Part I

Arduino Engineering Basics

In This Part

CHAPTER 1

Getting Up and Blinking with the Arduino

Parts You'll Need for This Chapter:

Arduino Uno

USB cable

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at `www.exploringarduino.com/content/ch1`.

In addition, all code can be found at `www.wiley.com/go/exploringarduino` on the Download Code tab. The code is in the chapter 01 download and individually named according to the names throughout the chapter.

Now that you have some perspective on the Arduino platform and its capabilities, it's time to explore your options in the world of Arduino. In this chapter, you examine the available hardware, learn about the programming environment and language, and get your first program up and running. Once you have a grip on the functionality that the Arduino can provide, you'll write your first program and get the Arduino to blink!

NOTE To follow along with a video that introduces the Arduino platform, visit `www.jeremyblum.com/2011/01/02/arduino-tutorial-series-it-begins/`. You can also find this video on the Wiley website shown at the beginning of this chapter.

Exploring the Arduino Ecosystem

In your adventures with the Arduino, you'll depend on three main components for your projects:

- The Arduino board itself
- External hardware (including both shields and hand-made circuits, which you'll explore throughout this book)
- The Arduino integrated development environment, or Arduino IDE

All these system components work in tandem to enable you do just about anything with your Arduino.

You have a lot of options when it comes to Arduino development boards, but this book focuses on using official Arduino boards. Because the boards are all designed to be programmable via the same IDE, you can generally use any of the modern Arduino boards to complete the projects in this book with zero or minor changes. However, when necessary, you'll see caveats about using different boards for various projects. The majority of the projects use the Arduino Uno.

You start by exploring the basic functionality baked in to every Arduino board. Then you examine the differences between each modern board so that you can make an informed decision when choosing a board to use for your next project.

Arduino Functionality

All Arduino boards have a few key capabilities and functions. Take a moment to examine the Arduino Uno (see Figure 1-1); it will be your base configuration. These are some key components that you'll be concerning yourself with:

- Atmel microcontroller
- USB programming/communication interface(s)
- Voltage regulator and power connections
- Breakout I/O pins
- Debug, Power, and RX/TX LEDs
- Reset button
- In-circuit serial programmer (ICSP) connector(s)

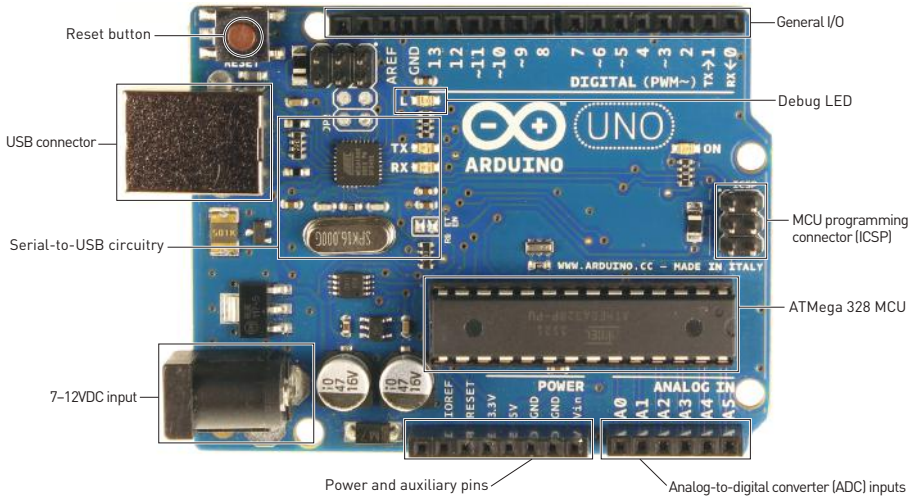


Figure 1-1: Arduino Uno components

Credit: Arduino, www.arduino.cc

Atmel Microcontroller

At the heart of every Arduino is an Atmel microcontroller unit (MCU). Most Arduino boards, including the Arduino Uno, use an AVR ATMega microcontroller. The Arduino Uno in Figure 1-1 uses an ATMega 328p. The Due is an exception; it uses an ARM Cortex microcontroller. This microcontroller is responsible for holding all of your compiled code and executing the commands you specify. The Arduino programming language gives you access to microcontroller peripherals, including analog-to-digital converters (ADCs), general-purpose input/output (I/O) pins, communication buses (including I^2C and SPI), and serial interfaces. All of this useful functionality is broken out from the tiny pins on the microcontroller to accessible female headers on the Arduino that you can plug wires or shields into. A 16 MHz ceramic resonator is wired to the ATMega's clock pins, which serves as the reference by which all program commands execute. You can use the Reset button to restart the execution of your program. Most Arduino boards come with a debug LED already connected to pin 13, which enables you to run your first program (blinking an LED) without connecting any additional circuitry.

Programming Interfaces

Ordinarily, ATMega microcontroller programs are written in C or Assembly and programmed via the ICSP interface using a dedicated programmer (see Figure 1-2). Perhaps the most important characteristic of an Arduino is that you can program it easily via USB, without using a separate programmer. This functionality is made possible by the Arduino bootloader. The bootloader is loaded onto the ATMega at the factory (using the ICSP header), which allows a serial USART (Universal Synchronous/Asynchronous Receiver/Transmitter) to load your program on the Arduino without using a separate programmer. (You can learn more about how the bootloader functions in "The Arduino Bootloader and Firmware Setup" sidebar.)

In the case of the Arduino Uno and Mega 2560, a secondary microcontroller (an ATMega 16U2 or 8U2 depending on your revision) serves as an interface between a USB cable and the serial USART pins on the main microcontroller. The Arduino Leonardo, which uses an ATMega 32U4 as the main microcontroller, has USB baked right in, so a secondary microcontroller is not needed. In older Arduino boards, an FTDI brand USB-to-serial chip was used as the interface between the ATMega's serial USART port and a USB connection.

Figure 1-2: AVR ISP MKII programmer

General I/O and ADCs

The part of the Arduino that you'll care the most about during your projects is the general-purpose I/O and ADC pins. All of these pins can be individually addressed via the programs you'll write. All of them can serve as digital inputs and outputs. The ADC pins can also act as analog inputs that can measure voltages between 0 and 5V (usually from resistive sensors). Many of these pins are also multiplexed to serve additional functions, which you will explore during your projects. These special functions include various communication interfaces, serial interfaces, pulse-width-modulated outputs, and external interrupts.

Power Supplies

For the majority of your projects, you will simply use the 5V power that is provided over your USB cable. However, when you're ready to untether your project from a computer, you have other power options. The Arduino can accept between 6V and 20V (7-12V recommend) via the direct current (DC) barrel jack connector, or into the V_{in} pin. The Arduino has built-in 5V and 3.3V regulators:

- 5V is used for all the logic on the board. In other words, when you toggle a digital I/O pin, you are toggling it between 5V and 0V.
- 3.3V is broken out to a pin to accommodate 3.3V shields and external circuitry.

THE ARDUINO BOOTLOADER AND FIRMWARE SETUP

A *bootloader* is a chunk of code that lives in a reserved space in the program memory of the Arduino's main MCU. In general, AVR microcontrollers are programmed with an ICSP, which talks to the microcontroller via a serial peripheral interface (SPI). Programming via this method is fairly straightforward, but necessitates the user having a hardware programmer such as an STK500 or an AVR ISP MKII programmer (see Figure 1-2).

When you first boot the Arduino board, it enters the bootloader, which runs for a few seconds. If it receives a programming command from the IDE over the MCU's UART (serial interface) in that time period, it loads the program that you are sending it into the rest of the MCU's program memory. If it does not receive a programming command, it starts running your most recently uploaded sketch, which resides in the rest of the program memory.

When you send an "upload" command from the Arduino IDE, it instructs the USB-to-serial chip (an ATMega 16U2 or 8U2 in the case of the Arduino Uno) to reset the main MCU, hence forcing it into the bootloader. Then, your computer immediately begins to send the program contents, which the MCU is ready to receive over its UART connection (facilitated by the USB-to-serial converter).

Bootloaders are great because they enable simple programming via USB with no external hardware. However, they do have two downsides:

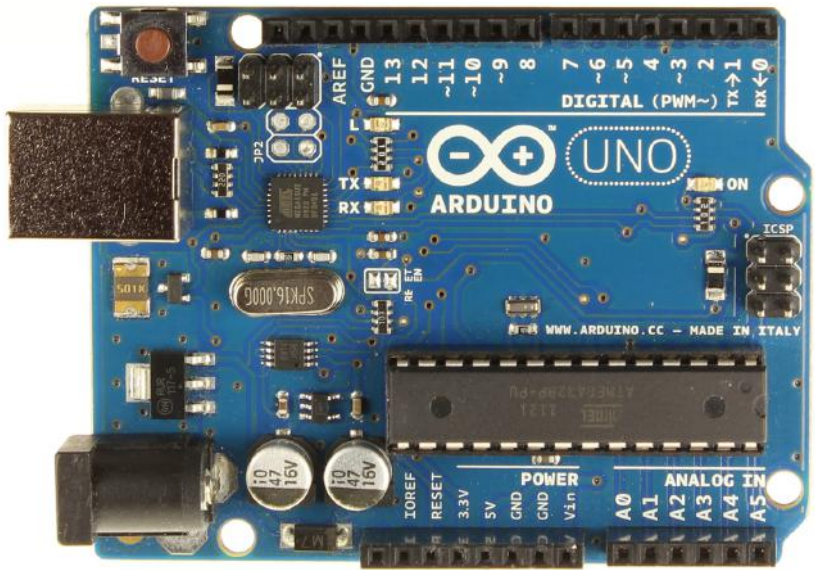
- First, they take up valuable program space. If you have written a complicated sketch, the approximately 2KB of space taken up by the bootloader might be really valuable.
- Second, using a bootloader means that your program will always be delayed by a few seconds at boot as the bootloader checks for a programming request.

If you have a programmer (or another Arduino that can be programmed to act as a programmer), you can remove the bootloader from your ATMega and program it directly by connecting your programmer to the ICSP header and using the File ⇨ Upload Using Programmer command from within the IDE.

Arduino Boards

This book cannot possibly cover all the available Arduino boards; there are many, and manufacturers are constantly releasing new ones with various features. The following section highlights some of the features in the official Arduino boards.

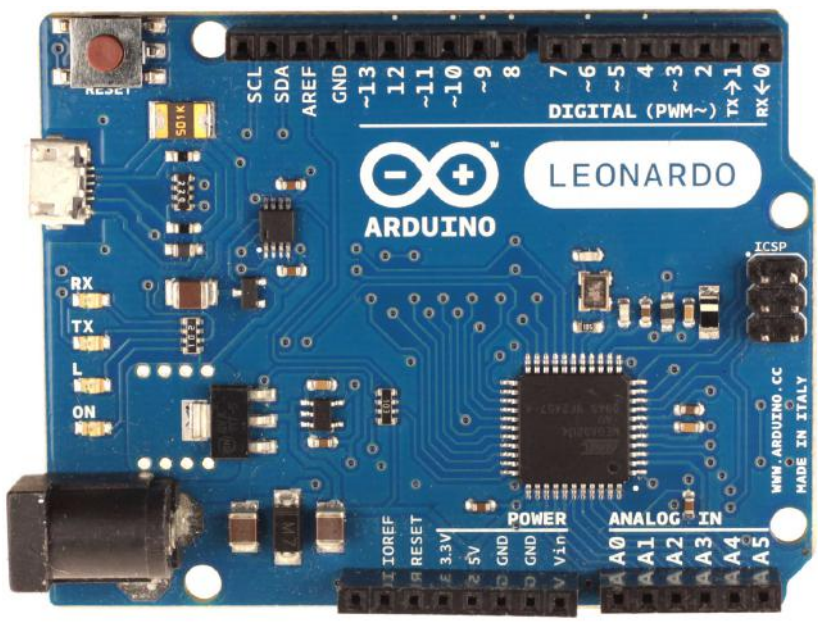
The Uno (see Figure 1-3) is the flagship Arduino and will be used heavily in this book. It uses a 16U2 USB-to-serial converter chip and an ATMega 328p as the main MCU. It is available in both DIP and SMD versions (which defines whether the MCU is removable).



Credit: Arduino, www.arduino.cc

Figure 1-3: The Arduino Uno

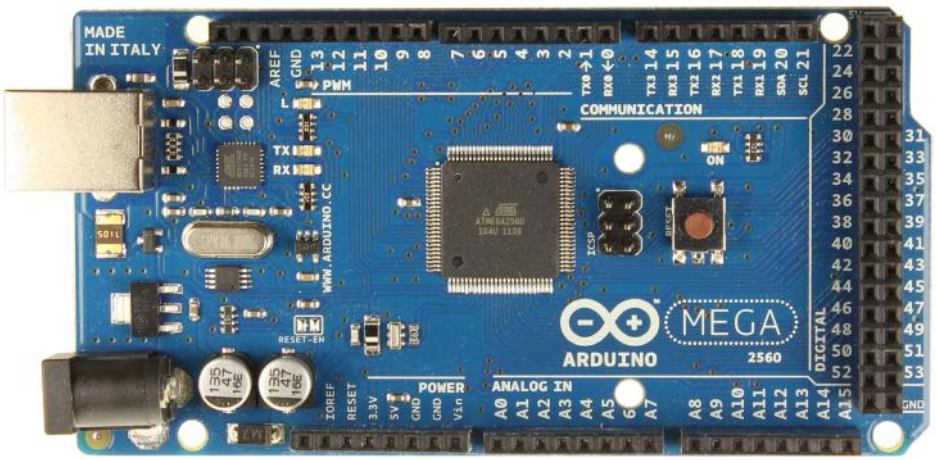
The Leonardo (see Figure 1-4) uses the 32U4 as the main microcontroller, which has a USB interface built in. Therefore, it doesn't need a secondary MCU to perform the serial-to-USB conversion. This cuts down on the cost and enables you to do unique things like emulate a joystick or a keyboard instead of a simple serial device. You will learn how to use these features in Chapter 6, "USB and Serial Communication."



Credit: Arduino, www.arduino.cc

Figure 1-4: The Arduino Leonardo

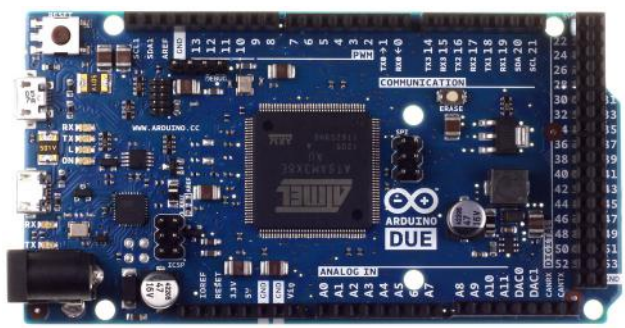
The Mega 2560 (see Figure 1-5) employs an ATMega 2560 as the main MCU, which has 54 general I/Os to enable you to interface with many more devices. The Mega also has more ADC channels, and has four hardware serial interfaces (unlike the one serial interface found on the Uno).



