Programming Language

A common misconception about Arduinos is that they have their own programming language. Actually, they are programmed in the language simply called C. This language has been around since the early days of computing. What Arduino does provide is a nice easy-to-use set of commands written in C that you can use in your programs.

Purists may wish to point out that Arduino uses C++, the object-oriented extension to C. Although, strictly speaking, this is true, having only 1 or 2kB of memory available generally means that the kinds of habits encouraged by object-oriented programming are not normally a good idea with Arduino, so aside from a few specialized areas, you are effectively programming in C.

Let's start by modifying the Blink sketch.

Modifying the Blink Sketch

It may be that your Arduino was already blinking when you first plugged it in. That is because the Arduino is often shipped with the Blink sketch installed.

If this is the case, then you might like to prove to yourself that you have actually done something by changing the blink rate. Let's look at the Blink sketch to see how to change it to make the LED blink faster.

The first part of the sketch is just a comment telling you what the sketch is supposed to do. A comment is not actual program code. Part of the preparation for the code being uploaded is for all such "comments" to be stripped out. Anything between /* and */ is ignored by the computer, but should be readable by humans.

```
/*
  Blink
  Turns on an LED for one second, then off for one second, repeatedly.

  This example code is in the public domain.
*/
```

Then, there are two individual line comments, just like the block comments, except they start with //. These comments tell you what is

happening. In this case, the comment helpfully tells you that pin 13 is the pin we are going to flash. We have chosen that pin because on an Arduino Uno it is connected to the built-in "L" LED.

```
// Pin 13 has an LED connected on most Arduino boards.
// give it a name:
int led = 13;
```

The next part of the sketch is the **setup** function. Every Arduino sketch must have a **setup** function, and this function runs every time the Arduino is reset, either because (as the comment says) the Reset button is pressed or the Arduino is powered up.

```
// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}
```

The structure of this text is a little confusing if you are new to programming. A *function* is a section of code that has been given a name (in this case, the name is **setup**). For now, just use the previous text as a template and know that you must start your sketch with the first line **void setup() {** and then enter the commands that you want to issue, each on a line ending with a semicolon (;). The end of the function is marked with a } symbol.

In this case, the only command Arduino will issue is the **pinMode(led, OUTPUT)** command that, not unsurprisingly, sets that pin to be an output.

Next comes the juicy part of the sketch, the **loop** function.

Like the **setup** function, every Arduino sketch has to have a **loop** function. Unlike **setup**, which only runs once after a reset, the **loop** function runs continuously. That is, as soon as all its instructions have been run, it starts again.

In the **loop** function, you turn on the LED by issuing the **digitalWrite(led, HIGH)** instruction. You then set the sketch to pause for a second by using the command **delay(1000)**. The value 1000 is for 1000 milliseconds or 1 second. You then turn the LED back on again and delay for another second before the whole process starts over.

```
// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);               // wait for a second
}
```

To modify this sketch to make the LED blink faster, change both occurrences of 1000 to be **200**. These changes are both in the **loop** function, so your function should now look like this:

```
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(200);                // wait for a second
  digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
  delay(200);                // wait for a second
}
```

If you try and save the sketch before uploading it, the Arduino IDE reminds you that it is a "read-only" example sketch, but it will offer you the option to save it as a copy, which you can then modify to your heart's content.

You do not have to do this; you can just upload the sketch unsaved. But if you do decide to save this or any other sketch, you will find that it then appears in the File | Sketchbook menu on the Arduino IDE.

So, either way, click the Upload button again, and when the uploading is complete, the Arduino resets itself and the LED should start to blink much faster.

Variables

Variables give a name to a number. Actually, they can be a lot more powerful than this, but for now, we'll use them for this purpose.

When defining a variable in C, you have to specify the type of variable. For example, if you want your variables to be whole numbers, you would use *int* (short for *integer*). To define a variable called **delayPeriod** with a value of **200**, you need to write:

```
int delayPeriod = 200;
```

Notice that because **delayPeriod** is a name, there cannot be any spaces between words. The convention is to start variables with a lowercase letter and begin each new word with an uppercase letter. Programmers often call this *bumpy case* or *camel case*.

