enabling you to connect traditional USB devices (keyboards, mice, Android phones) to an Arduino. Naturally, there must be appropriate drivers to support the device you are connecting to. For example, you cannot just connect a webcam to an Arduino Due and expect to be able to snap photos with no additional work. The Due presently

supports a USB host class that enables you to plug a keyboard or mouse into the Due's on-the-go USB port to control it. The Arduino Mega ADK uses the Android Open Accessory Protocol (AOA) to facilitate communication between the Arduino and an Android device. This is primarily used for controlling Arduino I/O from an application running on the Android device.

Two Arduino boards that have USB-host capabilities are the Arduino Due and the Arduino Mega ADK (based on Mega 2560).

Listening to the Arduino

The most basic serial function that you can do with an Arduino is to print to the computer's serial terminal. You've already done this in several of the previous chapters. In this section, you explore the functionality in more depth, and later in the chapter you build some desktop apps that respond to the data you send instead of just printing it to the terminal. This process is the same for all Arduinos.

Using print Statements

To print data to the terminal, you only need to utilize three functions:

- `Serial.begin(baud_rate)`
- `Serial.print("Message")`
- `Serial.println("Message")`

where `baud_rate` and `"Message"` are variables that you specify.

As you've already learned, `Serial.begin()` must be called once at the start of the program in `setup()` to prepare the serial port for communication. After you've done this, you can freely use `Serial.print()` and `Serial.println()` functions to write data to the serial port. The only difference between the two is that `Serial.println()` adds a carriage return at the end of the line (so that the next thing printed will appear on the following line). To experiment with this functionality, wire up a simple circuit with a potentiometer connected to pin A0 on the Arduino, as shown in Figure 6-6.

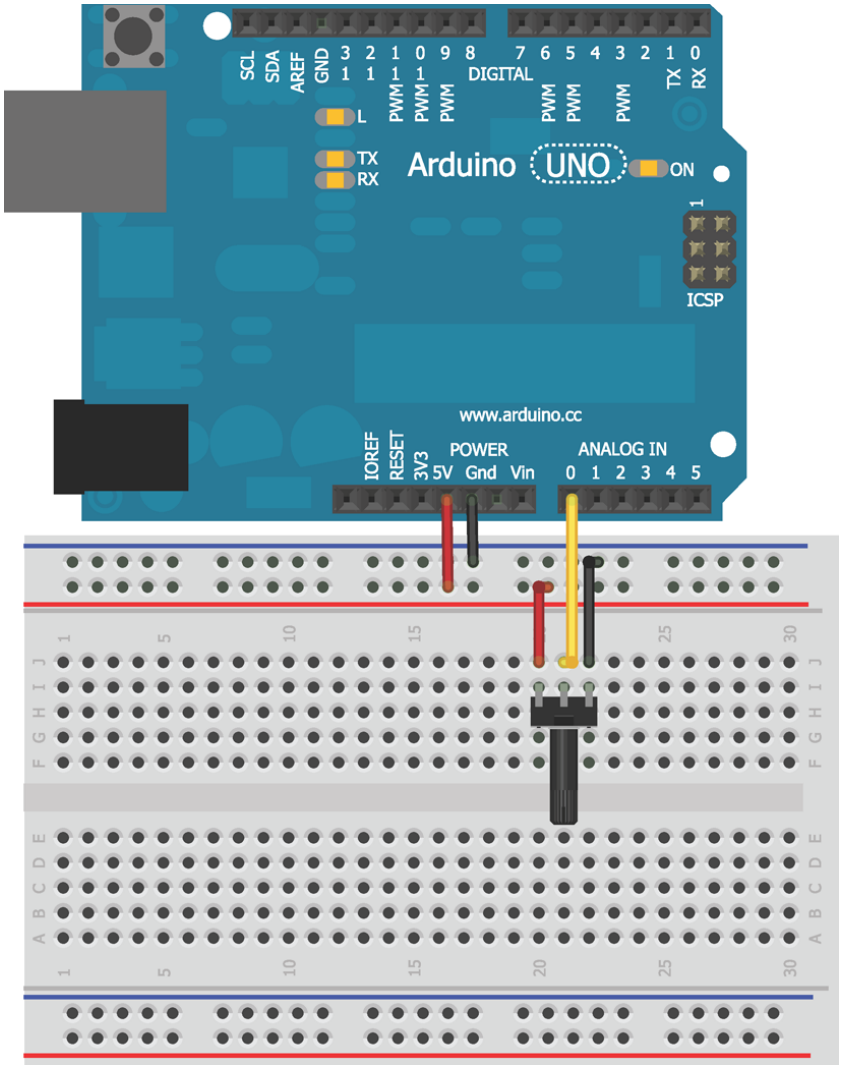

Image created with Fritzing.

Figure 6-6: Potentiometer wiring diagram

After wiring your potentiometer, load on the simple program shown in Listing 6-1 that will read the value of the potentiometer and print it as both a raw value and a percentage value.

Listing 6-1: Potentiometer Serial Print Test Program—pot.ino

```
//Simple Serial Printing Test with a Potentiometer

const int POT=0;  //Pot on analog pin 0

void setup()
{
  Serial.begin(9600); //Start serial port with baud = 9600
}

void loop()
{
  int val = analogRead(POT);           //Read potentiometer
  int per = map(val, 0, 1023, 0, 100); //Convert to percentage
  Serial.print("Analog Reading: ");
  Serial.print(val);                   //Print raw analog value
  Serial.print("  Percentage: ");
  Serial.print(per);                   //Print percentage analog value
  Serial.println("%");                 //Print % sign and newline
  delay(1000);                         //Wait 1 second, then repeat
}
```

Using a combination of `Serial.print()` and `Serial.println()` statements, this code prints both the raw and percentage values once per second. Note that by our using `Serial.println()` only on the last line, each previous transmission stays on the same line.

Open the serial monitor from the Arduino IDE and ensure that your baud rate is set to 9600 to match the value set in the Arduino sketch. You should see the values printing out once per second as you turn the potentiometer.

Using Special Characters

You can also transmit a variety of "special characters" over serial, which allow you to change the formatting of the serial data you are printing. You indicate these special characters with a slash escape character (`\`) followed by a command character. There are a variety of these special characters, but the two of greatest interest are the tab and newline characters. To insert a tab character, add a `\t` to the string. To insert a newline character, add a `\n` to the string. This proves particularly useful if you want a newline to be inserted at the beginning of a string, instead of at the end as the `Serial.println()` function does. If, for some reason, you actually want to print `\n` or `\t` in the string, you can do so by printing `\\n` or `\\t`, respectively. Listing 6-2 is a modification of the previous code to use these special characters to show data in a tabular format.

Listing 6-2: Tabular Printing using Special Characters—pot_tabular.ino

```
//Tabular serial printing test with a potentiometer

const int POT=0;  //Pot on analog pin 0

void setup()
{
  Serial.begin(9600); //Start Serial Port with Baud = 9600
}

void loop()
{
  Serial.println("\nAnalog Pin\tRaw Value\tPercentage");
  Serial.println("-----------------------------------------");
  for (int i = 0; i < 10; i++)
  {
    int val = analogRead(POT);           //Read potentiometer
    int per = map(val, 0, 1023, 0, 100); //Convert to percentage

    Serial.print("A0\t\t");
    Serial.print(val);
    Serial.print("\t\t");
    Serial.print(per);           //Print percentage analog value
    Serial.println("%");         //Print % sign and newline
    delay(1000);                 //Wait 1 second, then repeat
  }
}
```

As you turn the potentiometer, the output from this program should look something like the results shown in Figure 6-7.