Credit: Arduino, www.arduino.cc

Figure 1-5: The Arduino Mega 2560

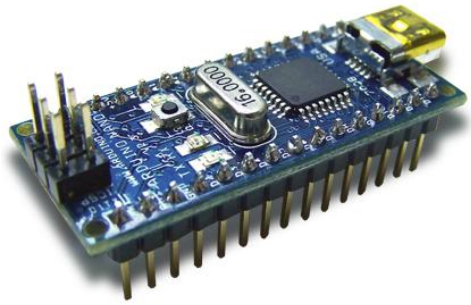
Unlike all the other Arduino variants, which use 8-bit AVR MCUs, the Due (see Figure 1-6) uses a 32-bit ARM Cortex M3 SAM3X MCU. The Due offers higher-precision ADCs, selectable resolution pulse-width modulation (PWM), Digital-to-Analog Converters (DACs), a USB host connector, and an 84 MHz clock speed.



Credit: Arduino, www.arduino.cc

Figure 1-6: The Arduino Due

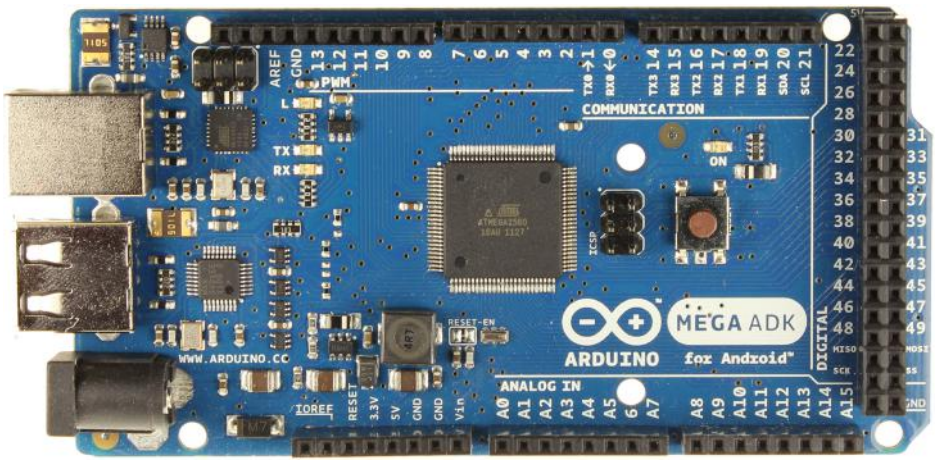
The Nano (see Figure 1-7) is designed to be mounted right into a breadboard socket. Its small form factor makes it perfect for use in more finished projects.



Credit: Cooking Hacks, www.cookinghacks.com

Figure 1-7: The Arduino Nano

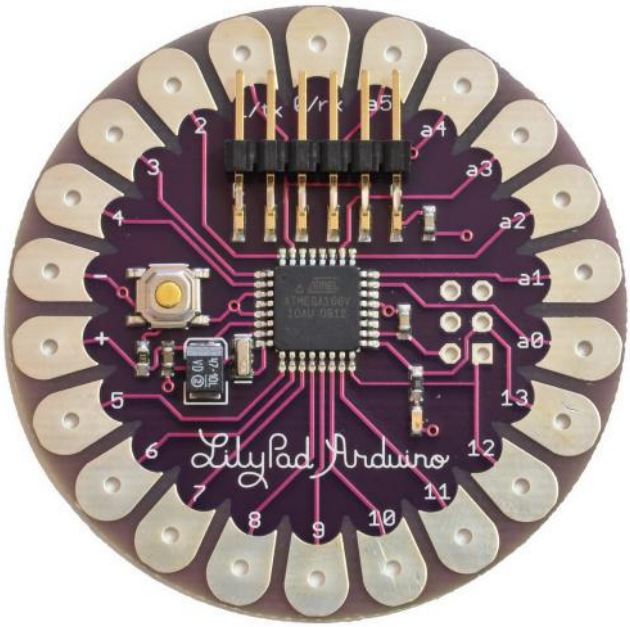
The Mega ADK (see Figure 1-8) is very similar to the Mega 2560, except that it has USB host functionality, allowing it to connect to an Android phone so that it can communicate with apps that you write.



Credit: Arduino, www.arduino.cc

Figure 1-8: The Arduino Mega ADK

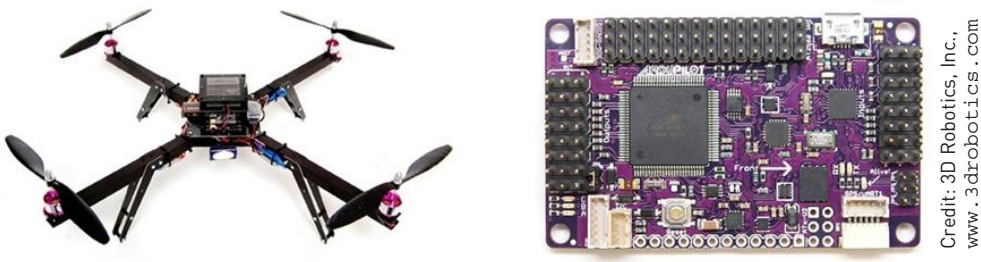
The LilyPad (see Figure 1-9) is unique because it is designed to be sewn into clothing. Using conductive thread, you can wire it up to sewable sensors, LEDs, and more. To keep size down, you need to program it using an FTDI cable.



Credit: Arduino, www.arduino.cc

Figure 1-9: The LilyPad Arduino

As explained in this book's introduction, the Arduino is open source hardware. As a result, you can find dozens and dozens of "Arduino compatible" devices available for sale that will work just fine with the Arduino IDE and all the projects you'll do in this book. Some of the popular third-party boards include the Seeeduino, the Adafruit 32U4 breakout board, and the SparkFun Pro Mini Arduino boards. Many third-party boards are designed for very particular applications, with additional functionality already built into the board. For example, the ArduPilot is an autopilot board for use in autonomous DIY quadcopters (see Figure 1-10). You can even find Arduino-compatible circuitry baked into consumer devices like the MakerBot Replicator and Replicator 2 3D printers.



Credit: 3D Robotics, Inc., www.3drobotics.com

Figure 1-10: Quadcopter and ArduPilot Mega controller

Creating Your First Program

Now that you understand the hardware that you'll be using throughout this book, you can install the software and run your first program. Start by downloading the Arduino software to your computer.

Downloading and Installing the Arduino IDE

Access the Arduino website at `www.arduino.cc` and download the newest version of the IDE from the Download page (see Figure 1-11).

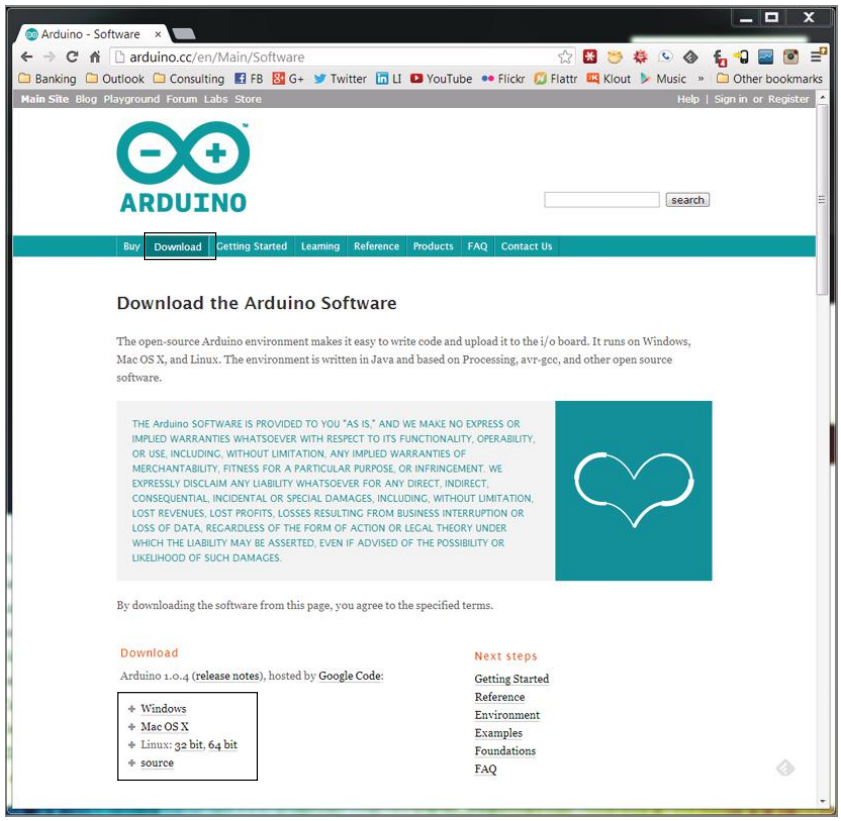


Figure 1-11: The Arduino.cc Download page

After completing the download, unzip it. Inside, you'll find the Arduino IDE. New versions of the Windows IDE are available as an installer that you can download and run, instead of downloading a ZIP file.

Running the IDE and Connecting to the Arduino

Now that you have the IDE downloaded and ready to run, you can connect the Arduino to your computer via USB, as shown in Figure 1-12. Mac and Linux machines install the drivers (mostly) automatically.

If you are using OS X, the first time you plug in an Uno or a Mega 2560, you will get a notification that a new network device has been added. Click the Network Preferences button. In the new window, click Apply. Even though the board will appear as "Not Configured" in the network device list, it will be ready to use. Now, quit System Preferences.

If you are using a modern Arduino on a Windows computer, you will probably need to install drivers. You can skip the following directions if you are not using a Windows computer that needs to have drivers installed. If you installed the IDE using the Windows installer, then these steps have been completed for you. If you downloaded the ZIP on your Windows machine, then you will need to follow the directions shown next.

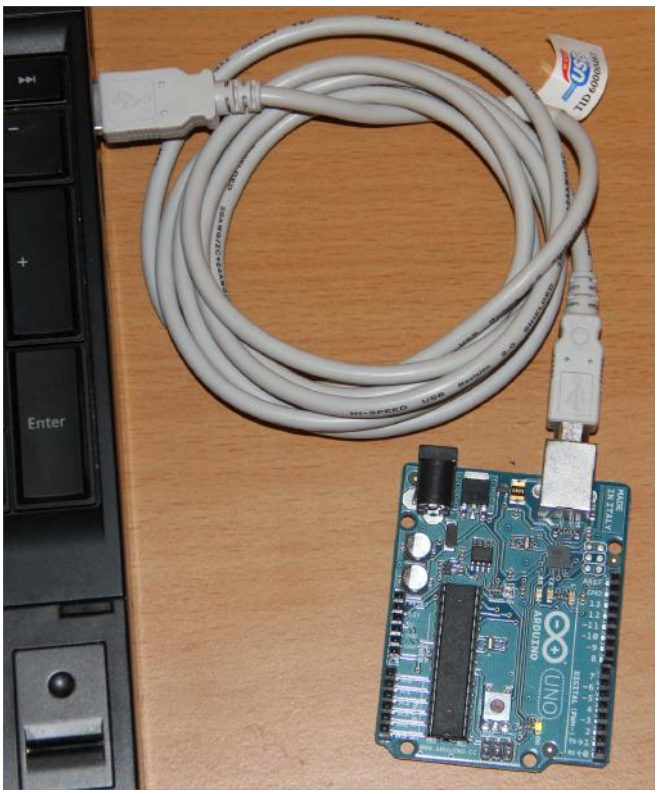


Figure 1-12: Arduino Uno connected to a computer via USB

On your Windows computer, follow these steps to install the drivers (instructions adapted from the Arduino.cc website):

1. Wait for the automatic install process to fail.
2. Open the Start menu, right-click My Computer, and select Properties.
3. Choose Device Manager.
4. Look under Ports (COM and LPT) for the Arduino that you connected.
5. Right-click it and choose Update Driver Software.
6. Choose to browse your computer for software.
7. Select the appropriate driver from the drivers directory of the Arduino IDE that you just downloaded (not the FTDI drivers directory).
8. Windows will now finish the driver installation.

Now, launch the Arduino IDE. You're ready to load your first program onto your Arduino. To ensure that everything is working as expected, you'll load the Blink example program, which will blink the onboard LED. Most Arduinos have an LED connected to pin 13. Navigate to File ➪ Examples ➪ Basic, and click the Blink program. This opens a new IDE window with the Blink program already written for you. First, you'll program the Arduino with this example sketch, and then you'll analyze the program to understand the important components so that you can start to write your own programs in the next chapter.

Before you load the program, you need to tell the IDE what kind of Arduino you have connected and what port it is connected to. Go to Tools ➪ Board and ensure that the right board is selected. This example uses the Uno, but if you are using a different board, select that one (assuming that it also has an LED connected to pin 13).

The last step before programming is to tell the IDE what port your board is connected to. Navigate to Tools ➪ Serial Port and select the appropriate port. On Windows machines, this will be `COM*`, where * is some number representing the serial port number.

TIP If you have multiple serial devices attached to your computer, try unplugging your board to see which COM port disappears from the menu; then plug it back in and select that COM port.

On Linux and Mac computers, the serial port looks something like `/dev/tty.usbmodem*` or `/dev/tty.usbserial*`, where `*` is a string of alphanumeric characters.

You're finally ready to load your first program. Click the Upload button ( ) on the top left of the IDE. The status bar at the bottom of the IDE shows a progress bar as it compiles and uploads your program. When the upload completes, the yellow LED on your Arduino should be blinking once per second. Congratulations! You've just uploaded your first Arduino program.

Breaking Down Your First Program

Take a moment to deconstruct the Blink program so that you understand the basic structure of programs written for the Arduino. Consider Figure 1-13. The numbered callouts shown in the figure correspond to the following list.