Let's fit this into the blink sketch, so that instead of "hard-coding" the value **200** for the length of delay, we use a variable instead:

```
int led = 13;
int delayPeriod = 200;

void setup()
{
  pinMode(led, OUTPUT);
}

void loop()
{
 digitalWrite(led, HIGH);
 delay(delayPeriod);
 digitalWrite(led, LOW);
 delay(delayPeriod);
}
```

At each place in the sketch where we used to refer to **200**, we now refer to **delayPeriod**.

Now, if you want to make the sketch blink faster, you can just change the value of **delayPeriod** in one place.

If

Normally, your lines of program are executed in order one after the other, with no exceptions. But what if you don't want to do that? What if you only want to execute part of a sketch if some condition is true?

A good example of that might be to only do something when a button, attached to the Arduino, is pressed. The code might look like this:

```
void setup()
{
  pinMode(5, INPUT_PULLUP);
  pinMode(9, OUTPUT);
}
```

```
void loop()
{
  if (digitalRead(5) == LOW)
  {
    digitalWrite(9, HIGH);
  }
}
```

In this case, the condition (after the **if**) is that the value read from pin 5 has a value of **LOW**. The double equals symbol == is used for comparing two values. It is easy to confuse it with a single equals sign that assigns a value to a variable. An **if** statement says, if this condition is true, then the commands inside the curly braces are executed. In this case, the action is to set digital output to **9, HIGH**.

If the condition is not true, then the Arduino just continues on with the next thing. In this case, that is the loop function, which runs again.

Loops

As well as conditionally performing some of the actions, you also need your sketch to be able to repeat actions over and over again. You get this for free of course by putting commands into the sketch's **loop** function. That is, after all, what happens with the Blink example.

Sometimes, however, you'll need to be more specific about the number of times that you want to repeat something. You can accomplish this with the **for** command, which allows you to use a counter variable. For example, let's write a sketch that blinks the LED ten times. Later, you'll see why this approach might be considered less than ideal under some circumstances, but for now, it will do just fine.

```
// sketch 01_01_blink_10
int ledPin = 13;
int delayPeriod = 200;
void setup()
{
  pinMode(ledPin, OUTPUT);
}
void loop()
{
```

```
  for (int i = 0; i < 10; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
  }
}
```

NOTE *As this is the first full sketch, it's named in a comment at the top of the file. All the sketches named in this way can be downloaded from the author's website at www.simonmonk.org.*

To install all the sketches into your Arduino environment, unzip the file containing the sketches into your Arduino directory, which you'll find in your Documents folder. The Arduino IDE automatically creates this folder for you the first time it is run.

The **for** command defines a variable called **i** and gives it an initial value of **0**. After the **;** the text **i < 10** appears. This is the condition for staying in the loop. In other words, while **i** is less than **10**, keep doing the things inside the curly brackets.

The last part of the **for** command is **i++**. This is C shorthand for "i = i + 1" which, not surprisingly, adds 1 to the value of **i**. One is added to the value of **i** each time around the loop. This is what ensures that you can escape from the loop, because if you keep adding 1 to **i**, eventually it will be greater than 10.

Functions

Functions are a way to group a set of programming commands into a useful chunk. This helps to divide your sketch into manageable chunks, making it easier to use.

For example, let's write a sketch that makes the Arduino blink rapidly 10 times when it first starts and then blink steadily once each second thereafter.

Read through the following listing, and then I'll explain what is going on.

```
// sketch 01_02_blink_fast_slow
int ledPin = 13;

void setup()
{
  pinMode(ledPin, OUTPUT);
  flash(10, 100);
}

void loop()
{
  flash(1, 500);
}

void flash(int n, int delayPeriod)
{
  for (int i = 0; i < n; i++)
  {
    digitalWrite(ledPin, HIGH);
    delay(delayPeriod);
    digitalWrite(ledPin, LOW);
    delay(delayPeriod);
  }
}
```

The **setup** function now contains a line that says **flash(10, 100);**. This means flash **10** times with a **delayPeriod** of **100** milliseconds. The **flash** command is not a built-in Arduino command; you are going to create this quite useful function yourself.

