Chapter 5: Basic IO

You now know the basics of how to get an Arduino-based robot to move. In this chapter you will look at how a robot can react to changes in its surroundings. One of the most basic ways for a robot to do this is through sensors. Sensors allow the robot to understand something about the world around it. For example, it can use a gyro to tell the robots current heading, an encoder to tell how far the robot has moved, a potentiometer to tell the position of an arm joint, an ultrasonic ragefinder to tell how far the robot is from a wall, and even more. Of course, your robot doesn't just know how to use these sensors; you have write code into your program to work with these sensors and take actions based on what they tell you. The first thing to learn about is the basic differences between different signal types.

5.1: Signal Types

Sensors communicate to the Arduino through use of input signals. There are two basic types of signals: digital and analog. A digital signal is very simple: it's either on or off and is represented by a value of 0-5V. An analog value is represented as a voltage that can have any value between 0 and 5V. In theory analog signals can have an infinite number of values but due to the limitations of the circuits they are limited to discrete values in the 0-5V range.

Signal Type	**How It's Represented**	**Typical Uses**
Digital Input	An input voltage that is either 0V (off) or 5V (on)	Devices that return on-off type signals such as limit switches (on or off), counters, encoders, etc.
Digital Output	An output voltage that is either 0V (off) or 5V (on)	Telling devices to do something such as LED indicators, actuators, relays, etc.
Analog Input	An input voltage in the range of 0V - 5V	Devices that return a range of values such as potentiometers, analog gyros, pressure sensors, etc.
Analog Output	An output voltage in the range of 0V - 5V	Not applicable to the Arduino without using additional electronics.
PWM Output	A pulsed digital signal where the pulse width represents a numeric value	Used for motor controllers, servos, etc.

A simple digital input sensor is a limit switch, which has two states: pressed or not pressed. More complicated sensors can use digital signals to communicate more than one value, such as an encoder that creates a series of pulses as a wheel turns.

Examples of sensors that have analog outputs are position sensors, such as gyros or potentiometers. For a gyro, the voltage corresponds to the current rate of rotation. For a potentiometer, the voltage corresponds to the current angle of the potentiometer shaft.

The Arduino requires extra electronics to supply analog outputs, even though it has an `analogWrite()` function.

5.2: Reading Digital Signals

Arduino digital ports can either act as digital inputs or digital outputs. When reading a digital signal on an Arduino port, you first need to set the port mode to input. To do this, the Arduino uses the function `pinMode()`, which looks like this:

```
// Set the mode of pin 1, which is a digital port
void setup(){
     pinMode(1, INPUT);
}
```

The Arduino has 3 modes that any digital pin can be set to: `OUTPUT`, `INPUT`, and `INPUT_PULLUP`. In the example above, we set pin 1 to the input mode. In input mode, the value read is 0 if the sensor pin is at 0V (`LOW`), and 1 if the sensor pin is at 5V (`HIGH`). We can also set the pin to `INPUT_PULLUP` mode which is the same as `INPUT` but the Arduino processor supplies pull-up resistors to make it work better with switches. You need to check to make sure that you are using the right mode for your sensor, or you may get unreliable input from it.

After the digital pin has been set to `INPUT` mode, it is possible to read in values from that pin. The function for reading a value from a digital pin is the `digitalRead()` function, which is used like this:

```
void loop() {
     digitalRead(1);
}
```

