9

MOTOR CONTROL WITH THE ARDUINO MEGA

The Arduino family of circuit boards is one of the core technologies that have made the Maker Movement possible. If you're new to gadget development and you want to get your feet wet, Arduino boards are ideal because they're so easy to use and easy to program. If you're an entrepreneur who wants to manufacture and sell a gadget, Arduino boards are ideal because they provide high reliability at low cost.

Many hundreds of thousands of Arduino boards have been sold, and across the world, electronic hobbyists have incorporated them into projects. Some projects are simple hobbyist gadgets, such as remote-controlled musical instruments, but many others have become viable products, including vehicles, household robots, and health-monitoring systems.

The goal of this chapter is to explain how Arduino technology can be used to control electric motors. There are three main parts:

- **The Arduino Mega**—Understanding the hardware and developing software
- **The Arduino Motor Shield**—Understanding motor-control devices
- **Motor control**—Developing software to control a brushed motor, stepper motor, and servomotor

This chapter covers a great deal of ground, and as much as I'd like to, I can't explore any specific subject in detail. Thankfully, there are countless Arduino resources on the Internet. In particular, I recommend the

documentation page at http://arduino.cc/en/Reference/HomePage and the Arduino forum at http://forum.arduino.cc.

9.1 The Arduino Mega

The Arduino Mega isn't the most powerful or the most recent Arduino board, but it's one of the most popular. It's also fully compatible with existing Arduino hardware and software. Table 9.1 presents basic information about the board.

Table 9.1 Specifications of the Arduino Mega

Parameter Name	Parameter Value
Dimensions	4 × 2.1 inches
Operating voltage	5 V
Recommended input voltage	7–12 V
Clock speed	16 MHz
Digital I/O pins	54
Analog input pins	16

With its limited resources, the Mega isn't suitable for computing tasks such as text editing or web surfing. However, when combined with the Arduino Motor Shield, it's capable of controlling brushed motors, stepper motors, and servomotors. This section discusses the Arduino Mega's circuit board and its brain, the ATmega2560 microcontroller. The motor shield is introduced in a later section.

9.1.1 The Arduino Mega Circuit Board

The design of the Mega makes evident Arduino's focus on simplicity. The power pins are grouped together and labeled POWER. The communication pins are grouped together under the heading COMMUNICATION. Figure 9.1 shows what this looks like.

Most of the board's perimeter is occupied with raised black components that receive single-wire connections. These are called *headers*, and the Mega's header connections are divided into five groups:

- **Power**—Receive or deliver external power
- **Analog input**—Receive analog data to be converted into digital signals for processing
- **Digital**—Receive or transmit digital signals
- **Communication**—Communicate signals through three serial ports
- **PWM**—Transmit control signals using pulse width modulation

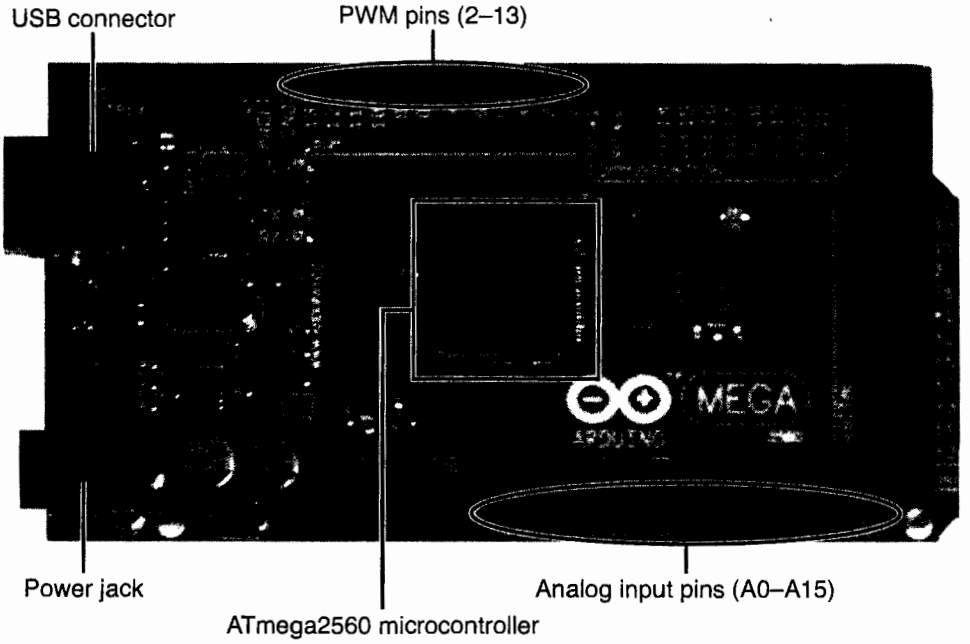

Figure 9.1
The Arduino Mega

For the purposes of this chapter, the most important header signals are in the PWM group. We'll use these to generate control signals for DC motors.

The left side of the board has a power jack, and the recommended input voltage is between 7 and 12 volts. However, I prefer to deliver power to the Mega through the USB connector. If the Mega is connected to a PC by USB, it will draw the current it needs to operate.

In addition to delivering power, the USB connection also makes it possible to transfer programs to the Mega. However, before we get into Arduino programming, I want to introduce the device that executes the programs: Atmel's ATmega2560 microcontroller.

9.1.2 Microcontrollers and the ATmega2560

As shown in Figure 9.1, the center of the circuit board is occupied by a 100-pin device called the ATmega2560. This device is a *microcontroller*, and in essence the entire purpose of an Arduino board is to provide access to this device. This discussion explains what microcontrollers are and presents the specific characteristics of the ATmega2560.

Microcontrollers

In my experience, the best way to introduce microcontrollers is to compare them with personal computers. A PC serves a wide variety of purposes involving data processing and digital communication. To serve these purposes, the PC requires multiple devices—the CPU to process data, RAM chips to store temporary data, and the hard disk to store programs, files, and the operating system.

A microcontroller (abbreviated MCU) serves similar purposes, but all of its resources are integrated onto a single chip. This self-containment provides a number of benefits, including low cost, low

power operation, and ease of circuit design. The drawback is that the MCU's on-chip resources aren't nearly as impressive as those you'd find in a PC.

An example will make this clear. My laptop has 8 gigabytes of RAM and its processor runs at 3 gigahertz. In contrast, the Mega's microcontroller has 8 kilobytes of RAM and processes data at 16 megahertz. This means my laptop runs nearly 200 times faster than the Mega and can store one million times as much data.

MCUs may not be suitable for personal computing, but they're ideal for maker projects. If you're building a simple robot or automated sensor system, the microcontroller's lack of resources won't be an issue. Instead, you'll appreciate the low cost, and because MCUs are so self-contained, you'll find it easy to design the circuit.

In general, gadgets with microcontrollers operate in three steps:

1. Read data from a sensor, such as a temperature sensor or a pressure sensor.
2. Process the data to judge the state of the system.
3. Use the state information to control a mechanism, such as a motor.