The definition of the function is at the end of the sketch. The first line of the function definition is

```
void flash(int n, int delayPeriod)
```

This tells the Arduino that you are defining your own function called **flash** and that it takes two parameters, both of which are **int**s. The first is **n**, which is the number of times to flash the LED, and the second is **delayPeriod**, which is the delay to use between turning the LED on or off.

These two parameter variables can only be used inside the function. So, **n** is used in the **for** command to determine how many times to repeat the loop, and **delayPeriod** is used inside the **delay** commands.

The sketch's **loop** function also uses the previous **flash** function, but with a longer **delayPeriod**, and it only makes the LED flash once. Because it is inside **loop**, it will just keep flashing anyway.

Digital Inputs

To get the most out of this section, you need to find a short length of wire or even a metal paperclip that has been straightened.

Load the following sketch and run it:

```
// sketch 01_03_paperclip
int ledPin = 13;
int switchPin = 7;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(switchPin, INPUT_PULLUP);
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {
    flash(100);
  }
  else
  {
    flash(500);
  }
}

void flash(int delayPeriod)
{
  digitalWrite(ledPin, HIGH);
  delay(delayPeriod);
  digitalWrite(ledPin, LOW);
  delay(delayPeriod);
}
```

Use your wire or paperclip to connect the GND pin to digital pin 7, as shown in Figure 1-13. You can do this with your Arduino plugged in, but

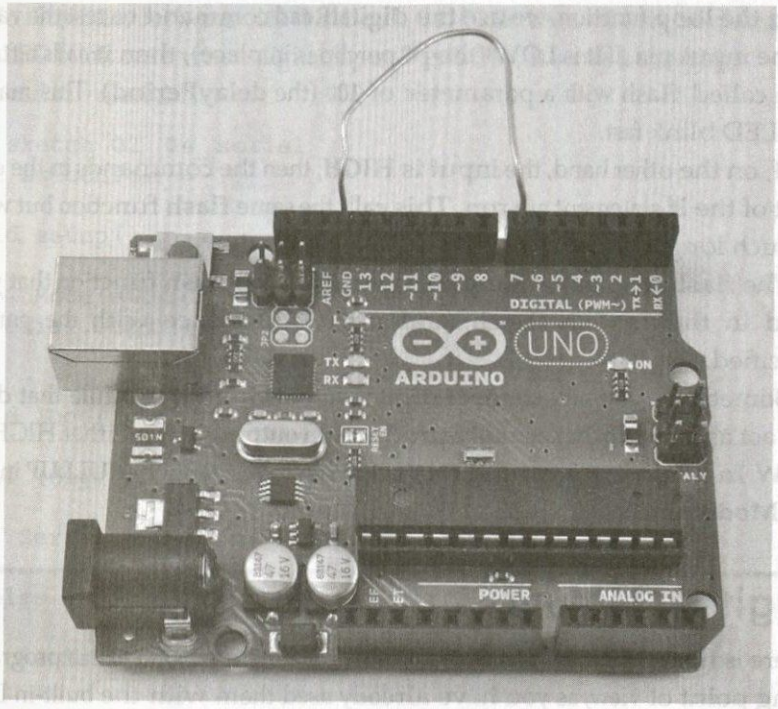

Figure 1-13 *Using a digital input*

only after you have uploaded the sketch. The reason is that if on some previous sketch pin 7 had been set to an output, then connecting it to the GND would damage the pin. Since the sketch sets pin 7 to be an input, this is safe.

This is what should happen: when the paperclip is connected, the LED will flash quickly, and when it is not connected, it will flash slowly.

Let's dissect the sketch and see how it works.

First, we have a new variable called **switchPin**. This variable is assigned to pin 7. So the paperclip is acting like a switch. In the **setup** function, we specify that this pin will be an input using the **pinMode** command. The second argument to **pinMode** is not simply **INPUT** but actually **INPUT_PULLUP**. This tells the Arduino that, by default, the input is to be **HIGH**, unless it is pulled **LOW** by connecting it to GND (with the paperclip).

In the **loop** function, we use the **digitalRead** command to test the value at the input pin. If it is **LOW** (the paperclip is in place), then it calls a function called **flash** with a parameter of **100** (the **delayPeriod**). This makes the LED blink fast.

If, on the other hand, the input is **HIGH**, then the commands in the **else** part of the **if** statement are run. This calls the same **flash** function but with a much longer delay, making the LED blink slowly.

