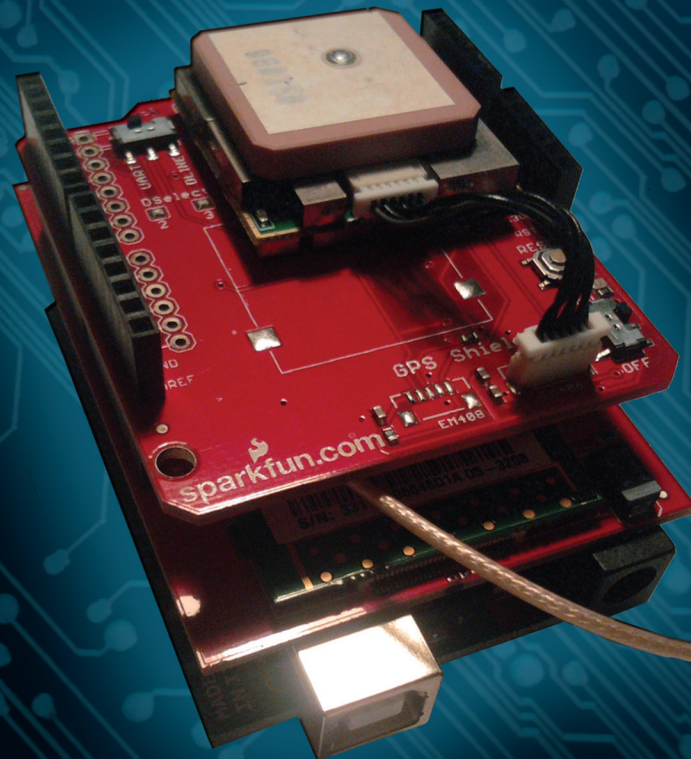




TECHNOLOGY IN ACTION™

Practical Arduino Engineering



Harold Timmis

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Contents at a Glance

■ About the Author.....	xii
■ About the Technical Reviewer	xiii
■ Acknowledgments	xiv
■ Preface.....	xv
■ Chapter 1: The Process of Arduino Engineering	1
■ Chapter 2: Understanding the Arduino Software	15
■ Chapter 3: Robot Engineering Requirements: Controlling Motion	27
■ Chapter 4: Adding Complexity to the Robot: Working with LCDs.....	61
■ Chapter 5: Robot Integration Engineering a GPS Module with the Arduino.....	97
■ Chapter 6: Interlude: Home Engineering from Requirements to Implementation	133
■ Chapter 7: Robot Perception: Object Detection with the Arduino	165
■ Chapter 8: Mature Arduino Engineering: Making an Alarm System Using the Arduino	197
■ Chapter 9: Error Messages and Commands: Using GSM Technology with Your Arduino	217
■ Chapter 10: Control and Instrumentation: The Xbox Controller and the LabVIEW Process	239
■ Chapter 11: Controlling Your Project: Bluetooth Arduino	277
■ Appendix A: Hardware and Tools.....	299
■ Index	303

The Process of Arduino Engineering

In this chapter, we will discuss the engineering process and how you can use it streamline your prototypes by avoiding problems with hardware and software and keeping to a fixed schedule. Throughout this book, you will have projects that will be organized into a sequence I like to call the “engineering process.” Here’s a quick summary of the sequence:

1. Requirements Gathering
2. Creating the requirements document
3. Gathering hardware
4. Configuring the hardware
5. Writing the software
6. Debugging the Arduino software
7. Troubleshooting the hardware
8. Finished prototype

As you can imagine, even this summary of engineering process is very effective when prototyping, which is why we will use it with the Arduino in this book. What is the Arduino? The Arduino is a very customizable microcontroller used by hobbyists and engineers alike. Also, it is open source, which means that the source code is available to you for your programming needs; the integrated development environment (IDE) (where you will be writing your software) is free, and most the resources you can find are open source. The only thing you have to buy is the Arduino microcontroller itself. The Arduino is supported very well on the Web and in books, which makes it very easy to research how-to topics; a few sites that will help you get started are www.Arduino.cc and <http://tronixstuff.wordpress.com/tutorials/>. But this book is more than simply a how-to reference; this book is going to teach you the engineering process—a skill that is useful for making projects more readable, efficient, and reliable.