In Step 1, sensor data can take an infinite number of values, and for this reason, the data is referred to as *analog*. Before the data can be processed, it must be converted into the ones and zeros required by processors. This is *digital* data. To make this possible, most modern MCUs have multiple analog-to-digital converters, or ADCs.

In Step 3, microcontrollers control mechanisms using pulse width modulation (PWM), which has been discussed at length throughout this book. Later in this chapter, I'll show how the Mega can use PWM to control brushed motors and servomotors.

The ATmega2560

Most Arduino boards contain Atmel MCUs, and the Mega is no exception. The Mega relies on Atmel's ATmega2560 microcontroller to process its data, and Table 9.2 lists a number of its important characteristics.

Table 9.2 Characteristics of Atmel's ATmega2560 Microcontroller

Parameter Name	Parameter Value
Clock speed	16 MHz
Flash memory	256 KB
SRAM	8 KB
EEPROM	4 KB
Number of pins	100
Analog conversion resolution	10-bit
PWM resolution	8-bit
Temperature range	–40° to 85° C

In looking at these values, it's important to understand the difference between the three different types of memory:

- Flash memory holds programs, which means the largest program that can run on the Mega is 256 KB.
- SRAM (static random access memory) stores temporary data used by the program.
- EEPROM (electrically erasable programmable read-only memory) stores settings and other parameters.

The SRAM memory is cleared whenever power is removed. In contrast, the Flash memory and EEPROM maintain their contents without power.

The ATmega2560 MCU has 100 pins: 11 power pins and 89 pins for input/output. Most of the I/O pins can serve multiple roles, and configuring the pins' roles is a major concern in MCU development.

Thankfully, the Arduino framework makes pin configuration easy: 54 of the ATmega2560's I/O pins are accessible through the Mega's headers, and the process of configuring their operation is almost trivially simple.

The ATmega2560 has many incredible characteristics that have endeared it to makers across the world, but to really appreciate this device, you need to get your hands dirty and start writing programs. The next section explains how this is done.

9.2 Programming the Arduino Mega

In general, microcontroller programming isn't pleasant. You need to be aware of memory maps, peripheral buses, interrupt vectors, and countless data/control registers. Also, when you change to a new MCU, you practically have to rewrite your code from scratch.

The great innovation of the Arduino framework is that it dramatically simplifies writing code for MCUs. If you're familiar with the C programming language, you can come up to speed with Arduino in minutes—and as new Arduino boards come out, you can compile and run your programs without modification.

This section explains how to write, compile, and execute Arduino programs, commonly called *sketches*. However, before you can start programming, you need to get the Arduino environment up and running.

note

This chapter assumes a basic familiarity with C programming. If you don't have this background, my favorite introductory book is *C for Dummies* by Dan Gookin.

9.2.1 Preparing the Arduino Environment

The Arduino environment is a software package that consists of three components:

- USB drivers needed to communicate with Arduino boards
- A compiler that converts sketch code into executables for microcontrollers

- An integrated development environment (IDE) for writing, editing, compiling, and uploading sketches

The Arduino environment can be downloaded from http://arduino.cc/en/Main/Software. Two releases are available, and it's important to understand the difference between them:

- **Arduino 1.0.*x***—This environment is stable and supports boards with 8-bit processors such as the Arduino Mega.
- **Arduino 1.5.*x***—This environment supports boards with 32-bit microcontrollers, such as the Arduino Yún and the Arduino Due. At the time of this writing, this is a *beta release*, and to quote the site: "You may encounter bugs or unexpected behaviours."

> **note**
> The newer environment software (version 1.5.*x*) has been in beta for a long time. Many people are happy with it, but I've found it to be unreliable. This is why this chapter relies on the Arduino Mega for motor control instead of a newer board.

To program the Arduino Mega, you'll need the first option. To download it, open http://arduino.cc/en/Main/Software in a browser, scroll until you see the Arduino 1.0.*x* heading, and click the link corresponding to your operating system.

After you've downloaded the Arduino file, you're ready to install the application. The installation process depends on your operating system:

- For Windows, the instructions are at http://arduino.cc/en/Guide/Windows.
- For Mac OS X, the instructions are at http://arduino.cc/en/Guide/MacOSX.
- For Linux, the operating instructions depend on the distribution. The list of supported distributions and their instructions can be found at http://playground.arduino.cc/Learning/Linux.

If you've completed the directions, you should have an executable that brings up the Arduino IDE. Figure 9.2 shows what this looks like on my Windows 7 computer.

When the environment starts for the first time, it has no idea what Arduino board you're using or how to access it. Therefore, before you start coding, you need to tell the environment about your board. For boards connected by USB, I recommend five steps:

1. Connect the Arduino Mega to a USB cable and connect the other end of the cable to a PC.
2. Launch the Arduino environment.
3. In the main menu, go to Tools > Board and select the option Arduino Mega 2560 or Mega ADK.
4. In the main menu, go to Tools > Serial Port and select the serial port to which the Arduino Mega is connected. The process of determining the connected serial port depends on the operating system.
5. In the main menu, go to File > Save As and enter **blink** for the name of the sketch. Click Save.

After these steps are completed, the environment should look similar to that shown in Figure 9.3.