Here's how the code works, piece by piece:

1. This is a multiline comment. Comments are important for documenting your code. Everything you write between these symbols will not be compiled or even seen by your Arduino. Multiline comments start with `/*` and end with `*/`. Multiline comments are generally used when you have to say a lot (like the description of this program).
2. This is a single-line comment. When you put `//` on any line, the compiler ignores all text after that symbol on the same line. This is great for annotating specific lines of code or for "commenting out" a particular line of code that you believe might be causing problems.
3. This code is a variable declaration. A variable is a place in the Arduino's memory that holds information. Variables have different types. In this case, it's of type `int`, which means it will hold an integer. In this case, an integer variable called `led` is being set to the value of `13`, the pin that the LED is connected to on the Arduino Uno. Throughout the rest of the program, we can simply use `led` whenever we want to control pin 13. Setting variables is useful because you can just change this one line if you hook up your LED to a different I/O pin later on; the rest of the code will still work as expected.
4. `void setup()` is one of two functions that must be included in every Arduino program. A *function* is a piece of code that does a specific task. Code within the curly braces of the `setup()` function is executed once at the start of the program. This is useful for one-time settings, such as setting the direction of pins, initializing communication interfaces, and so on.

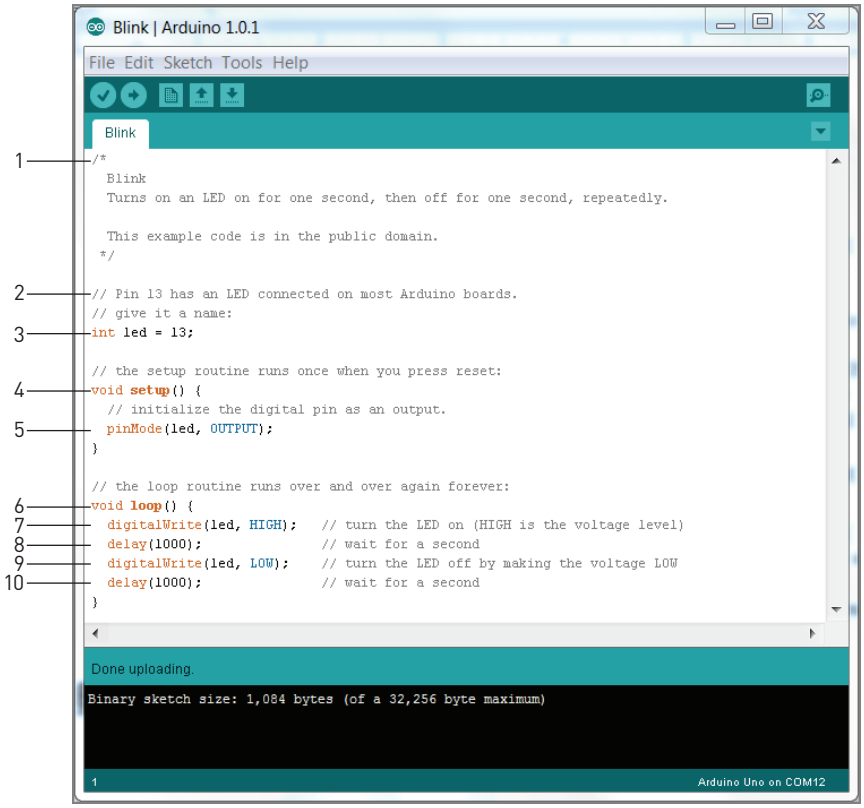


Figure 1-13: The components of the Blink program

5. The Arduino's digital pins can function as input or outputs. To configure their direction, use the command `pinMode()`. This command takes two arguments. An *argument* gives commands information on how they should operate. Arguments are placed inside the parentheses following a command. The first argument to `pinMode` determines which pin is having its direction set. Because you defined the `led` variable earlier in the program, you are telling the command that you want to set the direction of pin 13. The second argument sets the direction of the pin: `INPUT` or `OUTPUT`. Pins are inputs by default, so you need to explicitly set them to outputs if you want them to function as outputs. Because you want to light an LED, you have set the led pin to an output (current is flowing out of the I/O pin). Note that you have to do this only one time. It will then function as an output for the rest of the program, or until you change it to an input.

6. The second required function in all Arduino programs is `void loop()`. The contents of the loop function repeat forever as long as the Arduino is on. If you want your Arduino to do something once at boot only, you still need to include the loop function, but you can leave it empty.
7. `digitalWrite()` is used to set the state of an output pin. It can set the pin to either 5V or 0V. When an LED and resistor is connected to a pin, setting it to 5V will enable you to light up the LED. (You learn more about this in the next chapter.) The first argument to `digitalWrite()` is the pin you want to control. The second argument is the value you want to set it to, either `HIGH` (5V) or `LOW` (0V). The pin remains in this state until it is changed in the code.
8. The `delay()` function accepts one argument: a delay time in milliseconds. When calling `delay()`, the Arduino stops doing anything for the amount of time specified. In this case, you are delaying the program for 1000ms, or 1 second. This results in the LED staying on for 1 second before you execute the next command.
9. Here, `digitalWrite()` is used to turn the LED off, by setting the pin state to `LOW`.
10. Again, we delay for 1 second to keep the LED in the off state before the loop repeats and switches to the on state again.

That's all there is to it. Don't be intimidated if you don't fully understand all the code yet. As you put together more examples in the following chapters, you'll become more and more proficient at understanding program flow, and writing your own code.

Summary

In this chapter you learned about the following:

- All the components that comprise an Arduino board
- How the Arduino bootloader allows you to program Arduino firmware over a USB connection
- The differences between the various available Arduino boards
- How to connect and install the Arduino with your system
- How to load and run your first program

CHAPTER 2

Digital Inputs, Outputs, and Pulse-Width Modulation

Parts You'll Need for This Chapter:

Arduino Uno
Small breadboard
Jumper wires
1 10kΩ resistor
3 220Ω resistors
USB cable
Pushbutton
5mm single-color LED
5mm common-cathode RGB LED

CODE AND DIGITAL CONTENT FOR THIS CHAPTER

Code downloads, videos, and other digital content for this chapter can be found at `www.exploringarduino.com/content/ch2`.

In addition, all code can be found at `www.wiley.com/go/exploringarduino` on the Download Code tab. The code is in the chapter 02 download and individually named according to the names throughout the chapter.

Blinking an LED is great, as you learned in the preceding chapter, but what makes the Arduino microcontroller platform so useful is that the system is equipped with both inputs and outputs. By combining both, your opportunities are nearly limitless. For example, you can use a magnetic reed switch to play music when your door opens, create an electronic lockbox, or build a light-up musical instrument!

In this chapter, you start to learn the skills you need to build projects like these. You explore the Arduino's digital input capabilities, learn about pullup and pulldown resistors, and learn how to control digital outputs. Most Arduinos do not have analog outputs, but it is possible to use digital pulse-width modulation to emulate it in many scenarios. You learn about generating pulse-width modulated signals in this chapter. You will also learn how to debounce digital switches, a key skill when reading human input. By the end of the chapter, you will be able to build and program a controllable RGB (Red, Green, Blue) LED nightlight.

NOTE You can follow along with a video as I teach you about digital inputs and outputs, debouncing, and pulse-width modulation (PWM): `www.jeremyblum.com/2011/01/10/arduino-tutorial-2-now-with-more-blinky-things/`. You can also find this video on the Wiley website shown at the beginning of this chapter.

If you want to learn more about some of the basics of electrical engineering touched on in this chapter, watch this video: `www.jeremyblum.com/2011/01/17/electrical-engineering-basics-in-arduino-tutorial-3/`. You can also find this video on the Wiley website shown at the beginning of this chapter.

Digital Outputs

In Chapter 1, "Getting Up and Blinking with the Arduino," you learned how to blink an LED. In this chapter, you will further explore Arduino digital output capabilities, including the following topics:

- Setting pins as outputs
- Wiring up external components
- New programming concepts, including `for` loops and constants
- Digital versus analog outputs and pulse-width modulation (PWM)

Wiring Up an LED and Using Breadboards

In Chapter 1, you learned how to blink the onboard LED, but what fun is that? Now it is time to whip out the breadboard and wire up an external LED to pin 9 of your Arduino. Adding this external LED will be a stepping-stone towards helping you to understand how to wire up more complex external circuits in

the coming chapters. What's more, pin 9 is PWM-enabled, which will enable you to pursue the analog output examples later in this chapter.

Working with Breadboards

It is important to understand how breadboards work so that you can use them effectively for the projects in this book. A *breadboard* is a simple prototyping tool that easily allows you to wire up simple circuits without having to solder together parts to a custom printed circuit board. First, consider the blue and red lines that run the length of the board. The pins adjacent to these color-coded lines are designed to be used as power and ground buses. All the red pins are electrically connected, and are generally used for providing power. In the case of most Arduinos and the projects in this book, this will generally be at 5V. All the blue pins are electrically connected and are used for the ground bus. All the vertically aligned pins are also connected in rows, with a division in the middle to make it easy to mount integrated circuits on the breadboard. Figure 2-1 highlights how the pins are electrically connected, with all the thick lines representing connected holes.

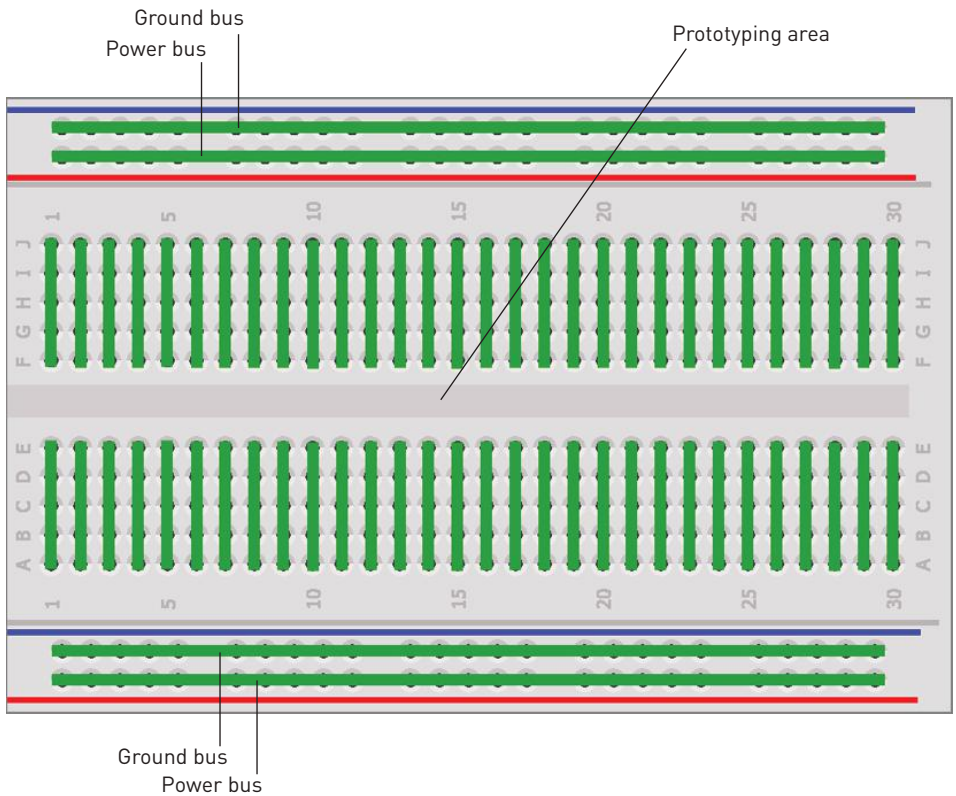


Figure 2-1: Breadboard electrical connections

Wiring LEDs

LEDs will almost certainly be one of the most-used parts in your projects throughout this book. LEDs are polarized; in other words, it matters in what direction you hook them up. The positive lead is called the *anode*, and the negative lead is called the *cathode*. If you look at the clear top of the LED, there will usually be a flat side on the lip of the casing. That side is the cathode. Another way to determine which side is the anode and which is the cathode is by examining the leads. The shorter lead is the cathode.

As you probably already know, LED stands for light-emitting diode. Like all diodes, LEDs allow current to flow in only one direction—from their anode to their cathode. Because current flows from positive to negative, the anode of the LED should be connected to the current source (a 5V digital signal in this case), and the cathode should be connected to ground. The resistor can be inserted in series on either side of the LED. Resistors are not polarized, and so you do not have to worry about their orientation.

You'll wire the LED into pin 9 in series with a resistor. LEDs must always be wired in series with a resistor to serve as a current limiter. The larger the resistor value, the more it restricts the flow of current and the dimmer the LED glows. In this scenario, you use a 220Ω resistor. Wire it up as shown in Figure 2-2.

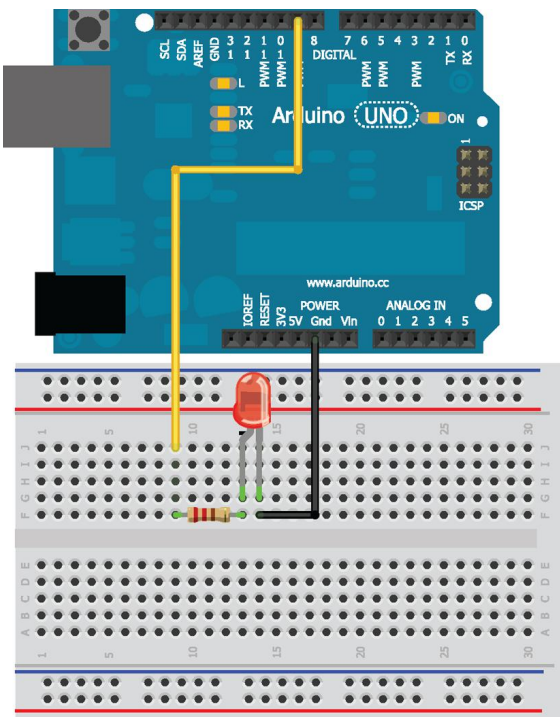


Image created with Fritzing.

Figure 2-2: Arduino Uno wired to an LED

OHM'S LAW AND THE POWER EQUATION

The most important equation for any electrical engineer to know is Ohm's law. Ohm's law dictates the relationship between voltage (measured in volts), current (measured in amps), and resistance (measured in ohms or Ω) in a circuit. A circuit is a closed loop with a source of electrical energy (like a 9V battery) and a load (something to use up the energy, like an LED). Before delving into the law, it is important to understand what each term means, at least at a basic level:

- *Voltage* represents the potential electrical difference between two points.
- *Current* flows from a point of higher potential energy to lower potential energy. You can think of current as a flow of water, and voltage as elevation. Water (or current) always flows from high elevation (higher voltage) to lower elevation (ground, or a lower voltage). Current, like water in a river, will always follow the path of least resistance in a circuit.
- *Resistance*, in this analogy, is representative of how easy it is for current to flow. When the water (the current) is flowing through a narrow pipe, less can pass through in the same amount of time as through a larger pipe. The narrow pipe is equivalent to a high resistance value because the water will have a harder time flowing through. The wider pipe is equivalent to a low resistance value (like a wire) because current can flow freely through it.