The **flash** function is a simplified version of the **flash** function that you used in the previous sketch, and it just blinks once with the period specified.

Sometimes you will connect digital outputs from a module that does not act as a switch, but actually produces an output that is either HIGH or LOW. In this case, you can use **INPUT** rather than **INPUT_PULLUP** in the **pinMode** function.

Digital Outputs

There is not really much new to say about digital outputs from a programming point of view, as you have already used them with the built-in LED on pin 13.

The essence of a digital output is that in your **setup** function you define them as being an output using this command:

```
pinMode(outputPin, OUTPUT);
```

When you want to set the output **HIGH** or **LOW**, you use the **digitalWrite** command:

```
digitalWrite(outputPin, HIGH);
```

The Serial Monitor

Because your Arduino is connected to your computer by USB, you can send messages between the two using a feature of the Arduino IDE called the *Serial Monitor*.

To illustrate, let's modify the sketch 01_03 so that, instead of changing the LED blink rate when digital input 7 is LOW, it sends a message.

Load this sketch:

```
// sketch 01_04_serial
int switchPin = 7;

void setup()
{
  pinMode(switchPin, INPUT_PULLUP);
  Serial.begin(9600);
}

void loop()
{
  if (digitalRead(switchPin) == LOW)
  {
    Serial.println("Paperclip connected");
  }
  else
  {
    Serial.println("Paperclip NOT connected");
  }
  delay(1000);
}
```

Now open the Serial Monitor on the Arduino IDE by clicking the icon that looks like a magnifying glass on the toolbar. You should immediately start to see some messages appear, once each second (Figure 1-14).

Disconnect one end of the paperclip, and you should see the message change.

Because you are no longer using the built-in LED, you do not need the **ledPin** variable any more. Instead, you need to use the **Serial.begin** command to start serial communications. The parameter is the baud rate. In Chapter 13, you will find out much more about serial communications.

To write messages to the Serial Monitor, all you need to do is use the **Serial.println** command.

In this example, the Arduino is sending messages to the Serial Monitor.

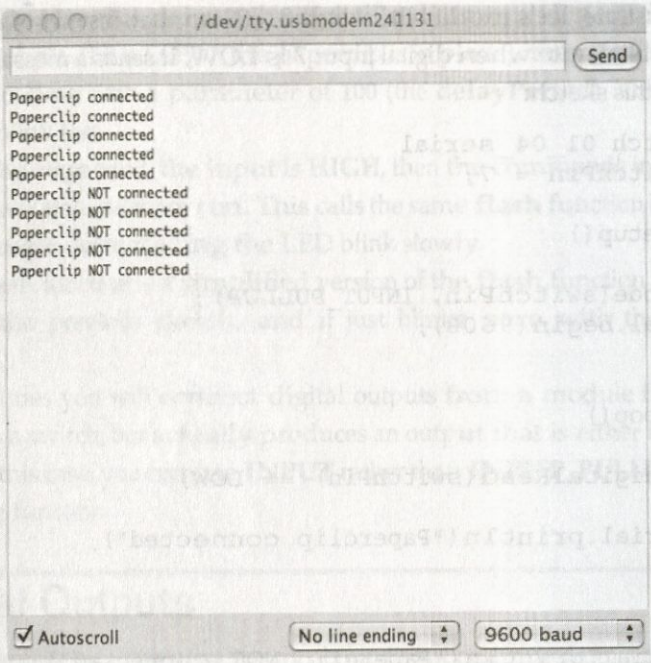

Figure 1-14 *The Serial Monitor*

Arrays and Strings

Arrays are a way of containing a list of values. The variables you have met so far have only contained a single value, usually an **int**. By contrast, an array contains a list of values, and you can access any one of those values by its position in the list.

C, in common with most programming languages, begins its index positions at 0 rather than 1. This means that the first element is actually element zero.

You have already met one kind of array in the last section when you learned about the Serial Monitor. Messages like **"Paperclip NOT connected"** are called *character arrays* because they are essentially collections of characters.

For example, let's teach Arduino to talk gibberish over the Serial Monitor.