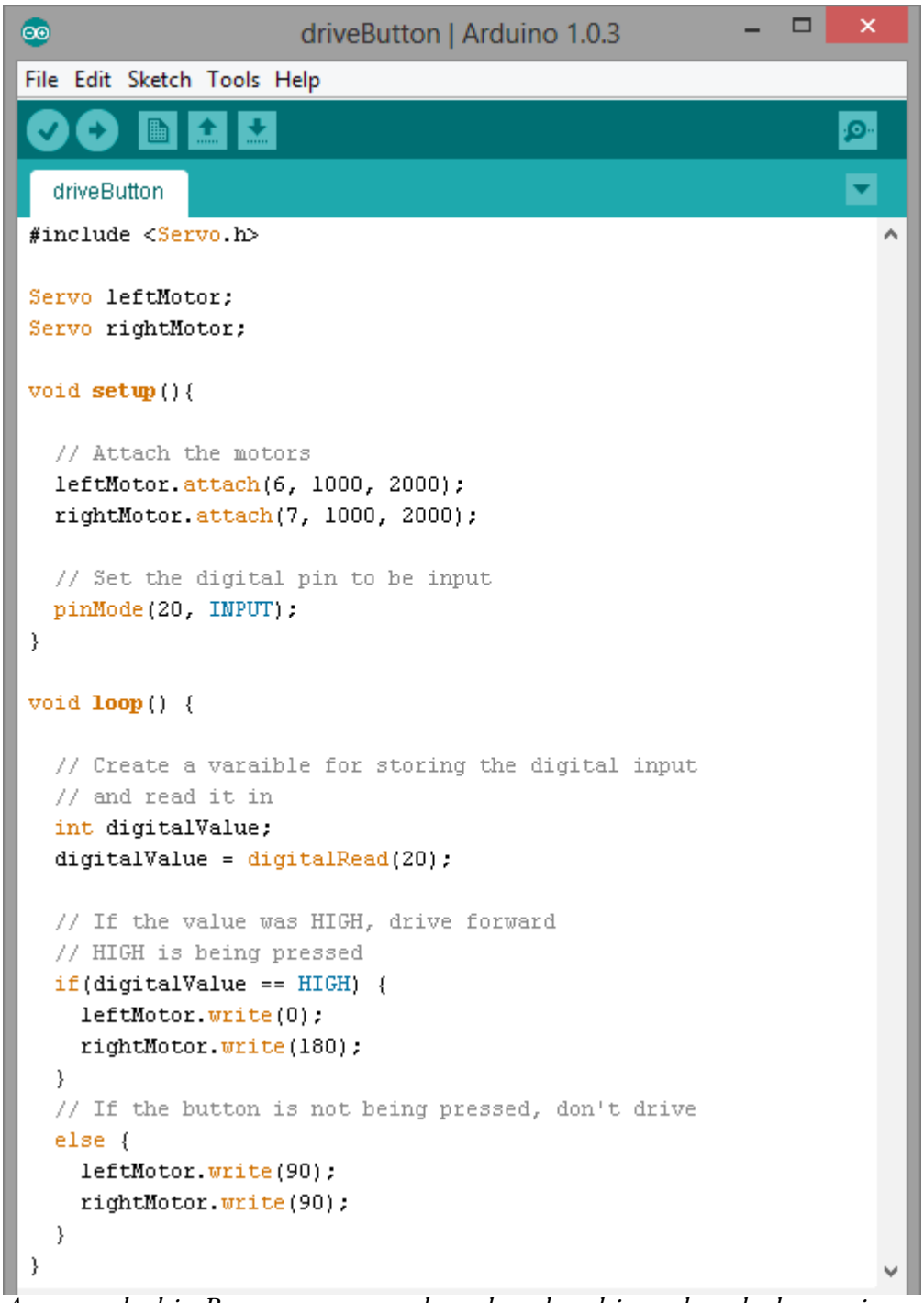
This calls the `digitalRead()` function, which takes an integer representing the port number that the sensor is connected to. It returns `LOW` or `HIGH` depending on the value of the sensor.

Now that you know how to read in a value from a port, it would be helpful to know how to do something with that value. The first step is to save the input value. You can do this by creating an integer variable and setting it equal to the result of `digitalRead()`, like this:

```
void loop() {
     int digitalValue;
     digitalValue = digitalRead(1);
}
```

This will read in the value from digital pin 1 and set digitalValue equal to it. This value can then be used in other places in your programs. For example, if you have a limit switch on digital input 1. If you want to drive forward when the limit switch is pressed, the code would look something like this.

Of course, there are much more complex things that you can do with digital input. In upcoming sections, you'll learn about multiple different types of sensors that you can use with the Arduino. First, though, it's important to know about digital output as well.



An example driveButton program, where the robot drives when the button is pressed