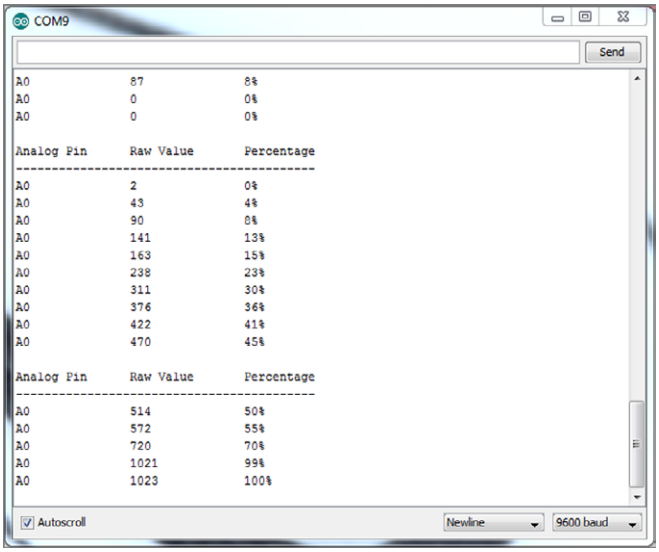

Figure 6-7: Screenshot of serial terminal with tabular data

Changing Data Type Representations

The `Serial.print()` and `Serial.println()` functions are fairly intelligent when it comes to printing out data in the format you are expecting. However, you have options for outputting data in various formats, including hexadecimal, octal, and binary. Decimal-coded ASCII is the default format. The `Serial.print()` and `Serial.println()` functions have an optional second argument that specifies the print format. Table 6-1 includes examples of how you would print the same data in various formats and how it would appear in your serial terminal.

Table 6-1: Serial Data Type Options

DATA TYPE	EXAMPLE CODE	SERIAL OUTPUT
Decimal	`Serial.println(23);`	`23`
Hexadecimal	`Serial.println(23, HEX);`	`17`
Octal	`Serial.println(23, OCT)`	`27`
Binary	`Serial.println(23, BIN)`	`00010111`

Talking to the Arduino

What good is a conversation with your Arduino if it's only going in one direction? Now that you understand how the Arduino sends data to your computer, let's spend some time discussing how to send commands from your computer to the Arduino. You've probably already noticed that the Arduino IDE serial monitor has a text entry field at the top, and a drop-down menu at the bottom. Figure 6-8 highlights both.

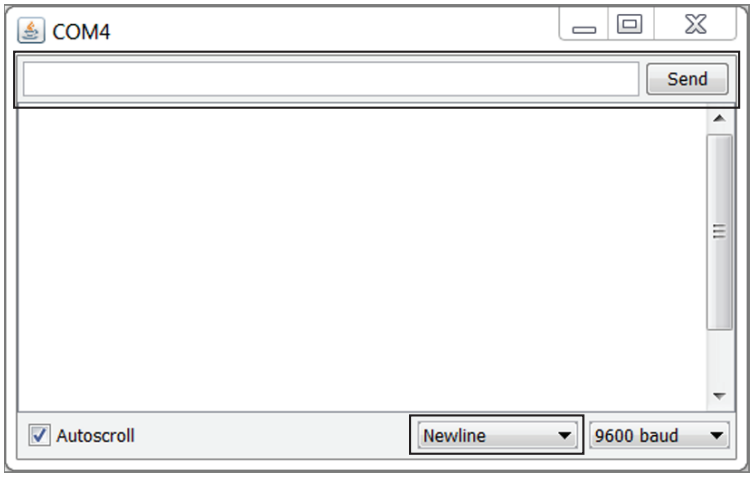

Figure 6-8: Screenshot of serial terminal highlighting text entry field and Line Ending Options drop-down menu

First, make sure that the drop-down is set to Newline. The drop-down menu determines what, if anything, is appended to end of your commands when you send them to the Arduino. The examples in the following sections assume that you have Newline selected, which just appends a `\n` to the end of anything that you send from the text entry field at the top of the serial monitor window.

Unlike with some other terminal programs, the Arduino IDE serial monitor sends your whole command string at one time (at the baud rate you specify) when you press the Enter key or the Send button. This is in contrast to other serial terminals like PuTTy (linked from this chapter's digital content page at `www.exploringarduino.com`) that send characters as you type them.

Reading Information from a Computer or Other Serial Device

You start by using the Arduino IDE serial monitor to send commands manually to the Arduino. Once that's working, you'll learn how to send multiple command values at once and how to build a simple graphical interface for sending commands.

It's important to recall that the Arduino's serial port has a buffer. In other words, you can send several bytes of data at once and the Arduino will queue them up and process them in order based on the content of your sketch. You do not need to worry about sending data faster than your loop time, but you do need to worry about sending so much data that it overflows the buffer and information is lost.

Telling the Arduino to Echo Incoming Data

The simplest thing you can do is to have the Arduino echo back everything that you send it. To accomplish this, the Arduino basically just needs to monitor its serial input buffer and print any character that it receives. To do this, you need to implement two new commands from the `Serial` object:

- `Serial.available()` returns the number of characters (or bytes) that are currently stored in the Arduino's incoming serial buffer. Whenever it's more than zero, you will read the characters and echo them back to the computer.
- `Serial.read()` reads and returns the next character that is available in the buffer.

Note that each call to `Serial.read()` will only return 1 byte, so you need to run it for as long as `Serial.available()` is returning a value greater than zero. Each time `Serial.read()` grabs a byte, that byte is removed from the buffer, as well, so the next byte is ready to be read. With this knowledge, you can now write and load the echo program in Listing 6-3 on to your Arduino.

Listing 6-3: Arduino Serial Echo Test—echo.ino

```
//Echo every character

char data; //Holds incoming character

void setup()
{
  Serial.begin(9600); //Serial Port at 9600 baud
}

void loop()
{
  //Only print when data is received
  if (Serial.available() > 0)
  {
    data = Serial.read(); //Read byte of data
    Serial.print(data);   //Print byte of data
  }
}
```

Launch the serial monitor and type anything you want into the text entry field. As soon as you press Send, whatever you typed is echoed back and displayed in the serial monitor. You have already selected to append a "newline" to the end of each command, which will ensure that each response is on a new line. That is why `Serial.print()` is used instead of `Serial.println()` in the preceding sketch.

Understanding the Differences Between Chars and Ints

When you send an alphanumeric character via the serial monitor, you aren't actually passing a "5", or an "A". You're sending a byte that the computer interprets as a character. In the case of serial communication, the ASCII character set is used to represent all the letters, number, symbols, and special commands that you might want to send. The base ASCII character set, shown in Figure 6-9, is a 7-bit set and contains a total of 128 unique characters or commands.

When reading a value that you've sent from the computer, as you did in Listing 6-3, the data must be read as a `char` type. Even if you are only expecting to send numbers from the serial terminal, you need to read values as a character first, and then convert as necessary. For example, if you were to modify the code to declare `data` as type `int`, sending a value of `5` would return `53` to the serial monitor because the decimal representation of the character 5 is the number 53. You can confirm this by looking at the ASCII reference table in Figure 6-9.

Hex	Dec	Char		Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL	null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH	Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX	Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX	End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT	End of transmission	0x24	36	$	0x44	68	D	0x64	100	d
0x05	5	ENQ	Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK	Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL	Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS	Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB	Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF	New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT	Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF	Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR	Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO	Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI	Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE	Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1	Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2	Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3	Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4	Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK	Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN	Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB	End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN	Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM	End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB	Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC	Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS	File separator	0x3C	60	<	0x5C	92	\	0x7C	124	|
0x1D	29	GS	Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS	Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US	Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

Credit: Ben Borowiec, www.benborowiec.com.

Figure 6-9: ASCII table

However, you'll often want to send numeric values to the Arduino. So how do you do that? You can do so in a few ways. First, you can simply compare the characters directly. If you want to turn an LED on when you send a `1`, you can compare the character values like this: `if (Serial.read() == '1')`. Note that the single quotes around the `'1'` indicate that it should be treated like a character.