The following sketch has an array of character arrays and will pick one at random and display it on the Serial Monitor after a random amount of time. This sketch has the added advantage of showing you how to produce random numbers with an Arduino.

```
// sketch 01_05_gibberish

char* messages[] = {
                   "My name is Arduino",
                   "Buy books by Simon Monk",
                   "Make something cool with me",
                   "Raspberry Pis are fruity"};

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

void loop()
{
  int delayPeriod = random(2000, 8000);
  delay(delayPeriod);
  int messageIndex = random(4);
  Serial.println(messages[messageIndex]);
}
```

Each of the messages, or *strings* as collections of characters are often called, has a data type of **char***. The * is a pointer to something. We'll get to the advanced topic of pointers in Chapter 6. The [] on the end of the variable declaration indicates that the variable is an array of **char*** rather than just a single **char*** on its own.

Inside the **loop** function, the value **delayPeriod** is assigned a random value between **2000** and **7999** (the second argument to "random" is exclusive). A pause of this length is then set using the **delay** function.

The **messageIndex** variable is also assigned a random value using **random**, but this time **random** is only given one parameter, in which case a random number between 0 and 3 is generated as the index for the message to be displayed.

Finally, the message at that position is sent to the Serial Monitor. Try out the sketch, remembering to open the Serial Monitor.

Analog Inputs

The Arduino pins labeled A0 to A5 can measure the voltage applied to them. The voltage must be between 0 and 5V. The built-in Arduino function that does this is **analogRead**, and it returns a value between 0 and 1023: 0 at 0V and 1023 at 5V. So to convert that number into a value between 0 and 5, you have to divide 1023/5 = 204.6.

To measure voltage, **int** is not the ideal data type as it only represents whole numbers and it would be good to see the fractional voltage, for which you need to use the **float** data type.

Load this sketch onto your Arduino and then attach the paperclip between A0 and 3.3V (Figure 1-15).

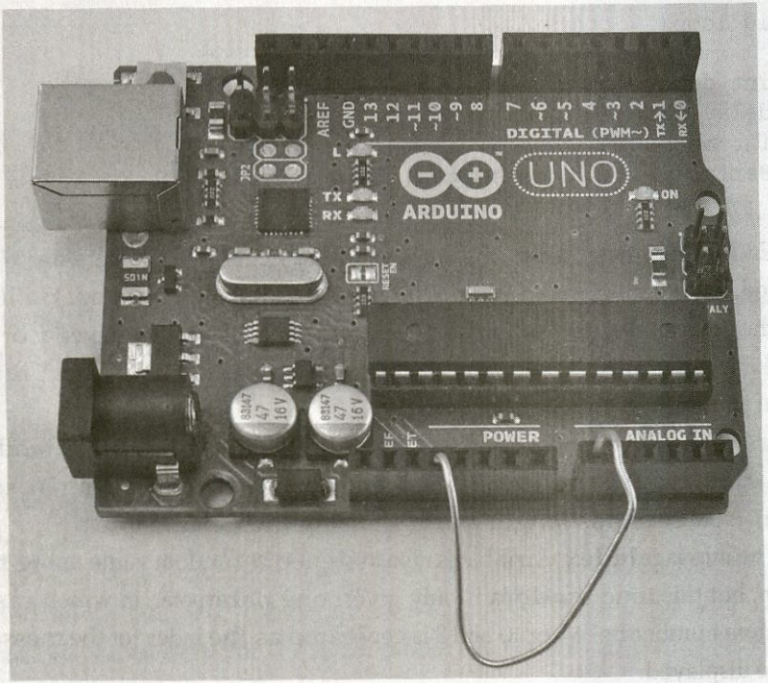

Figure 1-15 *Connecting 3.3V to A0*

```
// sketch 01_06_analog
int analogPin = A0;

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

void loop()
{
  int rawReading = analogRead(analogPin);
  float volts = rawReading / 204.6;
  Serial.println(volts);
  delay(1000);
}
```

Open the Serial Monitor, and a stream of numbers should appear (Figure 1-16). These should be close to 3.3.