Ohm's law is defined as follows:

$V = IR$

Where V is Voltage difference in volts, I is Current in amps, and R is the Resistance in ohms.

In a circuit, all voltage gets used up, and each component offers up some resistance that lowers the voltage. Knowing this, the above equation comes in handy for things like figuring out what resistor value to match up with an LED. LEDs have a predefined voltage drop across them and are designed to operate at a particular current value. The larger the current through the LED, the brighter the LED glows, up to a limit. For the most common LEDs, the maximum current designed to go through an LED is 20milliamps (a milliamp is 1/1000 of an amp and is typically abbreviated as mA). The voltage drop across an LED is defined in its datasheet. A common value is around 2V. Consider the LED circuit shown in Figure 2-3.

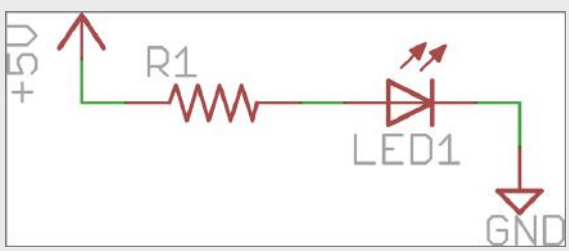


Image created with Eagle.

Figure 2-3: Simple LED circuit

Continues

continued

You can use Ohm's law to decide on a resistor value for this circuit. Assume that this is a standard LED with 20mA forward current and a 2V drop across it. Because the source voltage is 5V and it ends at ground, a total of 5V must drop across this circuit. Since the LED has a 2V drop, the other 3V must drop across the resistor. Knowing that you want approximately 20mA to flow through these components, you can find the resistor value by solving for R:

R = V/I

Where V = 3V and I = 20mA.

Solving for R, R = 3V / 0.02A = 150Ω. So, with a resistor value of 150Ω, 20mA flows through both the resistor and LED. As you increase the resistance value, less current is allowed to flow through. 220Ω is a bit more than 150Ω, but still allows the LED to glow sufficiently bright, and is a very commonly available resistor value.

Another useful equation to keep in mind is the power equation. The power equation tells you how much power, in watts, is dissipated across a given resistive component. Because increased power is associated with increased heat dissipation, components generally have a maximum power rating. You want to ensure that you do not exceed the maximum power rating for resistors because otherwise they might overheat. A common power rating for resistors is 1/8 of a watt (abbreviated as W, milliwatts as mW). The power equation is as follows:

P = IV

Where P is power in watts, and I and V are still defined as the current and voltage.

For the resistor defined earlier with a voltage drop of 3V and a current of 20mA, P = 3V × 0.02A = 60mW, well under the resistor's rating of 1/8W, or 125mW. So, you do not have to worry about the resistor overheating; it is well within its operating limits.

Programming Digital Outputs

By default, all Arduino pins are set to inputs. If you want to make a pin an output, you need to first tell the Arduino how the pin should be configured. In the Arduino programming language, the program requires two parts: the `setup()` and the `loop()`.

As you learned in Chapter 1, the `setup()` function runs one time at the start of the program, and the `loop()` function runs over and over again. Because you'll generally dedicate each pin to serve as either an input or an output, it is

common practice to define all your pins as inputs or outputs in the setup. You start by writing a simple program that sets pin 9 as an output and turns it on when the program starts.

To write this program, use the `pinMode()` command to set the direction of pin 9, and use `digitalWrite()` to make the output high (5V). See Listing 2-1.

Listing 2-1: Turning on an LED—led.ino

```
const int LED=9;              //define LED for pin 9
void setup()
{
  pinMode (LED, OUTPUT);    //Set the LED pin as an output
  digitalWrite(LED, HIGH);  //Set the LED pin high
}

void loop()
{
  //we are not doing anything in the loop!
}
```

Load this program onto your Arduino, wired as shown in Figure 2-2. In this program, also notice that I used the `const` operator before defining the pin integer variable. Ordinarily, you'll use variables to hold values that may change during program execution. By putting `const` before your variable declaration, you are telling the compiler that the variable is "read only" and will not change during program execution. All instances of `LED` in your program will be "replaced" with `9` when they are called. When you are defining values that will not change, using the `const` qualifier is recommended. In some of the examples later in this chapter, you will define non-constant variables that may change during program execution.

You must specify the type for any variable that you declare. In the preceding case, it is an integer (pins will always be integers), so you should set it as such. You can now easily modify this sketch to match the one you made in Chapter 1 by moving the `digitalWrite()` command to the loop and adding some delays. Experiment with the delay values and create different blink rates.

Using For Loops

It's frequently necessary to use loops with changing variable values to adjust parameters of a program. In the case of the program you just wrote, you can implement a `for` loop to see how different blink rates impact your system's operation. You can visualize different blink rates by using a `for` loop to cycle through various rates. The code in Listing 2-2 accomplishes that.

Listing 2-2: LED with Changing Blink Rate—blink.ino

```
const int LED=9;            //define LED for Pin 9
void setup()
{
  pinMode (LED, OUTPUT); //Set the LED pin as an output
}

void loop()
{
  for (int i=100; i<=1000; i=i+100)
  {
    digitalWrite(LED, HIGH);
    delay(i);
    digitalWrite(LED, LOW);
    delay(i);
  }
}
```

Compile the preceding code and load it onto your Arduino. What happens? Take a moment to break down the `for` loop to understand how it works. The `for` loop declaration always contains three semicolon-separated entries:

- The first entry sets the index variable for the loop. In this case, the index variable is `i` and is set to start at a value of `100`.
- The second entry specifies when the loop should stop. The contents of the loop will execute over and over again while that condition is true. `<=` indicates less than or equal to. So, for this loop, the contents will continue to execute as long as the variable `i` is still less than or equal to `1000`.
- The final entry specifies what should happen to the index variable at the end of each loop execution. In this case, `i` will be set to its current value plus `100`.

To better understand these concepts, consider what happens in two passes through the `for` loop:

1. `i` equals `100`.
2. The LED is set high, and stays high for 100ms, the current value of `i`.
3. The LED is set low, and stays low for 100ms, the current value of `i`.
4. At the end of the loop, `i` is incremented by `100`, so it is now `200`.
5. `200` is less than or equal to `1000`, so the loop repeats again.
6. The LED is set high, and stays high for 200ms, the current value of `i`.
7. The LED is set low, and stays low for 200ms, the current value of `i`.

8. At the end of the loop, `i` is incremented by `100`, so it is now `300`.
9. This process repeats until `i` surpasses `1000` and the outer loop function repeats, setting the `i` value back to `100` and starting the process again.

Now that you've generated digital outputs from your Arduino, you'll learn about using PWM to create analog outputs from the I/O pins on your Arduino.

Pulse-Width Modulation with analogWrite()

So, you have mastered digital control of your pins. This is great for blinking LEDs, controlling relays, and spinning motors at a constant speed. But what if you want to output a voltage other than 0V or 5V? Well, you can't—unless you are using the digital-to-analog converter (DAC) pins on the Due or are using an external DAC chip.

However, you can get pretty close to generating analog output values by using a trick called *pulse-width modulation* (PWM). Select pins on each Arduino can use the `analogWrite()` command to generate PWM signals that can emulate a pure analog signal when used with certain peripherals. These pins are marked with a ~ on the board. On the Arduino Uno, Pins 3, 5, 6, 9, 10, and 11 are PWM pins. If you're using an Uno, you can continue to use the circuit from Figure 2-1 to test out the `analogWrite()` command with your LED. Presumably, if you can decrease the voltage being dropped across the resistor, the LED should glow more dimly because less current will flow. That is what you will try to accomplish using PWM via the `analogWrite()` command. The `analogWrite()` command accepts two arguments: the pin to control and the value to write to it.

The PWM output is an 8-bit value. In other words, you can write values from 0 to 2^8-1, or 0 to 255. Try using a similar `for` loop structure to the one you used previously to cycle through varying brightness values (see Listing 2-3).

Listing 2-3: LED Fade Sketch—fade.ino

```
const int LED=9;   //define LED for Pin 9
void setup()
{
  pinMode (LED, OUTPUT);    //Set the LED pin as an output
}

void loop()
{
  for (int i=0; i<256; i++)
  {
    analogWrite(LED, i);
    delay(10);
```

```
  }
  for (int i=255; i>=0; i--)
  {
    analogWrite(LED, i);
    delay(10);
  }
}
```

What does the LED do when you run this code? You should observe the LED fading from off to on, then from on to off. Of course, because this is all in the main loop, this pattern repeats ad infinitum. Be sure to note a few differences in this `for` loop. In the first loop, `i++` is just shorthand code to represent `i=i+1`. Similarly, `i--` is functionally equivalent to `i=i-1`. The first `for` loop fades the LED up, and the second loop fades it down.

PWM control can be used in lots of circumstances to emulate pure analog control, but it cannot always be used when you actually need an analog signal. For instance, PWM is great for driving direct current (DC) motors at variable speeds (you experiment with this in later chapters), but it does not work well for driving speakers unless you supplement it with some external circuitry. Take a moment to examine how PWM actually works. Consider the graphs shown in Figure 2-4.

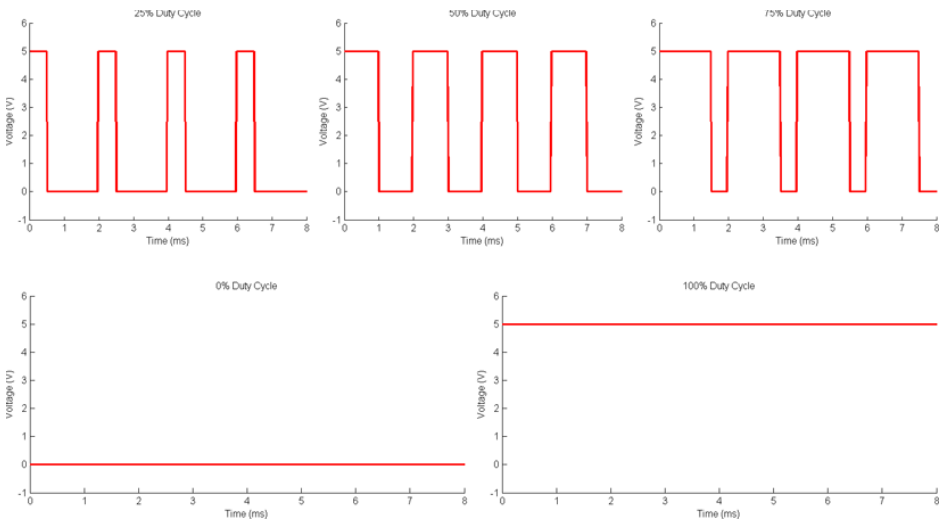


Image created with MATLAB.

Figure 2-4: PWM signals with varying duty cycles

PWM works by modulating the duty cycle of a square wave (a signal that switches on and off). *Duty cycle* refers to the percentage of time that a square wave is high versus low. You are probably most familiar with square waves that have a duty cycle of 50%—they are high half of the time, and low half of the time.

The `analogWrite()` command sets the duty cycle of a square wave depending on the value you pass to it:

- Writing a value of 0 with `analogWrite()` indicates a square wave with a duty cycle of 0 percent (always low).
- Writing a 255 indicates a square wave with a duty cycle of 100 percent (always high).
- Writing a 127 indicates a square wave with a duty cycle of 50 percent (high half of the time, low half of the time).

The graphs in Figure 2-4 show that for a signal with a duty cycle of 25 percent, it is high 25 percent of the time, and low 75 percent of the time. The frequency of this square wave, in the case of the Arduino, is about 490Hz. In other words, the signal varies between high (5V) and low (0V) about 490 times every second.

So, if you are not actually changing the voltage being delivered to an LED, why do you see it get dimmer as you lower the duty cycle? It is really a result of your eyes playing a trick on you! If the LED is switching on and off every 1ms (which is the case with a duty cycle of 50 percent), it appears to be operating at approximately half brightness because it is blinking faster than your eyes can perceive. Therefore, your brain actually averages out the signal and tricks you into believing that the LED is operating at half brightness.

Reading Digital Inputs

Now it is time for the other side of the equation. You've managed to successfully *generate* both digital and analog(ish) outputs. The next step is to *read* digital inputs, such as switches and buttons, so that you can interact with your project in real time. In this section, you learn to read inputs, implement pullup and pulldown resistors, and debounce a button in software.

Reading Digital Inputs with Pulldown Resistors

You should start by modifying the circuit that you first built from Figure 2-1. Following Figure 2-5, you'll add a pushbutton and a pulldown resistor connected to a digital input pin.

TIP Be sure to also connect the power and ground buses of the breadboard to the Arduino. Now that you're using multiple devices on the breadboard, that will come in handy.

Before you write the code to read from the pushbutton, it is important to understand the significance of the pulldown resistor used with this circuit. Nearly

all digital inputs use a pullup or pulldown resistor to set the "default state" of the input pin. Imagine the circuit in Figure 2-5 without the 10kΩ resistor. In this scenario, the pin would obviously read a high value when the button is pressed.

But, what happens when the button is not being pressed? In that scenario, the input pin you would be reading is essentially connected to nothing—the input pin is said to be "floating." And because the pin is not physically connected to 0V or 5V, reading it could cause unexpected results as electrical noise on nearby pins causes its value to fluctuate between high and low. To remedy this, the pulldown resistor is installed as shown in Figure 2-5.

Now, consider what happens when the button is not pressed with the pulldown resistor in the circuit: The input pin will be connected through a 10kΩ resistor to ground. While the resistor will restrict the flow of current, there is still enough current flow to ensure that the input pin will read a low logic value. 10kΩ is a fairly common pulldown resistor value. Larger values are said to be *weak pulldowns* because it easier to overcome them, and smaller resistor values are said to be *strong pulldowns* because it is easier for more current to flow through them to ground. When the button is pressed, the input pin is directly connected to 5V through the button.

Now, the current has two options:

- It can flow through a nearly zero resistance path to the 5V rail.
- It can flow through a high resistance path to the ground rail.