A second option is to convert each incoming byte to an integer by subtracting the zero-valued character, like this: `int val = Serial.read() - '0'`. However, this doesn't work very well if you intend to send numbers that are greater than 9, because they will be multiple digits. To deal with this, the Arduino IDE includes a handy function called `parseInt()` that attempts to extract integers from a serial data stream. The examples that follow elaborate on these techniques.

Sending Single Characters to Control an LED

Before your dive into parsing larger strings of multiple-digit numbers, start by writing a sketch that uses a simple character comparison to control an LED.

You'll send a `1` to turn an LED on, and a `0` to turn it off. Wire an LED up to pin 9 of your Arduino as shown in Figure 6-10.

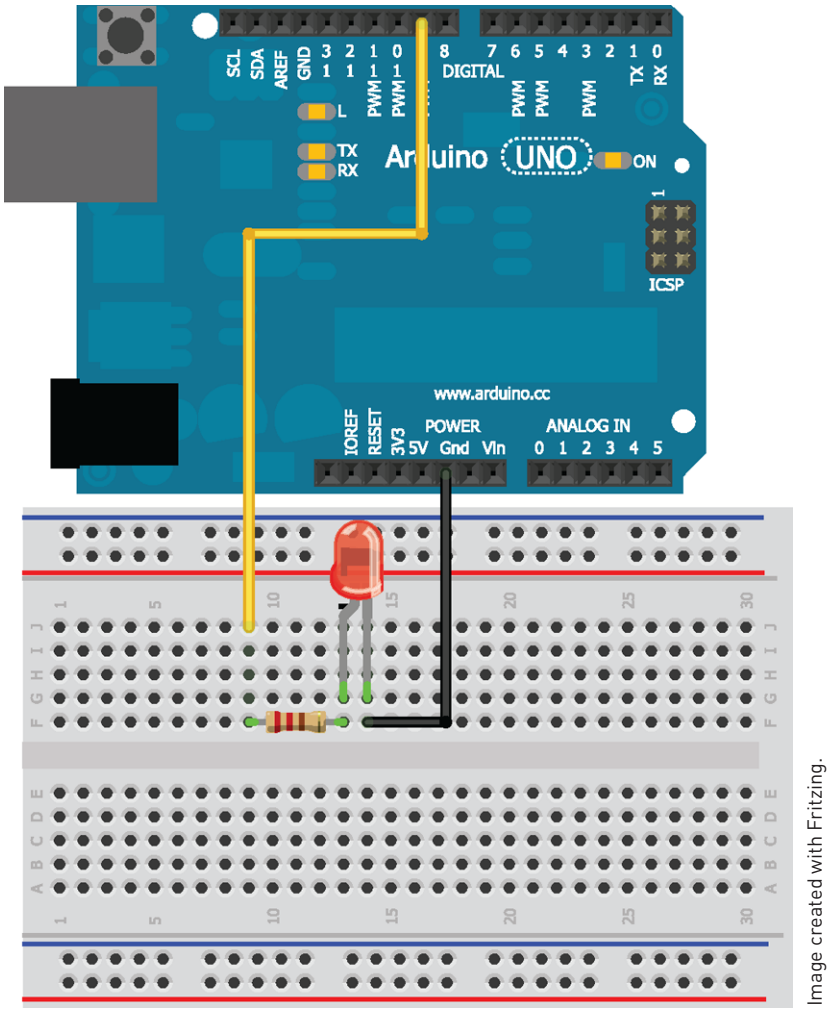

Figure 6-10: Single LED connected to Arduino on pin 9

As explained in the previous section, when only sending a single character, the easier thing to do is to do a simple character comparison in an `if` statement. Each time a character is added to the buffer, it is compared to a `'0'` or a `'1'`, and the appropriate action is taken. Load up the code in Listing 6-4 and experiment with sending a `0` or a `1` from the serial terminal.

Listing 6-4: Single LED Control using Characters—single_char_control.ino

```
//Single Character Control of an LED

const int LED=9;

char data; //Holds incoming character

void setup()
{
  Serial.begin(9600); //Serial Port at 9600 baud
  pinMode(LED, OUTPUT);
}

void loop()
{
  //Only act when data is available in the buffer
  if (Serial.available() > 0)
  {
    data = Serial.read(); //Read byte of data
    //Turn LED on
    if (data == '1')
    {
      digitalWrite(LED, HIGH);
      Serial.println("LED ON");
    }
    //Turn LED off
    else if (data == '0')
    {
      digitalWrite(LED, LOW);
      Serial.println("LED OFF");
    }
  }

}
```

Note that an `else if` statement is used instead of a simple `else` statement. Because your terminal is also set to send a newline character with each transmission, it's critical to clear these from the buffer. `Serial.read()` will read in the newline character, see that is not equivalent to a `'0'` or a `'1'`, and it will be overwritten the next time `Serial.read()` is called. If just an `else` statement were used, both `'0'` and `'\n'` would trigger turning the LED off. Even when sending a `'1'`, the LED would immediately turn off again when the `'\n'` was received!

Sending Lists of Values to Control an RGB LED

Sending a single command character is fine for controlling a single digital pin, but what if you want to accomplish some more complex control schemes? This section explores sending multiple comma-separate values to simultaneously command multiple devices. To facilitate testing this, wire up a common cathode RGB LED as shown in Figure 6-11.

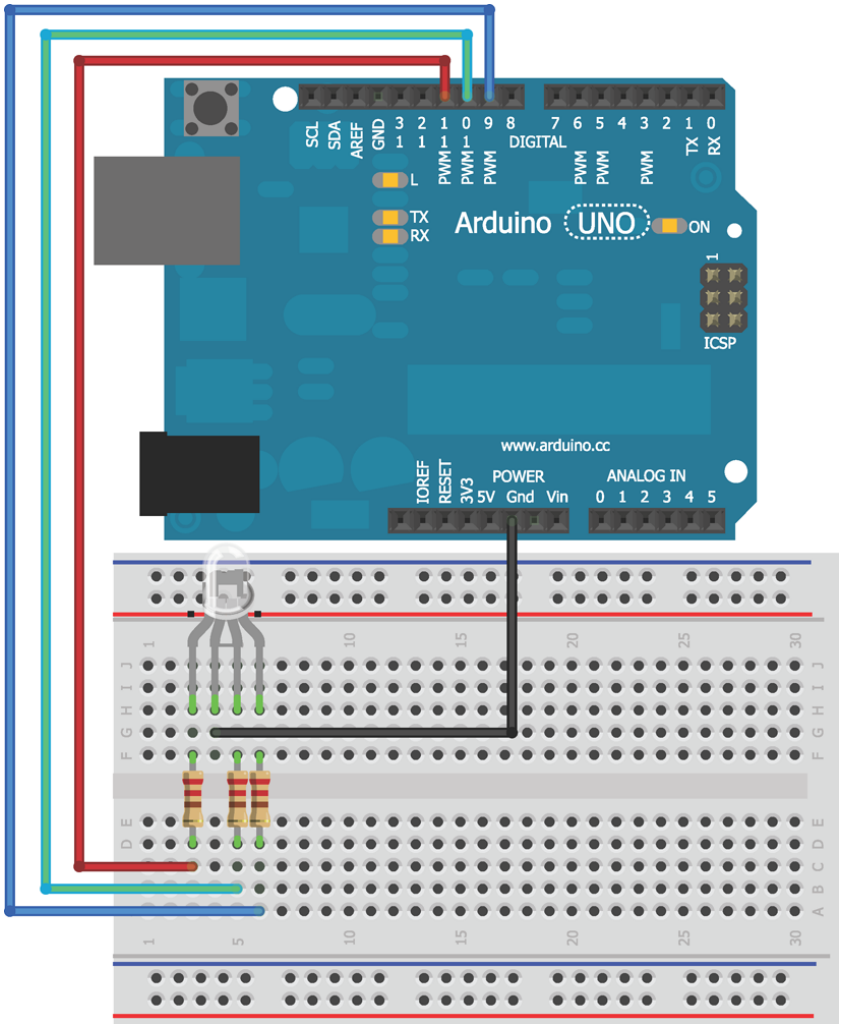

Figure 6-11: RGB LED connected to Arduino

To control this RGB LED, you send three separate 8-bit values (0–255) to set the brightness of each LED color. For example, to set all the colors to full brightness, you send `"255,255,255"`. This presents a few challenges:

- You need to differentiate between numbers and commas.
- You need to turn this sequence of characters into integers that you can pass to `analogWrite()` functions.
- You need to be able to handle the fact that values could be one, two, or three digits.