5.3: Digital Output

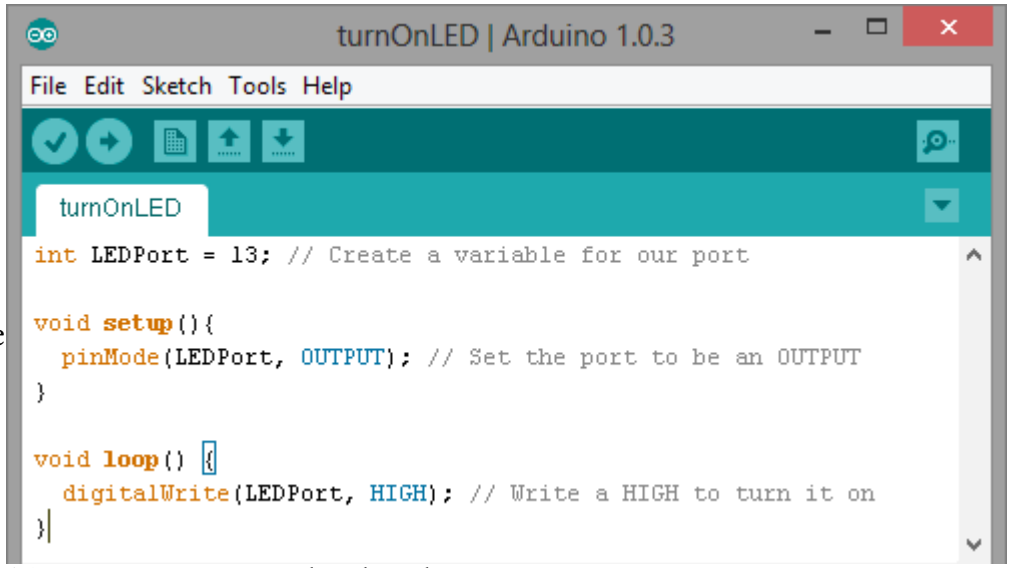
The principle of digital output is very similar to digital input: you have two values, `HIGH` and `LOW`. To write to a port, you once again must first set the pinMode of the port, in this case to `OUTPUT`, like this;

```
void setup() {
     pinMode(1, OUTPUT);
}
```

Once you have set the mode to be an output, you control it with the `digitalWrite()` function, like this:

```
void loop() {
     digitalWrite(1, HIGH);
}
```

Here's an example of using the `digitalWrite()` function. The Uno that comes in your kit has a built-in LED on digital pin 13. You control this LED using digital write. The code to turn it on is as follows.



Turning on an LED with a digital port

Of course, you can do much more complex stuff than just turning on an LED. Try modifying this code so that it blinks the LED at different rates. Remember you can use the `delay()` function to have your program wait.

Another thing that uses digital output (at least on the Arduino) is PWM. The PWM wave is made by pulsing the digital port for the length of time specified by the value you are writing to the PWM. If you take a look at the PWM animation in the previous chapter, you will see that the output is either LOW or HIGH, so it is very feasible to generate using a digital port. PWM square waves are a type of square waves, which are simple waves that are either LOW or HIGH, and nothing in between. The function `analogWrite()`, while sounding like it will allow you to do analog output on an analog pin, will actually generate a square wave on the specified digital pin with the specified duty cycle.

Now that you know about digital input and output, it's time to cover the other type of input: analog input.

5.4: Analog Input

5.4.1: 10 bit ADC

The basics of analog input are very similar to digital input. The difference is, of course, with digital input you get one of two values: 1 or 0. With analog input, you can get a range of values from 0 to 1023. This range is because the Arduino has a 10-bit analog to digital converter (abbreviated ADC) which takes the raw voltage of 0 to 5 volts on the Mega and 0 to 3.3 volts on the Uno and converts it to an integer value. In order for you to understand why this range is 0 to 1023, you'll have to learn about binary. Take a look at this Wikipedia article on the binary

number system, and read the sections on representation, counting, and conversion to other number systems. https://en.wikipedia.org/wiki/Binary_numeral_system#Representation

Now that you've read this, you'll be able to understand what a ten-bit ADC means. A bit is a single binary number, so either 0 or 1. A 10 bit binary number is a binary number composed of 10 bits. A sample 10 bit number could look like this: 1001101001. In base 10 (the traditional 0-9 system you use every day), this number is 617.

The maximum value for a 10 bit number is 210 - 1, or 1023. We subtract 1 because we start at 0, not at one. This is a concept that you will need to become very familiar with as you move on in the robotics track, as it is very important for more advanced electrical and programming related topics.

So, now you know that 10 bits can hold a maximum of 1024 values. This is the reason that the values you get from an analog device on the Arduino are integers ranging from 0 to 1023. This leads to what is called the resolution of the ADC, or how accurate it can be. Since, on the Mega, we have 1023 value for 5 volts, we have a resolution of .0049 volts/value that is returned. On the Uno, where we have a range of 0 to 3.3 volts, the resolution is .0032 volts/value. This is generally accurate enough, as many sensors won't be accurate to the millivolt level.