CAUTION *Do not connect any of the supply voltages together (5V, 3.3V, or GND). Creating such a short circuit would probably damage your Arduino and could even damage your computer.*

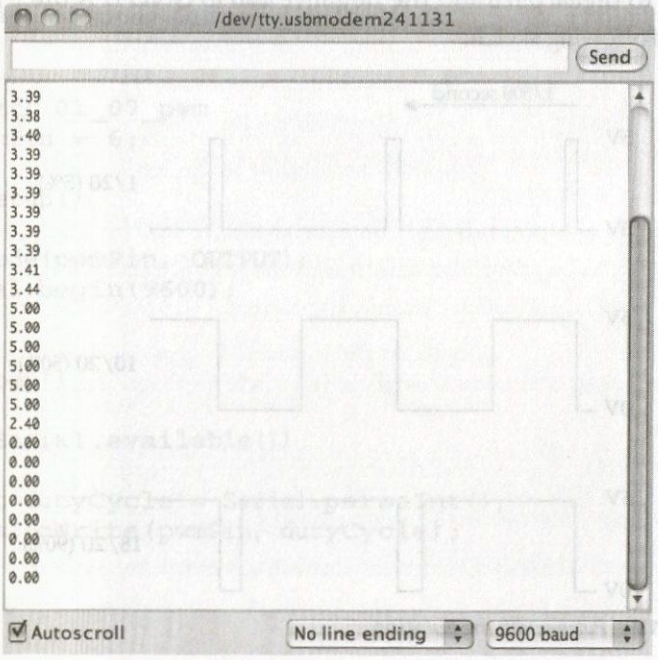

Figure 1-16 *Voltage readings*

If you now keep one end of the paperclip in A0 but move the other end of the paperclip to 5V, the readings will change to around 5V. Moving the same end to GND gives you a reading of 0V.

Analog Outputs

The Arduino Uno does not produce true analog outputs (for that you need an Arduino Due), but it does have a number of outputs that are capable of producing a pulse-width modulation (PWM) output. This approximates to an analog output by controlling the length of a stream of pulses, as you can see in Figure 1-17.

The longer the pulse is high, the higher the average voltage of the signal. Since there are about 600 pulses per second and most things that you would connect to a PWM output are quite slow to react, the effect is of the voltage changing.

On an Arduino Uno, the pins marked with a little ~ (pins 3, 5, 6, 9, 10, and 11) can be used as analog outputs.

If you have a voltmeter, set it to its 0..20V DC range and attach the positive lead to digital pin 6 and the negative lead to GND (Figure 1-18). Then load the following sketch:

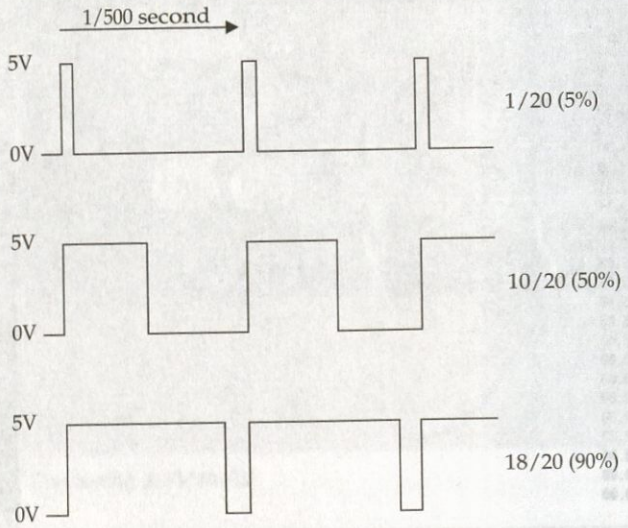

Figure 1-17 *Pulse-width modulation*

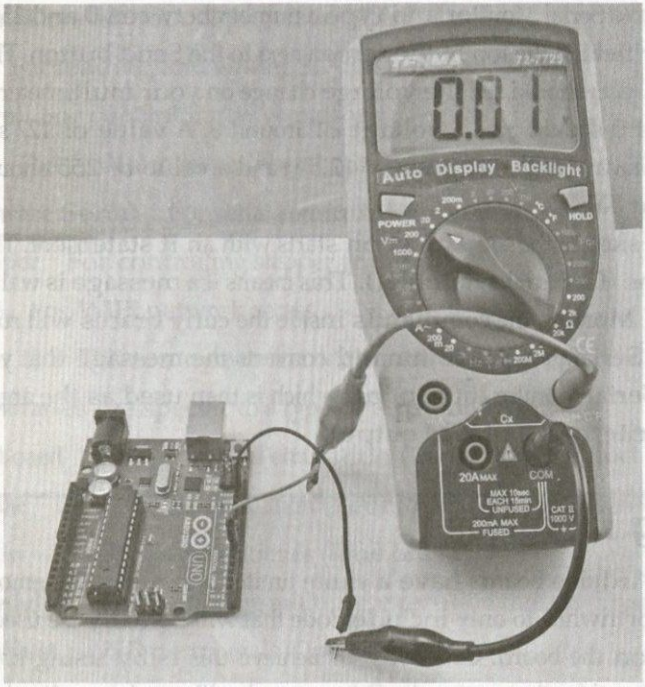

Figure 1-18 *Measuring the output voltage*

```
// sketch 01_07_pwm
int pwmPin = 6;

void setup()
{
  pinMode(pwmPin, OUTPUT);
  Serial.begin(9600);
}

void loop()
{
  if (Serial.available())
  {
    int dutyCycle = Serial.parseInt();
    analogWrite(pwmPin, dutyCycle);
  }
}
```

Open the Serial Monitor and type a number between 0 and 255 into the text entry field at the top of the screen next to the Send button. Then press Send and you should see the voltage change on your multimeter. Sending a value of 0 should give a voltage of around 0. A value of 127 should be about halfway between 0 and 5V (2.5V) and a value of 255 should give a value near 5V.

In this sketch, the **loop** function starts with an **if** statement. The condition for the **if** is **Serial.available()**. This means if a message is waiting from the Serial Monitor, the commands inside the curly braces will run. In this case, the **Serial.parseInt** command converts the message that you typed into the Serial Monitor into an **int**, which is then used as the argument to **analogWrite** to set the PWM output.

Using Libraries

Because Arduino boards have a quite limited amount of memory, you'll find it worthwhile to only include code that will actually be used in what ends up on the board. One way to achieve this is by using libraries. In Arduino, and for that matter in C in general, a *library* is a collection of useful functions.

So, for example, the Arduino IDE includes a library for using an LCD display. This uses about 1.5kB of program memory. There is no point in this library being included unless you are using it, so such libraries are "included" when needed.

You accomplish this using the **#include** directive at the beginning of your sketch. You can add an **include** statement for any libraries that the Arduino IDE has installed using the Sketch | Import Library... menu option.

The Arduino IDE comes with a large selection of "official" libraries, including:

- **EEPROM** For storing data in EEPROM memory
- **Ethernet** For network programming
- **Firmata** The serial communications standard for Arduino to computer

- **LiquidCrystal** For alphanumeric LCD displays
- **SD** For reading and writing SD flash memory cards
- **Servo** For controlling servo motors
- **SPI** The Arduino to peripheral communication bus
- **Software Serial** For serial communication using nonserial pins
- **Stepper** For controlling stepper motors
- **WiFi** For WiFi network access
- **Wire** For I2C communication with peripherals

Some libraries are specific to a type of Arduino board:

- **Keyboard** USB keyboard emulation (Leonardo, Due, and Micro)
- **Mouse** USB mouse emulation (Leonardo, Due, and Micro)
- **Audio** Audio playing utilities (Due only)
- **Scheduler** For managing multiple execution threads (Due only)
- **USBHost** USB peripherals (Due only)

Finally, there are a huge number of libraries that other Arduino users have written that can be downloaded from the Internet. Some of the more popular ones are

- **OneWire** For reading data from Dallas Semiconductor's range of digital devices using the 1-wire bus interface
- **Xbee** For Wireless serial communication
- **GFX** A graphics library for many different types of display from Adafruit
- **Capacitive Sensing** For proximity detection
- **FFT** Frequency analysis library

New libraries appear all the time and you may find them on the official Arduino site (http://arduino.cc/en/Reference/Libraries) or you may find them with an Internet search.

If you want to use one of these last categories of libraries, then you need to install it by downloading the library and then saving it to the Libraries

folder within your Arduino folder (in your Documents folder). Note that if there is no Libraries folder, you will need to create it the first time that you add a library.

For the Arduino IDE to become aware of a library that you have installed, you need to exit and restart the IDE.