Thankfully, the Arduino IDE implements a very handy function for identifying and extracting integers: `Serial.parseInt()`. Each call to this function waits until a non-numeric value enters the serial buffer, and converts the previous digits into an integer. The first two values are read when the commas are detected, and the last value is read when the newline is detected.

To test this function for yourself, load the program shown in Listing 6-5 on to your Arduino.

Listing 6-5: RGB LED Control via Serial—list_control.ino

```
//Sending Multiple Variables at Once

//Define LED pins
const int RED   =11;
const int GREEN =10;
const int BLUE  =9;

//Variables for RGB levels
int rval = 0;
int gval = 0;
int bval = 0;

void setup()
{
  Serial.begin(9600); //Serial Port at 9600 baud

  //Set pins as outputs
  pinMode(RED, OUTPUT);
  pinMode(GREEN, OUTPUT);
  pinMode(BLUE, OUTPUT);
}

void loop()
{
  //Keep working as long as data is in the buffer
  while (Serial.available() > 0)
```

```
  {
    rval = Serial.parseInt();  //First valid integer
    gval = Serial.parseInt();  //Second valid integer
    bval = Serial.parseInt();  //Third valid integer

    if (Serial.read() == '\n') //Done transmitting
    {
      //set LED
      analogWrite(RED, rval);
      analogWrite(GREEN, gval);
      analogWrite(BLUE, bval);
    }
  }
}
```

The program keeps looking for the three integer values until a newline is detected. Once this happens, the values that were read are used to set the brightness of the LEDs. To use this, open the serial monitor and enter three values between 0 and 255 separated by a comma, like `"200,30,180"`. Try mixing all kinds of pretty colors!

Talking to a Desktop App

Eventually, you're bound to get bored of doing all your serial communication through the Arduino serial monitor. Conveniently, just about any desktop programming language you can think of has libraries that allow it to interface with the serial ports in your computer. You can use your favorite desktop programming language to write programs that send serial commands to your Arduino and that react to serial data being transmitted from the Arduino to the computer.

In this book, Processing is the desktop programming language of choice because it is very similar to the Arduino language that you have already become familiar with. In fact, the Arduino programming language is based on Processing! Other popular desktop languages (that have well-documented serial communication libraries) include Python, PHP, Visual Basic, C, and more. First, you'll learn how to read transmitted serial data in Processing, and then you'll learn how you can use Processing to create a simple graphical user interface (GUI) to send commands to your Arduino.

Talking to Processing

Processing has a fairly simple programming interface, and it's similar to the one you've already been using for the Arduino. In this section, you install Processing, and then write a simple graphical interface to generate a graphical

output based on serial data transmitted from your Arduino. Once that's working, you implement communication in the opposite direction to control your Arduino from a GUI on your computer.

Installing Processing

First things first, you need to install Processing on your machine. This is the same process that you followed in the first chapter to get the Arduino IDE installed. Visit `http://processing.org/download/` (or find the download link on the digital content page for this chapter on `www.exploringarduino.com`) and download the compressed package for your operating system. Simply unzip it to your preferred location and you are ready to go! Run the Processing application, and you should see an IDE that looks like the one shown in Figure 6-12.

Figure 6-12: The Processing IDE. Does it look familiar?

Controlling a Processing Sketch from Your Arduino

For your first experiment with Processing, you use a potentiometer connected to your Arduino to control the color of a window on your computer. Wire up your Arduino with a potentiometer, referencing Figure 6-6 again. You already know the Arduino code necessary to send the analog values from the potentiometer to your computer. The fact that you are now feeding the serial data into Processing does not have any impact on the way you transmit it.

Reference the code in Listing 6-6 and load it on to your Arduino. It sends an updated value of the potentiometer to the computer's serial port every 50 milliseconds. The 50ms is important; if you were to send it as fast as possible, the Processing sketch wouldn't be able to handle it as quickly as you are sending it, and you would eventually overflow the serial input buffer on your computer.

Listing 6-6: Arduino Code to send Data to the Computer—pot_to_processing/arduino_read_pot

```
//Sending POT value to the computer

const int POT=0; //Pot on analog pin 0

int val; //For holding mapped pot value

void setup()
{
  Serial.begin(9600); //Start Serial
}

void loop()
{
  val = map(analogRead(POT), 0, 1023, 0, 255); //Read and map POT
  Serial.println(val);                         //Send value
  delay(50);                                   //Delay so we don't flood
                                               //the computer
}
```

Now comes the interesting part: writing a Processing sketch to do something interesting with this incoming data. The sketch in Listing 6-7 reads the data in the serial buffer and adjusts the brightness of a color on the screen of your computer based on the value it receives. First, copy the code from Listing 6-7 into a new Processing sketch. You need to change just one important part. The Processing sketch needs to know which serial port to expect data to arrive on. This is the same port that you've been programming the Arduino from. In the

following listing, replace `"COM3"` with your serial port number. Remember that on Linux and Mac it will look like `/dev/ttyUSB0`, for example. You can copy the exact name from within the Arduino IDE if you are unsure.

```
port = new Serial(this, "COM3", 9600); //setup serial
```

Listing 6-7: Processing Code to Read Data and Change Color on the Screen—pot_to_processing/processing_display_color

```
//Processing Sketch to Read Value and Change Color on the Screen

//Import and initialize serial port library
import processing.serial.*;
Serial port;

float brightness = 0; //For holding value from pot

void setup()
{
  size(500,500);                          //Window size
  port = new Serial(this, "COM3", 9600); //Set up serial
  port.bufferUntil('\n');                 //Set up port to read until
                                          //newline
}

void draw()
{
  background(0,0,brightness); //Updates the window
}

void serialEvent (Serial port)
{
  brightness = float(port.readStringUntil('\n')); //Gets val
}
```

After you've loaded the code into your Processing IDE and set the serial port properly, make sure that the Arduino serial monitor isn't open. Only one program on your computer can have access to the serial port at a time. Click the Run button in the Processing IDE (the button in the top left of the window with a triangle); when you do so, a small window will pop up (see Figure 6-13). As you turn the potentiometer, you should see the color of the window change from black to blue.

Now that you've seen it working, let's walk through the code to gain a better understanding of how the Processing sketch is working. Unlike in Arduino, the serial library is not imported automatically. By calling `import processing.serial.*;` and `Serial port;` you are importing the serial library and making a serial object called `port`.