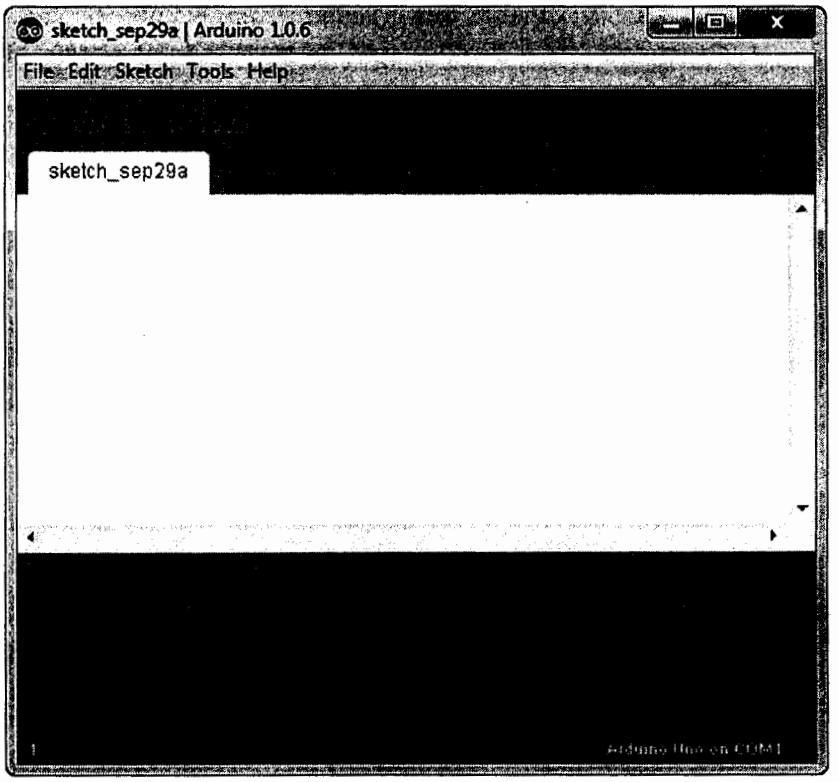

Figure 9.2
The Arduino development environment

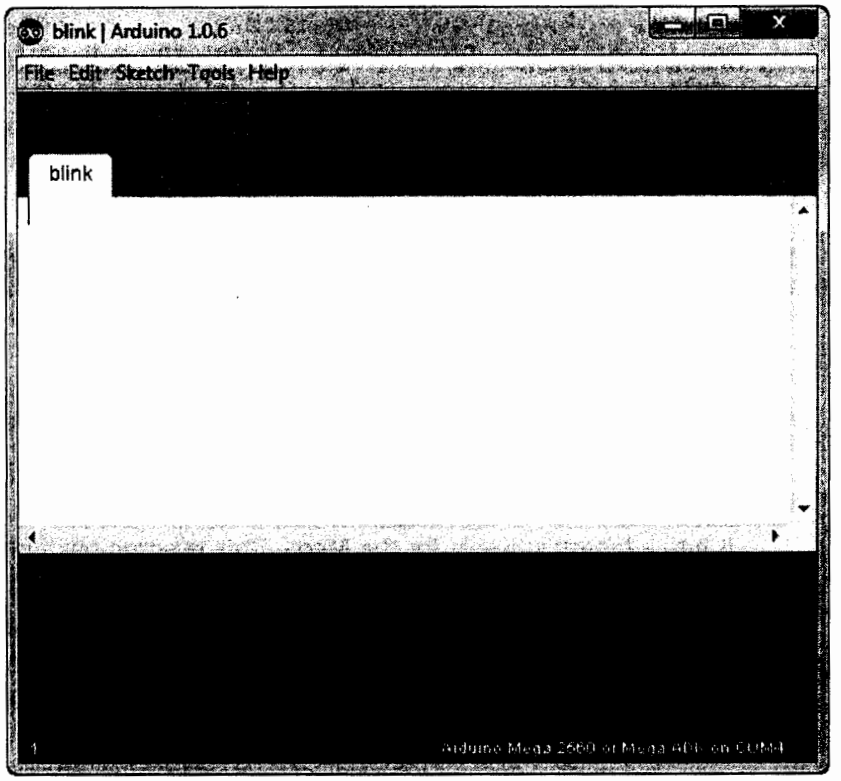

Figure 9.3
The fully configured environment

The final step saves an empty sketch to a file called blink.ino (*.ino is the extension for Arduino sketches). By default, this is saved in the Arduino\blink directory, which is located in the user's documents folder. For example, on my Windows system, blink.ino is saved to the C:\Users\Matt\My Documents\Arduino\blink directory.

9.2.2 Using the Environment

Once you've configured the environment for your board, the hard part is over. Now all you need to do is edit code and use the buttons above the editor.

To explain how this is done, I'm going to walk through the process of compiling and uploading a simple sketch. Listing 9.1 presents the code for blink.ino.

Listing 9.1 Ch9/blink.ino—Causing an LED to Blink

```
/* This sketch sets the voltage of Pin 13 high and low.
This causes the LED connected to the pin to blink. */

// Assign a name to Pin 13
int led_pin = 13;

// At startup, configure Pin 13 to serve as output
void setup() {
  pinMode(led_pin, OUTPUT);
}

// Repeatedly change the voltage of Pin 13
void loop() {
  digitalWrite(led_pin, HIGH);  // set the pin voltage high
  delay(1000);                  // delay one second
  digitalWrite(led_pin, LOW);   // set the pin voltage low
  delay(1000);                  // delay one second
}
```

If you'd rather not code this by hand, you can load my blink.ino sketch. Every code example in this book is contained in an archive called mfm.zip, which can be freely downloaded from http://www.motorsformakers.com.

After downloading the archive and extracting its contents, you'll find blink.ino in the Ch9 directory. You can load this into the environment by going to the main menu and selecting File > Open.

After the code has been entered, the next step is to compile it into a binary suitable for the Mega. This is accomplished by clicking the leftmost button above the editor, which features a check mark. If the code has errors, an error message in orange text will identify the first error and the line of code that produced it. If there are no errors, a "Done compiling" message will be displayed.

note

If you enter the code by hand, you'll see the number in the lower-left corner of the editor change as you type. This identifies the line number the cursor is on, and it took me far too long to figure that out.

If clicked, the button with a right-pointing arrow will recompile the sketch and upload it to the board. If this succeeds, a "Done uploading" message will be displayed and the program will start executing on the board. On the Arduino Mega, the LED next to Pin 13 will start blinking—one second on, one second off.

9.2.3 Arduino Programming

An Arduino program consists of statements that have the same structure and syntax as statements in the C programming language. That is, each statement ends with a semicolon and statements can be grouped into named blocks called *functions*. Arduino supports many of the basic C data types, and for many Mega programs the only data type you'll need is the `int` type.

Unlike C, sketches don't have a top-level `main` function. Instead, every sketch can be divided into three parts:

- **Global variables**—This part declares and initializes variables that can be used throughout the sketch.
- **`setup()`**—Contains statements to be executed when the board starts up or resets.
- **`loop()`**—Contains statements to be repeated after the `setup` function finishes.

A simple example will clarify how Arduino programs work. The code in Listing 9.1 repeatedly sets the voltage for Pin 13 high and low, delaying one second with each change. This causes the LED connected to Pin 13 to blink.

To understand the blink sketch, you need to be familiar with the functions provided by the Arduino framework. This discussion doesn't cover all of them, but focuses on functions in four categories: digital I/O, timing, analog read, and analog write. Table 9.3 lists each of them with a description.

note

`HIGH` and `LOW` are regular `int` values. `HIGH` equals 1 and `LOW` equals 0.

Table 9.3 Important Sketch Functions

Category	Function	Description
Digital I/O	`pinMode(int pin_num, int mode_type)`	Configures whether a pin's mode is `INPUT`, `OUTPUT`, or `INPUT_PULLUP`
	`digitalRead(int pin_num)`	Returns `HIGH` or `LOW`, depending on the input voltage
	`digitalWrite (int pin_num, int level)`	Sets the output pin's voltage to `HIGH` or `LOW`
Timing	`delay(int time)`	Waits a number of milliseconds before completing
	`delayMicroseconds (int time)`	Waits a number of microseconds before completing

Category	Function	Description
	millis()	Returns the number of milliseconds since the program started
	micros()	Returns the number of microseconds since the program started
Analog read	analogReference (int ref_type)	Sets the maximum voltage for analog input
	analogRead()	Returns the input analog voltage
Analog write	analogWrite (int pin_num, int duty_cycle)	Delivers PWM pulses with the desired duty cycle

This table lists less than half of the functions available for Arduino programming. For the full list, visit the reference site at http://arduino.cc/en/Reference/HomePage.

