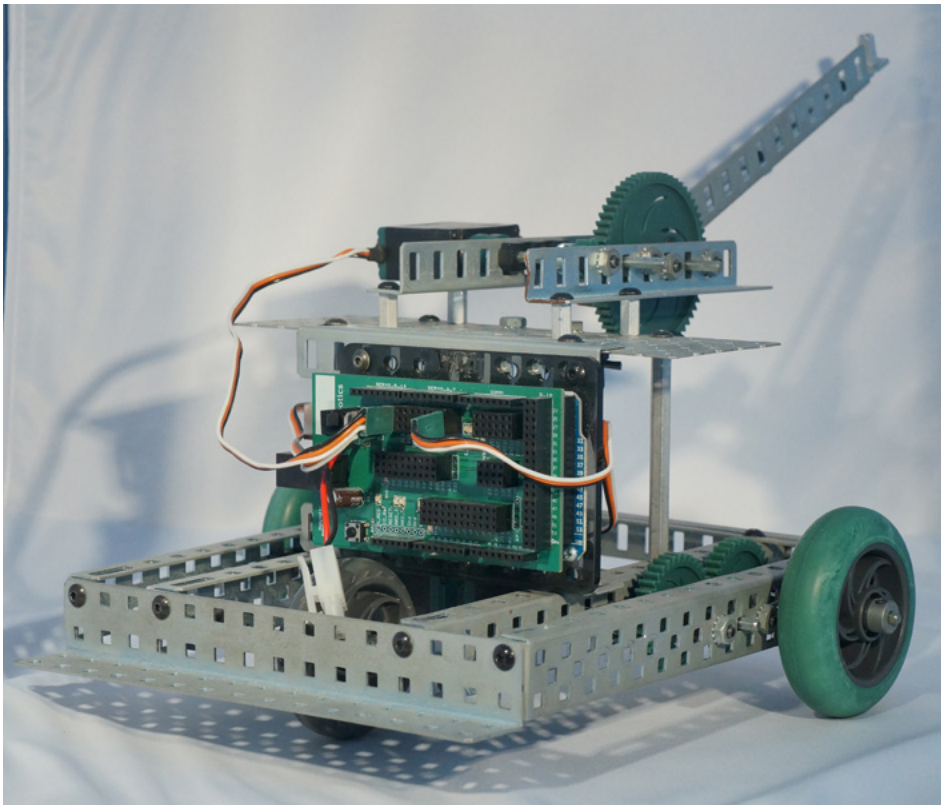
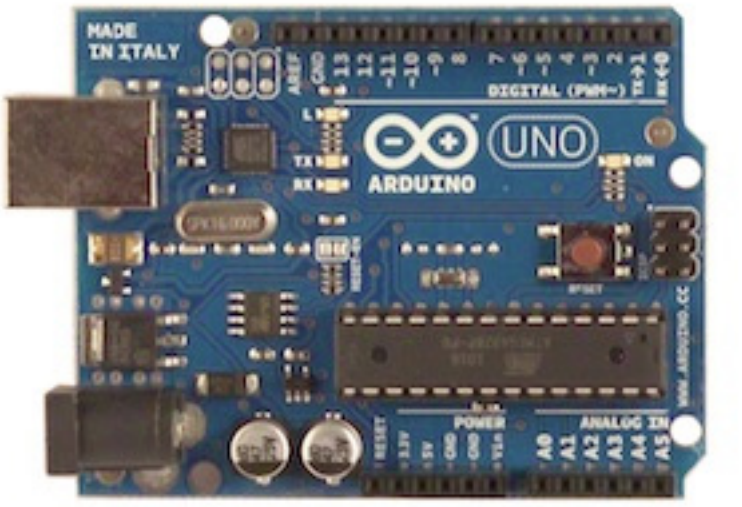
Chapter 1: Getting Started

Hello, and welcome to RBE 1001! This book will help you learn about how to use your Arduino to control a robot, including how to use various types of motors and sensors with your Arduino. In this first chapter, we'll be covering the basics of how to interact with motors using Electronic Speed Controllers, how to use servos, and the structure of a PWM signal. Additionally, we're going to be looking at basic coding structures and practices, such as the basic structure of an Arduino program, good commenting, and code reuse through functions.

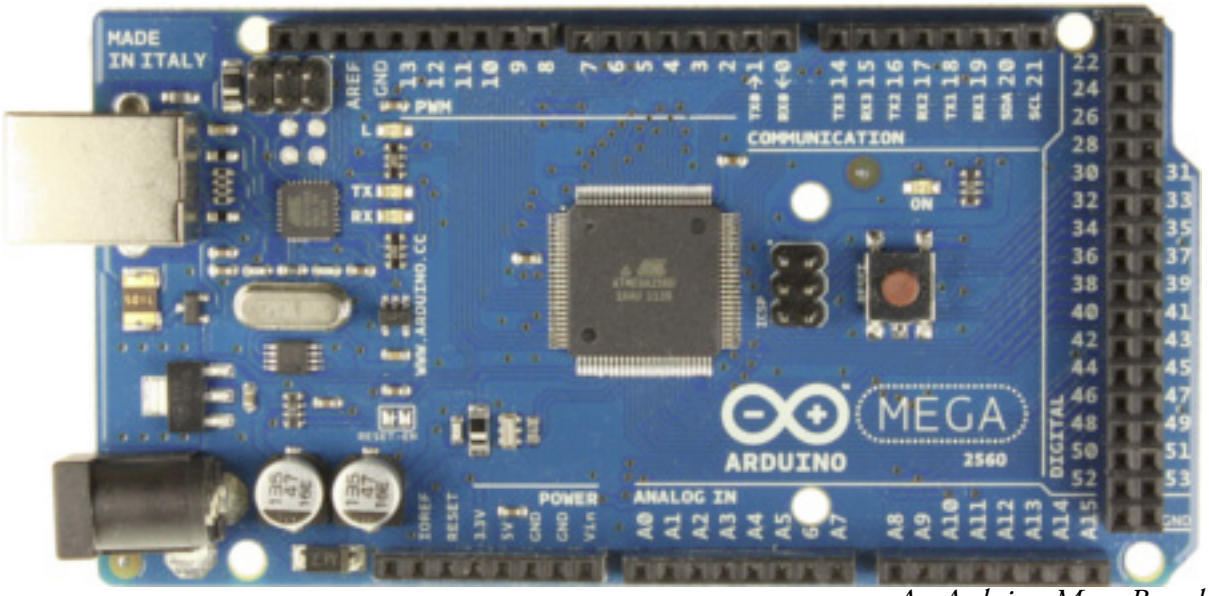


1.1: The Arduino

You will be programming all of your robot projects and homework using the Arduino platform. The Arduino was designed to be a very inexpensive and easy to use prototyping platform for electronics projects. The Arduino can read values from sensors, drive servo motors, output signals, and communicate with other computers. The Arduino is programmed using one of two computer languages, C or C++. There are several other languages which are unofficially supported, this course will be focused on C++. C and C++ are among the most widely used computer languages for embedded computer applications. For a programming environment, this book and the class expect you to be using the Arduino IDE (Integrated Development Environment). This includes all of the necessary tools for programming any Arduino board, and features to make programming the Arduino easy. There are other environments that you can use, but we will not be covering them in this book or providing support for them in class.