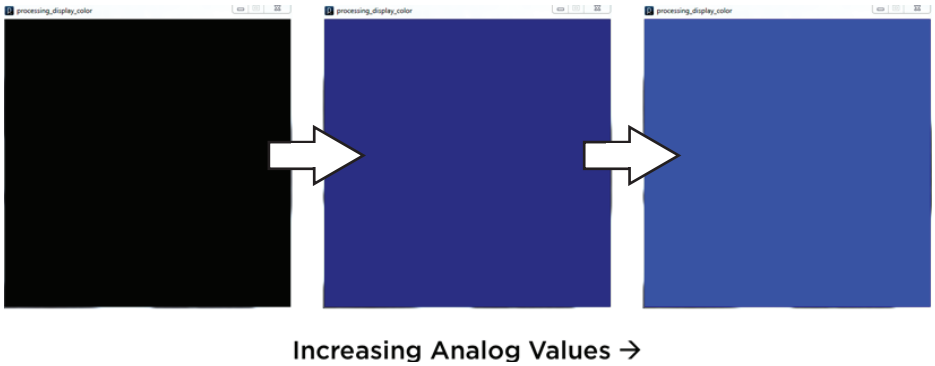

Figure 6-13: Example windows from Processing sketch

Like the Arduino, Processing has a `setup()` function that runs once at the beginning of the sketch. In this sketch, it sets up the serial port and creates a window of size 500×500 pixels with the command `size(500,500)`. The command `port = new Serial(this, "COM3", 9600)` tells Processing everything it needs to know about creating the serial port. The instance (referred to as "port") will run in this sketch and communicate on COM3 (or whatever your serial port is) at 9600 baud. The Arduino and the program on your computer must agree on the speed at which they communicate; otherwise, you'll get garbage characters. `port.bufferUntil('\n')` tells Processing to buffer the serial input and not do anything with the information until it sees a newline character.

Instead of `loop()`, Processing defines other special functions. This program uses `draw()` and `serialEvent()`. The `draw()` function is similar to Arduino's `loop()`; it runs continuously and updates the display. The `background()` function sets the color of the window by setting red, green, and blue values (the three arguments of the function). In this case, the value from the potentiometer is controlling the blue intensity, and red and green are set to `0`. You can change what color your pot is adjusting simply by swapping which argument `brightness` is filling in. RGB color values are 8-bit values ranging from `0` to `255`, which is why the potentiometer is mapped to those values before being transmitted.

`serialEvent()` is called whenever the `bufferUntil()` condition that you specified in the `setup()` is met. Whenever a newline character is received, the `serialEvent()` function is triggered. The incoming serial information is read as a string with `port.readStringUntil('\n')`. You can think of a string as an array of text. To use the string as a number, you must convert it to a floating-point number with `float()`. This sets the brightness variable, changing the background color of the application window.

To stop the application and close the serial port, click the Stop button in the Processing IDE; it's the square located next to the Run button.

SUDOGLOVE PROCESSING DEBUGGER

The SudoGlove is a control glove that drives RC cars and controls other hardware. I developed a Processing debugging display that graphically shows the values of various sensors. You can learn more about it here: `www.sudoglove.com`.

Download the source code for the Processing display here: `www.jeremyblum.com/2011/03/25/processing-based-sudoglove-visual-debugger/`. You can also find this source code on the Wiley website shown at the beginning of this chapter.

Sending Data from Processing to Your Arduino

The obvious next step is to do the opposite. Wire up an RGB LED to your Arduino as shown in Figure 6-11 and load on the same program from earlier that you used to receive a string of three comma-separated values for setting the red, green, and blue intensities (Listing 6-5). Now, instead of sending a string of three values from the serial monitor, you select a color using a color picker.

Load and run the code in Listing 6-8 in Processing, remembering to adjust the serial port number accordingly as you did with the previous sketch. Processing sketches automatically load collateral files from a folder called "data" in the sketch folder. The hsv.jpg file is included in the code download for this chapter. Download it and place it in a folder named "data" in the same directory as your sketch. Processing defaults to saving sketches in your Documents folder. The structure will look similar to the one shown in Figure 6-14.

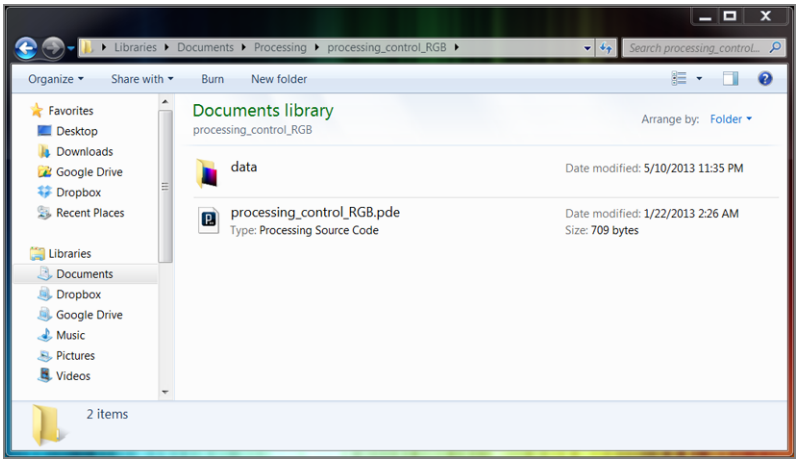

Figure 6-14: Folder structure

The image in the data folder will serve as the color selector.

Listing 6-8: Processing Sketch to Set Arduino RGB Colors— processing_control_RGB/processing_control_RGB

```
import processing.serial.*; //Import serial library
PImage img;                 //Image object
Serial port;                //Serial port object

void setup()
{
  size(640,256);                          //Size of HSV image
  img = loadImage("hsv.jpg");             //Load in background image
  port = new Serial(this, "COM9", 9600); //Open serial port
}

void draw()
{
  background(0);   //Black background
  image(img,0,0);  //Overlay image
}

void mousePressed()
{
  color c = get(mouseX, mouseY); //Get the RGB color where mouse was
pressed
  String colors = int(red(c))+","+int(green(c))+","+int(blue(c))+"\n"; //
extract
values from color
  print(colors);      //Print colors for debugging
  port.write(colors); //Send values to Arduino
}
```

When you execute the program, you should see a screen like the one shown in Figure 6-15 pop up. Click different colors and the RGB values will be transmitted to the Arduino to control the RGB LED's color. Note that the serial console also displays the commands being sent to assist you in any debugging.

After you've finished staring at all the pretty colors, look back at the code and consider how it's working. As before, the serial library is imported and a serial object called `port` is created. A `PImage` object call `img` is also created. This will hold the background image. In the `setup()`, the serial port is initialized, the display window is set to the size of the image, and image is imported into the image object by calling `img = loadImage("hsv.jpg")`.

In the `draw()` function, the image is loaded in the window with `image(img,0,0)`. `img` is the image you want to draw in the window, and `0, 0` are coordinates where the image will start to be drawn. `0,0` is the top left of the application window.