Digital I/O

As discussed earlier, the Mega's pins can be divided into five groups: power, analog input, communication, digital, and PWM. With the exception of the power pins, every pin on the Mega can be configured to serve one of three roles:

- `INPUT`—An input pin's digital voltage level (`HIGH` or `LOW`) can be read with `digitalRead`.
- `INPUT_PULLUP`—An input pin whose default state is `HIGH`.
- `OUTPUT`—An output pin's voltage can be set with `digitalWrite`.

A pin's mode determines whether it's an input pin or an output pin. To configure this mode, the `pinMode` function requires two arguments: the pin's number and the desired mode. By default, every pin's mode is set to `INPUT`. The following code sets Pin 10 to behave as an `OUTPUT` pin:

```
pinMode(10, OUTPUT);
```

If `pinMode` sets a pin in `INPUT` or `INPUT_PULLUP` mode, `digitalRead` returns an `int` corresponding to its voltage level. For `INPUT` mode, `digitalRead` returns `HIGH` if connected to a voltage greater than 3 V and `LOW` if connected to a voltage less than 2 V. For `INPUT_PULLUP` mode, `digitalRead` returns `HIGH` by default, and returns `LOW` when the pin is connected to ground.

If `pinMode` sets a pin's mode to `OUTPUT`, then `digitalWrite` can be called to set its voltage level. This function accepts two arguments: the pin number and the voltage level. If the second argument is `HIGH`, then `digitalWrite` sets the pin's voltage to 5 V. If the second argument is low, the pin's voltage is set to 0 V.

As a brief example, the following code reads the voltage level of Pin 7 and writes the voltage level to Pin 9:

```
res = digitalRead(7);
digitalWrite(8, res);
```

If this code is placed inside the `loop` function, it will be executed repeatedly. If it's placed inside the `setup` function, it will be run once per execution of the program.

Timing

The Arduino timing functions are easy to understand and use. There are four in total: Two specify timing delays and two identify how long the program has been running.

When a program updates a pin's state, such as when `digitalWrite` changes a pin's voltage, you may want to maintain this state for a duration of time. This is made possible by the `delay` and `delayMicroseconds` functions.

For example, the blink application sets Pin 13's voltage from `HIGH` to `LOW` and back again. With each change, the `delay` function maintains the state for 1 second. This is accomplished with the following code:

```
delay(1000);
```

Once `delay` starts, further statements won't execute until it finishes. Its argument identifies the wait time in milliseconds. If the argument is 250, `delay` will prevent statements from executing for a quarter of a second.

In many applications, a millisecond can be too long. If this is the case, you can call `delayMicroseconds`. This is similar to `delay`, but the argument identifies the wait time in microseconds. A microsecond is one-thousandth of a millisecond, so `delayMicroseconds(500)` waits for 500 microseconds, which equals one-half of a millisecond, which equals 0.0005 seconds.

In a sketch, the `loop` function executes until power is removed. This means you can't halt the `loop` function or break out of it. However, if you want to execute code for a specific duration of time, you can use the `millis` and `micros` functions. These tell you how long the program has been running, and their return values are given in milliseconds and microseconds, respectively.

An example will show how `millis` is used in practice. The following code sets Pin 13 to `HIGH` for the first 5 seconds, `LOW` for the next 5 seconds, and back to `HIGH`:

```
if (millis() < 5000)
  digitalWrite(13, HIGH);
else if (millis() < 10000)
  digitalWrite(13, LOW);
else
  digitalWrite(13, HIGH);
```

`micros` allows more precise time measurement. This can be helpful when you're controlling a mechanism or communicating with another device.

Analog Read

If you want to read data from a sensor or another analog device, Arduino provides two crucial functions: `analogReference` and `analogRead`.

An analog signal can take an infinite number of values, but the Mega's pins can't read an infinite number of values. Therefore, when writing a sketch that reads analog data, you need to know the maximum voltage that can be read.

By default, the maximum analog voltage that can be read by the Mega is 5 V. This means the analog inputs can only distinguish inputs between 0 V and 5 V. This maximum can be changed with the `analogReference` function, whose argument can take one of four values:

- `DEFAULT`—The default value of 5 V.
- `INTERNAL1V1`—A maximum of 1.1 V.
- `INTERNAL2V56`—A maximum of 2.56 V.
- `EXTERNAL`—The maximum is set by the voltage on the AREF pin.

If the board is oriented as shown in Figure 9.1, AREF is the leftmost pin in the top header. If `analogReference` is called with its argument set to `EXTERNAL`, the board's analog pins will read input voltages between 0 V and the voltage on AREF. Note that AREF must be set to a voltage between 0 V and 5 V.

Like `digitalRead`, `analogRead` accepts the pin number that the voltage should be read from. Also like `digitalRead`, `analogRead` returns an `int`. However, there are two major differences between these functions:

- The `int` returned by `analogRead` ranges from 0 and 1023, where 0 represents a voltage of 0 V and 1023 represents the maximum voltage.
- `analogRead` can only be called for specially configured analog input pins. As shown in Figure 9.1, the Arduino Mega has 16 analog input pins, A0–A15.

An example will make this clear. The following code reads the analog voltage on Pin A5:

```
analog_v = analogRead(A5);
```

Another point about the analog input pins is that they can be accessed by the digital I/O functions, `digitalRead` and `digitalWrite`. For example, the following code reads the digital voltage level from Pin A5:

```
digital_v = digitalRead(A5);
```

Like regular digital pins, the analog input pins are configured in `INPUT` mode by default. With the `pinMode` function, they can be configured in the `OUTPUT` or `INPUT_PULLUP` mode.

Analog Write

The `analogWrite` function is so important that it deserves its own category. When I first saw this function, I assumed the microcontroller had digital-to-analog converters (DACs) capable of converting integer values into true analog outputs. Unfortunately, the Mega doesn't have any DACs, so it's incapable of producing real analog values.

Instead, `analogWrite` on the Mega produces a train of pulses formatted with pulse width modulation (PWM). As introduced in Chapter 2, "Preliminary Concepts," PWM is the primary mechanism for controlling most DC motors. Pulses in a PWM signal have the same height and period, but the pulse width may vary over the course of the signal. The ratio of the pulse width to the period is the duty cycle.

To generate a PWM signal, the `analogWrite` function needs two arguments:

- **Pin number**—`analogWrite` is available only for a specific set of pins (2–13, 44–46 on the Mega). It cannot be called on the analog input pins.
- **Duty cycle**—This value determines the time width of the pulse relative to the time between pulses. This takes a value between 0 (always off) and 255 (always on).

The code in Listing 9.2 shows how `analogWrite` can be used to generate pulses.