Gathering Your Hardware

Before we examine the engineering process steps, it’s important to know some of the parts and materials you’ll need. Throughout this book, you will need the following pieces of hardware to complete the

various projects we'll be working on (for a complete list of hardware used in this book, please see Appendix A):

- *Arduino Duemilanove or UNO:* You can use either the Duemilanove or the UNO micro-controller for this book (see Figure 1-1). They have multiple I/O ports for sensors and motors. We will be using these I/O points to control and keep track of the various projects in this book.

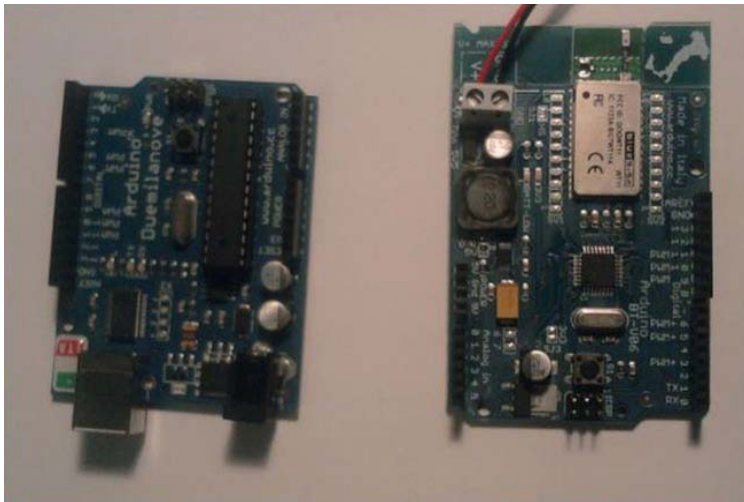


Figure 1-1. Arduino UNO (left) and Duemilanove (right)

- *ArduinoBT or Bluetooth Mate Silver:* I suggest using the Bluetooth Mate Silver modem for this book because it can make your Arduino Duemilanove or UNO behave like an ArduinoBT at half the cost. Also, the ArduinoBT does not have a 3.3V output point, so you would need to add circuitry to the Arduino in order to get 3.3V, which you need in Chapter 6 of this book. Figure 1-2 illustrates these two pieces of hardware.

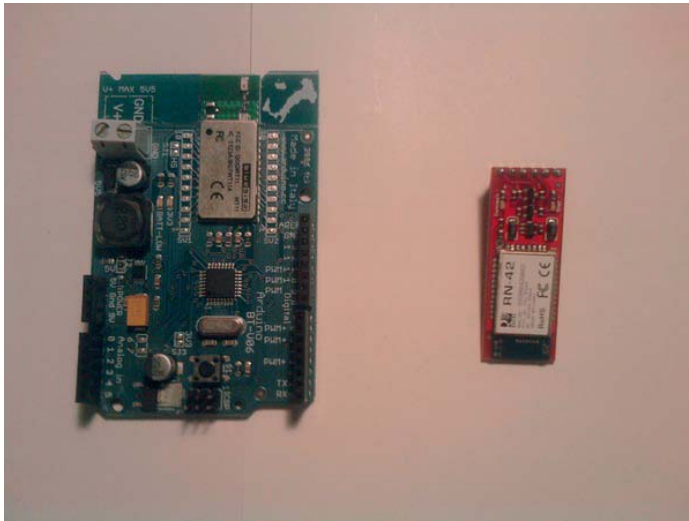


Figure 1-2. ArduinoBT (left) and Bluetooth Mate Silver (right)

- *Solderless breadboard:* Another very important piece of hardware is the solderless breadboard (see Figure 1-3), which is used to implement your circuitry. For this book, you need to have a midsize solderless breadboard. It will be used in both the design and troubleshooting phases of the projects.