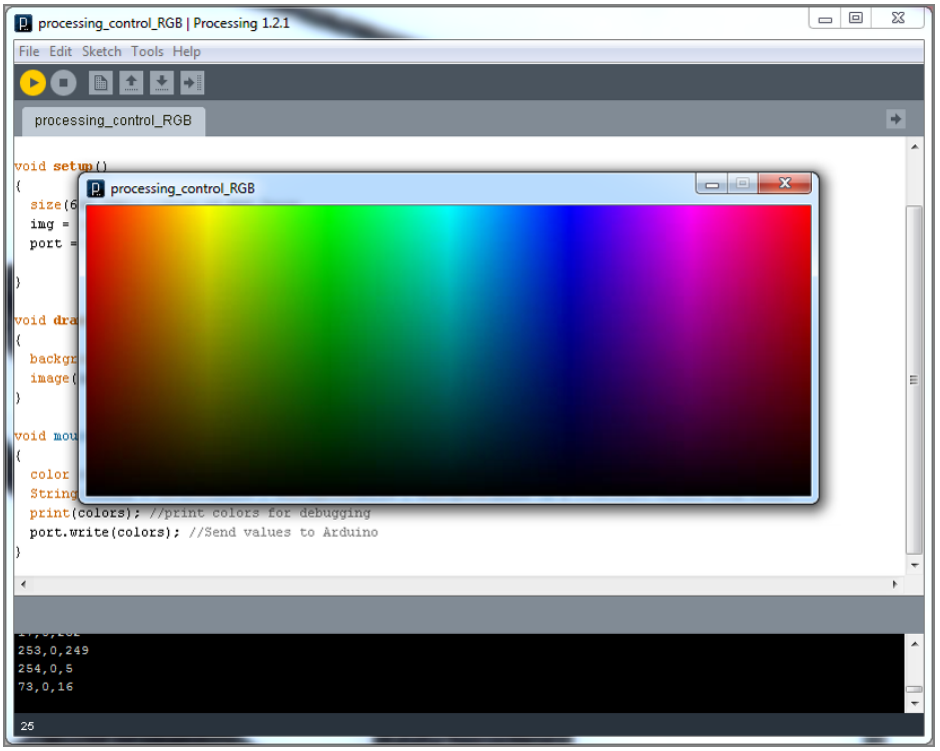

Figure 6-15: Processing color selection screen

Every time the mouse button is pressed, the `mousePressed()` function is called. The color of the pixel where you clicked is saved to a `color` object named `c`. The `get()` method tells the application where to get the color from (in this case, the location of the mouse's X and Y position in the sketch). The sketch converts the object `c` into a string that can be sent to the Arduino by converting to integers representing red, green, and blue. These values are also printed to the Processing console so that you can see what is being sent.

Ensure that the Arduino is connected and programmed with the code from Listing 6-5. Run the processing sketch (with the correct serial port specified) and click around the color map to adjust the color of the LED connected to your Arduino.

Learning Special Tricks with the Arduino Leonardo (and Other 32U4-Based Arduinos)

The Leonardo, in addition to other Arduinos that implement MCUs that connect directly to USB, has the unique ability to emulate nonserial devices such as a keyboard or mouse. In this section you learn about using a Leonardo to

emulate these devices. You need to be extremely careful to implement these functions in a way that does not make reprogramming difficult. For example, if you write a sketch that emulates a mouse and continuously moves your pointer around the screen, you might have trouble clicking on the Upload button in the Arduino IDE! In this section, you learn a few tricks that you can use to avoid these circumstances.

TIP If you get stuck with a board that's too hard to program due to its keyboard or mouse input, hold down the Reset button and release it while pressing the Upload button in the Arduino IDE to reprogram it.

When you first connect a Leonardo to a Windows computer, you need to install drivers, just as you did with the Arduino Uno in the first chapter. Follow the same directions at `http://arduino.cc/en/Guide/ArduinoLeonardoMicro#toc8` for Leonardo-specific instructions. (These instructions are also linked from the digital content page for this chapter from `www.exploringarduino.com`.)

Emulating a Keyboard

Using the Leonardo's unique capability to emulate USB devices, you can easily turn your Arduino into a keyboard. Emulating a keyboard allows you to easily send key-combination commands to your computer or type data directly into a file that is open on your computer.

Typing Data into the Computer

The Leonardo can emulate a USB keyboard, sending keystrokes and key combinations. This section explores how to use both concepts. First, you write a simple program that records data from a few analog sensors into a comma-separated-value (.csv) format that you can later open up with Excel or Google spreadsheets to generate a graph of the values.

Start by opening the text editor of your choice and saving a blank document with a .csv extension. To do this, you can generally choose the file type in the Save dialog, select "All Files," and manually type the file name with the extension, such as "data.csv." The demo video also shows how to create a .csv file.

Next, create a simple circuit like the one shown in Figure 6-16. It will monitor both light and temperature levels using analog sensors that you have already seen in Chapter 3, "Reading Analog Sensors." In addition to the sensors, the circuit includes a button for turning the logging functionality on and off, and an LED that will indicate whether it is currently logging data.

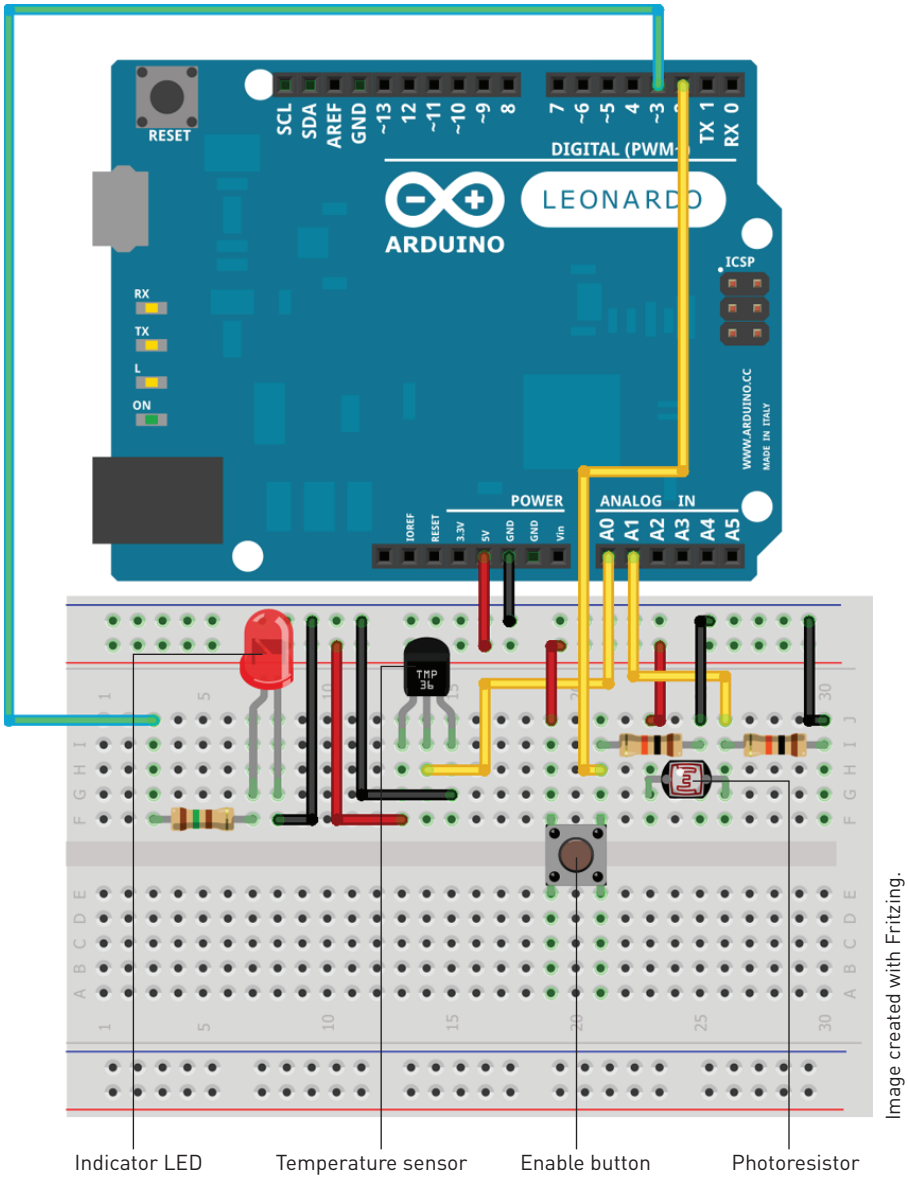

Image created with Fritzing.