5.4.2: Reading Analog Input

Now that you know why the Arduino returns 0-1023 for analog input, it would be helpful to know how to read in analog values. With a digital sensor, you start by setting the mode of the pin you want to read in on; the analogous action with analog sensors is to set the reference voltage level, which is the voltage level used for the top of the range. The default reference is 5V for the Mega and 3.3V for the Uno, and these will be fine in most cases. However, there could be some cases where you need to change it. The function for setting the reference level is `analogReference()`, and you use it like this:

```
void setup() {
     analogReference(DEFAULT);
}
```

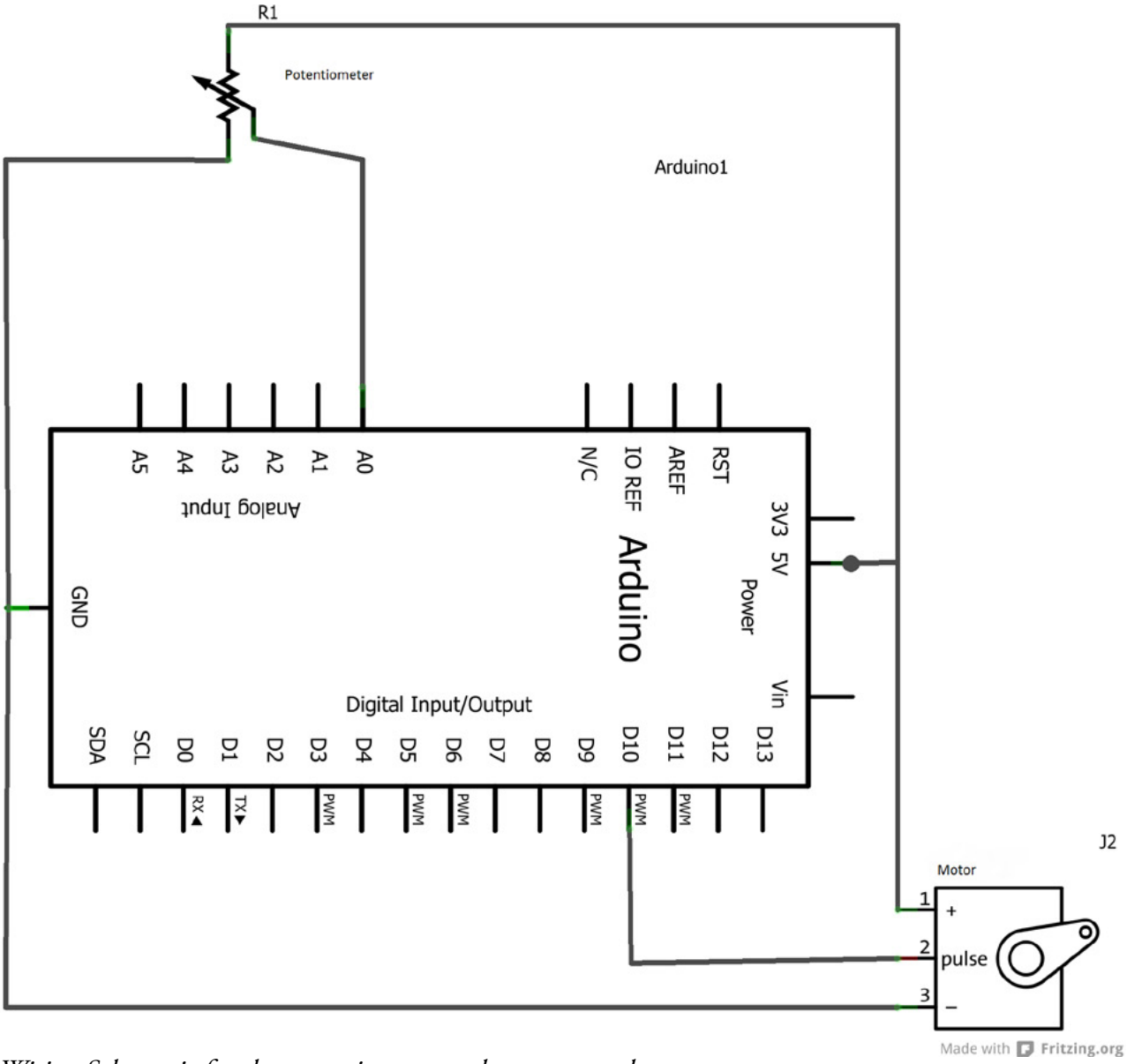
As mentioned, there are several other levels you can use, and they are fully covered in the documentation in the Arduino reference section at http://arduino.cc/en/Reference/HomePage.