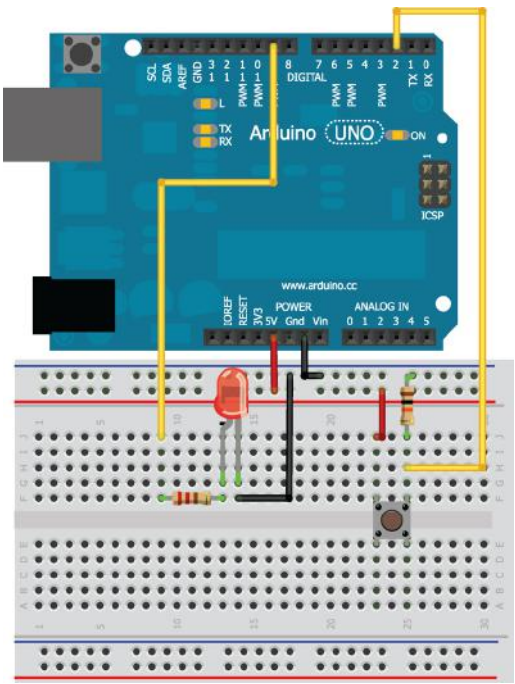


Image created with Fritzing.

Figure 2-5: Wiring an Arduino to a button and an LED

Recall from the previous sidebar on Ohm's law and the power equation that current will always follow the path of the least resistance in a circuit. In this scenario, the majority of the current flows through the button, and a high logic level is generated on the input pin, because that is the path of least resistance.

NOTE This example uses a pulldown resistor, but you could also use a pullup resistor by connecting the resistor to 5V instead of ground and by connecting the other side of the button to ground. In this setup, the input pin reads a high-logic value when the button is unpressed and a low-logic value when the button is being pressed.

Pulldown and pullup resistors are important because they ensure that the button does not create a short circuit between 5V and ground when pressed and that the input pin is never left in a floating state.

Now it is time to write the program for this circuit! In this first example, you just have the LED stay on while the button is held down, and you have it stay off when the button is released (see Listing 2-4).

Listing 2-4: Simple LED Control with a Button—led_button.ino

```
const int LED=9;      //The LED is connected to pin 9
const int BUTTON=2;      //The Button is connected to pin 2

void setup()
{
  pinMode (LED, OUTPUT);     //Set the LED pin as an output
  pinMode (BUTTON, INPUT);   //Set button as input (not required)
}

void loop()
{
  if (digitalRead(BUTTON) == LOW)
  {
    digitalWrite(LED, LOW);
  }
  else
  {
    digitalWrite(LED, HIGH);
  }
}
```

Notice here that the code implements some new concepts, including `digitalRead` and `if/else` statements. A new `const int` statement has been added for the button pin. Further, this code defines the button pin as an input in the `setup` function. This is not explicitly necessary, though, because pins are inputs by default; it is shown for completeness. `digitalRead()` reads the

value of an input. In this case, it is reading the value of the `BUTTON` pin. If the button is being pressed, `digitalRead()` returns a value of `HIGH`, or `1`. If it is not being pressed, it returns `LOW`, or `0`. When placed in the `if()` statement, you're checking the state of the pin and evaluating if it matches the condition you've declared. In this `if()` statement, you're checking to see if the value returned by `digitalRead()` is `LOW`. The `==` is a comparison operator that tests whether the first item (`digitalRead()`) is equal to the second (`LOW`). If this is true (that is, the button is not being pressed), the code inside the brackets executes, and the LED set to `LOW`. If this is not true (the button is being pressed), the `else` statement is executed, and the LED is turned `HIGH`.

That's it! Program your circuit with this code and confirm that it works as expected.

Working with "Bouncy" Buttons

When was the last time you had to hold a button down to keep a light on? Probably never. It makes more sense to be able to click the button once to turn it on and to click the button again to turn it off. This way, you do not have to hold the button down to keep the light on. Unfortunately, this is not quite as easy as you might first guess. You cannot just look for the value of the switch to change from low to high; you need to worry about a phenomenon called *switch bouncing*.

Buttons are mechanical devices that operate as a spring-damper system. In other words, when you push a button down, the signal you read does not just go from low to high, it bounces up and down between those two states for a few milliseconds before it settles. Figure 2-6 illustrates the expected behavior next to the actual behavior you might see when probing the button using an oscilloscope (though this figure was generated using a MATLAB script):

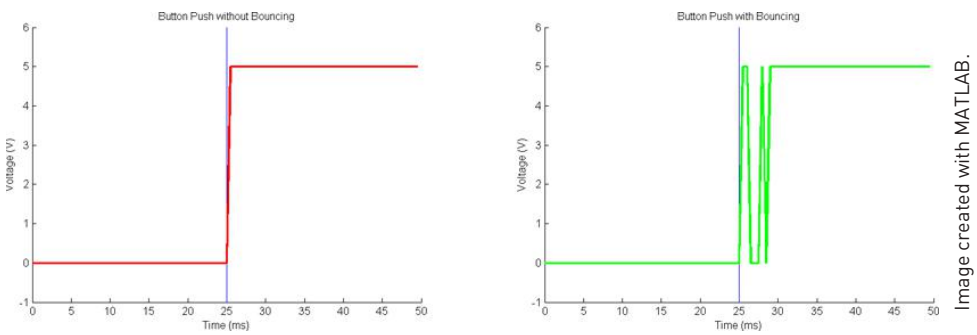


Image created with MATLAB.

Figure 2-6: Bouncing button effects.

The button is physically pressed at the 25ms mark. You would expect the button state to be immediately read as a high logic level as the graph on the left

shows. However, the button actually bounces up and down before settling, as the graph on the right shows.

If you know that the switch is going to do this, it is relatively straightforward to deal with it in software. Next, you write switch-debouncing software that looks for a button state change, waits for the bouncing to finish, and then reads the switch state again. This program logic can be expressed as follows:

1. Store a previous button state and a current button state (initialized to `LOW`).
2. Read the current button state.
3. If the current button state differs from the previous button state, wait 5ms because the button must have changed state.
4. After 5ms, reread the button state and use that as the current button state.
5. If the previous button state was low, and the current button state is high, toggle the LED state.
6. Set the previous button state to the current button state.
7. Return to step 2.

This is a perfect opportunity to explore using *functions* for the first time. *Functions* are blocks of code that can accept input arguments, execute code based on those arguments, and optionally return a result. Without realizing it, you've already been using predefined functions throughout your programs. For example, `digitalWrite()` is a function that accepts a pin and a state, and writes that state to the given pin. To simplify your program, you can define your own functions to encapsulate actions that you do over and over again.

Within the program flow (listed in the preceding steps) is a series of repeating steps that need to be applied to changing variable values. Because you'll want to repeatedly debounce the switch value, it's useful to define the steps for debouncing as a function that can be called each time. This function will accept the previous button state as an input and outputs the current debounced button state. The following program accomplishes the preceding steps and switches the LED state every time the button is pressed. You'll use the same circuit as the previous example for this. Try loading it onto your Arduino and see how it works (see Listing 2-5).

Listing 2-5: Debounced Button Toggling—debounce.ino

```
const int LED=9;              //The LED is connected to pin 9
const int BUTTON=2;           //The Button is connected to pin 2
boolean lastButton = LOW;     //Variable containing the previous
                              //button state
boolean currentButton = LOW; //Variable containing the current
                              //button state
```

```
boolean ledOn = false;       //The present state of the LED (on/off)

void setup()
{
  pinMode (LED, OUTPUT);     //Set the LED pin as an output
  pinMode (BUTTON, INPUT);   //Set button as input (not required)
}

/*
* 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
}

void loop()
{
  currentButton = debounce(lastButton);           //read debounced state
  if (lastButton == LOW && currentButton == HIGH) //if it was pressed...
  {
    ledOn = !ledOn;                               //toggle the LED value
  }
  lastButton = currentButton;                     //reset button value

  digitalWrite(LED, ledOn);                       //change the LED state

}
```

Now, break down the code in Listing 2-5. First, constant values are defined for the pins connected to the button and LED. Next, three Boolean *variables* are declared. When the `const` qualifier is not placed before a variable declaration, you are indicating that this variable can change within the program. By defining these values at the top of the program, you are declaring them as *global* variables that can be used and changed by any function within this sketch. The three Boolean variables declared at the top of this sketch are *initialized* as well, meaning that they have been set to an initial value (`LOW`, `LOW`, and `false` respectively). Later in the program, the values of these variables can be changed with an assignment operator (a single equals sign: `=`).

Consider the function definition in the preceding code: `boolean debounce(boolean last)`. This function accepts a Boolean (a data type that has only two states: true/false, high/low, on/off, 1/0) input variable called `last` and returns a Boolean value representing the current debounced pin value. This function compares the current button state with the previous (`last`) button state that was passed to it as an argument. The `!=` represents inequality and is used to compare the present and previous button values in the `if` statement. If they differ, then the button must have been pressed and the `if` statement will execute its contents. The `if` statement waits 5ms before checking the button state again. This 5ms gives sufficient time for the button to stop bouncing. The button is then checked again to ascertain its stable value. As you learned earlier, functions can optionally return values. In the case of this function, the `return current` statement returns the value of the `current` Boolean variable when the function is called. `current` is a *local* variable—it is declared and used only within the debounce function. When the debounce function is called from the main loop, the returned value is written to the *global* `currentButton` variable that was defined at the top of the sketch. Because the function was defined as `debounce`, you can call the function by writing `currentButton = debounce(lastButton)` from within the `setup` or `loop` functions. `currentButton` will be set equal to the value that is returned by the `debounce` function.

After you've called the function and populated the `currentButton` variable, you can easily compare it to the previous button state by using the `if` statement in the code. The `&&` is a logical operator that means "AND". By joining two or more equality statements with an `&&` in an `if` statement, you are indicating that the contents of the `if` statement block should execute only if both of the equalities evaluate to `true`. If the button was previously `LOW`, and is now `HIGH`, you can assume that the button has been pressed, and you can invert the value of the `ledOn` variable. By putting an `!` in front of the `ledOn` variable, you reset the variable to the opposite of whatever it currently is. The loop is finished off by updating the previous button variable and writing the updated LED state.

This code should change the LED state each time the button is pressed. If you try to accomplish the same thing without debouncing the button, you will find the results unpredictable, with the LED sometimes working as expected and sometimes not.

Building a Controllable RGB LED Nightlight

In this chapter, you have learned how to control digital outputs, how to read debounced buttons, and how to use PWM to change LED brightness. Using those skills, you can now hook up an RGB LED and a debounced button to cycle

through some colors for a controllable RGB LED nightlight. It's possible to mix colors with an RGB LED by changing the brightness of each color.

In this scenario, you use a common cathode LED. That means that the LED has four leads. One of them is a cathode pin that is shared among all three diodes, while the other three pins connect to the anodes of each diode color. Wire that LED up to three PWM pins through current-limiting resistors on the Arduino as shown in the wiring diagram in Figure 2-7.

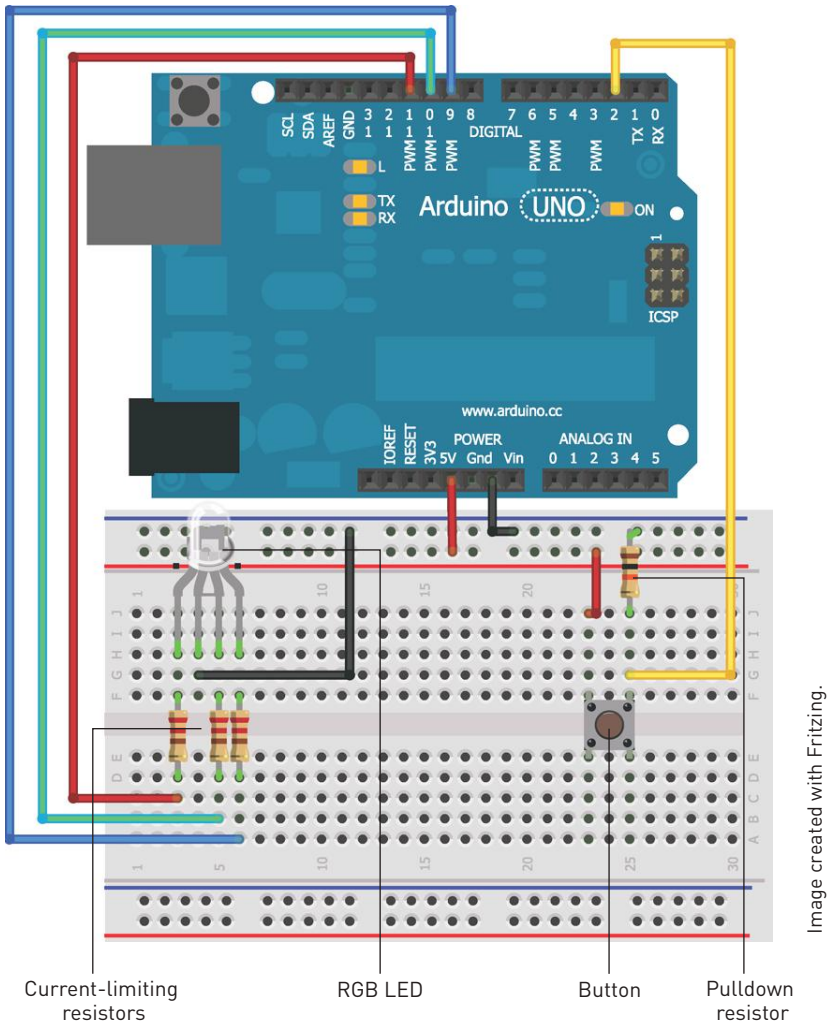


Figure 2-7: Nightlight wiring diagram

You can configure a debounced button to cycle through a selection of colors each time you press it. To do this, it is useful to add an additional function to

set the RGB LED to the next state in the color cycle. In the following program (see Listing 2-6), I have defined seven total color states, plus one off state for the LED. Using the `analogWrite()` function, you can choose your own color-mixing combinations. The only change to the `loop()` from the previous example is that instead of flipping a single LED state, an LED state counter is incremented each time the button is pressed, and it is reset back to zero when you cycle through all the options. Upload this to your Arduino connected to the circuit you just built and enjoy your nightlight. Modify the color states by changing the values of `analogWrite()` to make your own color options.