Figure 6-16: Temperature and light sensor circuit

Using the same debouncing function that you implemented in Chapter 2, "Digital Inputs, Outputs, and Pulse-Width Modulation," you use the pushbutton to toggle the logging mode on and off. While in logging mode, the Arduino polls the sensors and "types" those values into your computer in a comma-separated format once every second. An indicator LED remains illuminated while you are logging data. Because you want the Arduino to be constantly polling the state

of the button, you cannot use a `delay()` function to wait 1000ms between each update. Instead, you use the `millis()` function, which returns the number of milliseconds since the Arduino was last reset. You can make the Arduino send data every time the `millis()` function returns a multiple of 1000ms, effectively creating a nonblocking 1-second delay between transmissions. To do this, you can use the modulo operator (`%`). Modulo returns the remainder of a division. If, for example, you executed `1000%1000`, you would find that the result is 0 because 1000/1000=1, with a remainder of 0. `1500%1000`, on the other hand, would return 500 because 1500/1000 is equal to 1, with a remainder of 500. If you take the modulus of `millis()` with `1000`, the result is zero every time `millis()` reaches a value that is a multiple of 1000. By checking this with an `if()` statement, you can execute code once every second.

Examine the code in Listing 6-9 and load it onto your Arduino Leonardo. Ensure that you've selected "Arduino Leonardo" from the Tools > Board menu in the Arduino IDE.

Listing 6-9: Temperature and Light Data Logger—csv_logger.ino

```
//Light and Temp Logger

const int TEMP   =0;      //Temp sensor on analog pin 0
const int LIGHT  =1;      //Light sensor on analog pin 1
const int LED    =3;      //Red LED on pin 13
const int BUTTON =2;      //The button is connected to pin 2

boolean lastButton = LOW;    //Last button state
boolean currentButton = LOW; //Current button state
boolean running = false;     //Not running by default
int counter = 1;             //An index for logged data entries

void setup()
{
  pinMode (LED, OUTPUT);   //Set blue LED as output
  Keyboard.begin();        //Start keyboard emulation
}

void loop()
{
  currentButton = debounce(lastButton);           //Read debounced state

  if (lastButton == LOW && currentButton == HIGH) //If it was pressed…
    running = !running;                           //Toggle running state

  lastButton = currentButton;                     //Reset button value

  if (running)                              //If logger is running
```

```
  {
    digitalWrite(LED, HIGH);              //Turn the LED on
    if (millis() % 1000 == 0)             //If time is multiple
                                          //of 1000ms
    {
      int temperature = analogRead(TEMP); //Read the temperature
      int brightness = analogRead(LIGHT); //Read the light level
      Keyboard.print(counter);            //Print the index number
      Keyboard.print(",");                //Print a comma
      Keyboard.print(temperature);        //Print the temperature
      Keyboard.print(",");                //Print a comma
      Keyboard.println(brightness);       //Print brightness, newline
      counter++;                          //Increment the counter
    }
  }
  else
  {
    digitalWrite(LED, LOW);  //If logger not running, turn LED off
  }
}

/*
* Debouncing Function
* Pass it the previous button state,
* and get back the current debounced button state.
*/
boolean debounce(boolean last)
{
  boolean current = digitalRead(BUTTON);       //Read the button state
  if (last != current)                         //If it's different…
  {
    delay(5);                                  //Wait 5ms
    current = digitalRead(BUTTON);             //Read it again
  }
  return current;                              //Return the current
                                               //value
}
```

Before you test the data logger, let's highlight some of the new functionality that has been implemented in this sketch. Similarly to how you initialized the serial communication, the keyboard communication is initialized by putting `Keyboard.begin()` in the `setup()`.

Each time through `loop()`, the Arduino checks the state of the button and runs the debouncing function that you are already familiar with. When the button is pressed, the value of the *running* variable is inverted. This is accomplished by setting it to its opposite with the `!` operator.

While the Arduino is in *running* mode, the logging step is executed only every 1000ms using the logic described previously. The keyboard functions work very similarly to the serial functions. `Keyboard.print()` "types" the given string into

your computer. After reading the two analog sensors, the Arduino sends the values to your computer as keystrokes. When you use `Keyboard.println()`, the Arduino emulates pressing the Enter or Return key on your keyboard after sending the given string. An incrementing counter and both analog values are entered in a comma-separated format.

Follow the demo video from this chapter's web page to see this sketch in action. Make sure that your cursor is actively positioned in a text document, and then press the button to start logging. You should see the document begin to populate with data. Hold your hand over the light sensor to change the value or squeeze the temperature sensor to see the value increase. When you have finished, press the button again to stop logging. After you save your file, you can import it into the spreadsheet application of your choice and graph it over time. This is shown in the demo video.

NOTE To watch a demo video of the live temperature and light logger, visit `www.exploringarduino.com/content/ch6`. You can also find this video on the Wiley website shown at the beginning of this chapter.

Commanding Your Computer to Do Your Bidding

In addition to typing like a keyboard, you can also use the Leonardo to emulate key combinations. On Windows computers, pressing the Windows+L keys locks the computer screen (On Linux, you can use Control+Alt+L). Using that knowledge paired with a light sensor, you can have your computer lock automatically when you turn the lights off. OS X uses the Control+Shift+Eject, or Control+Shift+Power keys to lock the machine, which can't be emulated by the Leonardo because it cannot send an Eject or Power simulated button press. In this example, you learn how to lock a Windows computer. You can continue to use the same circuit shown in Figure 6-16, though only the light sensor will be used in this example.

Run the previous sketch at a few different light levels and see how the light sensor reading changes. Using this information, you should pick a threshold value below which you will want your computer to lock. (In my room, I found that with the lights off the value was about `300`, and it was about `700` with the lights on. So, I chose a threshold value of `500`.) When the light sensor value drops below that value, the `lock` command will be sent to the computer. You might want to adjust this value for your environment.

Load the sketch in Listing 6-10 on to your Arduino. Just make sure you have your threshold set to a reasonable value first, by testing what light levels in your room correspond to various analog levels. If you pick a poorly calibrated value, it might lock your computer as soon as you upload it!

Listing 6-10: Light-Based Computer Lock—lock_computer.ino

```
//Locks your computer when you turn off the lights

const int LIGHT     =1;   //Light sensor on analog pin 1
const int THRESHOLD =500;  //Brightness must drop below this level
                           //to lock computer

void setup()
{
  Keyboard.begin();
}

void loop()
{
  int brightness = analogRead(LIGHT);    //Read the light level

  if (brightness < THRESHOLD)
  {
    Keyboard.press(KEY_LEFT_GUI);
    Keyboard.press('l');
    delay(100);
    Keyboard.releaseAll();
  }
}
```

After loading the program, try flipping the lights off. Your computer should lock immediately. The following video demo shows this in action. This sketch implements two new keyboard functions: `Keyboard.press()` and `Keyboard.releaseAll()`. Running `Keyboard.press()` is equivalent to starting to hold a key down. So, if you want to hold the Windows key and the L key down at the same time, you run `Keyboard.press()` on each. Then, you delay for a short period of time and run the `Keyboard.releaseAll()` function to let go of, or release, the keys. Special keys are defined on the Arduino website: `http://arduino.cc/en/Reference/KeyboardModifiers`. (This definition table is also linked from the content page for this chapter at `www.exploringarduino.com/content/ch6`.)

NOTE To watch a demo video of the light-activated computer lock, visit `www.exploringarduino.com/content/ch6`. You can also find this video on the Wiley website shown at the beginning of this chapter.

Emulating a Mouse

Using a two-axis joystick and some pushbuttons, you can use an Arduino Leonardo to make your own mouse! The joystick will control the mouse location, and the buttons will control the left, middle, and right buttons of the mouse.

Just like with the keyboard functionality, the Arduino language has some great functions built in that make it easy to control mouse functionality.

First things first, get your circuit set up with a joystick and some buttons as shown in Figure 6-17. Don't forget that your buttons need to have pull-down resistors! The joystick will connect to analog pins 0 and 1. (Joysticks are actually just two potentiometers hooked up to a knob.) When you move the joystick all the way in the x direction, it maxes out the x potentiometer, and the same goes for the y direction.