To read in a value on a sensor port, we will use the analogRead() function. It takes in the port number that you want to read in, and returns an integer from 0 to 1023 representing the voltage. Here's how you would use it:

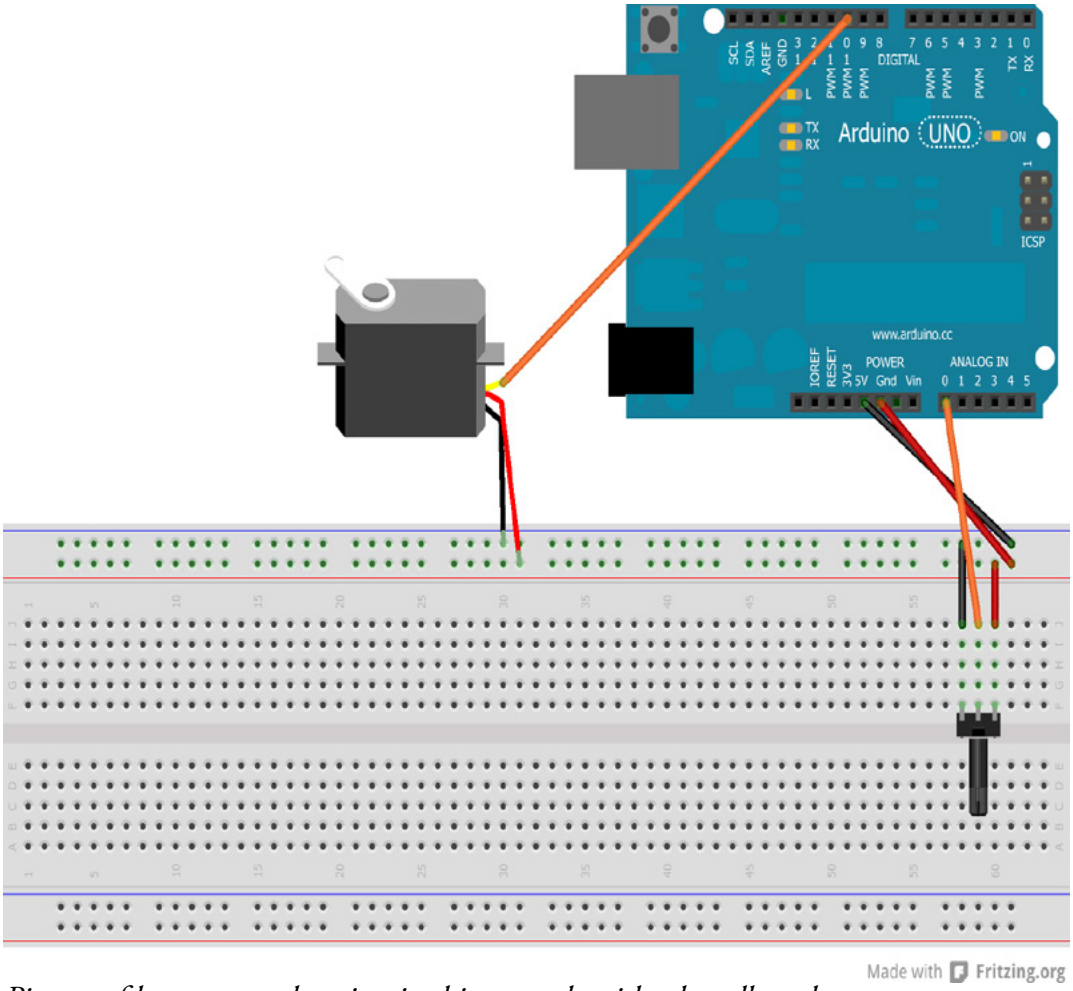
```
void loop() {
     int analogVal;
     analogVal = analogRead(1);
     // Do something with analogVal here
}
```

As an example, here's an example of controlling a motor with a potentiometer. A potentiometer is essentially a variable voltage divider, it will be covered more in depth in the next chapter. For now, know that a potentiometer is a sensor that is used to tell position, usually rotation. There are three potentiometers included in your kit; they are small, with blue casing, a white knob on top, and 3 pins on the bottom.

The first step to controlling a motor with a potentiometer is to read in the current position of the potentiometer. To do this, you'll need to use the breadboard included in your kit to wire the potentiometer and motor. This picture shows you how to wire the motor and potentiometer, as well as showing you what it looks like once they are fully wired together.



Wiring Schematic for the potentiometer and servo example



Picture of how to run the wires in this example with a breadboard.

As you can see, the +5v and ground from the Uno is run to the breadboard. The +5v and ground are then run them to the potentiometer. For the potentiometer, the outer two pins are +5v and ground. Since the potentiometer is just a variable voltage divider, it doesn't matter which pin is +5v and which is ground. The center pin is the signal pin, and that is run directly to analog pin 0 on the Arduino.