Listing 9.2 Ch9/pwm.ino—Pulse Width Modulation

```
/* This sketch produces a pulse-width modulation (PWM) signal
whose duty-cycle switches between 0%, 25%, 50%, and 75%. */

// Assign a name to Pin 13
int pwm_pin = 13;

// Configure Pin 13 as an output pin
void setup() {
  pinMode(pwm_pin, OUTPUT);
}

// Switch the duty-cycle between 25% and 75%
void loop() {
  analogWrite(pwm_pin, 0);       // set duty cycle to 0%
  delay(1000);                   // delay one second
  analogWrite(pwm_pin, 64);      // set duty cycle to 25%
  delay(1000);                   // delay one second
  analogWrite(pwm_pin, 128);     // set duty cycle to 50%
  delay(1000);                   // delay one second
  analogWrite(pwm_pin, 192);     // set duty cycle to 75%
  delay(1000);                   // delay one second
}
```

In this sketch, `setup` configures Pin 13 in `OUTPUT` mode. Then `loop` calls `analogWrite` four times, changing the duty cycle from 0% to 25% to 50% to 75%. This changes the brightness of the LED connected to Pin 13. After each change, the sketch delays for one second. Figure 9.4 gives an idea of what these pulses look like.

The time between pulses (period) varies from board to board, and occasionally from pin to pin. Based on my tests on the Mega, the period for Pins 2, 3, and 5–12 is 2.05 ms. The period for Pins 4 and 13 is about 1.025 ms. Put another way, the PWM frequency for Pins 2, 3, and 5–12 is 488 Hz and the PWM frequency for Pins 4 and 13 is 976 Hz.

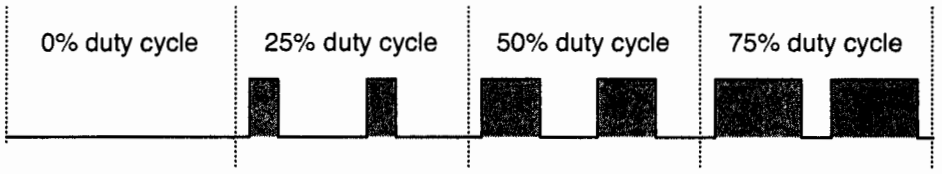

Figure 9.4
PWM output from `analogWrite`

It's possible to change the PWM frequencies of the Arduino Mega with special code. This topic lies beyond the scope of this book, but there are plenty of online resources that explain how it can be done.

9.3 The Arduino Motor Shield

The Arduino Mega has many capabilities, but it can't deliver enough current to control a motor. It also lacks the H bridge needed to reverse a motor's direction. Therefore, before the Mega can control a motor, it must be connected to the Arduino Motor Shield.

In Arduino parlance, a *shield* is a secondary circuit board that can be connected on top of an Arduino board. Many different types of shields are available for Arduino devices, including shields for wireless communication, GPS tracking, and MP3 playing. The motor shield contains resources needed for motor control, and Figure 9.5 shows what it looks like.

The motor shield may seem confusing because of the many connections to support different types of motors. The goal of this section is to explain how it works. Later sections explain how the shield can be used to control specific motors.

9.3.1 Power

The logic devices on the motor shield receive power from the Arduino Mega, but this isn't sufficient to deliver power to a motor. For this reason, the motor shield has its own power connections: the Vin and GND screw terminals in the lower-left of the figure. Vin accepts voltages between 7 V and 12 V and can receive as much as 2 A of current per motor.

If Vin is set to a high voltage, it's important to keep the shield's power from affecting the Arduino Mega. The official directions recommend removing the "Vin Connect" jumper on the underside of the shield. I recommend bending the shield's Vin pin (at the far right of the POWER header) so that it doesn't connect to the Mega.

Above the Vin and GND screw terminals, the motor shield has four connections for output power. These can provide power to two brushed motors or one stepper motor. Later sections explain how this is accomplished.

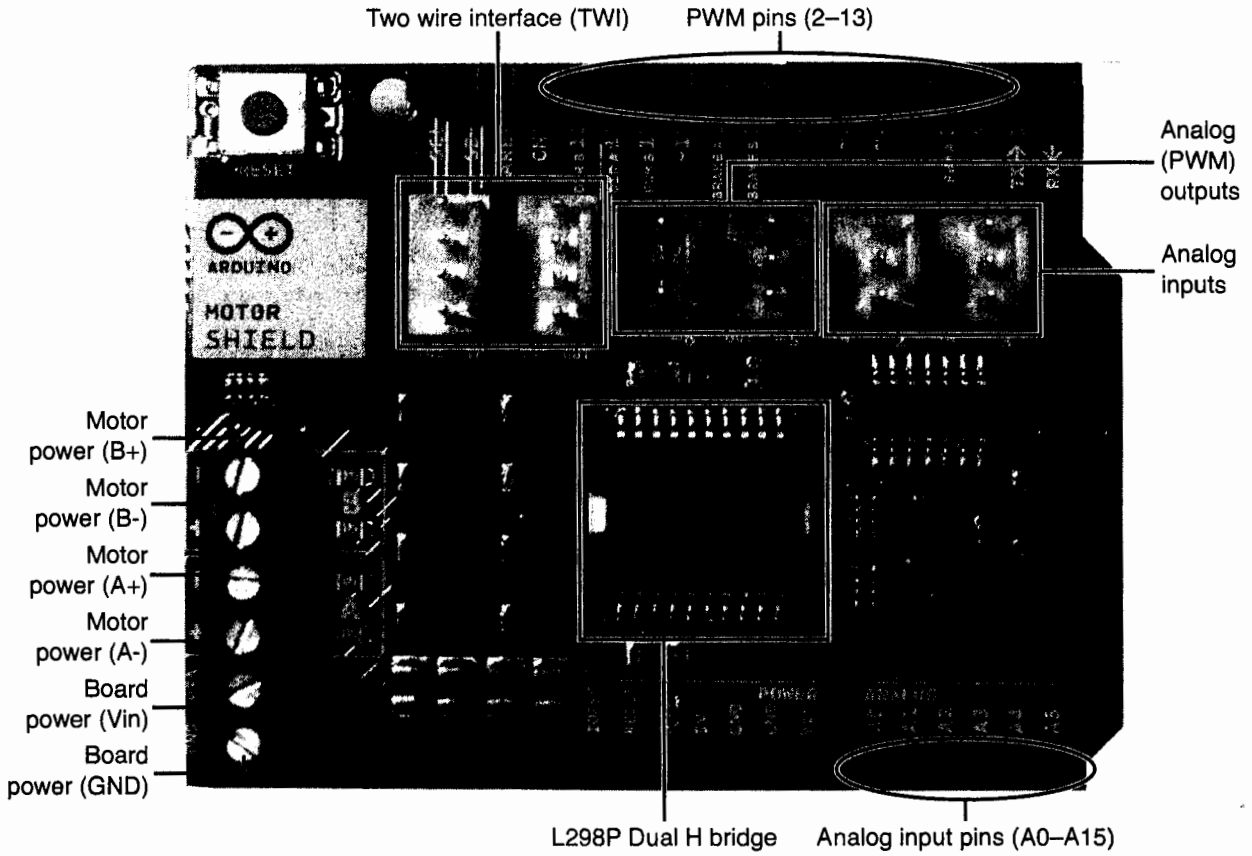

Figure 9.5
The Arduino Motor Shield (v1.1)

9.3.2 The L298P Dual H Bridge Connections

Chapter 3, "DC Motors," introduced the H bridge, whose four switches make it possible to reverse current to a motor. The motor shield contains two H bridges in the form of the L298P integrated circuit. This chip uses bipolar junction transistors (BJTs) to serve as switches, and Figure 9.6 shows how the first H bridge is connected to the shield's signals.