Listing 2-6: Toggling LED Nightlight—rgb_nightlight.ino

```
const int BLED=9;    //Blue LED on Pin 9
const int GLED=10;   //Green LED on Pin 10
const int RLED=11;   //Red LED on Pin 11
const int BUTTON=2;    //The Button is connected to pin 2

boolean lastButton = LOW;    //Last Button State
boolean currentButton = LOW; //Current Button State
int ledMode = 0;             //Cycle between LED states

void setup()
{
  pinMode (BLED, OUTPUT);   //Set Blue LED as Output
  pinMode (GLED, OUTPUT);   //Set Green LED as Output
  pinMode (RLED, OUTPUT);   //Set Red LED as Output
  pinMode (BUTTON, INPUT);  //Set button as input (not required)
}

/*
* 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
}

/*
* LED Mode Selection
* Pass a number for the LED state and set it accordingly.
*/
void setMode(int mode)
```

```
{
  //RED
  if (mode == 1)
  {
    digitalWrite(RLED, HIGH);
    digitalWrite(GLED, LOW);
    digitalWrite(BLED, LOW);
  }
  //GREEN
  else if (mode == 2)
  {
    digitalWrite(RLED, LOW);
    digitalWrite(GLED, HIGH);
    digitalWrite(BLED, LOW);
  }
  //BLUE
  else if (mode == 3)
  {
    digitalWrite(RLED, LOW);
    digitalWrite(GLED, LOW);
    digitalWrite(BLED, HIGH);
  }
  //PURPLE (RED+BLUE)
  else if (mode == 4)
  {
    analogWrite(RLED, 127);
    analogWrite(GLED, 0);
    analogWrite(BLED, 127);
  }
  //TEAL (BLUE+GREEN)
  else if (mode == 5)
  {
    analogWrite(RLED, 0);
    analogWrite(GLED, 127);
    analogWrite(BLED, 127);
  }
  //ORANGE (GREEN+RED)
  else if (mode == 6)
  {
    analogWrite(RLED, 127);
    analogWrite(GLED, 127);
    analogWrite(BLED, 0);
  }
  //WHITE (GREEN+RED+BLUE)
  else if (mode == 7)
  {
    analogWrite(RLED, 85);
    analogWrite(GLED, 85);
    analogWrite(BLED, 85);
  }
```

```
  //OFF (mode = 0)
  else
  {
    digitalWrite(RLED, LOW);
    digitalWrite(GLED, LOW);
    digitalWrite(BLED, LOW);
  }
}

void loop()
{
  currentButton = debounce(lastButton);           //read debounced state
  if (lastButton == LOW && currentButton == HIGH) //if it was pressed...
  {
    ledMode++;                                    //increment the LED value
  }
  lastButton = currentButton;                     //reset button value
  //if you've cycled through the different options,
  //reset the counter to 0
  if (ledMode == 8) ledMode = 0;
  setMode(ledMode);                              //change the LED state
}
```

This might look like a lot of code, but it is nothing more than a conglomeration of code snippets that you have already written throughout this chapter.

How else could you modify this code? You could add additional buttons to independently control one of the three colors. You could also add blink modes, using code from Chapter 1 that blinked the LED. The possibilities are limitless.

Summary

In this chapter you learned about the following:

- How a breadboard works
- How to pick a resistor to current-limit an LED
- How to wire an external LED to your Arduino
- How to use PWM to write "analog" values to LEDs
- How to read a pushbutton
- How to debounce a pushbutton
- How to use `for` loops
- How to utilize pullup and pulldown resistors

CHAPTER

3

Reading Analog Sensors

Parts You'll Need for This Chapter

Arduino Uno

Small breadboard

Jumper wires

10kΩ potentiometer

10kΩ resistor (×2)

220Ω resistor (×3)

USB cable

Photoresistor

TMP36 temperature sensor (or any other 5V analog sensor)

5mm common-cathode RGB LED (All examples in this book use a common-cathode RGB LED. If you use a common-anode RGB LED, you'll need to invert the LED control logic, connect the anode to the 5V, and connect each of the cathode pins through resistors to I/O pins.)

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/ch3`.

In addition, all code can be found at www.wiley.com/go/exploringarduino on the Download Code tab. The code is in the chapter 03 download and individually named according to the names throughout the chapter.

The world around you is analog. Even though you might hear that the world is "going digital," the majority of observable features in your environment will always be analog in nature. The world can assume an infinite number of potential states, whether you are considering the color of sunlight, the temperature of the ocean, or the concentration of contaminants in the air. This chapter focuses on developing techniques for discretizing these infinite possibilities into palatable digital values that can be analyzed with a microcontroller system like the Arduino.

In this chapter, you will learn about the differences between analog and digital signals and how to convert between the two, as well as a handful of the analog sensors that you can interface with your Arduino. Using skills that you acquired in the preceding chapter, you will add light sensors for automatically adjusting your nightlight. You will also learn how to send analog data from your Arduino to your computer via a USB-to-serial connection, which opens up enormous possibilities for developing more complex systems that can transmit environmental data to your computer.

NOTE You can follow along with a video as I teach you about reading from analog inputs: www.jeremyblum.com/2011/01/24/arduino-tutorial-4-analog-inputs/. You can also find this video on the Wiley website shown at the beginning of this chapter.

If you want to learn more about the differences between analog and digital signals, check out this video that explains each in depth: www.jeremyblum.com/2010/06/20/lets-get-digital-or-analog/. You can also find this video on the Wiley website shown at the beginning of this chapter.

Understanding Analog and Digital Signals

If you want your devices to interface with the world, they will inevitably be interfacing with analog data. Consider the projects you completed in the preceding chapter. You used a switch to control an LED. A switch is a digital input—it has only two possible states: on or off, high or low, 1 or 0, and so on. Digital information (what your computer or the Arduino processes) is a series of binary (or digital) data. Each bit has only has one of two values.

The world around you, however, rarely expresses information in only two ways. Take a look out the window. What do you see? If it's daytime, you probably see sunlight, trees moving in the breeze, and maybe cars passing or people walking around. All these things that you perceive cannot readily be classified

as binary data. Sunlight is not on or off; its brightness varies over the course of a day. Similarly, wind does not just have two states; it gusts at different speeds all the time.

Comparing Analog and Digital Signals

The graphs in Figure 3-1 show how analog and digital signals compare to each other. On the left is a square wave that varies between only two values: 0 and 5 volts. Just like with the button that you used in the preceding chapter, this signal is only a "logic high" or "logic low" value. On the right is part of a cosine wave. Although its bounds are still 0 and 5 volts, the signal takes on an infinite number of values between those two voltages.

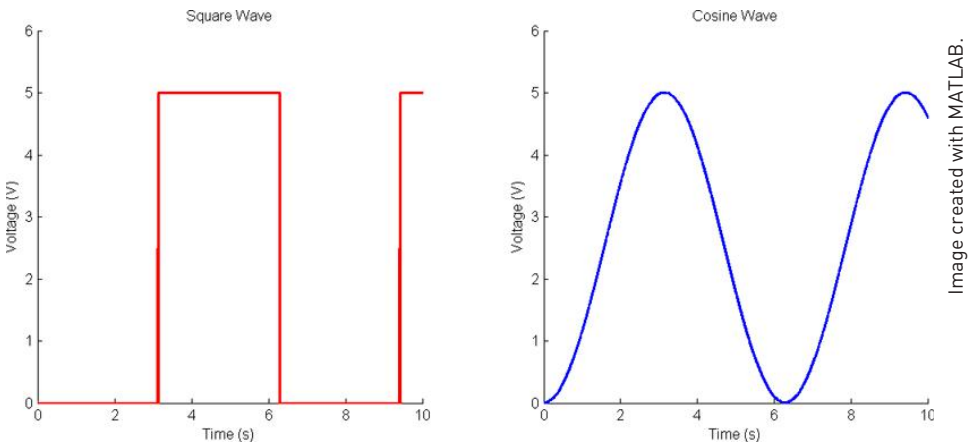


Figure 3-1: Analog and digital signals

Analog signals are those that cannot be discretely classified; they vary within a range, theoretically taking on an infinite number of possible values within that range. Think about sunlight as an example of an analog input you may want to measure. Naturally, there is a reasonable range over which you might measure sunlight. Often measured in lux, or luminous flux per unit area, you can reasonably expect to measure values between 0 lux (for pitch black) and 130,000 lux in direct sunlight. If your measuring device were infinitely accurate, you could measure an infinite number of values between those two. An indoor setting might be 400 lux. If it were slightly brighter, it could be 401 lux, then 401.1 lux, then 401.11 lux, and so on. A computer system could never feasibly measure an infinite number of decimal places for an analog value because memory and computer power must be finite values. If that's the case, how can you interface your Arduino with the "real world?" The answer is analog-to-digital converters (ADC), which can convert analog values into digital representations with a finite amount of precision and speed.

Converting an Analog Signal to a Digital One

Suppose that you want to measure the brightness of your room. Presumably, a good light sensor could produce a varying output voltage that changes with the brightness of the room. When it is pitch black, the device would output 0V, and when it's completely saturated by light, it would output 5V, with values in between corresponding to the varying amount of light. That's all well and good, but how do you go about reading those values with an Arduino to figure out how bright the room is? You can use the Arduino's analog-to-digital converter (ADC) pins to convert analog voltage values into number representations that you can work with.

The accuracy of an ADC is determined by the resolution. In the case of the Arduino Uno, there is a 10-bit ADC for doing your analog conversions. "10-bit" means that the ADC can subdivide (or quantize) an analog signal into 2^{10} different values. If you do the math, you'll find that $2^{10} = 1024$; hence, the Arduino can assign a value from 0 to 1023 for any analog value that you give it. Although it is possible to change the reference voltage, you'll be using the default 5V reference for the analog work that you do in this book. The reference voltage determines the max voltage that you are expecting, and, therefore, the value that will be mapped to 1023. So, with a 5V reference voltage, putting 0V on an ADC pin returns a value of 0, 2.5V returns a value of 512 (half of 1023), and 5V returns a value of 1023. To better understand what's happening here, consider what a 3-bit ADC would do, as shown in Figure 3-2.

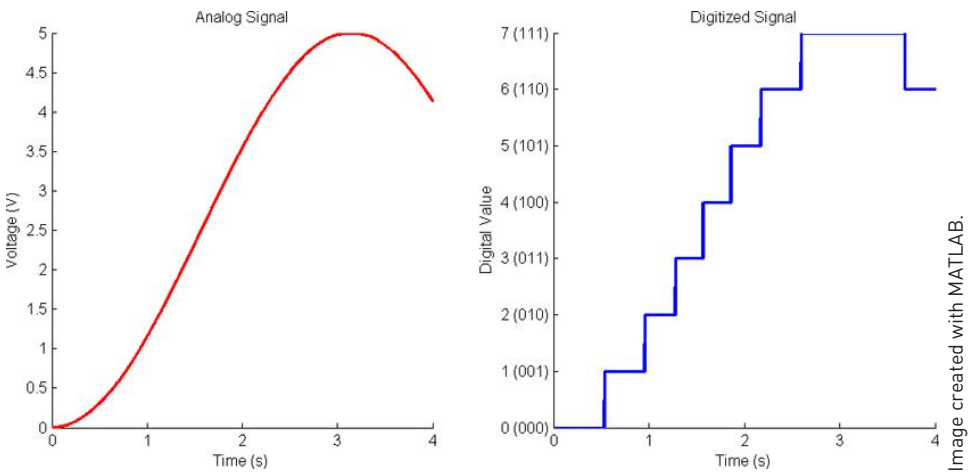


Figure 3-2: 3-bit analog quantization

NOTE If you want to learn more about using your own reference voltage or using a different internal voltage reference, check out the *analogReference()* page on the Arduino website: `www.arduino.cc/en/Reference/AnalogReference`.

A 3-bit ADC has 3 bits of resolution. Because $2^3=8$, there are 8 total logic levels, from 0 to 7. Therefore, any analog value that is passed to a 3-bit ADC will have to be assigned a value from 0 to 7. Looking at Figure 3-2, you can see that voltage levels are converted to discrete digital values that can be used by the microcontroller. The higher the resolution, the more steps that are available for representing each value. In the case of the Arduino Uno, there are 1024 steps rather than the 8 shown here.

Reading Analog Sensors with the Arduino: analogRead()

Now that you understand how to convert analog signals to digital values, you can integrate that knowledge into your programs and circuits. Different Arduinos have different numbers of analog input pins, but you read them all the same way, using the `analogRead()` command. First, you'll experiment with a potentiometer and a packaged analog sensor. Then, you'll learn how voltage dividers work, and how you can use them to make your own analog sensors from devices that vary their resistance in response to some kind of input.

Reading a Potentiometer

The easiest analog sensor to read is a simple potentiometer (a pot, for short). Odds are that you have tons of these around your home in your stereos, speakers, thermostats, cars, and elsewhere. Potentiometers are variable voltage dividers (discussed later in this chapter) that look like knobs. They come in lots of sizes and shapes, but they all have three pins. You connect one of the outer pins to ground, and the other to the 5V. Potentiometers are symmetrical, so it doesn't matter which side you connect the 5V and ground to. You connect the middle pin to analog input 0 on your Arduino. Figure 3-3 shows how to properly hook up your potentiometer to an Arduino.

As you turn the potentiometer, you vary the voltage that you are feeding into analog input 0 between 0V and 5V. If you want, you can confirm this with a multimeter in voltage measurement mode by hooking it up as shown Figure 3-4 and reading the display as you turn the knob. The red (positive) probe should be connected to the middle pin, and the black (negative) probe should be connected to whichever side is connected to ground. Note that your potentiometer and multimeter might look different than shown here.

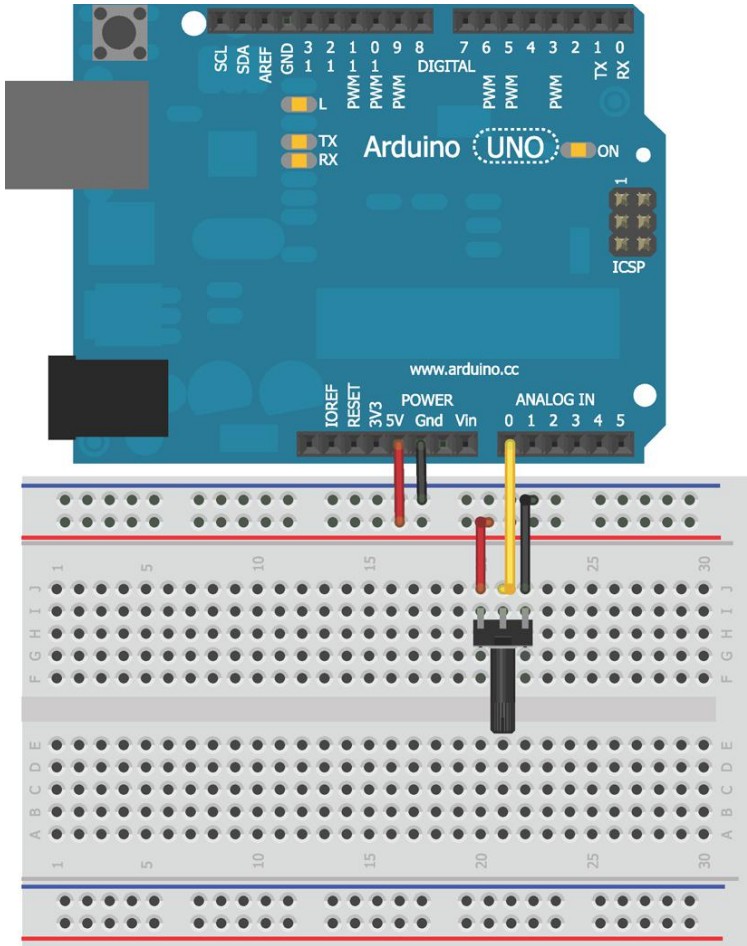


Image created with Fritzing.