The motor also has +5v and ground run to it from the breadboard. The black wire is ground, the red wire is +5v, and the white wire is signal. In this case, it does matter which pin is +5v and which is ground, as you can damage the ESC in the motor if it's wired incorrectly. The signal wire is also run directly to digital pin 10. If you look on the Arduino, pin 10 has a ~ in front of it. Any digital ports marked with a ~ are PWM compatible ports, so make sure you use one of them if you want to control a motor.

Now that the test board is wired, it needs power, and then it needs code. The power for the motor and the board is provided by plugging in the USB cable to your computer. It will provide enough power to run both the Arduino and the motor, but not too much more than that. If you want to run more, you'll have to get a power adapter that will allow you

to plug the Arduino into a wall or battery. The shield on the Mega has an adaptor for power for just this reason.

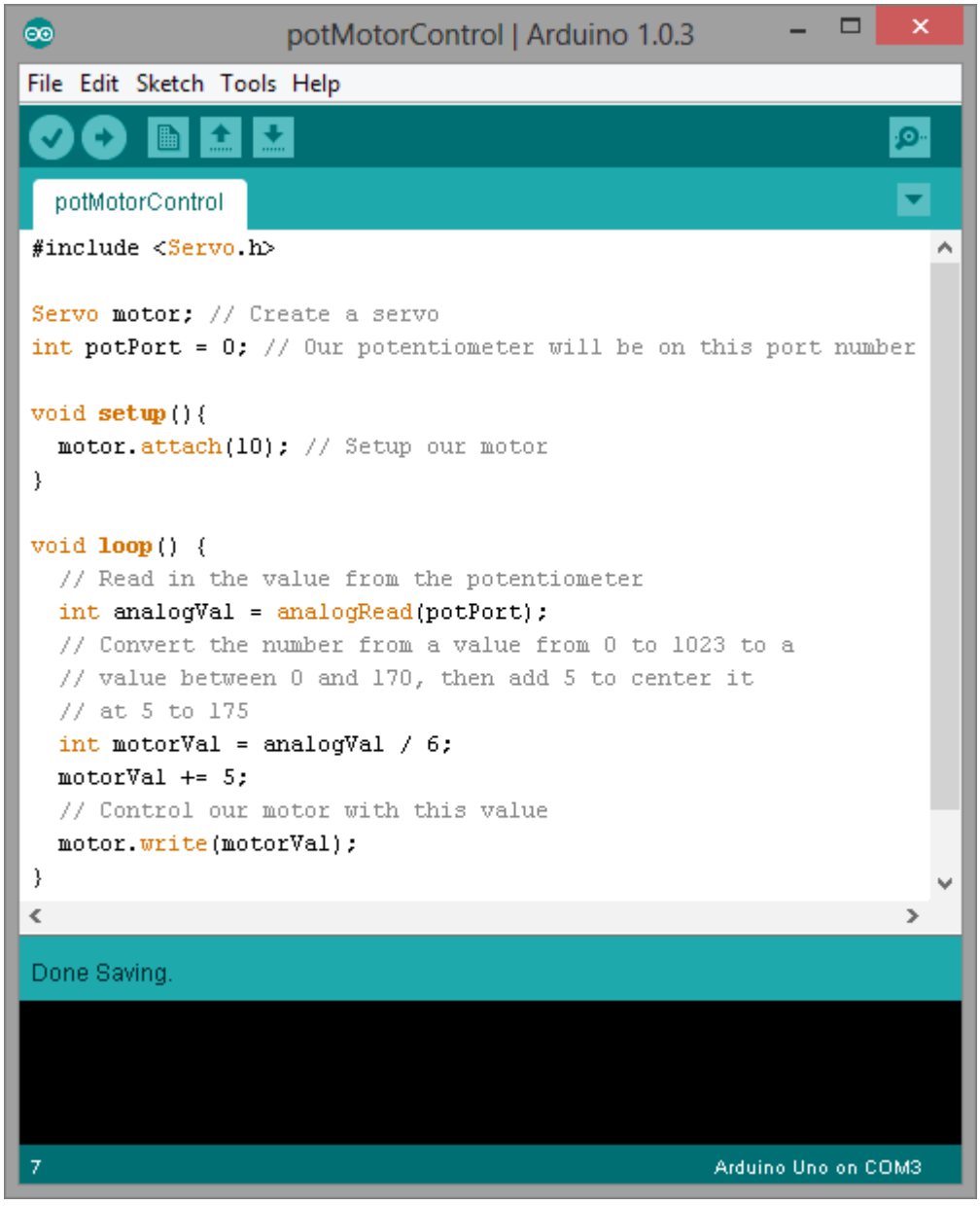
Writing the code to do this is relatively simple. The first step is reading in the value of the potentiometer. This is accomplished by the `analogRead()` command, which takes a port number and returns the integer value of the voltage on that port. Since the potentiometer's output is plugged into port 0, the value is read like this:

```
void loop() {
     int analogVal;
     analogVal = analogRead(0);
}
```