This circuit is complex, but keep in mind that its primary purpose is to deliver power to the motor outputs, MOT_A+ and MOT_A-. To turn the motor in the forward direction, MOT_A+ should be connected to Vin and MOT_A- should be connected to GND. To reverse the motor's direction, MOT_A+ should be connected to GND and MOT_A- should be connected to Vin.

The PWM_A signal receives PWM pulses from the Arduino Mega board. When this is high, the circuit functions normally. When it's low, no voltage is applied to the switches' inputs, which means MOT_A+ and MOT_A- are left unconnected.

If PWM_A is high, the states of the four switches are controlled by DIR_A and BRAKE_A. If DIR_A is high, S_0 connects MOT_A+ to Vin. If DIR_A is low, S_2 connects MOT_A+ to GND.

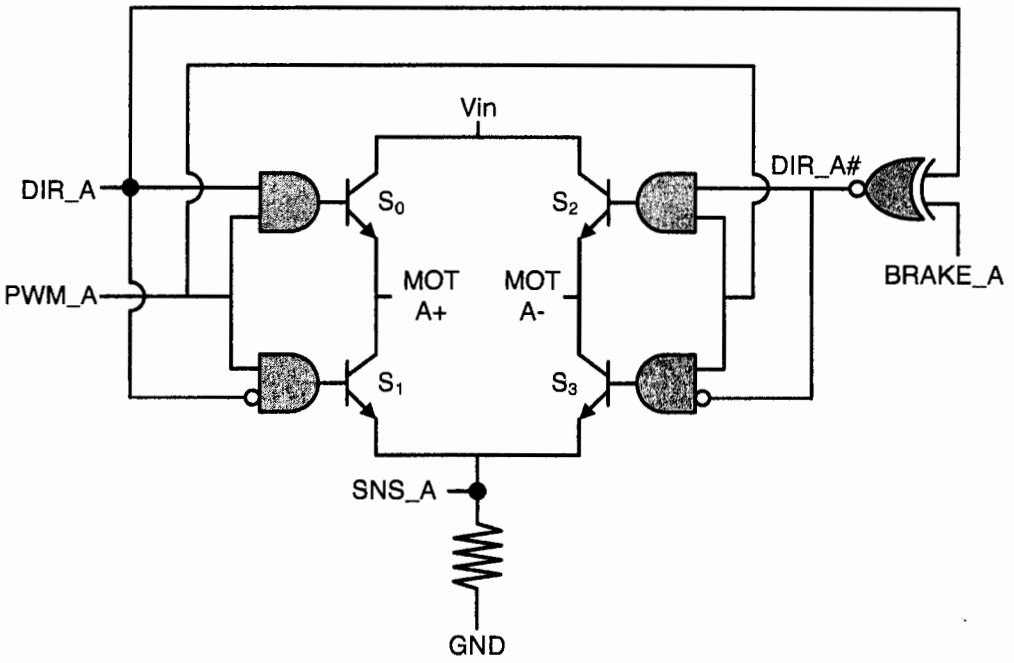

Figure 9.6 H bridge connections

The right of the diagram shows how BRAKE_A affects the circuit. When BRAKE_A is low, DIR_A# is the inverse of DIR_A. This means MOT_A- is connected to GND when MOT_A+ is connected to Vin, and vice versa.

However, when BRAKE_A is high, DIR_A# equals DIR_A. This means MOT_A+ and MOT_A- are always connected to the same source. Because the voltage difference between MOT_A+ and MOT_A- is zero, no current flows through the motor, so it comes to a halt.

To control a motor properly, it's important to know how the signals in Figure 9.6 relate to the Arduino Mega's pins. Table 9.4 lists each of the motor signals and their corresponding pins.

Table 9.4 Motor Signals and Arduino Pins

Motor Signal	Arduino Mega Pin	Description
DIR_A	12	Controls the direction of Motor A
DIR_B	13	Controls the direction of Motor B
PWM_A	3	PWM signal for Motor A
PWM_B	11	PWM signal for Motor B
BRAKE_A	9	Halts Motor A when high
BRAKE_B	8	Halts Motor B when high
SNS_A	A0	Current sensing for Motor A
SNS_B	A1	Current sensing for Motor B

9.3.3 Controlling a Brushed Motor

An example will clarify how the dual H bridge and its connections can be used to control a motor. The code in Listing 9.3 controls a brushed DC motor whose wires are connected to the shield's Motor Power A+ and Motor Power A- screw terminals.

Listing 9.3 Ch9/brushed.ino—Brushed DC Motor Control

```
/* This sketch controls a brushed motor. It drives it in the
forward direction at 75% duty cycle and halts. Then it
drives it in reverse at 75% duty cycle and halts. */

// Assign names to motor control pins
int dir_a = 12;
int pwm_a = 3;
int brake_a = 9;

// Configure the motor control pins in output mode
void setup() {
  pinMode(dir_a, OUTPUT);
  pinMode(pwm_a, OUTPUT);
  pinMode(brake_a, OUTPUT);
}

// Deliver power to the motor
void loop() {

  // Drive the motor forward at 75% duty cycle
  digitalWrite(brake_a, LOW);
  digitalWrite(dir_a, HIGH);
  analogWrite(pwm_a, 192);
  delay(2000);

  // Halt the motor for a second
  digitalWrite(brake_a, HIGH);
  delay(1000);

  // Drive the motor in reverse at 75% duty cycle
  digitalWrite(brake_a, LOW);
  digitalWrite(dir_a, LOW);
  analogWrite(pwm_a, 192);
  delay(2000);

  // Halt the motor for a second
  digitalWrite(brake_a, HIGH);
  delay(1000);
}
```

When the processing loop starts, DIR_A is set to 1 and PWM_A is set to 192. This drives the motor forward at 75% duty cycle. After a halt period, DIR_A is set to 0 and PWM_A is set to 192. This drives the motor in reverse at 75% duty cycle.

9.4 Stepper Motor Control

Of the many motors discussed in this book, stepper motors are the easiest to understand: They rotate through a fixed angle and halt. But they aren't the easiest motors to control. Bipolar steppers have four connections that require signals, and unipolar steppers have six.

The motor shield makes it straightforward to control a stepper. Not only is the shield's hardware ideally suited for the purpose, but Arduino provides free software to get your sketches working. This free software is packaged in the form of a *library*, so the first part of this section explains how to obtain the Stepper library and use its functions.

9.4.1 The Stepper Library

When you install the Arduino environment, you can call about 40 functions, not including the Stream and Serial functions. This set of functions can be extended using libraries. For example, one library contains functions for communicating across a serial peripheral interface (SPI) bus. Another contains functions that control liquid crystal displays (LCDs).