An Arduino Uno Board

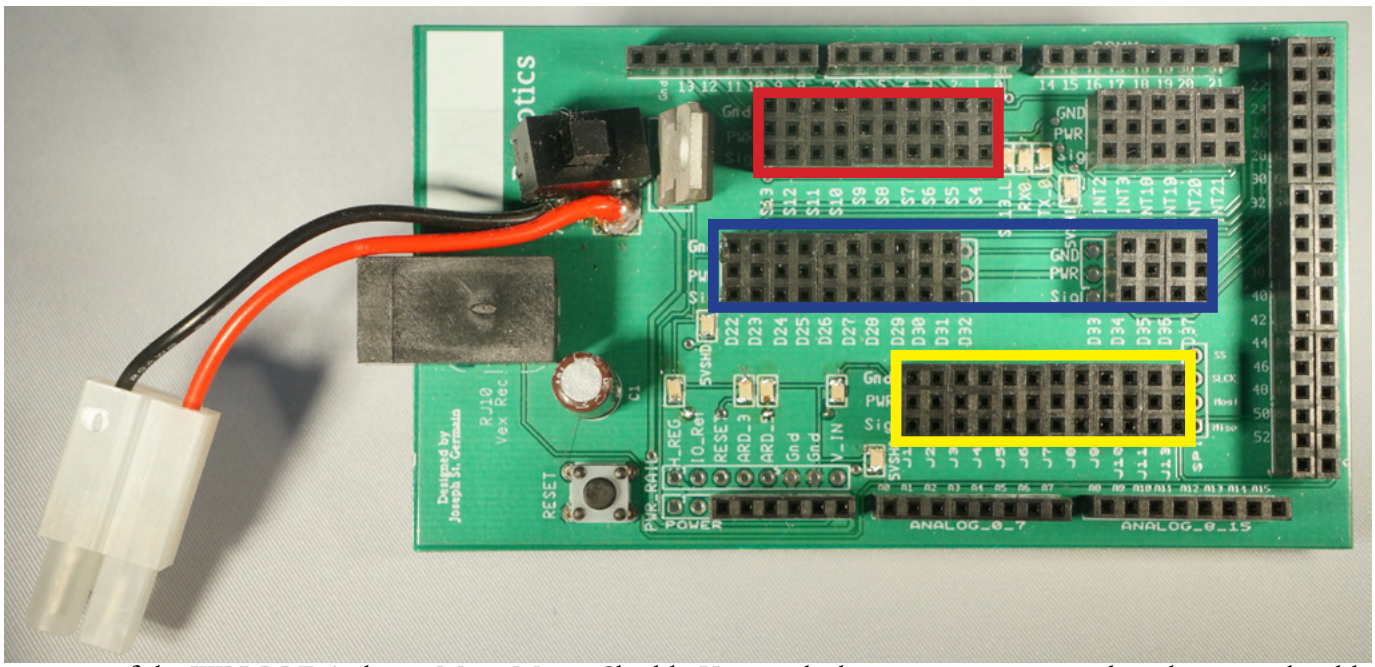


An Arduino Mega Board

You will be using one of two Arduinos for this class, either an Arduino Uno in your lab kit for doing homework and individual experiments or the Arduino Mega for your lab projects. The Mega is more expandable and is a better choice for building more complex robot projects, but both are programmed exactly the same way. In fact, the Mega was designed to be compatible with hardware and software designed for the Uno. Anything you learn on your Uno will apply to the Mega as well.

1.2: Getting started with your Arduino

Before going further you should read the Arduino introduction from the Arduino website. You'll find this website an invaluable resource for finding information for using your Arduino. The introduction is located here:
http://arduino.cc/en/Guide/Introduction
After reading the introduction, install the Arduino development tools on your personal computer so that you can follow along with exercises in this book as well as complete homework assignments. The Arduino development environment works on Linux, Mac OSX, or Windows and is very easy to install. The directions are here:
http://arduino.cc/en/Guide/HomePage
If you followed all the steps installing the Arduino software, you should have run the "Blink" example program to verify that the your software was correctly installed. If you had trouble, be sure you followed the steps to set the serial port and the Arduino board type correctly.
Arduinos are extended by adding on circuit boards called shields. Shields plug into a standard set of connectors on the Arduino board and provide additional electronics. This class provides a shield that makes it easy to wire Vex motors and sensors and to use an external power source with the Arduino. This shield plugs into the mega, and provides traditional 3-wire connection ports. If you did not have access to a shield like this, you would have to use a breadboard to wire power and signal to the correct ports on the Arduino. Hover over each section on the shield next to this text to see descriptions of each component, and what they're used for.



Revision 1 of the WPI RBE Arduino Mega Motor Shield. You might have a newer version, but the ports should still be in the same general location

1.3: Programming in C/C++

You'll be writing programs in C++ for this class. C++ is a superset of the C language and was developed after the C language to provide Object Oriented Programming (OOP) additions. You'll learn more about that later. You ran the Blink program in the previous section. Now read the tutorial about Arduino programs, called Sketches: http://arduino.cc/en/Tutorial/Sketch.

1.3.1: Variables

One of the most important concepts in programming are variables. A variable is essentially a value that can be changed while the program is being run. The value has a name and is associated with a spot in the Arduinos memory to hold the value. There are various types of variables:

C++ Name	Bytes of Storage Used	Range of Values
`char`	1	-128 to 127
`int`	2	-32,768 to 32,767
`long`	4	–2,147,483,648 to 2,147,483,647
`unsigned char`	1	0 to 255
`unsigned int`	2	0 to 65,535
`unsigned long`	4	0 to 4,294,967,295
`float`	4	Approx. 7 Decimal Digits
`double`	8	Approx. 16 Decimal Digits
`boolean`	1	true or false

Unless it is absolutely required you should avoid the use of floating point types (float and double) because operations on these types are much slower and the variables use more memory than most integer types.

To declare a variable, we start by declaring the type, and then the variable name. This is how you would declare a variable:

```
int intVar;
```

This declares a variable of type signed integer, which is abbreviated as int in C and C++, and name `intVar`. A signed integer can hold any value between -32,768 and 32,767. An unsigned integer is declared very similarly, and it can hold any integer value between 0 and 65,535.

```
unsigned int unsignedIntVar;
```

Once a variable has been declared, values of that variable's type can be stored in it using the assignment operator, which in C/C++ is the = sign. Here's an example of assigning 3 to `intVar`:

```
intVar = 3;
```

Now, `intVar` contains the value of 3. `intVar` can now be used in an expression instead of using a number. For example, you can say this:

```
unsignedIntVar = intVar + 3;
```

This adds 3 to the value currently in `intVar`, and then stores the resulting number, 6, into `unsignedIntVar`.

Another important concept that example showed is variable conversion. In this example, a variable of type signed integer was stored in a variable of type unsigned integer. This works as expected, as long as the value being stored is not a negative number. If a negative number, such as -6, was stored in `unsignedIntVar`, it would store the value as 65,530. This is because computers store negative numbers using a technique called 2's complement. Negative numbers are represented by taking the max value for an unsigned integer, 65,535, and adding the negative number + 1. So a -1 would be represented as 65,535, a -2 would be 65,354, and so on. Note that doing this conversion does not cause the program to throw any errors, but it could cause the program to behave incorrectly if this is not intended behaviour, so make sure to do type conversion carefully.

Another very common type is a floating point number. These are traditional numbers with a decimal point. For example, 3.14 is a floating point number, as is -6.28. Unlike integers, floating point numbers do not have unsigned versions: all floating point numbers can be positive or negative. However, there are multiple levels of precision for floating point numbers: a standard (single-precision) floating point number can represent about 6-7 digits, while a double-precision floating point number can represent about 16 digits. As you might expect, though, a double precision floating point number takes up double the space of a single-precision floating point number and uses more processing power, so if memory and/or computing power is an issue (which it can be on an Ardunio) make sure to use the minimum precision necessary.