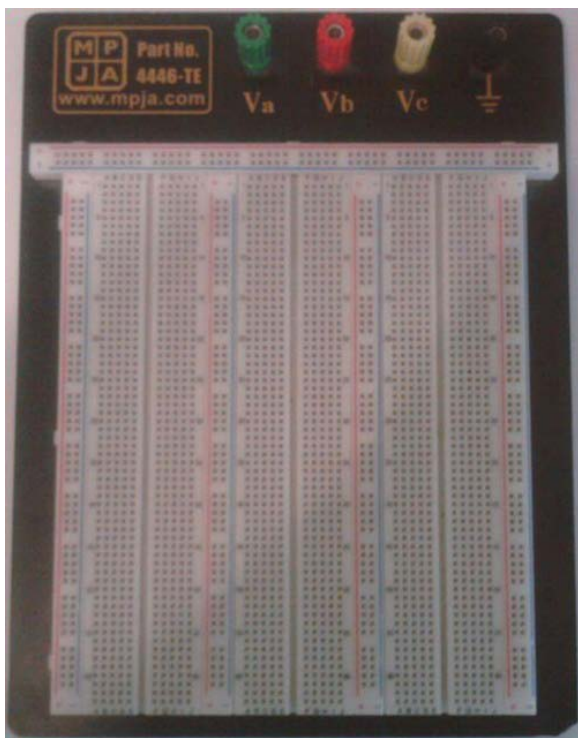


Figure 1-3. An example of a solderless breadboard

- *Wire:* We will use a large quantity of wire in this book; you can get a wire jumper kit at almost any electronics store.
- *Arduino shields:* We will be using several shields in this book, including the Motor, GPS, GSM, and LCD shields (see Figure 1-4).

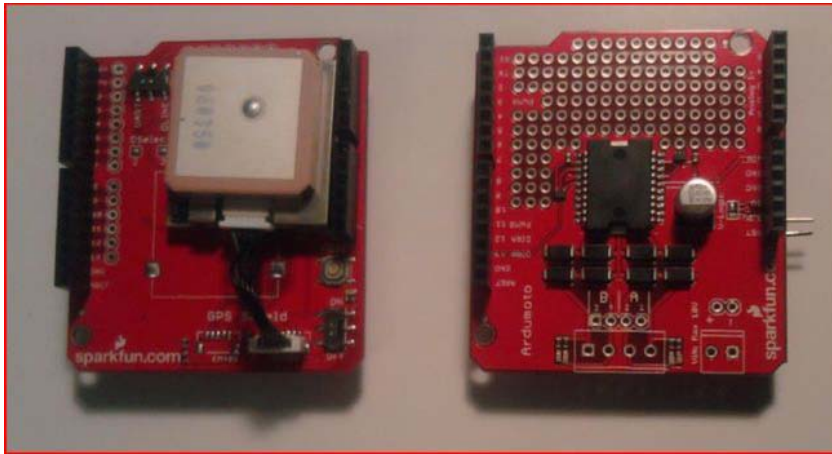


Figure 1-4. A couple of Aeduino Shields the GPS Shield on the left and the motor shield on the right.

- *Motor shield:* This shield is used to control motors up to 18V. It includes a surface-mount H-bridge, which allows for a higher voltage motor to be used as well as for control of two motors. For more information on H-bridges, please see Chapter 3.
- *GPS shield:* This shield is used to get positioning information from GPS satellites. It uses the National Marine Electronics Association (NMEA) standard, which can be parsed to tell you any number of things such as longitude and latitude, whether the GPS has a fix, what type of fix, a timestamp, and the signal-to-noise ratio. For more information on GPS systems, please see Chapter 5.
- *GSM shield:* This shield will allow you to use the power of the Global System for Mobile Communications (GSM) to send text messages back and forth at great distances; this shield also uses a standard protocol called the GSM protocol.
- *LCD shield:* We will use this to add images and life to our robots. The LCD shield can also be used to create your own user interface for your robot or any other project that you would like to pursue.
- *Sensors:* These are very important because they give your projects life. Some sensor examples are PIR (Passive Infrared), sonar, and temperature (see Figure 1-5).

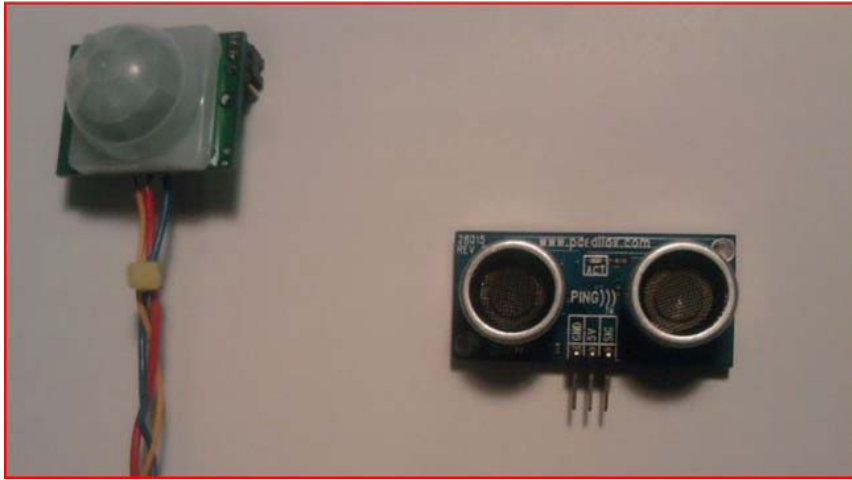


Figure 1-5. A PIR sensor (left) and a Sonar sensor (right)