Figure 3-3: Potentiometer circuit

Before you use the potentiometer to control another piece of hardware, use the Arduino's serial communication functionality to print out the potentiometer's ADC value on your computer as it changes. Use the `analogRead()` function to read the value of the analog pin connected to the Arduino and the `Serial.println()` function to print it to the Arduino IDE serial monitor. Start by writing and uploading the program in Listing 3-1 to your Arduino.

Figure 3-4: Multimeter measurement

Listing 3-1: Potentiometer Reading Sketch—pot.ino

```
//Potentiometer Reading Program

const int POT=0;  //Pot on analog pin 0
int val = 0;      //variable to hold the analog reading from the POT

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

void loop()
{
  val = analogRead(POT);
  Serial.println(val);
  delay(500);
}
```

You'll investigate the functionality of the serial interface more in later chapters. For now, just be aware that the serial interface to the computer must be started in the `setup`. `Serial.begin()` takes one argument that specifies the communication speed, or baud rate. The *baud rate* specifies the number of bits being transferred per second. Faster baud rates enable you to transmit more data in less time, but can also introduce transmission errors in some communication systems. 9600 baud is a common value, and it's what you use throughout this book.

In each iteration through the loop, the `val` variable is set to the present value that the ADC reports from analog pin 0. The `analogRead()` command requires the number of the ADC pin to be passed to it. In this case, it's 0 because that's what you hooked the potentiometer up to. You can also pass `A0`, though the `analogRead()` function knows you must be passing it an analog pin number, so you can pass `0` as shorthand. After the value has been read (a number between `0` and `1023`), `Serial.println()` prints that value over serial to the computer's serial terminal, followed by a "newline" that advances the cursor to the next line. The loop then delays for half a second (so that the numbers don't scroll by faster than you can read them), and the process repeats.

After loading this onto your Arduino, you'll notice that the TX LED on your Arduino is blinking every 500ms (at least it should be). This LED indicates that your Arduino is transmitting data via the USB connection to the serial terminal on your computer. You can use a variety of terminal programs to see what your Arduino is sending, but the Arduino IDE conveniently has one built right in! Click the circled button shown in Figure 3-5 to launch the serial monitor.

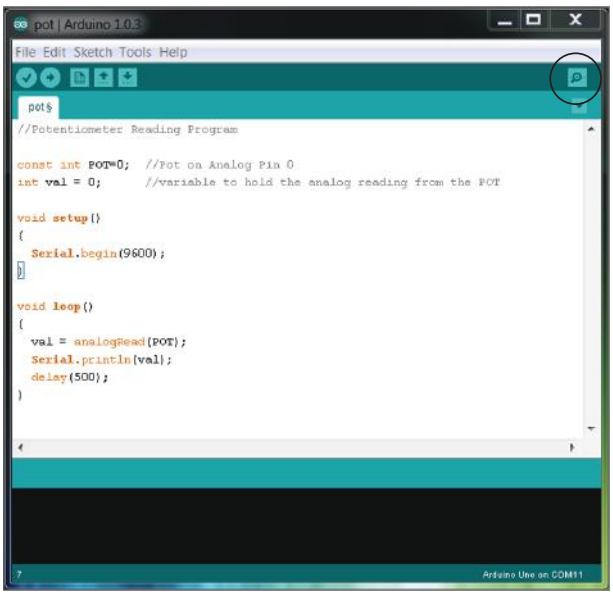


Figure 3-5: Serial monitor button

After launching the serial monitor, you should see a window with numbers streaming by. Turn the dial and you'll see the numbers go up and down to correspond with the position of the potentiometer. If you turn it all the way in one direction, the numbers should approach `0`, and if you turn it all the way in the other direction, the numbers should approach `1023`. It will look like the example shown in Figure 3-6.

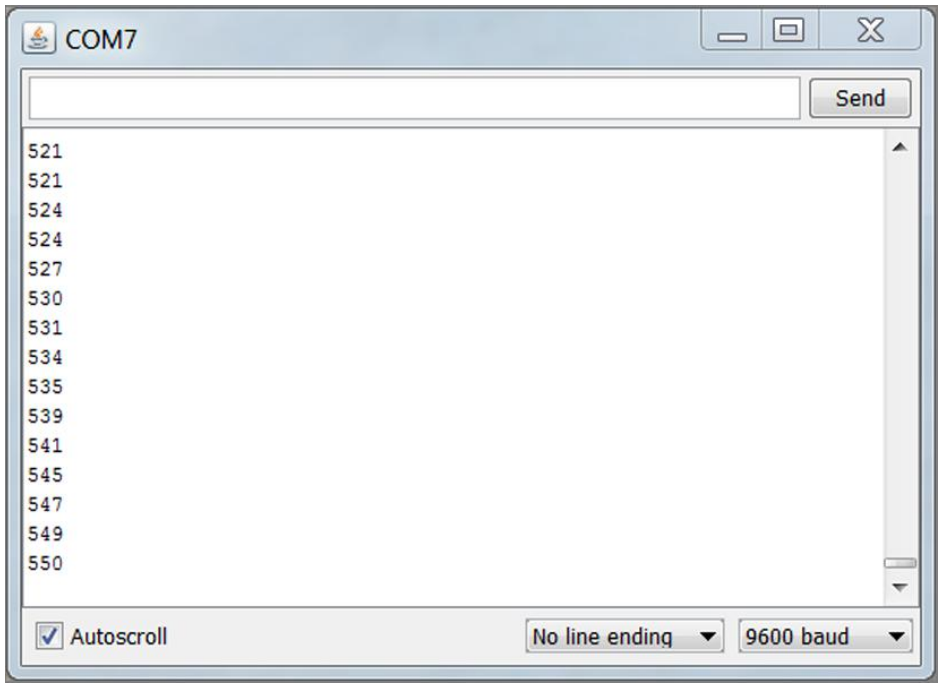


Figure 3-6: Incoming serial data

NOTE If you're getting funky characters, make sure that you have the baud rate set correctly. Because you set it to 9600 in the code, you need to set it to 9600 in this window as well.

You've now managed to successfully turn a dial and make some numbers change; pretty exciting, right? No? Well, this is the just the first step. Next, you learn about other types of analog sensors and how you can use the data from analog sensors to control other pieces of hardware. For now, you use the familiar LED, but in later chapters you use motors and other output devices to visualize your analog inputs.

Using Analog Sensors

Although potentiometers generate an analog voltage value on a pin, they aren't really sensors in the traditional sense. They "sense" your turning of the dial, but that gets boring pretty quickly. The good news is that all kinds of sensors generate analog output values corresponding to "real-world" action. Examples of such include the following:

- Accelerometers that detect tilting (many smartphones and tablets now have these)
- Magnetometers that detect magnetic fields (for making digital compasses)
- Infrared sensors that detect distance to an object
- Temperature sensors that can tell you about the operating environment of your project

Many of these sensors are designed to operate in a manner similar to the potentiometer you just experimented with: You provide them with a power (VCC) and ground (GND) connection, and they output an analog voltage between VCC and GND on the third pin that you hook up to your Arduino's ADC.

For this next experiment, you get to choose what kind of analog sensor you want to use. They all output a value between 0V and 5V when connected to an Arduino, so they will all work the same for your purposes. Here are some examples of sensors that you can use:

- **Sharp Infrared Proximity Sensor**

  `www.exploringarduino.com/parts/IR-Distance-Sensor`

  Connector: `www.exploringarduino.com/parts/JST-Wire`

  The Sharp infrared distance sensors are popular for measuring the distance between your project and other objects. As you move farther from the object you are aiming at, the voltage output decreases. Figure 5 in the datasheet from the part webpage linked above shows the relationship between voltage and measured distance.

- **TMP36 Temperature Sensor**

  `www.exploringarduino.com/parts/TMP36`

  The TMP36 temperature sensor easily correlates temperature readings in Celsius with voltage output levels. Since every 10mV corresponds to 1°C, you can easily create a linear correlation to convert from the voltage you measure back to the absolute temperature of the ambient environment: °C = [(Vout in mV) – 500]/10. The offset of –500 is for dealing with temperatures below 0°C. The graph in Figure 3-7 (extracted from the datasheet) shows this conversion.

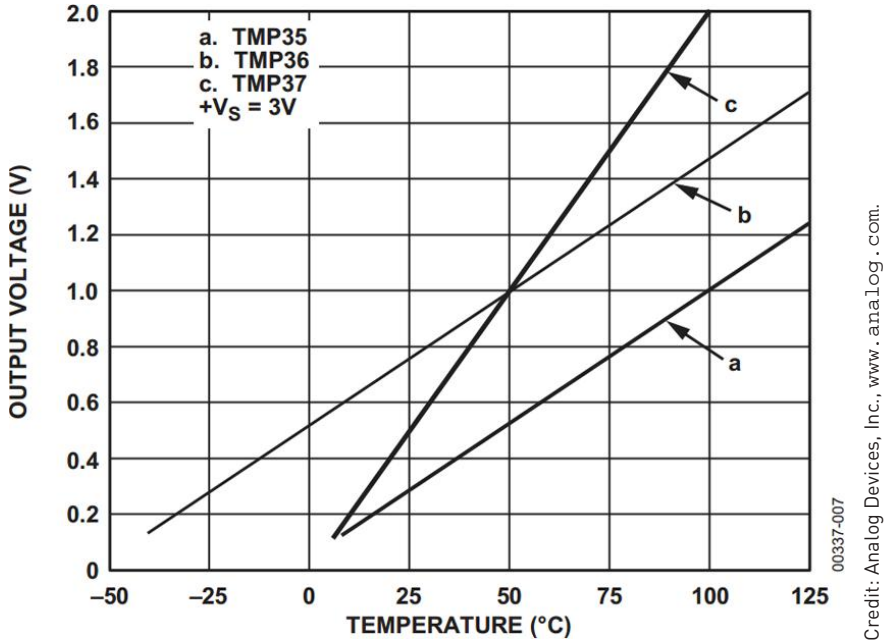


Figure 3-7: Voltage to Temperature Correlation

- **Triple Axis Analog Accelerometer**

  `www.exploringarduino.com/parts/TriAxis-Analog-Accelerometer`

  Triple axis accelerometers are great for detecting orientation. Analog accelerometers output an analog value corresponding to each axis of movement: X, Y, and Z (each on a different pin). Using some clever math (trigonometry and knowledge of gravity), you can use these voltage values to ascertain the position of your project in 3D space! Importantly, many of these sensors are 3.3V, so you will need to use the `analogReference()` command paired with the AREF pin to set a 3.3V voltage reference to enable you to get the full resolution out of the sensor.

- **Dual Axis Analog Gyroscope**

  `www.exploringarduino.com/parts/DualAxis-Analog-Gyroscope`

  Gyroscopes, unlike accelerometers, are not affected by gravity. Their analog output voltages fluctuate in accordance with angular acceleration around an axis. These prove particularly useful for detecting twisting motions. For an example of a gyroscope in action with an Arduino, check out my SudoGlove, a glove I designed that captures hand gestures to control hardware like music synthesizers and RC cars: `www.sudoglove.com`. Like accelerometers, be aware that many gyroscopes are 3.3V parts.

Now that you've chosen a sensor, it's time to put that sensor to use.

Working with Analog Sensors to Sense Temperature

This simple example uses the TMP36 temperature sensor mentioned in the previous section. However, feel free to use any analog sensor you can get your hands on. Experiment with one of the examples listed earlier, or find your own. (It should be 5V compliant if you are using the Arduino Uno.) The following steps are basically the same for any analog sensor you might want to use.

To begin, wire up your RGB LED as you did in the preceding chapter, and wire the temperature sensor up to analog input 0 as shown in the Figure 3-8.

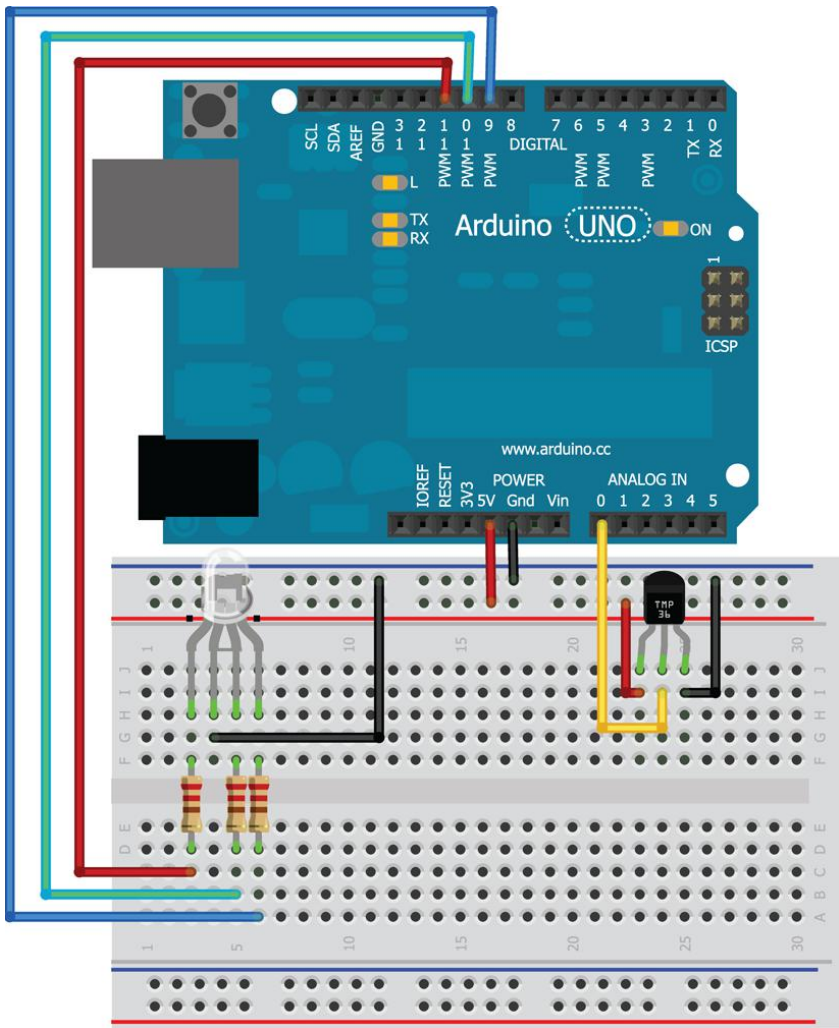


Image created with Fritzing.