Here's an example of declaring a single-precision floating point number, which has the type keyword float:

```
float singleFloat;
```

Just like `intVar`, you can now use `singleFloat` in any expression where you would normally use a floating point number, and assign a value to it with the assignment operator. To declare a double-precision floating point number, the double keyword is used:

```
double doubleFloat;
```

Most computer scientist don't want to say single-precision floating point number and double-precision floating point number all the time, so they generally refer to them as floats and doubles, respectively.
Floats and doubles can also be converted between each other. However, if a double with an 11 digits of precision is stored in a float, it will be truncated to only 7 digits of precision, meaning that the last 4 significant figures are lost. If you had a double that was 1234.5678901 and stored it in a float, then it would be truncated to 1234.567. There is, however, no loss of precision when going from float to double.
Additionally, it is possible to convert between integers and floating point numbers. When going from integer to float, no precision is lost. When going from a float to an int, however, everything to the right of the decimal point is truncated. So 3.14 and 3.94 would both be stored as 3. If you want correct rounding behaviour, so that 3.14 would be stored as 3 and 3.94 would be stored as 4, it must be done manually.

Question: how would you assign a floating point variable called floatValue to an integer variable called intValue and have intValue rounded properly?

In addition to `ints`, `unsigned ints`, `floats`, and `doubles`, there are also character and boolean types, whose keywords are `char` and `bool`. A `char` can represent a single character. A chraracter value can be assigned to a char variable by surrounding the character with single quotation marks, like this:

```
char charVariable;
charVariable = 'a';
```

Character variables are represented in computer memory as a single byte and can also store integer values that fall within a single byte range as described in the table above.

A boolean variable contains a boolean value, which is either `true` or `false`. In some languages such as C, there is no explicit boolean type, and `true` is stored as a non-zero value, usually the integer 1, and `false` is an integer 0. In C++, there are boolean types, so you can simply have a variable that has a value of `true` or `false`. However, the old C-style boolean still is valid, which is useful for evaluating if statements (these will be covered later in this chapter). Here is an example of a boolean variable.

```
bool booleanVar;
bool dark;
booleanVar = true;
dark = lightSensorValue < 40;
```

In the second case, an expression that returns a boolean value (less than in this case) can be used to set the variable. Boolean expressions will be covered in more detail later.

For all variables, you can assign a value on the same line as an the initialization, so that you can take these two lines of code:

```
int intVar;
intVar = 3;
```

and combine them to this one line:

```
int intVar = 3;
```

This works for all variable types. Additionally, you must make sure to assign a value to a variable before you attempt to use it, or you will get errors. Take this code block, where `intVar` is used before it is given a value:

```
int intVar;
float floatVar;

floatVar = intVar + 3;
```

In this case `intVar` was never assigned a value so its value is undefined (whatever was in memory when the statement is executed.) If the last variable to use that area of memory was the value 8, then `floatVar` will now equal 11. If it was 63, then `floatVar` will be 66. The C++ compiler will try its best to warn you of uninitialized variables but it can't always figure it out in a complex program.

By now, you will have noticed that we name our variables starting out with lowercase letters, and capitalizing each new word, like `intVar` or `singleFloat`. This is called camel case, and it is one of the naming conventions for variables in C and C++. If you had a variable for a sensor value, then it might be called `sensorValue`. Another common naming style that you might see is using underscored between words, such as `sensor_value`. In this book, we will use camel case, as it is the more common naming convention.

1.3.2: Setup and Loop

Recall from running the blink program that there are two main pieces of code that allow your Arduino programs to run. These are called the `setup()` and `loop()` functions. Functions will be covered more in depth later in this chapter, but for now a function is a self-contained piece of code that can be called from another place in your code.

The `setup()` function is called once at the beginning of your code. It's a good place to set up (thus the name `setup()`) motors, sensors, and anything else that you want to run only once on your robot. Here's an

example of a `setup()` function: it first creates a `Servo` (used for controlling motors, it will be covered in this chapter) variable, then inside the setup function sets up the motor variable to control the motor plugged into port 4.

```
Servo myServo;

setup() {
     myServo.attach(4);
}
```

The other main function in the Arduino code is the `loop()` function. This function will be executed repeatedly until the robot is turned off. This means that any code put into the loop function will loop (thus the name `loop()`) until the robot is either manually turned off or runs out of battery. Here's an example `loop()` function that will set the speed of the motor from the previous example to a value of 180, using the `Servo.write()` function:

```
loop() {
     myServo.write(180);
}
```

As long as the robot is on, it will continue to run the code that you put into `loop()`. An example of something slightly more complex is driving for 5 seconds, then stopping for 5 seconds, and repeating that code until the robot is turned off. This uses the `delay(x)` function, which causes the Arduino to stop executing your program for x number of milliseconds. The servo will continue running at the last speed it was set to until it is set to a new value, so the program sets it to a speed, then delays, then sets a different speed, and finally delays again.

```
loop() {
     myServo.write(180);
     delay(5000);
     myServo.write(90);
     delay(5000);
}
```

This will go full speed for 5 seconds then stop for 5 seconds, as long as the robot is on.

Sidebar: Servos and Motor Control

Motor control on the Arduino is done using the Servo library of functions. A servo is just a motor with a sensor on it, and the Servo.write(x) function will tell the servo to go to angle x. When used on a motor, that number represents the speed you want the motor to move at. This will be explained in much more detail later in the chapter.

1.3.3: How C and C++ programs really start

C and C++ programs don't normally have `setup()` and `loop()` functions. This is part of the Arduino toolkit to make it easier to get your programs working right away. If you write a C program anywhere else you would find that you are required to provide a function called `main()` that is started when the program starts. What is happening

behind the scenes in the Arduino is that the `main()` function is called on startup. It in turn does the following:

```
main() {
     setup();  // call your setup function
     while (1) {
          loop();   // call the loop function repeatedly
     }
}
```

The `setup()` function is called once at the start of `main()`, then there is a C while loop (a loop that runs repeatedly while the given condition is true) that repeatedly calls your `loop()` function. This all happens behind the scenes, but makes it easier for you to construct your Arduino programs. The condition for this while loop is `while(1)`. Recall from the previous discussion on booleans that a 1 is true. Therefore, the program never terminates. This is called an infinite loop, and the `loop()` function will continue getting called until you turn the power off, so keep that in mind when writing your programs.

1.3.4: Comments

One important thing to always include in your programs, in any language, is comments. These allow you write in plain English how your program works, for reference later by yourself or another person. It is especially important when working in teams so that other people can see what you're doing with a complicated block of code. In C and C++, line comments are started with a double forward slash, like this:

```
// This is a comment.  Anything written on this line will not
// be read by the rest of your program

Servo myservo;  // You can also start comments after you've
                // written a line of code

// Anything on the right-hand side of
// the double forward slash is ignored
```

Additionally, you might want to write out several lines of description for a function. Instead of putting a double slash in front of every line, you can surround your comment with these symbols:

```
/* This is a block comment.
Even though this line doesn't have a // in front of it,
it is still ignored by your code.
A block comment continues on until it finds the ending symbol,
which is this: */
Servo myServo;
myServo.attach(4);
```

The Arduino IDE will color code different elements of your program to help you see what you are writing. Comment text is shown in gray to indicate that it is being ignored by the compiler.