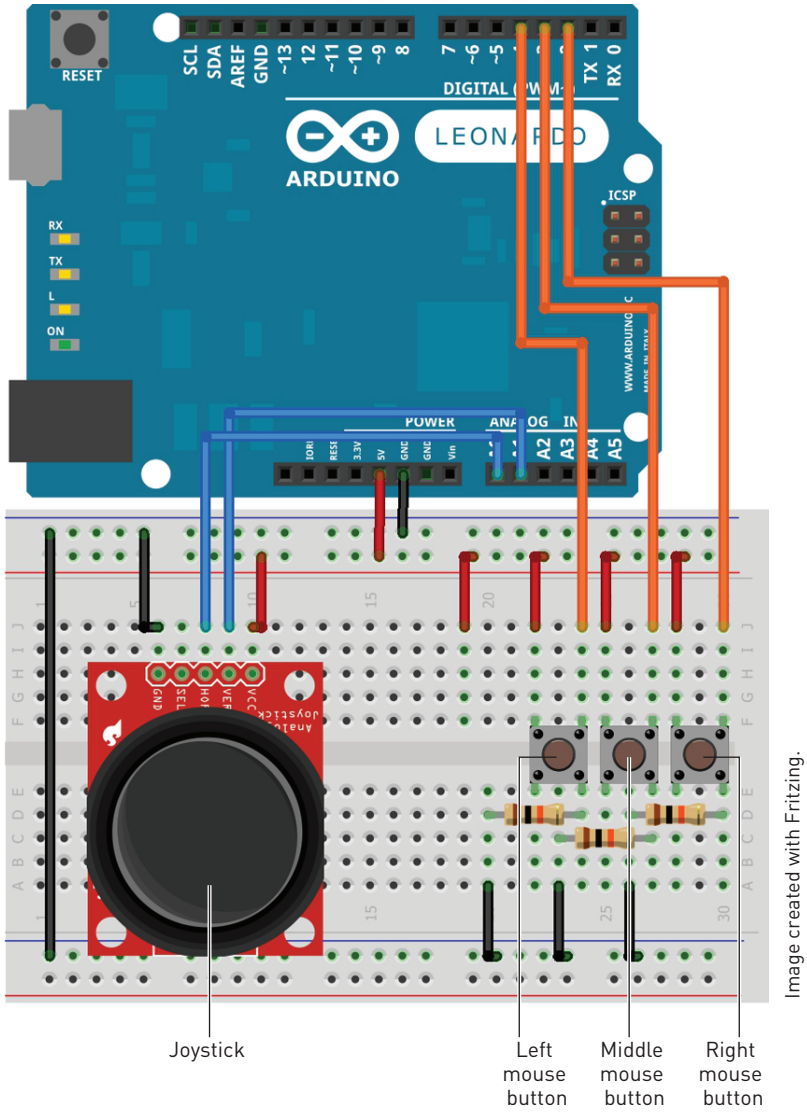

Figure 6-17: Joystick Leonardo mouse circuit

The diagram shows a SparkFun joystick, but any will do. (In the video described after the listing, I used a Parallax joystick.) Depending on the orientation of the joystick, you might need to adjust the bounds of the map function or swap the x/y in the code below.

After you've wired the circuit, it's time to load some code onto the Leonardo. Load up the code in Listing 6-11 and play with the joystick and buttons; the pointer on your screen should respond accordingly.

Listing 6-11: Mouse Control Code for the Leonardo—mouse.ino

```
// Make a Mouse!

const int LEFT_BUTTON   =4;  //Input pin for the left button
const int MIDDLE_BUTTON =3;  //Input pin for the middle button
const int RIGHT_BUTTON  =2;  //Input pin for the right button
const int X_AXIS        =0;  //Joystick x-axis analog pin
const int Y_AXIS        =1;  //Joystick y-axis analog pin

void setup()
{
  Mouse.begin();
}

void loop()
{
  int xVal = readJoystick(X_AXIS);         //Get x-axis movement
  int yVal = readJoystick(Y_AXIS);         //Get y-axis movement

  Mouse.move(xVal, yVal, 0);               //Move the mouse

  readButton(LEFT_BUTTON, MOUSE_LEFT);     //Control left button
  readButton(MIDDLE_BUTTON, MOUSE_MIDDLE); //Control middle button
  readButton(RIGHT_BUTTON, MOUSE_RIGHT);   //Control right button

  delay(5);                                //This controls responsiveness
}

//Reads joystick value, scales it, and adds dead range in middle
int readJoystick(int axis)
{
  int val = analogRead(axis);              //Read analog value
  val = map(val, 0, 1023, -10, 10);        //Map the reading

  if (val <= 2 && val >= -2)               //Create dead zone to stop mouse
drift
    return 0;

  else                                     //Return scaled value
    return val;
```

```
}

//Read a button and issue a mouse command
void readButton(int pin, char mouseCommand)
{
  //If button is depressed, click if it hasn't already been clicked
  if (digitalRead(pin) == HIGH)
  {
    if (!Mouse.isPressed(mouseCommand))
    {
      Mouse.press(mouseCommand);
    }
  }
  //Release the mouse if it has been clicked.
  else
  {
    if (Mouse.isPressed(mouseCommand))
    {
      Mouse.release(mouseCommand);
    }
  }
}
```

This is definitely one of the more complicated sketches that have been covered so far, so it's worth stepping through it to both understand the newly introduced functions and the program flow used to make the joystick mouse.

Each of the button and joystick pins are defined at the top of the sketch, and the mouse library is started in the setup. Each time through the loop, the joystick values are read and mapped to movement values for the mouse. The mouse buttons are also monitored and the button presses are transmitted if necessary.

A `readJoystick()` function was created to read the joystick values and map them. Each joystick axis has a range of 1024 values when read into the analog-to-digital converter (ADC). However, mouse motions are relative. In other words, passing a value of 0 to `Mouse.move()` for each axis will result in no movement on that axis. Passing a positive value for the x-axis will move the mouse to the right, and a negative value will move it to the left. The larger the value, the more the mouse will move. Hence, in the `readJoystick()` function, a value of 0 to 1023 is mapped to a value of `-10` to `10`. A small buffer value around `0` is added where the mouse will not move. This is because even while the joystick is in the middle position, the actual value may fluctuate around `512`. By setting the desired distance back to 0 after being mapped within a certain range, you guarantee that the mouse will not move on its own while the joystick is not being actuated. Once the values are ascertained, `Mouse.move()` is given the x and y values to move the mouse. A third argument for `Mouse.move()` determines the movement of the scroll wheel.

To detect mouse clicks, the `readButton()` function was created so that it can be repeated for each of the three buttons to detect. The function detects the current state of the mouse with the `Mouse.isPressed()` command and controls the mouse accordingly using the `Mouse.press()` and `Mouse.release()` functions.

NOTE To watch a demo video of the joystick mouse controlling a computer pointer, check out `www.exploringarduino.com/content/ch6`. You can also find this video on the Wiley website shown at the beginning of this chapter.

Summary

In this chapter you learned about the following:

- Arduinos connect to your computer via a USB-to-serial converter.
- Different Arduinos facilitate a USB-to-serial conversion using either dedicated ICs or built-in USB functionality.
- Your Arduino can print data to your computer via your USB serial connection.
- You can use special serial characters to format your serial printing with newlines and tabs.
- All serial data is transmitted as character that can be converted to integers in a variety of ways.
- You can send comma-separated integer lists and use integrated functions to parse them into commands for your sketch.
- You can send data from your Arduino to a Processing desktop application.
- You can receive data from a Processing application on your desktop to control peripherals connected to your Arduino.
- An Arduino Leonardo can be used to emulate a keyboard or mouse.