- *PIR sensor:* This is an outstanding sensor for detecting changes in infrared light and can detect changes in temperature. It is also great at detecting motion, and that's what we will use it for.
- *Sonar sensor:* Sonar sensors are good at detecting objects in their surroundings. The Sonar sensor we will be using is a Parallax sensor that uses digital pinging to tell how far away an object is.
- *Temperature sensor:* These sensors are used to read temperature. To use them, you first scale the voltage to the temperatures you want to record; you can find more information about this sensor in Chapter 6.
- *Servos and motors:* We will be using motors and servos to control many aspects of the projects (see Figure 1-6).



Figure 1-6. Examples of a few motors:

- *Miscellaneous:* These are the most common components, such as resistors, capacitors, LEDs, diodes, and transistors (see Figure 1-7).



Figure 1-7. Miscellaneous pieces of hardware (terminal blocks, capacitors, resistors, LEDs, and switches)

Gathering Your Tools

You will also use a variety of tools; this section will briefly describe them. For a full list of tools you will need, please see Appendix A.

- *Soldering iron:* This tool is used to connect circuits to each other; we will use it mostly to connect wire to circuits (see Figure 1-8).



Figure 1-8. A Soldering iron and its stand

- *Solder:* You will use this in conjunction with the soldering iron; it is the metal that connects the circuits together. Solder has a very low melting point.
- *Needle-nose pliers:* These pliers are very important; they are used to hold wire and circuits in place, wrap wire around circuitry, and so on (see Figure 1-9).
- *Magnifying glass:* This tool is used to better see circuitry.
- *Dikes:* These are used to cut wires (see Figure 1-9).
- *Wire stripper:* This tool is used to take off wire insulation (see Figure 1-9).
- *Multimeter:* Possibly the most important tool you can own, this tool allows you to read voltage (AC (alternate current) and DC (direct current)), amps (ampere), and ohms (see Figure 1-9).
- *Scientific calculator:* This allows you to do various calculations (Ohm's Law, voltage divider, etc.).



Figure 1-9. Additional Tools from left to right: multimeter, needle-nose pliers, wire clippers (top), wire stripper (bottom).

Understanding the Engineering Process

The engineering process is very useful in making your designs more efficient, streamlined, and comprehensible. The process consists of gathering requirements, creating the requirements document, gathering the correct hardware, configuring the hardware, writing the software, debugging the software, troubleshooting the hardware, and the signing off on the finished prototype.

Requirements Gathering

One Day, when you're an engineer, you may be asked to go to a company and assess its needs for a particular project. This part of the engineering process is crucial; everything will depend on the requirements you gather at this initial meeting. For example, assume you learn that your client needs to **blink an LED at a certain speed**[insert an example case here to flush out the idea]. And for that task, you and the client determine that the Arduino microprocessor is the best choice. To use the Arduino to blink the LED[insert the example task here], a customer needs an LED to blink at 100ms intervals.

Creating the Requirements Document

Based on the client's needs and your proposed solution, the following is a very simple requirements document:

- Hardware
 - Arduino
 - LED

- 9V battery
- 9V battery connector
- 22ohm resistor
- Software
 - A program that blinks an LED at 100ms intervals

Mind you, this is a very simple example, but we will be using this format for the rest of this book. One of the reasons you create a requirements document is to stop *feature creep*. This happens when a customer keeps adding features to the software and/or hardware. This is, of course, a problem because you will be working more hours without more pay on a project that may never end. You should create a requirements document, so you and the client know what you are doing and the features you will be creating for your project. After you have created the requirements document, you can create a flowchart that will help you debug the software later in the design process (see Figure 1-10).