You'll recall that this analog value is a range from 0 to 1023. In order to run the motor, it needs to be converted to go from 0 to 180. To do this, divide the input value by 6. Now, the value goes from 0 to 170. Now, this new value is centered on 90 by adding 5, so the final value's range is 5 to 175. The extra 5 degrees does not make much of a difference with motor control, so this range is fine. Finally, this value is used as the motor speed.

The program will then come back through this `loop()` again.

Here is the final code that will run the motor:



One bit of syntax you may have never seen before is `motorVal += 5;`. This is a more concise notation and exactly equivalent to:

```
motorVal = motorVal + 5;
```

This can also be used for other operators besides addition: -=, *=, and /=.

The code in the previous example will set the motor speed based on a potentiometer. You'll notice that the Vex PWM timings of 1000 and 2000 weren't used. That's because the default times of 544 to 2400 are correct for the motor in the kit, so it isn't necessary to specify them. Try doing this example with your own kit.

Sidebar: Scaling Values

Scaling values like this is very easy: start by following a simple formula of $\frac{inputHigh - inputLow}{outputHigh - outputLow}$. In the example above, you need to convert 0-1023 to 0-180, so the formula looks like this: $\frac{1023 - 0}{180 - 0}$, which is about 6. Approximation is ok in this instance, since the final scaling will be done using integer math, in which there are no decimals. The next step is to center the range obtained through this division on the same place as the desired output. In this example, 1023/6 is about 170 (again, with integer math) and 0/6 is 0, so the new range is 0-170, which has a center of 85. However, the desired range is 0-180, which has a center of 90. Therefore, to line up the centers, you must add 5. With all of these operations, 0-1023 is converted to 5-175. Try doing the same thing, but with a range of (-90)-90 instead.

Try to do these other experiments:

1. In the program shown above the motor runs between full counter-clockwise to full clockwise. Modify the program so the motor runs in one direction from full off to full on when rotating the potentiometer from one stop to the other.
2. Use the potentiometer to control how fast the LED on the board (digital port 13) blinks. Try taking your code from the blink program from the previous section and modifying the delay value to be controlled by the potentiometer's value.

5.5: Serial IO

A very useful technique for program debugging is to print intermediate program values. For example, printing sensor values on the screen is a way of determining if they are what is expected. If you are already familiar with C/C++, you may have tried to use printf(), but this function does not work on the Arduino. Instead, the Arduino uses a serial protocol to send data to be printed on the computer. Serial communications is a technique for sending data a single bit at a time rather than many bits in parallel. To learn more about serial data transmission a good place to

look is at the serial standard RS-232 used for years for communicating data from one device to another. If you are interested see this article to learn more about how it works: https://en.wikipedia.org/wiki/RS-232. The Arduino virtualizes the RS-232 protocol, which is then sent over the USB cable to the computer.