Figure 3-8: Temperature sensor circuit

Using this circuit, you'll make a simple temperature alert system. The light will glow green when the temperature is within an acceptable range, will turn red when it gets too hot, and will turn blue when it gets too cold.

First things first, you need to ascertain what values you want to use as your cutoffs. Using the exact same sketch from Listing 3-1, use the serial monitor to figure out what analog values correspond to the temperature cutoffs you care about. My room is about 20°C, which corresponds to an analog reading of about `143`. These numbers might differ for you, so launch the sketch from before, open the serial terminal, and take a look at the readings you are getting. You can confirm the values mathematically using the graph from Figure 3-7. In my case, a value of 143/1023 corresponds to a voltage input of about 700mV. Deriving from the datasheet, the following equation can be used to convert between the temperature (°C) and the voltage (mV):

$$\text{Temperature(°C)} \times 10 = \text{voltage (mV)} - 500$$

Plugging in the value of 700mV, you can confirm that it equates to a temperature of 20°C. Using this same logic (or by simply observing the serial window and picking a value), you can determine that 22°C is a digital value of `147` and 18°C is a digital value of `139`. Those values will serve as the cutoffs that will change the color of the LED to indicate that it is too hot or too cold. Using the `if` statements, the `digitalWrite` function, and the `analogRead` function that you have now learned about, you can easily read the temperature, determine what range it falls in, and set the LED accordingly.

NOTE Before you copy the code in Listing 3-2, try to write this yourself and see whether you can make it work. After giving it a shot, compare it with the code here. How did you do?

Listing 3-2: Temperature Alert Sketch—tempalert.ino

```
//Temperature Alert!
const int BLED=9;          //Blue LED on pin 9
const int GLED=10;         //Green LED on pin 10
const int RLED=11;         //Red LED on pin 11
const int TEMP=0;          //Temp Sensor is on pin A0

const int LOWER_BOUND=139; //Lower Threshold
const int UPPER_BOUND=147; //Upper Threshold

int val = 0;               //Variable to hold analog reading


void setup()
{
  pinMode (BLED, OUTPUT); //Set Blue LED as Output
  pinMode (GLED, OUTPUT); //Set Green LED as Output
```

```
  pinMode (RLED, OUTPUT); //Set Red LED as Output
}

void loop()
{
  val = analogRead(TEMP);

  if (val < LOWER_BOUND)
  {
    digitalWrite(RLED, LOW);
    digitalWrite(GLED, LOW);
    digitalWrite(BLED, HIGH);
  }
  else if (val > UPPER_BOUND)
  {
    digitalWrite(RLED, HIGH);
    digitalWrite(GLED, LOW);
    digitalWrite(BLED, LOW);
  }
  else
  {
    digitalWrite(RLED, LOW);
    digitalWrite(GLED, HIGH);
    digitalWrite(BLED, LOW);
  }
}
```

This code listing doesn't introduce any new concepts; rather, it combines what you have learned so far to make a system that uses both inputs and outputs to interact with the environment. To try it out, squeeze the temperature sensor with your fingers or exhale on it to heat it up. Blow on it to cool it down.

Using Variable Resistors to Make Your Own Analog Sensors

Thanks to physics, tons of devices change resistance as a result of physical action. For example, some conductive inks change resistance when squished or flexed (force sensors and flex sensors), some semiconductors change resistance when struck by light (photoresistors), and some polymers change resistance when heated or cooled (thermistors). These are just a few examples of components that you can take advantage of to build your own analog sensors. Because these sensors are changing resistance and not voltage, you need to create a voltage divider circuit so that you can measure their resistance change.

Using Resistive Voltage Dividers

A resistive voltage divider uses two resistors to output a voltage that is some fraction of the input voltage. The output voltage is a function directly related to the value of the two resistors. So, if one of the resistors is a variable resistor, you can monitor the change in voltage from the voltage divider that results from the varying resistance. The size of the other resistor can be used to set the sensitivity of the circuit, or you can use a potentiometer to make the sensitivity adjustable.

First, consider a fixed voltage divider and the equations associated with it, as shown in Figure 3-9. A0 in the Figure 3-9 refers to analog pin 0 on the Arduino.

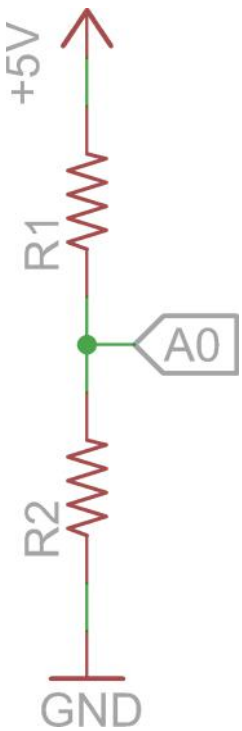


Figure 3-9: Simple voltage divider circuit

The equation for a voltage divider is as follows:

$$Vout = Vin(R2/(R1 + R2))$$

In this case, the voltage input is 5V, and the voltage output is what you'll be feeding into one of the analog pins of the Arduino. In the case where R1 and R2 are matched (both 10kΩ for example), the 5V is divided by 2 to make 2.5V at the analog input. Confirm this by plugging values into the equation:

$$Vout = 5V(10k/(10k + 10k)) = 5V \times .5 = 2.5V$$

Now, suppose one of those resistors is replaced with a variable resistor, such as a photoresistor. Photoresistors (see Figure 3-10) change resistance depending on the amount of light that hits them. In this case, I'll opt to use a 200kΩ photoresistor. When in complete darkness, its resistance is about 200kΩ; when saturated with light, the resistance drops nearly to zero. Whether you choose to replace R1 or R2 and what value you choose to make the fixed resistor will affect the scale and precision of the readings you receive. Try experimenting with different configurations and using the serial monitor to see how your values change. As an example, I will choose to replace R1 with the photoresistor, and I'll make R2 a 10kΩ resistor (see Figure 3-11). You can leave the RGB LED in place for now, though you'll only use one of the colors for this exercise.

Credit: element14, www.element14.com

Figure 3-10: Photoresistor

Load up your trusty serial printing sketch again (Listing 3-1) and try changing the lighting conditions over the photoresistor. Hold it up to a light and cup it with your hands. Odds are, you aren't going to be hitting the full range from 0 to 1023 because the variable resistor will never have a resistance of zero. Rather, you can probably figure out the maximum and minimum values that you are likely to receive. You can use the data from your photoresistor to make a more intelligent nightlight. The nightlight should get brighter as the room gets darker, and vice versa. Using your serial monitor sketch, pick the values that represent when your room is at full brightness or complete darkness. In my case, I found that a dark room has a value of around `200` and a completely bright room has a value around `900`. These values will vary for you based upon your lighting conditions, the resistor value you are using, and the value of your photoresistor.

Using Analog Inputs to Control Analog Outputs

Recall that you can use the `analogWrite()` command to set the brightness of an LED. However, it is an 8-bit value; that is, it accepts values between 0 and 255 only, whereas the ADC is returning values as high as 1023. Conveniently, the Arduino programming language has two functions that are useful for mapping between two sets of values: the `map()` and `constrain()` functions. The `map()` function looks like this:

```
output = map(value, fromLow, fromHigh, toLow, toHigh)
```

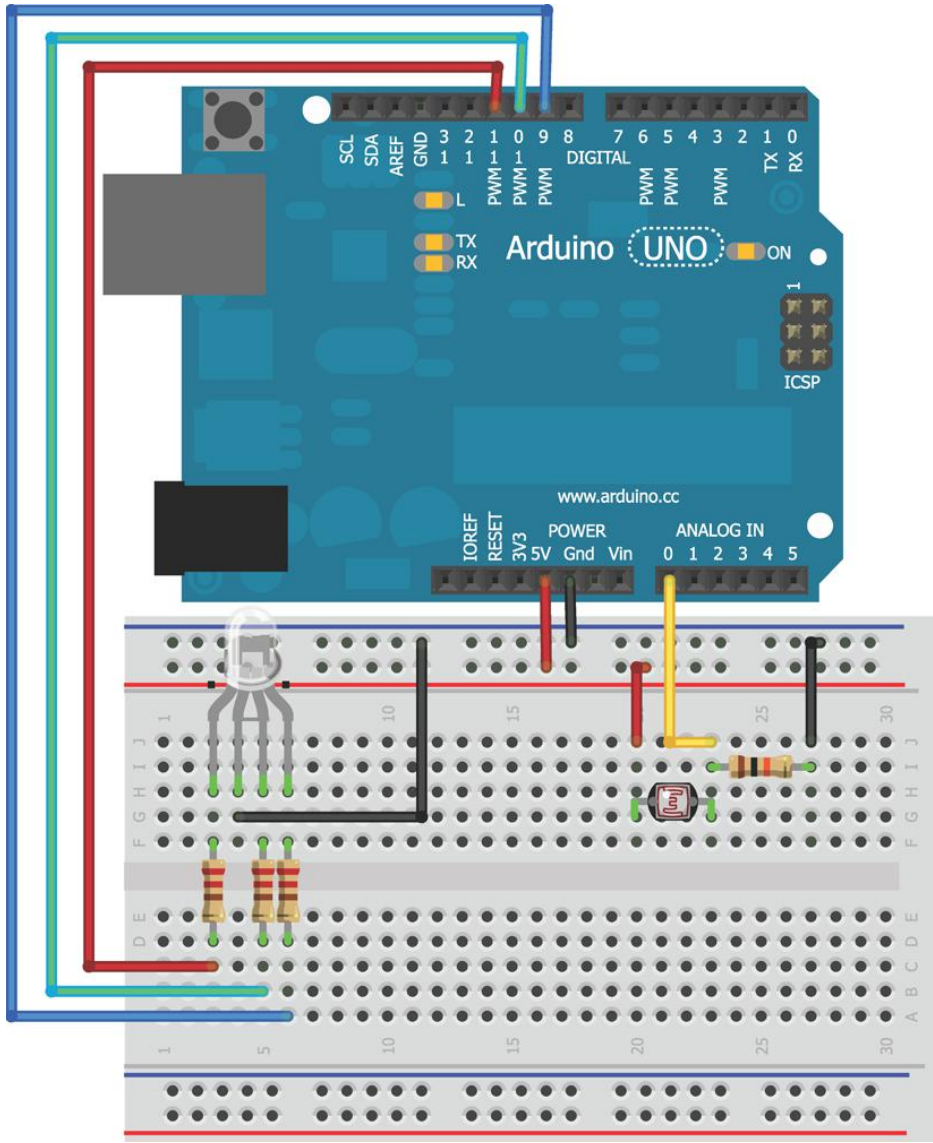


Figure 3-11: Photoresistor circuit

`value` is the information you are starting with. In your case, that's the most recent reading from the analog input. `fromLow` and `fromHigh` are the input boundaries. These are values you found to correspond to the minimum and maximum brightness in your room. In my case, they were `200` and `900`. `toLow` and `toHigh` are the values you want to map them to. Because `analogWrite()` expects value between `0` and `255`, you use those values. However, we want a darker room to map to a brighter LED. Therefore, when the input from the ADC is a low value, you want the output to the LED to be a high value, and vice versa.

Conveniently, the `map` function can handle this automatically; simply swap the high and low values so that the low value is `255` and the high value is 0. The `map()` function creates a linear mapping. For example, if your `fromLow` and `fromHigh` values are `200` and `900`, respectively, and your `toLow` and `toHigh` values are `255` and `0`, respectively, `550` maps to `127` because `550` is halfway between `200` and `900` and `127` is halfway between `255` and `0`. Importantly, however, the `map()` function does not constrain these values. So, if the photoresistor does measure a value below `200`, it is mapped to a value above `255` (because you are inverting the mapping). Obviously, you don't want that because you can't pass a value greater than `255` to the `analogWrite()` function. You can deal with this by using the `constrain()` function. The `constrain()` function looks like this:

```
output = constrain(value, min, max)
```

If you pass the output from the `map` function into the `constrain` function, you can set the `min` to `0` and the `max` to `255`, ensuring that any numbers above or below those values are constrained to either `0` or `255`. Finally, you can then use those values to command your LED! Now, take a look at what that final sketch will look like (see Listing 3-3).

Listing 3-3: Automatic Nightlight Sketch—nightlight.ino

```
//Automatic Nightlight

const int RLED=9;         //Red LED on pin 9 (PWM)
const int LIGHT=0;        //Lght Sensor on analog pin 0
const int MIN_LIGHT=200; //Minimum expected light value
const int MAX_LIGHT=900; //Maximum Expected Light value
int val = 0;              //variable to hold the analog reading

void setup()
{
  pinMode(RLED, OUTPUT); //Set LED pin as output
}

void loop()
{
  val = analogRead(LIGHT);                      //Read the light sensor
  val = map(val, MIN_LIGHT, MAX_LIGHT, 255, 0); //Map the light reading
  val = constrain(val, 0, 255);                 //Constrain light value
  analogWrite(RLED, val);                       //Control the LED
}
```

Note that this code reuses the `val` variable. You can alternatively use a different variable for each function call. In functions such as `map()` where `val` is both the input and the output, the previous value of `val` is used as the input, and its value is reset to the updated value when the function has completed.

Play around with your nightlight. Does it work as expected? Remember, you can adjust the sensitivity by changing the minimum and maximum bounds of the mapping function or changing the fixed resistor value. Use the serial monitor to observe the differences with different settings until you find one that works the best. Can you combine this sketch with the color-selection nightlight that you designed in the preceding chapter? Try adding a button to switch between colors, and use the photoresistor to adjust the brightness of each color.

Summary

In this chapter you learned about the following:

- The differences between analog and digital signals
- How to convert analog signals to digital signals
- How to read an analog signal from a potentiometer
- How to display data using the serial monitor
- How to interface with packaged analog sensors
- How to create your own analog sensors
- How to map and constrain analog readings to drive analog outputs