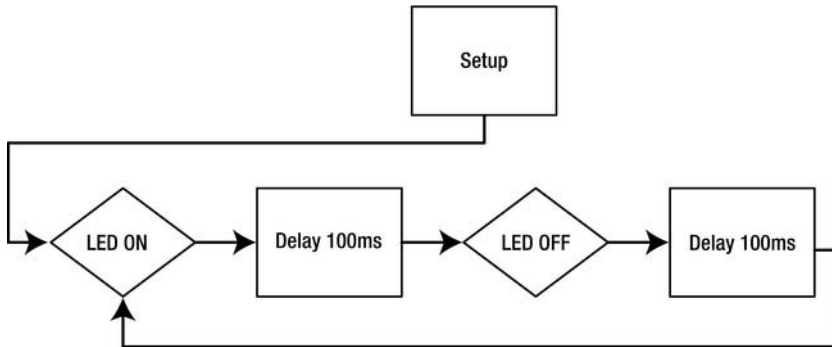


Figure 1-10. Blinking LEDs processes

Gathering the Hardware

The next very important part of the engineering process is making sure you have the right hardware. Your hardware needs should be decided as you're gathering requirements, but it is important to make sure all your hardware is compatible. If it is not compatible, hardware changes may be necessary, but you should consult the company you are working with to make sure it is satisfied with any hardware changes.

Configuring the Hardware

Once you have the correct hardware, it is time to configure it. Depending on the hardware required for the project, the configuration can change. For example, let's take a look at the hardware configuration for the blinking LED project.

- Arduino
- LED
- 9V battery

- 9V connector

To set up the hardware, we need to connect the LED to digital pin 13 and ground on the Arduino(see Figure 1-11), as illustrated in the schematic shown in Figure 1-12.

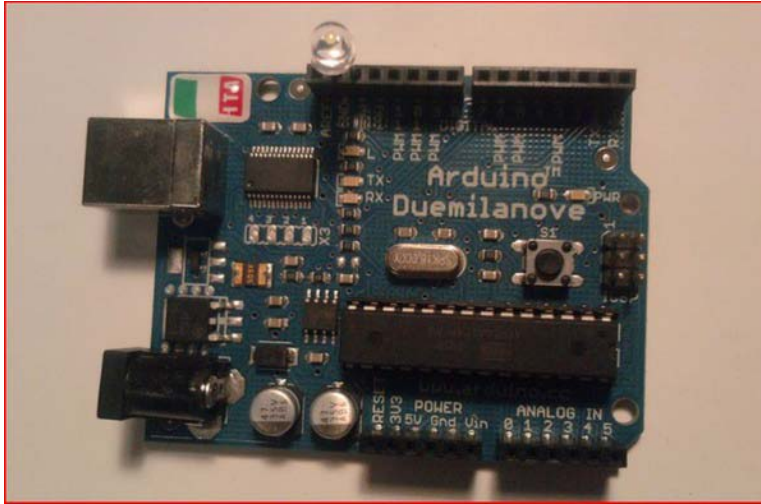


Figure 1-11. The hardware setup for the Blinking LED project

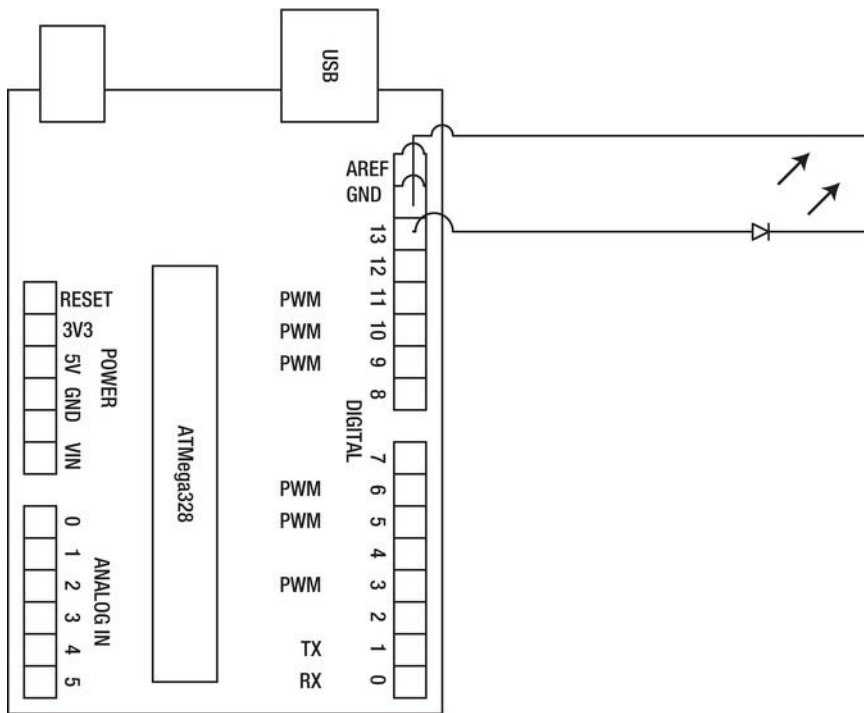


Figure 1-12. A schematic of Blinking LED project

First, we need to install the Arduino IDE. To do this, go to www.Arduino.cc/en/Main/Software. The Arduino IDE will work with Windows Vista or 7, Mac OS X, and Linux systems. After the Arduino IDE is downloaded to your desktop, it will be in a zipped format, so unzip the Arduino-0022 file to your desktop. The Arduino IDE is now installed.