1.4: Using Motors and Servos with the Arduino

One of the most basic parts of a robot are actuators that are often implemented with motors or servos and can be controlled through an Arduino. The motors that will be used in class and in labs run on electricity; current in one direction causes the motor to turn clockwise, and current in the opposite direction causes it to turn counterclockwise. The direction is dependent on the motor. You'll go over the specifics of how motors work in class.

There are two problems with controlling a motor with an Arduino:

1. Motors usually require higher current than the microprocessor on the Arduino can supply
2. The connections to the motor have to reverse polarity in order for the motor to run in the opposite direction.

To solve these problems an external device is needed that can supply the higher current needed and convert the digital signals from the microprocessor (either 0 or 5v) to a speed and direction.

1.4.1: Electronic Speed Controllers

The circuit that does this is called an electronic speed controller or ESC. The ESC does two things: it takes a signal that is compatible with the Arduino outputs and generates a voltage that is capable of reversing polarity; and it uses separate power from that signal to be able to supply the necessary current to drive the motor. This second point is important, because when using larger motors the controller (in this class the Arduino is the controller, but there are many different controllers) cannot safely supply enough power to run a motor, so the ESC allows you to have a separate power source and still control the motor. Here is an example of a speed controller used with a high powered motor in a larger application. The motor that this ESC is controlling is far more powerful than the motors that you are using on your robots, and it draws far more current than the Arduino can supply. You can see the signal wire going into the ESC; this powers the ESC and tells it how to run the motor. The thicker power wires going into and out of the ESC supply the power to the motor, and the ESC regulates how much power the motor gets.

With most Vex motors, you don't have to worry about putting separate speed controllers on the robot; an ESC is already integrated right into the motor housing. If you were to disassemble a vex motor (don't try this, you'll probably break the motor) you would be able to see a small ESC. Larger motors need larger ESCs, as they use more current. Some of the newer, high powered Vex motors use a separate ESC such as any of the two-wire Vex motors, including the high-power 393 motors found in the lab.

1.4.2: PWM Signals

As previously mentioned, an ESC is controlled using a digital signal. The type of signal used is called Pulse-Width Modulation, or PWM. An excellent article on PWM signals can be found here: http://www.jameco.com/Jameco/workshop/howitworks/how-servo-motors-work.html?sp_rid=MjI4MTA0MzM5ODIS1&sp_mid=4334065. It looks at PWM signals relating to another type of motor, called a servo. We'll be taking a look at servos a little later in this chapter.

At the most basic level, a PWM signal is pulsed DC electricity. The width of this pulse tells the ESC how fast to turn the motor. The period, or length of time between pulses, is 20 milliseconds. That means that the fastest that you can change a motor's speed is once every 20 milliseconds, or 50 times every second. A PWM pulse for the Vex motors in class is about 2 milliseconds long to set the motor to go full forwards, and is 1 millisecond long to the motor to go full backwards power. Put another way, when the PWM signal is 1 millisecond long, then the Vex motor moves counterclockwise. At 2 milliseconds long, the motor moves clockwise. At 1.5 milliseconds, the motor stops. Take a look at this PWM animation: http://rbe1001.wpi.edu/Chapter1/index.html. Try moving the slider, and watch what happens to waveform as you do so. See how the motor moves as the waveform changes? That is how the PWM waveform looks if you were to view a PWM signal with an oscilloscope.

Another important thing to note is that despite the fact that the PWM signal period is 20 milliseconds, the motor is only being controlled for 2 milliseconds of that period. So what happens to that other 18 milliseconds? We can do something called multiplexing signals with that other 18 milliseconds. In a nutshell, it means controlling more motors off that one signal. So the first few milliseconds are for ESC 1, the next few are for ESC 2, and so on. This is used with the joystick controller, which multiplexes all of the different channels of control on one signal.

The Arduino Uno can control up to 12 motors; the Mega can control up to 48. For more information on controlling motors on the Arduino, take a look at this page: http://www.arduino.cc/en/Reference/Servo. Addi-

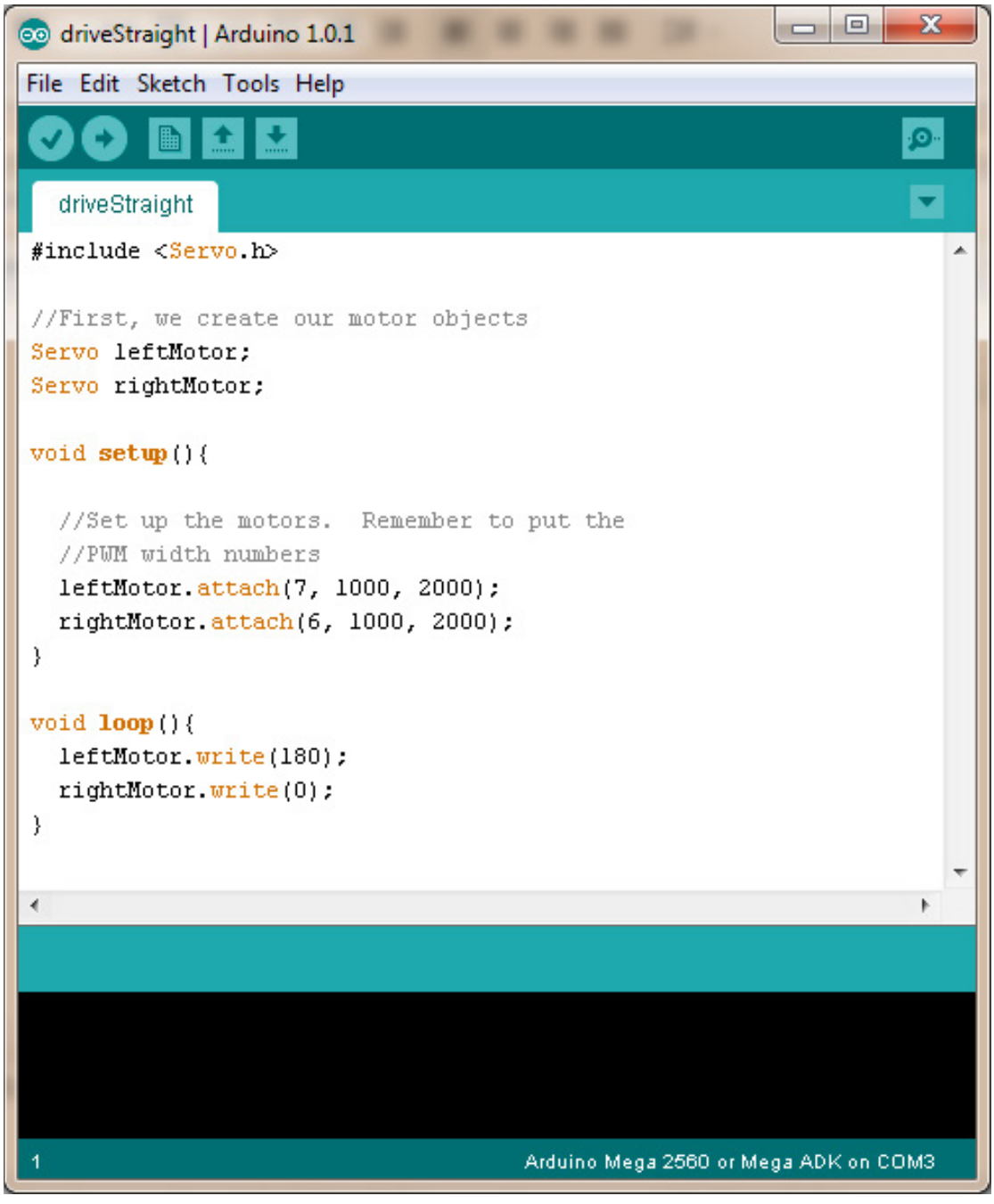
tionally, do some Google searches; there are lots of people with Arduinos making amazing programs, and there's a lot of good documentation available.