To see what libraries are available, visit http://arduino.cc/en/Reference/Libraries. Most of these libraries fall into one of two categories: standard libraries and contributed libraries. Contributed libraries must be downloaded and installed into the Arduino environment. Standard libraries don't have to be downloaded or installed. They're already included in the environment.

To access functions from a standard library, open a sketch in the environment's editor. Then, in the main menu, go to Sketch > Import Library and select the library you're interested in. This section relies on the Stepper library, and if you select this option, the following code will be prepended to the sketch:

```
#include <Stepper.h>
```

With this line added to the sketch, you can call the functions of the Stepper library. Table 9.5 lists each of them and provides a description of its purpose.

Table 9.5 Functions of the Stepper Library

Function	Description
`Stepper(int steps_per_rev, int pin1, int pin2)`	Returns a `Stepper` object with the given number of steps per revolution and connection pins
`Stepper(int steps_per_rev, int pin1, int pin2, int pin3, int pin4)`	Returns a `Stepper` object with the given number of steps per revolution and connection pins
`setSpeed(int rpm)`	Used to set the stepper speed in revolutions per minute
`step(int steps)`	Used to tell the stepper to turn one or more steps

These functions are straightforward if you know what objects and classes are. Just in case you don't, I'll provide a brief overview of object-oriented theory. Then I'll explain how to use the functions in Table 9.5.

Objects and Classes

The first two functions in Table 9.5 aren't like any of the other functions discussed in this chapter. That is, they're not called in the `setup` or `loop` method. Instead, their purpose is to create a new global variable, and for this reason, the `Stepper` function must be coded above the `setup` function.

The variable created by the `Stepper` functions isn't an `int` or a `float`, but has the `Stepper` data type. Technically speaking, `Stepper` is a *class*, and any variable created by the `Stepper` method is an *object*. Object-oriented programming (OOP) is a deep topic, and many books have been written on it. However, for Arduino development, you need to understand only four points:

- Every object is created by a function called a *constructor*. A class may have multiple constructors, and each must have the same name as its class.
- An object may contain variables of its own. These are called *member variables*.
- An object may contain functions of its own. These are called *member functions*.
- An object's member variables and functions can be accessed by following the object name with a dot (.) and the name of the variable or function.

In Table 9.5, the first two functions are constructors and the second two functions are member functions. As an example, the following code creates a `Stepper` object using the first constructor in the table. Later in the sketch, the `loop` function calls one of the object's member functions.

```
Stepper s = Stepper(200, 6, 5);
...
loop() {
  ...
  s.step(1);
  ...
}
```

The first line calls the `Stepper` constructor. Like a regular function, the constructor accepts arguments and returns a variable. In this case, the variable is a `Stepper` object named `s`.

This object has two member functions it can call: `setSpeed` and `step`. In this code, the object's `step` function is called inside `loop`. The dot between `s` and `step` identifies `step` as a member of the `s` object. It's important to note that a member function must be called with its associated object.

Stepper Functions

In both `Stepper` constructors, the first argument sets the number of steps the motor needs to turn to complete a revolution. For example, if each turn of the stepper rotates by an angle of 1.8°, the number of steps in a revolution is 360°/1.8° = 200. The constructor requires that this be given as an `int`.

The constructors' other arguments identify which pins control the stepper motor. Depending on the nature of the motor, it may be connected to two or four pins.

After the `Stepper` object is created, `setSpeed` specifies the desired speed of the motor. By combining the steps per revolution and the revolutions per minute, the program determines the time delay between steps.

For example, suppose the motor takes 150 steps to complete a revolution and `setSpeed` is called with a speed of 20 revolutions per minute. The program will determine that the motor should take 150 × 20 = 3000 steps in a minute, or 50 steps per second. Therefore, the program will delay 1/50 = 0.02 seconds between steps.

The last function in the table is `step`, whose argument identifies how many times the motor should step. If the argument is 1, the motor will step once and control will return to the program. If the argument is greater than 1, the motor will step multiple times. In this case, the program will halt until the steps are completed. If the argument is negative, the stepper will rotate in the reverse direction.

9.4.2 Controlling a Stepper Motor

As discussed in Chapter 4, "Stepper Motors," stepper motors come in two types: bipolar and unipolar. As a quick review, here are their characteristics:

- Bipolar steppers have four wires. Unipolar motors have five or six.
- Bipolar steppers require two H bridges for control. Unipolar stepper control is less complex.
- Bipolar steppers are significantly more efficient because they utilize the entire length of each winding when energized.

The drawback of using bipolar steppers is the need for two H bridges, but the motor shield's L298P provides two H bridges, so this isn't a concern. Therefore, this discussion focuses on controlling bipolar steppers. If you have a unipolar stepper, these directions still apply. Just ignore the center tap connections and connect the remaining wires as needed for a bipolar stepper.

Both types of stepper motors have two phases: A/A' and B/B'. Figure 9.7 shows how these phases are related to motor's windings and external connections.

Looking back at Figure 9.6, it should be clear that A and A' should be connected to the MOT_A+ and MOT_A- screw terminals in the lower-left corner of the shield. Similarly, B and B' should be connected to MOT_B+ and MOT_B-. In Figure 9.5, these connections are labeled Motor Power (A+), Motor Power (A-), Motor Power (B+), and Motor Power (B-).

The output of the two H bridges is determined by DIR_A and DIR_B, which correspond to Pins 12 and 13. Therefore, these are the pins provided to the `Stepper` constructor. This is shown in Listing 9.4.

note

Don't be concerned if the colors in Figure 9.7 don't match the wires of your stepper. You can tell which wires are paired together by testing the resistance between them. Also, it doesn't matter if you connect B/B' in place of A/A'. The stepper will still rotate.