Now that you have the Arduino IDE installed on your computer, you need to make sure it is configured correctly. To do this, open the Arduino IDE, and go to Tools → Serial port; select the serial port your Arduino is connected to. Next, select Tools → Boards, and select the Arduino board you are using (this book uses the Arduino Duemilanove Atmega328). Once your hardware is configured, it is time to write the software.

Writing the Software

Now, let's consider the software we need to write. This part of the engineering process is crucial. Let's take a look at the blinking LED software requirements document to decide what the software will need to do: the LED needs to blink in 100msn intervals. The software might look something like this:

```
// This code blinks an LED at 100ms

const int LEDdelay = 100; // delay time

void setup()
{
```

```

    pinMode(13, OUTPUT); // makes pin 13 an output
}

void loop()
{
    digitalWrite(13, HIGH); // this writes a high bit to pin 13
    delay(LEDdelay);        // delay 100ms
    digitalWrite(13, LOW);
    delay(LEDdelay)          // this will throw a syntax error due to a missing semicolon
}

```

■ **Note** When you try to compile this program to your Arduino, it gives you an error. This is because of a syntax error that we will debug in the next section.

Debugging the Arduino Software

The last program failed to compile because of a *syntax error*. This type of error is because of incorrectly formatted code, such as a missing semicolon (which is why the last program didn't compile). Here is the revised code:

```

// This code blinks an LED at 100ms

const int LEDdelay = 100; // delay time

void setup()
{
    pinMode(13, OUTPUT); // makes pin 13 an output
}

void loop()
{
    digitalWrite(13, HIGH); // this writes a high bit to pin 13
    delay(LEDdelay);        // delay 100ms
    digitalWrite(13, LOW);
    delay(LEDdelay);        // the semicolon is now present and the code will compile
}

```

Syntax errors are not the worst errors out there. The worst errors you can receive are *logical errors*; these errors allow the code to compile, but the end result is unexpected. For example, using a greater-than symbol instead of less-than is a logical error, and if it is in a project with thousands of lines, it can be almost impossible to fix.

■ **Note** A logical error in the blinking LED project would be if you put `digitalWrite(13, HIGH);` for both digital write functions.

We can debug logical errors in an Arduino program by using a flowchart to figure out where the values are not lining up.

Troubleshooting the Hardware

The number one tool used to troubleshoot hardware is the multimeter. This tool can save your hardware from being damaged. For instance, if your multimeter detects that your power supply is more than is required, the hardware setup for the blinking LED project could use a 22ohm resistor to keep the LED from burning out.

Finished Prototype

Once you have finished debugging the software and troubleshooting the hardware, you should have a completed prototype that will work under most circumstances. In this chapter, we used a very simple project, but in future chapters, the projects will get more complicated, and the engineering process will become even more necessary to make sure our code is efficient, streamlined, and comprehensible.

Summary

In this chapter, you learned the different pieces of hardware and various tools such as: The Arduino, Arduino Shields, Multimeter, and needle-nose pliers; just to name a few that will be used throughout this book. We then went over the engineering process, which is a sequence you can use to solve problems that provides the format for this book. The steps in the process are requirements gathering, creating the requirements document, gathering the hardware, configuring the hardware, writing the software, debugging the Arduino software, troubleshooting the hardware, and finished prototype. I also defined a few new terms that will help you understand the engineering process, and you learned the differences between logical and syntax errors.

Understanding the Arduino Software

In this chapter, we will discuss the various programming components that will be used throughout this book. If you have programmed in C, you will find that programming for the Arduino is very similar. If not, this chapter will teach you the basic concepts. Why is it important for you to learn the basics of programming the Arduino? In the long run, this will help keep your code clean and readable. Also, learning the basic loops and structures initially will allow us to focus more on the libraries later. Libraries can be sets of classes, types, or functions and can be called by using keywords in your program. The purpose of a library is to readily add more functionality to your program by using code that has been created previously; this promotes code reuse. The libraries we will go over briefly in this chapter are `NewSoftwareSerial`, `LCD Library`, `TinyGPS`, and a few others.