1.4.3: Controlling Motors with an Arduino

In order to control a motor with an Arduino, you'll need to write some code. The process to control a motor with the Arduino has been shown, but not explained. Now it can be explained.

Take a look at the following program:

The first thing that you see is `#include <Servo.h>`. This tells the program to look for function definitions in a file called Servo.h, which the Arduino IDE knows how to find. This file defines the functions necessary to run motors. The Arduino treats motors as servos: there is no distinction between the two. Next, outside of any function, two Servo variables are created for the left and right motors. Anytime that control of the left motor is desired, you can refer to the `leftMotor` variable. Now look at the setup function and recall that this is the first thing executed when the program starts. Here, the motors are set up using the `Servo.attach(pin, min, max)` function. Calling `leftMotor.attach(7, 1000, 2000)` tells the `leftMotor` variable that the mo-



tor to control is located on port number 7. The 1000 and 2000 are the PWM pulse widths that we talked about in the previous section. The min and max on Vex motors are 1000 microseconds and 2000 microseconds, respectively. If we don't include them, your program could work, but you will either have deadbands at the end of the range where you reach max speed before 0 and 180, or you will not be able to reach full speed.

Next look at the `loop()` function. Here, `leftMotor.write(180)` and `rightMotor.write(0)` are continuously called. The `Servo.write(number)` function causes the motor to operate where 180 is full clockwise and 0 is full counterclockwise. 90 is stopped, and values in-between are varying speeds of clockwise and counterclockwise. The reason 0 and 180 are chosen for the limits has to do with the operation of servos, but this will be covered later in this chapter. In the example `loop()` function the left motor will drive full clockwise and the right motor will drive full counterclockwise until you turn off your robot. Remember, the motors on your basebot face in opposite directions, so this causes the robot drive straight until you turn it off. This sample program, along with several others that will be shown later, can be found on myWPI.

1.4.4: Servos

Servos have been brought up a lot, and now its time to talk about what they are and how they work. A servo is a motor with built-in electronics and a sensor that allow it to be commanded by the Arduino to move to specific shaft angles and stop. This makes the servo very useful for actions that require you to precisely position a moving object, such as moving flaps on radio controlled airplanes, or positioning CNC tools. Servos don't require the Arduino program to read the sensor and drive the motor, it simply tells it to move to the desired angle. Servos have a limitation, however: they have constrained range of motion. Make sure to check the specifications to be sure that servos range of motion matches your application.

This is the reason that the servo library use values between 0 to 180. Typical hobby servos have 180 degrees of movement. Writing a 0 will tell these servos to go to 0 degrees, 180 will tell it to move to 180 degrees and 90 will tell it to move to 90 degrees. Writing anything in-between will move the servo to that location. The Vex servos only have 100 degrees of movement so the value will be scaled, so writing a 180 will move the servo to 100 degrees and writing a 90 will move it to 50 degrees.

1.5: Program Control

Now you have all the tools needed to make a robot drive straight or turn and to set a servo to a desired angle. However, programs that have been shown to this point have been very linear: they move from one thing to the next, and cannot dynamically change based on a variable. To take of this, it is time to introduce if statements, which are a method of program control.

1.5.1: If Statements

An if statement allows programmers to ask question, and make a decision based on the answer. A sample if statement looks like this:

```
//First, we ask the question
if (sensorValue > 10) {
     // If the sensor value is greater than 10,
     // run the code in here
} else {
     // If it is not greater than 10,
     // run the code in here
}
```

In an if statement, there is the if keyword, followed by an expression surrounded by parenthesis. This expression is a boolean expression: it evaluates to either true or false. If the value is true, then the content of the if statement is executed. If the value is not true, it moves to the else statement, and executes the code in there. The program will only execute one of these statements, not both.

The expression in an if statement often involves a relational operator. These operators are: `==`, `!=`, `>`, `<`, `>=`, `<=` and return a boolean value. These stand for equals, not equals, greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to, respectively. They have an operator on both sides. In our example, we have `sensorValue > 10`. This statement is saying return true if the value of the variable sensorValue is greater than 10, and false if not.

Note that == is different from =. The = operator is the assignment operator: it sets the value of the variable on the left side of operator to the expression on the right side. The == operator evaluates whether the left side operand is equal to the right side operand. If they are, it returns true. If not, it returns false. Take a look at this code:

```
if (variable = 1) {
     // Do something
}
```

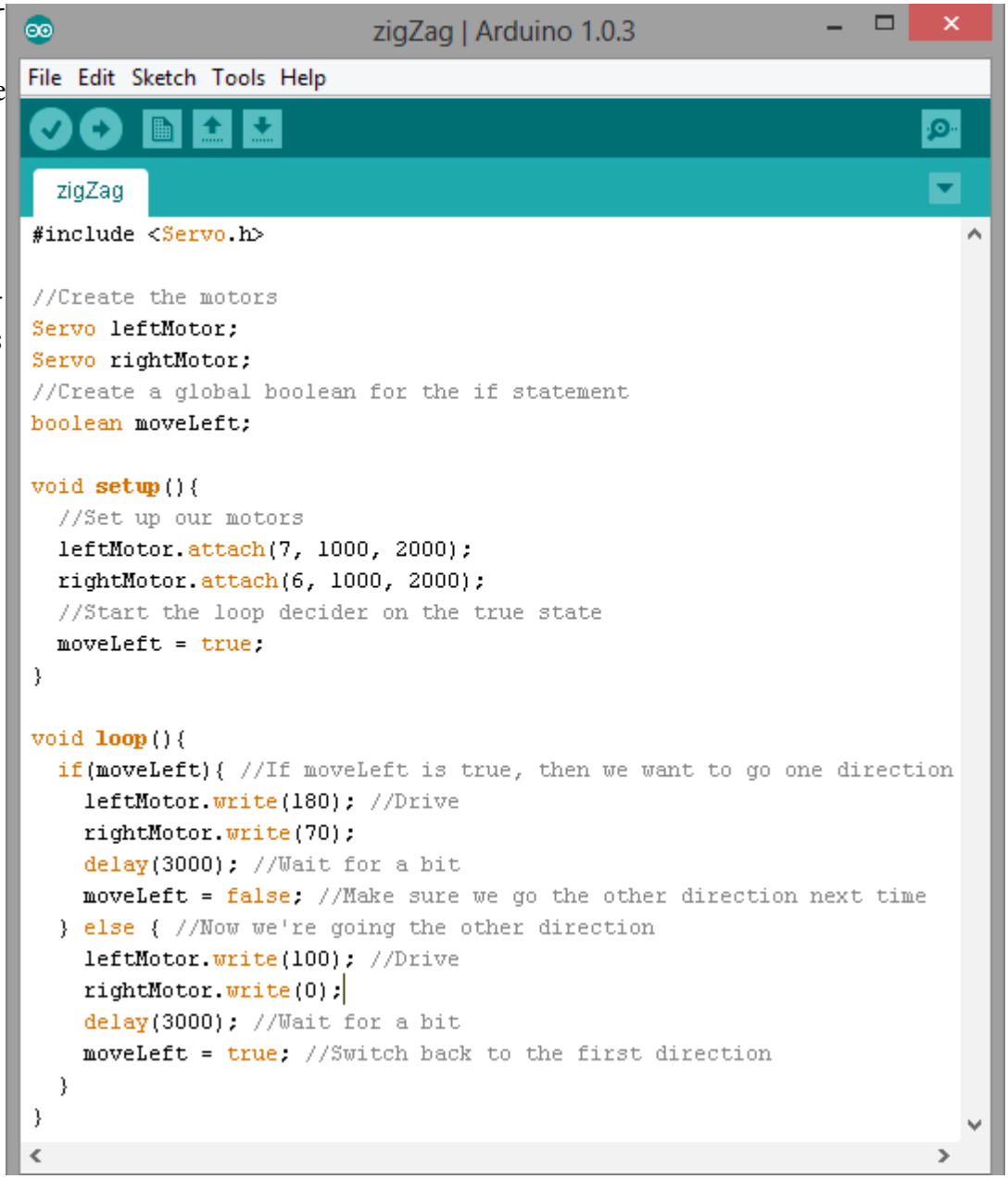
This is a common programming error where the intent was probably to check to see if variable is equal to 1. However, the = operator was used, so regardless of what the value of the variable was before the if statement, it is set to 1. The expression will always evaluate to true, because variable is now a non-zero number, so the content of the if statement is executed. This is not the desired behaviour. To fix this, be sure to use the correct operator. Here is the fixed if statement:

```
if (variable == 1) {
     // Do something
}
```