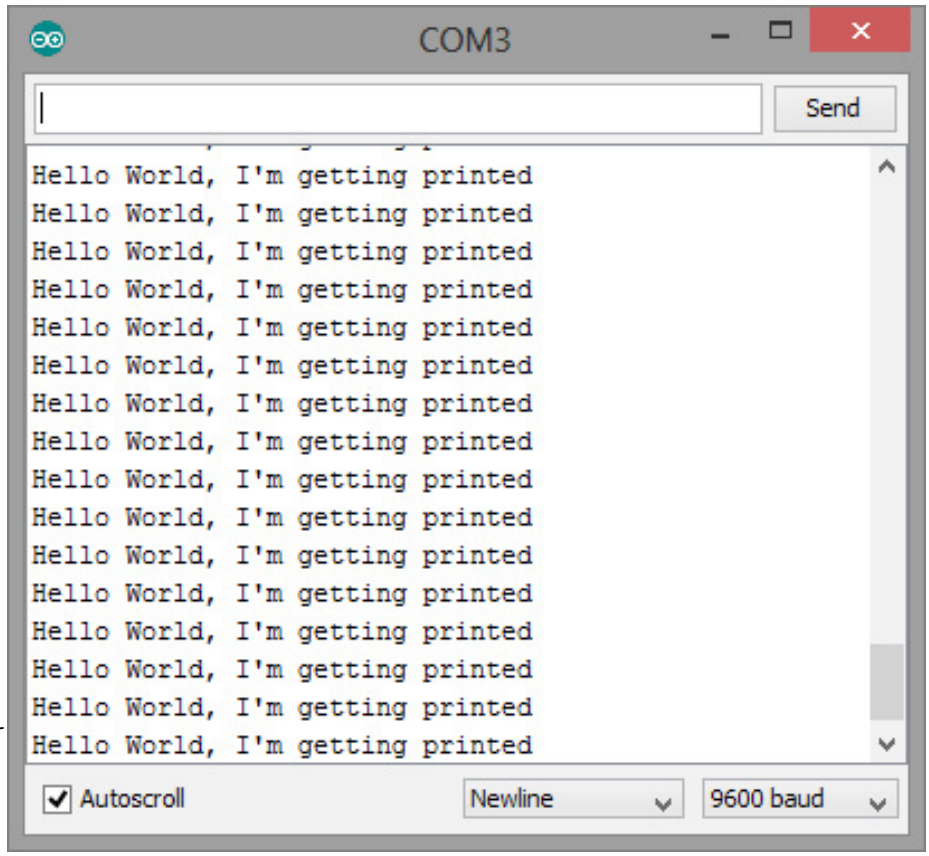
To communicate with the robot, the first step is to start the connection. This is accomplished by using the `Serial.begin()` function, which takes a communications rate and starts up the communication channel. The usual rate is 9600 baud. A baud is a unit of measurement expressing the speed of electronic signals, where 1 baud is 1 bit/second. So 9600 baud is 9600 bits/second, more than fast enough for routine communications. The 9600 is an artifact of the RS-232 standard we mentioned earlier. Here's a code example of initializing communications.

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

Now that communications have been set up, you can start to communicate back and forth with the Arduino. In the Arduino IDE, use the Serial Monitor to see messages coming from the Arduino. This is under Tools -> Serial Monitor, or it can be brought up with the keyboard shortcut Ctrl + Shift + M. To print messages to the console from the Arduino, use the `Serial.println()` function. Essentially, `Serial.println()` takes a string, like this.

```
void loop() {
     Serial.print("Hello World, I'm getting printed");
}
```

This will continually print on the console. Every time `Serial.println()` is called, the output is put on a new line. The output of this command looks like this:



The output of Serial.println("Hello World, I'm getting printed");

To print a number, such a variable, don't enclose it quotes, like this:

```
void loop() {
    int printVar = 5;
    Serial.println(printVar);
}
```

Additionally, if you want to print both a variable and a string on the same line, you need the `Serial.print()` function. This function is identical to `Serial.println()`, except that it doesn't add a newline after it has printed. Take a look at this example:

```
void loop() {
    int printVar = 5;
    Serial.print("The value of printVar is: ");
    Serial.println(printVar);
}
```

Notice that there is a space on the end of the output string; this is because neither `Serial.print()` or `Serial.println()` put spaces in their output. If that trailing space wasn't there, there would be no space between the : and `printVar`.

5.5.1: Special Characters

What if you want to use a quotation mark in a string? If you just put a quotation mark in, it will end the string. To solve this, computer scientists have designated the backslash character, "\", to be what is called the escape character. This means that when it is used, it changes the way the computer perceives the value of the next character. For example, if you want to use a quotation mark, you put a backslash in front of it, like this:

```
void loop() {
     Serial.println("I'm using a \" mark in this string");
}
```

` Instead of ending the string, the quotation mark becomes part of the string. An apostrophe is normally used to enclose character sequences (different from strings). To use one in a string, use "\' ". To use a backslash in a string, use this: " \\". This will put one backslash in the string. In short, putting a backslash in front of anything will change it's value in some way, even a space character.

In C, these backlash-prefixed sequences are called "escape sequences". For newline use "\n". Several others are "\t" for a tab and "\v" for a vertical tab. Take a look at this article for a few more: https://en.wikipedia.org/wiki/Escape_sequences_in_C.

`Serial.println()` uses these escape sequences as well, by appending a "\n" at the end of the input given to it. Take a look at these two statements:

```
void loop() {
   Serial.print("This will print on its own line\n");
   Serial.println("This will also print on its own line, just using the println function");
}
```

These statements do the exact same thing, and in fact, the `Serial.println()` function simply adds a newline on the end of the string and calls `Serial.print()` with the new string.

5.6: Conclusion

Now you know how to read raw sensor values on the Arduino, and how to print to the console. You should experiment with sensors, seeing how they work and thinking of ways to use them to control a robot. Now, it's time to take a look at several available sensors, how to get values from them, and how control an Arduino based on this input.