Arduino Data Types

A variable of type **int** in Arduino C uses 2 bytes of data. Unless a sketch becomes very memory hungry, then **int**s tend to be used for almost everything, even for Boolean values and small integers that could easily be represented in a single byte value.

Table 1-1 contains a full list of the data types available.

Type	Memory (bytes)	Range	Notes
boolean	1	true or false (0 or 1)	Used to represent logical values.
char	1	–128 to +128	Used to represent an ASCII character code; for example, A is represented as 65. Negative numbers are not normally used.
byte	1	0 to 255	Often used for communicating serial data, as a single unit of data. See Chapter 9.
int	2	–32768 to +32767	These are signed 16 bit values.
unsigned int	2	0 to 65536	Used for extra precision when negative numbers are not needed. Use with caution as arithmetic with **int**s may cause unexpected results.
long	4	2,147,483,648 to 2,147,483,647	Needed only for representing very big numbers.
unsigned long	4	0 to 4,294,967,295	*See* unsigned int.
float	4	–3.4028235E+38 to + 3.4028235E+38	Used to represent floating point numbers.
double	4	as float	Normally, this would be 8 bytes and higher precision than **float** with a greater range. However, on Arduino **double** is the same as **float**.

Table 1-1 *Data Types in Arduino C*

Arduino Commands

A large number of commands are available in the Arduino library, and a selection of the most commonly used commands is listed, along with examples, in Table 1-2.

Command	Example	Description
	Digital I/O	
pinMode	**pinMode(8, OUTPUT);**	Sets pin 8 to be an output. The alternative is to set it to be **INPUT** or **INPUT_PULLUP**.
digitalWrite	**digitalWrite(8, HIGH);**	Sets pin 8 high. To set it low, use the constant **LOW** instead of **HIGH**.
digitalRead	**int i;** **i = digitalRead(8);**	Sets the value of **i** to **HIGH** or **LOW**, depending on the voltage at the pin specified (in this case, pin 8).
pulseIn	**i = pulseIn(8, HIGH)**	Returns the duration in microseconds of the next **HIGH** pulse on pin 8.
tone	**tone(8, 440, 1000);**	Makes pin 8 oscillate at 440 Hz for 1000 milliseconds.
noTone	**noTone();**	Cuts short the playing of any tone that was in progress.
	Analog I/O	
analogRead	**int r;** **r = analogRead(0);**	Assigns a value to **r** of between 0 and 1023: 0 for 0V, 1023 if pin0 is 5V (3.3V for a 3V board).
analogWrite	**analogWrite(9, 127);**	Outputs a PWM signal. The duty cycle is a number between 0 and 255, 255 being 100%. This must be used by one of the pins marked as PWM on the Arduino board (3, 5, 6, 9, 10, and 11).

Table 1-2 *Arduino Library Functions* (continued)

Command	Example	Description
	Time Commands	
millis	**unsigned long l;** **l = millis();**	The variable type **long** in Arduino is represented in 32 bits. The value returned by **millis()** is the number of milliseconds since the last reset. The number wraps around after approximately 50 days.
micros	**long l;** **l = micros();**	See **millis**, except this is microseconds since the last reset. It wraps after approximately 70 minutes.
delay	**delay(1000);**	Delays for 1000 milliseconds or 1 second.
delayMicroseconds	**delayMicroseconds(100000);**	Delays for 100,000 microseconds. Note the minimum delay is 3 microseconds; the max is around 16 milliseconds.
	Interrupts (see Chapter 3)	
attachInterrupt	**attachInterrupt(1, myFunction, RISING);**	Associates the function **myFunction** with a rising transition on interrupt 1 (D3 on an Uno).
detachInterrupt	**detachInterrupt(1);**	Disables any interrupt on interrupt 1.

Table 1-2 *Arduino Library Functions*

For a full reference to all the Arduino commands, see the official Arduino documentation at http://arduino.cc.

Summary

By necessity, this chapter has been a very condensed introduction to the world of Arduino. If you require more information about the basics, then there are many online resources, including free Arduino tutorials at http://www.learn.adafruit.com.

In the next chapter, we will dig under the surface of Arduino and see just how it works and what is going on inside the nice, easy-to-use Arduino environment.