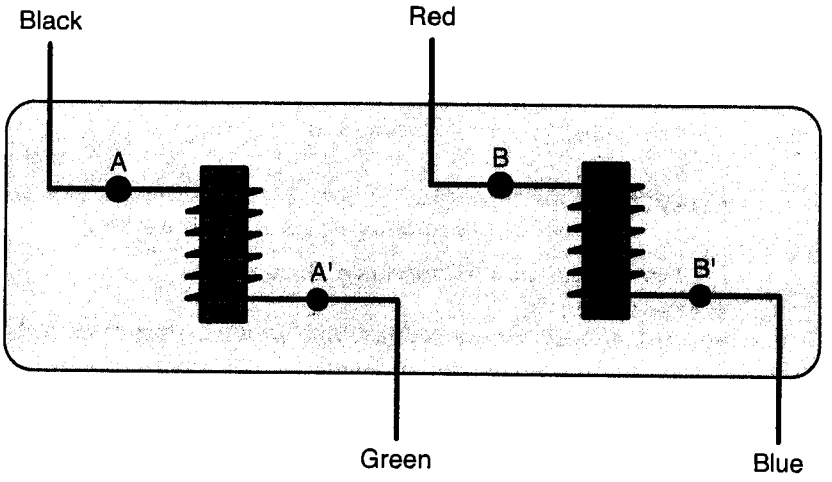

Figure 9.7
Connections of a bipolar stepper

Listing 9.4 Ch9/stepper.ino—Stepper Motor Control

```
/*
This sketch controls a bipolar stepper motor,
stepping ten times in the forward direction and
ten times in the reverse direction.
The steps/revolution is set to 200 (1.8 deg/step)
and the speed is set to 10 RPM.
 */

#include <Stepper.h>

// Set the pin numbers
int pwm_a = 3;
int pwm_b = 11;
int dir_a = 12;
int dir_b = 13;

// Create a stepper object
Stepper s = Stepper(200, dir_a, dir_b);

void setup() {

  // Set speed to 10 revs/min
  s.setSpeed(10);

  // Make sure the two H Bridges are always on
  pinMode(pwm_a, OUTPUT);
  pinMode(pwm_b, OUTPUT);
  digitalWrite(pwm_a, HIGH);
```

```
  digitalWrite(pwm_b, HIGH);
}

void loop() {

  // Ten steps in the forward direction
  s.step(10);
  delay(1000);

  // Ten steps in the reverse direction
  s.step(-10);
  delay(1000);
}
```

In addition to DIR_A and DIR_B, this sketch sets the values of PWM_A and PWM_B. These signals need to be set high to ensure that both H bridges will function normally. Note that, when controlling a stepper, you don't really need PWM signals or braking.

9.5 Servomotor Control

Chapter 5, "Servomotors," discussed the topic of servos and how most servos used by makers don't provide feedback to the controller. This section explains how to control these hobbyist servos by accessing the Arduino Motor Shield. The first part of the section discusses the Servo library and its functions. The second part presents the servo-control code.

9.5.1 The Servo Library

Like the Stepper library discussed earlier, the Servo library is a standard library, which means it comes preinstalled with the Arduino environment. To access this library in a sketch, go to the main menu, select Sketch > Import Library, and select the Servo option. This will prepend the following code to the sketch:

```
#include <Servo.h>
```

Just as the Stepper library defines a class called `Stepper`, the Servo library defines a class called `Servo`. This class contains a number of methods, and Table 9.6 lists each of them.

Table 9.6 Functions of the Servo Library

Function	Description
`attach(int pin)`	Associates the Servo object with the given pin
`attach(int pin, int min, int max)`	Associates the Servo object with the pin and sets the pulse widths for the min/max servo angle
`attached()`	Returns 1 if the servo is attached to a pin and 0 if not

Function	Description
`detach()`	Used to delete the association between the Servo object and its pin
`write(int angle)`	Used to set the servo's angular position
`writeMicroseconds(int time)`	Used to set the pulse width of the signal delivered to the servo
`read()`	Returns the last angle written to the servo

None of these functions is a constructor, so these functions can't create `Servo` objects. Instead, a `Servo` object can be declared like any other variable. This is shown in the following line of code:

```
Servo sv;
```

The `Servo` object must be associated with the pin that will deliver PWM control signals to the motor. This association is created by the `attach` function, which can be called with one argument or three arguments.

When you're controlling a servo's shaft, the minimum pulse width produces the minimum angle (usually 0°) and the maximum pulse width produces the maximum angle (usually 180°). If `attach` is called with one argument (the PWM pin number), the minimum pulse width is 544 and the maximum pulse width is 2400.

If `attach` is called with three arguments, the first argument is the PWM pin number, the second is the minimum pulse width, and the third is the maximum pulse width. As an example, the following code associates `sv` with Pin 8 and sets the min/max pulse widths to 900/2100:

```
sv.attach(8, 900, 2100);
```

After the `Servo` object is associated with its pin, the angle of the motor's shaft can be set with `write` or `writeMicroseconds`. The `write` function accepts the desired angle in degrees, and the program determines the appropriate pulse width. If the desired pulse width is already known, `writeMicroseconds` accepts the pulse width in milliseconds.

9.5.2 Controlling the Servomotor

As discussed in Chapter 5, hobbyist servomotors generally have three wires:

- **Power**—Provides about 5–6 V to power the motor (usually the red wire)
- **Signal**—Controls the servo with PWM signals
- **Ground**—Provides electrical ground (usually the black wire)

In Figure 9.5, the shield has two orange connectors labeled Analog (PWM) Outputs. These connectors have three pins each: power, PWM control, and ground, in that order.

Unfortunately, the connections of my hobbyist servos are ordered with the PWM control signal first, followed by the power and ground. For this reason, I've found it necessary to separate the servomotor's wires and manually connect them to the shield's header connections.

The code in Listing 9.5 shows how to control a servomotor whose PWM control wire is connected to Pin 6.

Listing 9.5 Ch9/servo.ino—Servomotor Control

```
/*
This sketch controls a hobbyist servomotor.
It rotates the shaft 180 degrees forward and 180 degrees back.
 */

#include <Servo.h>

Servo sv;        // Servo object
int angle;       // servo's angular position

void setup() {

  // Attach the Servo object to Pin 6
  sv.attach(6, 800, 2200);
}

void loop() {

  // Rotate from 0 to 180 degrees
  for(angle = 0; angle < 180; angle += 1) {
    sv.write(angle);
    delay(10);
  }

  // Rotate from 180 to 0 degrees
  for(angle = 180; angle >= 1; angle -=1) {
    sv.write(angle);
    delay(10);
  }
}
```

In this case, the minimum angle corresponds to a pulse width of 800 microseconds and the maximum angle corresponds to a pulse width of 2200 microseconds. These values are specified in the `attach` function.

9.6 Summary

After so many chapters discussing motor theory, it's good to see how real-world motors can be controlled with real-world electronics. Many motor control systems cost a great deal of money, but the Arduino Mega and Arduino Motor Shield are both inexpensive and easy to use. With the Arduino programming environment, you can have a motor control sketch coded and compiled in less than an hour.

The primary device of the Arduino Mega is the ATmega2560 microcontroller. This single chip contains all the ROM, RAM, and processing power needed to execute Arduino sketches, but the Mega doesn't have the resources needed to control motors.

In contrast, the Arduino Motor Shield is designed to control brushed DC motors, stepper motors, and servomotors. Its primary device is the L298P, which contains two H bridges. The H bridge connections are complex, but they allow the shield to halt a motor, deliver PWM signals, and reverse the motor's direction. In addition, the shield can deliver more electrical power than the Mega can.

The capabilities of the Arduino programming environment can be extended with libraries. The first library discussed in this chapter is the Stepper library, which makes it possible to control stepper motors. This library has four functions, and two of them are constructors that return a `Stepper` object. After this object is created, its member functions—`setSpeed` and `step`—can be called.

The functions of the Servo library make it possible to control servomotors. To accomplish this in code, a sketch needs to declare a `Servo` object and associate it with a pin capable of delivering PWM signals. When you're using this library, it's important to know the minimum and maximum pulse widths needed to set the angle of the servomotor's shaft.