Getting Started with `setup()` and `loop()`

All Arduino programs must have two main components to work properly—`setup()` and `loop()`—and they are implemented like this:

```
// Basic Arduino Program

void setup()
{
    // Set up I/Os here
}

void loop()
{
    // Do something
}
```

`setup()` is used to set up your I/O ports such as LEDs, sensors, motors, and serial ports. Careful setup is important because in order to use the pins on the Arduino, we need to tell Arduino that they are going to be used.

`loop()` holds all of the code that controls your I/O ports. For instance, here you'd tell your motor to go a certain speed. I will explain how to set up and control your I/O's in the next sections.

Arduino programs also have subroutines—very useful extra functions you can call within `loop()` or its subroutines. To use a subroutine, you must first initialize it at the beginning of your program; this initial mention is called a *function prototype*. Here is an example:

```
// Function Prototype
void delayLED();

void setup()
{

}

void loop()
{

}

// Subroutine Example

void delayLED()
{
    // This will go after the loop() structure.
}
```

Initializing Variables

Variables are the most basic programming building blocks; they are used to pass data around your program and are used in every program we will write in this book. We can write several types of variables to the Arduino language; Table 2-1 illustrates them.

Table 2-1. Types of variables

Type Name	Type Value	Type Range
char	'a'	-128 to 127
byte	1011	0 to 255
Int	-1	-32,768 to 32,767
unsigned int	5	0 to 65,535
long	512	-2,147,483,648 to 2,147,483,647
unsigned long	3,000,000	0 to 4,294,967, 295
float	2.513	-3.4028235E+38 to 3.4028235E+38
double	2.513	-3.4028235E+38 to 3.4028235E+38

Now that you know what types of variables are out there, you need to know how to declare those variables. In order to declare them, you need to know in what *scope* those variables can be used and then specify (or *declare*) the scope that meets your needs. In this book, we will declare two scopes for variables: local variables and global variables. A *local variable* only works in its given scope. For instance, a *for* loop keeps its declared variables only within its parentheses, so those variables are local to the *for* loop. A *global variable* can be called at any point in your program. To define a global variable, initialize it at the beginning of your program. The following program illustrates how to initialize local and global variables:

```
// Initialize Variable

int x; // This variable is declared globally and is available for access throughout this
      // program.

void setup()
{

}

void loop()
{
    x = 1 + 2; // Assigns the value 3 to x
    for(int i; i <= 100; i++)
    {
        // i is a local variable and can only be called in this for loop.
    }
}
```

The rest of the declarations are set up the same way until you start using arrays. *Arrays* allow you to pass multiple values of the same type, for example, if you want to pass multiple digital pins without having to declare each one individually:

```
int pins[] = {13,9,8};
```

It is a good idea to declare the size of the array, as in the following example:

```
const int NumOfPins = 3;
int pins[NumOfPins] = {13,9,8};
```

This will allow you to access your array's information, and then, you can pass that information to a digital pin or whatever else you want. Now that you have declared variables, how do you use them? This will be discussed in the next few sections of this chapter.

■ **Note** *Whitespacing* means that you've added blank lines and spaces in your code to make it more readable.

Writing Conditional Statements

Conditional statements can be used to control the flow of your program. For instance, say you want to turn a motor on only when a button is pressed; you can do so using a conditional statement. We will be discussing the following conditional statements: *if*, *if-elseif*, *if-else*, and *switch* statements.

An *if* statement is a very important conditional statement; it can be used in any Boolean capacity for a variety of reasons, such as limiting testing. Here is an example of an *if* statement:

```
int i;
if (i < 10)
{
    i++;
}
```

You can also add *elseif* statements to the end of your *if* statement to add other conditions to your program and create an *if-elseif* statement, for example:

```
int i;
if (i < 10)
{
    i++;
}
else if (i > 10)
{
    i--;
}
```

A practical use of a conditional statement would be to reading a value from a potentiometer, as in the following example:

```
potValue = analogRead(potPin);

if (potValue <= 500)
{
    digitalWrite(motorpin, 1);
}
else
{
    digitalWrite(motorpin, 0);
}
```

■ **Note** You must remember set up your Arduino's pins before you call them in a loop.