Now, the code will execute correctly.

Take a look at this sample program called `ZigZag`. This program will cause the robot to move one way for 3 seconds, then a different way for 3 seconds. It will continue this behaviour until the robot is shut off.

First, the left and right motor variables are created. Next, a variable called `moveLeft` is created, which will control the if statement the program takes. This is a boolean variable, which means that it will either contain true or false. Then, in `setup()`, the servo motors are attached to the correct ports. `moveLeft` is also initialized to be the value true: this will determine which direction the robot starts driving in. It could start out as false as well; this decision is up to the programmer. Finally, the `loop()` function is entered. The if statement here is `if(moveLeft)`. This works because `moveLeft` is a boolean value, and that's all that an if statement needs. If `moveLeft`



is true, then we set the motors and delay for 3 seconds. Very important to note: the `delay()` function uses time in milliseconds, not seconds. If you specify `delay(3)`, it will wait for less time than it takes you to react to anything. Thus, 3000 milliseconds, or 3 seconds, is used. The final thing that is done is to set `moveLeft` to be false. This means that in the next loop, the program will execute the else statement, not the if statement. The `else` statement does almost exactly the same thing, except going in a different direction and setting `moveLeft` to be true.

1.5.2: Else If

What happens if your question is not a simple yes or no question with two outcomes? What if you want to zig, zag, drive straight, and repeat? To do this you can connect if statements together as shown:

```
// Declare a variable
int variable = 1;

// Is the variable 0?
if (variable == 0) {
     // Code here
}
// How about 1?
else if (variable == 1) {
     // Other code here
}
// 2?
else if (variable == 2) {
     // Still more code here
}
// This could keep going ad infinitum, but skip to the else case
else {
     // Final code here
}
```

In this case each if statement executes only if all the previous if statements evaluated to false. And an important thing to note is that the program will stop evaluating the expressions once one of them evaluates to true. For example, you could have:

```
if (variable == 1) {
     // Code
}
else if variable > 0 {
     // Other code
}
```

In this case “variable” is both equal to one and greater than zero, however, only the first statement will be executed.
Consider this example:

```
if (variable == 1) {
     // Code
}
if (variable > 0) {
     // Other code
}
```

In this case the two if statements are independent of each other and both pieces of code will execute.

1.5.3: Logical Operators

It’s very common to want to have a multi-part question. You could have input from two sensors, and want to drive forward when both sensor values are true. One obvious solution would be to use a nested if statement, such as:

```
if (variable1 == 1) {
     if (variable2 == 2) {
          // Code
     }
}
```

In this case the code will be executed only if both variable1 is equal to 1 and variable2 is equal to 2. However, this is bulky and inefficient. To solve this, there are logical operators, which take in two boolean expressions and return a boolean output based on these inputs. The logical operations are && (for AND), || (for OR), ! (for NOT), and ^ (for Exclusive OR) often abbreviated as XOR. AND will return true if both inputs are true, and false otherwise. So, true && true will return true, and 1 && 1 will also return true (remember that numeric values that are nonzero are considered true). OR will return true if either of it’s inputs are true, and false otherwise. If you had the statement true || false, it would return true. 0 || 1 would also return true. NOT will return the opposite of what it is given, so true if the input is false and false if the input is true. !true would return false, and !0 would return true. XOR will return true if one, but not both, of its inputs is true, and false otherwise. So false ^ true would return true, but 1 ^ 1 would return false. Here is a truth table for AND, OR, and XOR. A truth table has two columns for input 1 and input 2, and columns for what the logical operators would produce if given these inputs. T represents true, and F represents false.

Input 1	Input 2	&&: Output	||: Output	^: Output
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

Let's revisit the nested if statement from the start of this section. While some nested if statements do have a purpose, the one that was shown earlier can be more clearly written making use of logical operators:

```
if ((variable1 == 1) && (variable2 == 2))  {
     // Code
}
```

This will execute the code in the if statement if and only if both conditions are true. Another important thing to note about boolean operators is that expressions are evaluated from left to right, and only evaluate as far as needed. For example in the above example, if variable1 was not 1, then the program will not evaluate the second part of the expression, as there is no situation in which AND will return anything but false. Similarly, if OR had been used, then it would stop evaluating if variable1 was 1, as there isn't a situation in which OR will not return true.

For a short, but very good explanation of some deeper nuances of logical operators, take a look at chapter 2.6 of the C Programming Language, which can be found here: http://net.pku.edu.cn/~course/cs101/2008/resource/The_C_Programming_Language.pdf. There are a couple mentions of getChar() and EOF: these have to do with input and output in C. If you want to read more about them you are welcome to, but they do not pertain to the content in this book and you do not have to understand them. Simply understand that EOF is a value, and getChar() is a function built into the C programming language.

You can also use multiple boolean operators in order to test three or more conditions in one statement. In fact, it doesn't matter how many operators you have in a single expression, so long as the statements are still legal. (ie, variable1 && && variable2 variable3 is not legal, but variable1 && variable2 && variable3 is). There are many ways you can combine boolean logic: this is called boolean algebra and is an important topic for those who will be taking more advanced Computer Science courses, but it is not relevant to this book.

1.6: Functions

1.6.1: What are Functions?

The example programs up to this point have been pretty simple, but as programs get larger and more complex they can become disorganized and have a lot of repeated code. To allow for better organization of programs and make it possible to reuse code within a program, just about every programming language has functions: a named block of code that you can call, just like you would call delay() or Servo.write(). The idea is that any block of code that you might reuse over and over in a single program or between programs can be made into a function. All functions have follow a certain pattern, and it's important to use this basic format when writing your own functions.

1.6.2: Creating Functions

The basic structure of a function is as follows:

```
return-type function-name (input-type input-name) {
     // Your code goes here
}
```

The first thing that we specify is the type of value the function returns. In C and C++ there are several basic types that can be returned. You can return any of the basic types that we covered previously, including `ints`, `unsigned ints`, `floats`, `doubles`, `chars`, and `bools`. Some functions don't return values and for these the return type is written as `void`. `setup()` and `loop()` are both void functions, and they don't return values.

The next thing that is specified is the name of the function. This is what is called when you want to execute this function. For example, the `delay()` function's name is delay, and it is specified in the function-name space in a separate file. In C, every function must have a unique name. If we have two functions that perform the same task, but one takes slightly different inputs than the other, then they must have different names. For example, consider functions that add numbers together. One might be able to two numbers, another one could add 3 numbers together. In C, they must have unique names, so one could be `addTwo`, and the other could be called `addThree`. C++ has function overloading; this allows us to have multiple functions with the same name. For example, when setting up a `Servo`, you call `Servo.attach()`. You can give this function two different sets of parameters: you only specify the port

number, such as `Servo.attach(1)`, or you can also specify the PWM lengths, with `Servo.attach(1, 1000, 2000)`. All overloaded functions must have different sets of parameters, so that the compiler knows which one should be used.

Next, function inputs are specified. Each input has a type, which can be any of the legal C/C++ types, and a name, which can be any non-reserved name. A non-reserved name means that you can't declare a variable with a name such as int, because int is a reserved word in C/C++, in this case a type. Additionally, you can have as many variables as you want, from none to as many as your application requires. However, since the point of functions is to simplify code and break it up into readable chunks, you generally won't see or create functions with more than 5-6 parameters. Exceptions occur, of course, but if your function needs 20 inputs, you may want to see if you can rethink how your code works.

The final step in a function is to return the type you specified at the beginning of the function. If you specified that you are going to return an integer, then the last thing that you do must be to return an integer value. You return a value by using the return statement. When the program encounters the return statement, your function ends, and that value is passed back to the calling statement. If your function is of type void, then you don't need a return statement, as you aren't returning anything. But you can write a return statement with no value to return from the middle of the function, say as part of an if statement.

Here's an example of a function that will drive two motors, named `leftMotor` and `rightMotor`, at speeds specified by the inputs to the function.

```
void driveMotors(int leftSpeed, int rightSpeed) {
     // Drive the left and right motors
     // based on the input speed
     leftMotor.write(leftSpeed);
     rightMotor.write(rightSpeed);
}
```

So this function's name is `driveMotors`. It doesn't return a value (so we write `void` for the return type) and takes two integer parameters, corresponding to left speed and right speed. If you call this function with a statement like:

```
driveMotors(0, 180);
```

The parameter `leftSpeed` will be assigned a value of 0 and `rightSpeed` will be assigned a value of 180 throughout the function.

Notice how this can make programs more readable by not duplicating the `Servo.write()` methods every time you want to drive the robot.

1.6.3: Commenting Functions

It is very important to document how functions that you create work, so that you and other programmers are never confused about how it works. For large projects, you will not be working alone on code and comments let your team members understand the code that you wrote. Take a look at these two functions:

```
void function1() {
     leftMotor.write(90);
     rightMotor.write(90);
}

bool function2(int x, int y) {
     if (x + y <= 360 && x + y >= 0) {
          leftMotor.write(x);
          rightMotor.write(y);
          return true;
     } else {
          return false;
     }
}
```

It's not easy to immediately see what these functions are doing: the function names aren't descriptive, and there are no comments to tell you what each program does. The actual description of the first program is that it stops both the left and the right motors. The second program drives the motors at a given speed, but only if the sum of the speeds is between 0 and 360. If the speed was valid, it return true. If the speed is invalid, it returns false.

These descriptions should be written as comments. In fact, the whole program should be modified to make it more readable. Here are the same programs, but modified to make them easy to quickly understand what is happening

```
/* stopDriving:
 * This function stops the left and right motors.
 * It has no inputs and returns void
 */
void stopDriving() {
     leftMotor.write(90);  // Stop the left motor
     rightMotor.write(90);  // Stop the right motor
}
```

```
/* driveAtSpeed:
 * This function takes in two variables, leftSpeed and rightSpeed.
 *  It drives the left motor at leftSpeed and drives the right motor
 * at rightSpeed, but only if the sum of these speeds is between 0
 * and 360.
 * If the speeds are valid, then the function returns true.
 *  If the speeds are invalid, it returns false.
 */
bool driveAtSpeed(int leftSpeed, int rightSpeed) {
     // First, check to see if left and right speed
     // are between 0 and 360
     // This is the formatting you will see if the line is too long
     if (leftSpeed + rightSpeed <= 360 &&
               leftSpeed + rightSpeed >= 0) {
          // They are, so set the motor speeds and return true
          leftMotor.write(leftSpeed);
          rightMotor.write(rightSpeed);
          return true;
     } else {
          // They are not, so return false
          return false;
     }
}
```

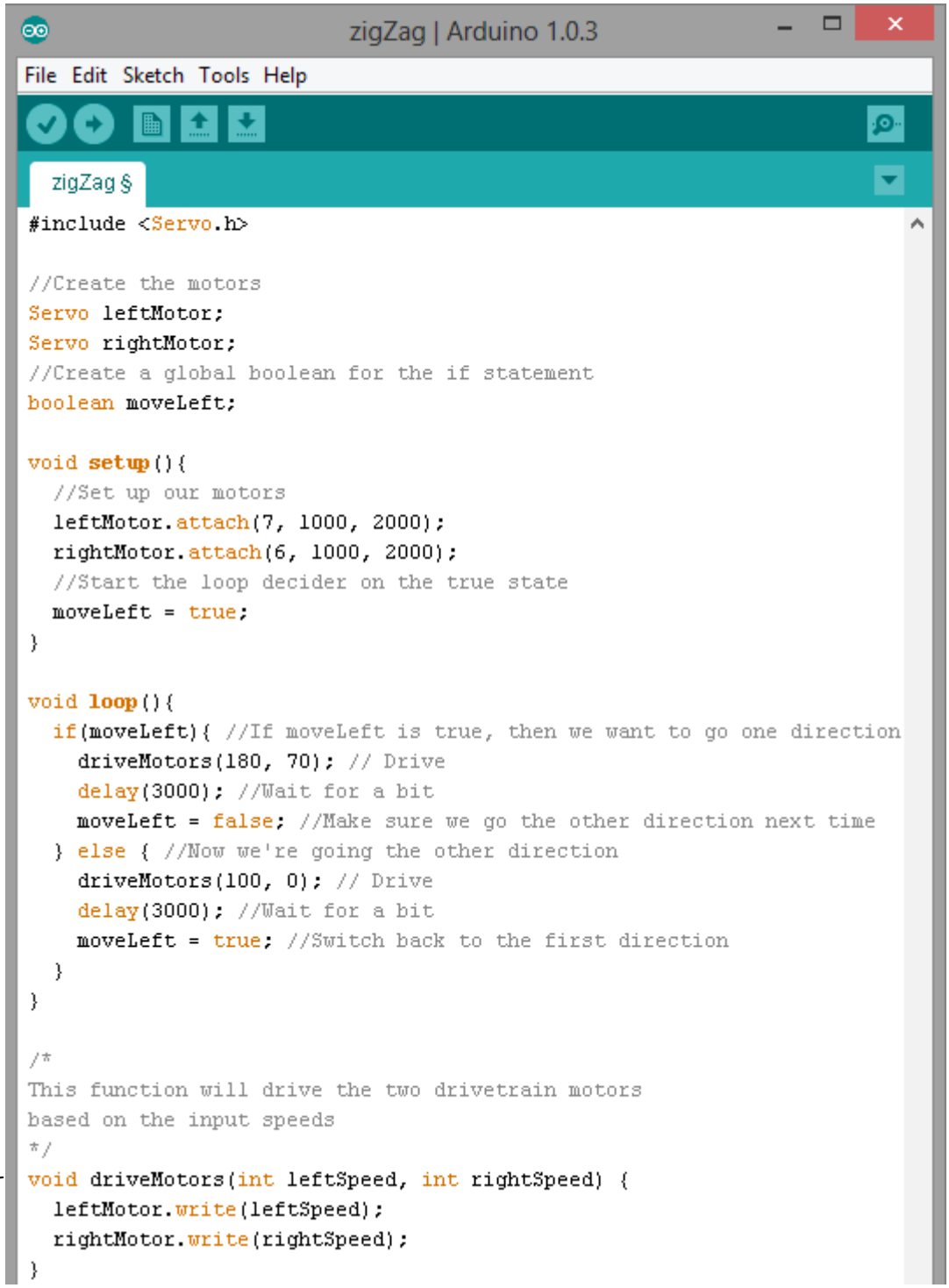
You'll notice a couple of things. First, in the comment above each function, the name of the function is called. It describes what the function takes for inputs, what the function does, and what it returns for outputs. In the body of the function, there are comments for every step, so that it's easy to look at any part of the function and quickly understand what the function is supposed to be doing at that step. These comments are extremely important, and they are required for all code questions that you get as part of this class.

1.6.4: Working with Functions

Functions will allow us to simplify the `ZigZag` program significantly. The loop function currently has a lot of repeated code that is not easy to read, but it can be abstracted out into separate functions that make it clean, and allow for code reuse. To start, replace the mirrored calls to the `leftMotor.write()` and `rightMotor.write()` with the `driveMotors` function, which has been shown previously. Here's what the program looks like now.

As you can see, all the `leftMotor.write()` and `rightMotor.write()` calls have been replaced with a call to `driveMotors()`. In this small function, the number of lines of code has actually increased, but the function is now easier to understand at first glance.

The next piece of repeated code across both cases of the if statement are the calls to delay, and changing `moveLeft` to be the opposite of what it is going into the loop. These can also be reduced by creating a function, this time called waitInvert. However, it is necessary to take a break from the `ZigZag` function for a minute, to talk about an important topic when using functions: variable scope.



1.6.4: Variable Scope

Take a look at this page: http://www.tutorialspoint.com/cplusplus/cpp_variable_scope.htm. It has a good introduction to variable scope. It is assumed that you have read this page going forward.

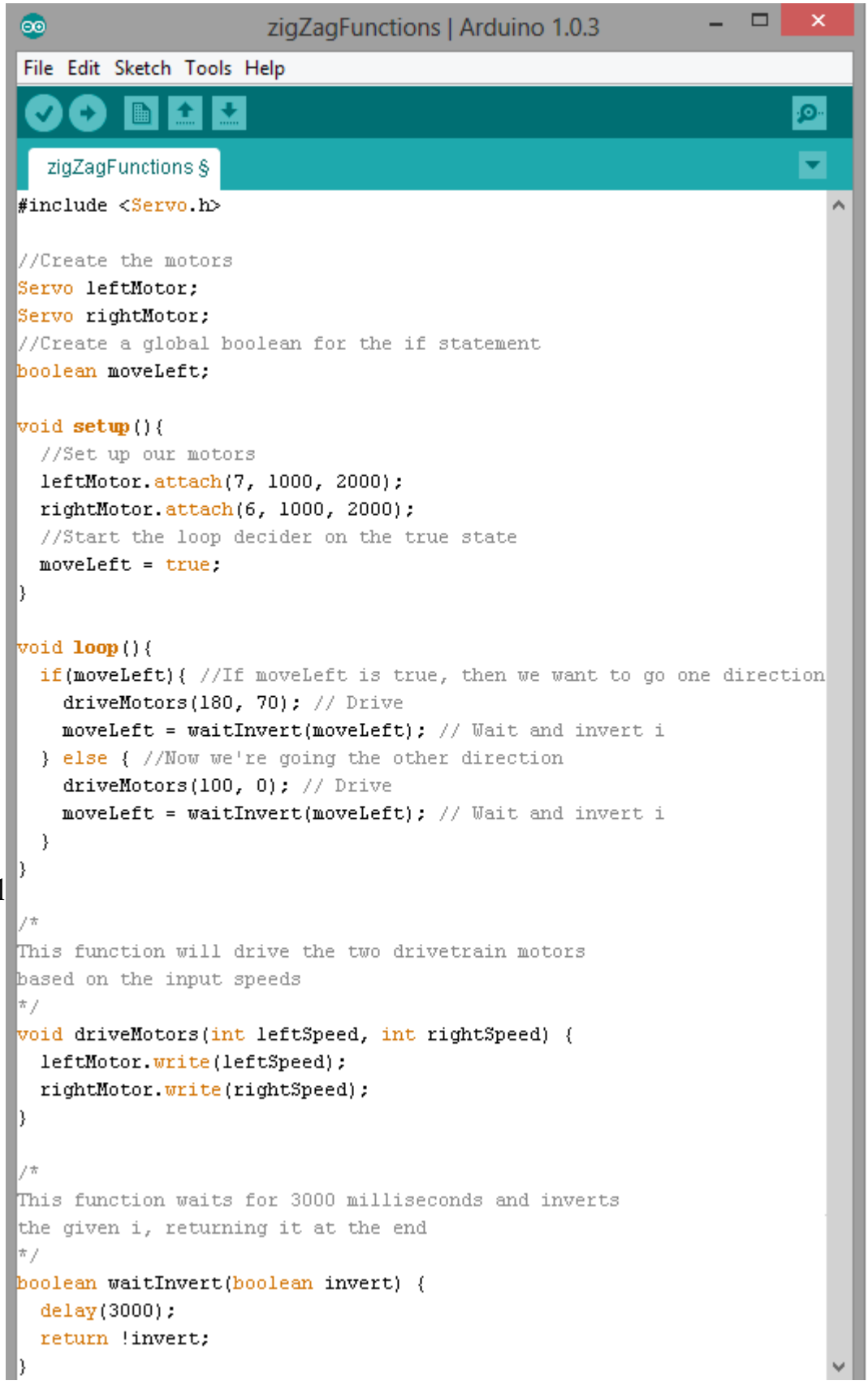
Variable scope is very important when looking at programming. Let's look at the `ZigZag` example again. `moveLeft` is a global variable, just like `leftMotor` and rightMotor. That means that it can be accessed from `waitInvert`, just like `driveMotors` can access `leftMotor` and `rightMotor`. However, `waitInvert` is going to treat `moveLeft` like a

local variable here in order to show you how to deal with local variables that need to be modified by another function. The code for the `waitInvert()` function looks like this.

```
boolean waitInvert(boolean invert) {
     delay(3000); // Delay the program for 3 seconds
     // Return the inverted input
     return !invert;
}
```

As you can see, `waitInvert` takes in a boolean value, and returns a boolean value. This is because it is treating `moveLeft` as a local variable. When `waitInvert(moveLeft)` is called, `moveLeft` is being "passed by value". Passing a variable by value means that the called function gets a copy of the variable, not the actual variable itself. This means that any changes the called function makes to that variable do not affect the original version of the variable. Take a handout from the teacher in class. The professor makes a photocopy of the original handout and gives it to the members of the class. Any notes that students in the class write on the handout don't magically show up on the original, because it is a copy. This is the same concept with passing by value.

Here is the final `ZigZag` function, with both



`waitInvert()` and `driveMotors()` implemented.

1.7: Wrap-Up

In this chapter, you learned about the basic structure of the Arduino, and the breakout boards that you will be using in the labs. You saw the basics of programming in C++, how to incorporate motors into your program, and how to break up your program in different functions. In the next chapter, you will be learning about basic analog and digital input and output, and how to use them in your programs.