A *switch* statement is used if you have multiple conditions because it cleans up your code. Here is an example of a *switch* statement:

```
switch (potValue){
case 500;
    digitalWrite(motorPin,1);
case 501;
```

```

    digitalWrite(ledPin,1);
    break;
default:
    digitalWrite(motorPin,0);
    digitalWrite(ledPin,0);

```

In this example, if `potValue` is equal to 500, the motor will turn on, and if `potValue` is equal to 501, an LED will turn on. The default case is true when the `potValue` equals neither 500 nor 501, and in that case, the motor and LED are both turned off.

Working with Loops

Loops have many uses including getting rid of redundant code and iterating through arrays. The loops we will use are `for`, `while`, and `do...while`. These loops will allow us to run through code while a condition is true (or false, in some circumstances).

- **for loop:** This loop is used to repeat a block of code a fixed number of times. The `for` loop's basic set up is

```

for(int i = 0; i <= 10; i++)
{
    // Place statements here
}

```

- A practical application for a `for` loop is to use it to update multiple `pinMode` settings:

```

int pins[] = {13,9,8};

void setup()
{
    for(int i = 0; i<=2;i++) // Sets up each pin
    {
        pinMode(pin[i], OUTPUT);
    }
}

void loop()
{
    // Put code here
}

```

■ **Note** `pinMode` is used to set up your I/O pins on the Arduino.

- **while loop:** This loop will run until a condition has been met; if its first condition is false, it will not run at all. For example, you'd use a `while` loop if you wanted to run code until a certain value came from a sensor. The following example illustrates this principle:

```

int potPin = A1;

```

```

int motorPin = 9;
int potVal;

void setup()
{
    pinMode(motorPin,OUTPUT);
    pinMode(potPin,INPUT);
}
void loop()
{
    potVal = analogRead(potPin);
    while(potVal <= 100) // Runs until potVal is greater than 100
    {
        digitalWrite(motorPin,1);
    }
}

```

- The first thing this code does is initialize the potentiometer and motor pins; then, it declares `potVal`, our variable that holds the potentiometer value. Next, we set the `motorPin` to an output, and the `potPin` to an input. Finally, we use a `while` Loop with a condition `potVal <= 100`, and while that condition is true, the motor will be on.
- `do . . . while` loop: This is the same as the `while` loop except that the conditional statement is checked at the end of the loop, so this loop will run at least one time. Here's an example:

```

do
{
    i++; // Increment i
}while(i <= 100);

```

Communicating Digitally

Throughout this book, we will be communicating different types of I/O through the digital pins, so it is important to understand how that communication works. Specifically, we use the `digitalWrite(pin,HIGH/LOW)` and `digitalRead(pin)` commands to communicate with the digital pins. An example of this is shown in Listing 2-1.

Listing 2-1. Digital Commands

```

int button = 12;
int led = 13;
int buttonState;

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

```

```

}
void loop()
{
    buttonState = digitalRead(button); // Assigns button to buttonState
    if(buttonState == 1)
    {
        digitalWrite(led,HIGH); // Writes a 1 to pin 13
    }
    Else
    {
        digitalWrite(led,LOW); // Writes 0 to pin 13
    }
}

```

This program uses `digitalWrite()` and `digitalRead()` to get the value of the button pin and writes a new value to it (in this case, high or low).

■ **Note** Use the PWM digital pins to control motor speed and LED brightness.

Communicating with Analog Components

You can also use analog communication with sensors and motors, meaning you can connect potentiometers to control motor speed through a pulse width modulation (PWM) pin on the Arduino. The functions for analog communication are `analogRead(value)` and `analogWrite(pin,value)`. The only thing you need to remember is that a potentiometer will give a value of 0 to 1024, so you will have to scale `analogWrite` from 0 to 255, for example:

```
analogWrite(LED,ledValue/4); // 1024/4 = 255
```

Serial Communication

We will be using serial communication throughout this book. Serial communication allows us to communicate with a computer, LCD, and many other devices, as you will see in the next several chapters. Some serial commands are `Serial.begin(baud)`, `Serial.Println("anything you want to write to the serial pin")`, `Serial.read()`, `Serial.write(Binary data)`, `Serial.available()`, and `Serial.end()`. These commands allow us to read and write to any serial peripheral we want. Here is a brief description of each of these serial commands:

- `Serial.begin(baud)`: You will put this command inside your `setup()` structure and put the appropriate baud rate for the device with which the serial will be communicating, for example:

```

void setup()
{
    Serial.begin(9600); // 9600 baud rate to communicate with a computer
}

```

- `Serial.println()`: Use this command to write values to the serial port, for example:

```
void loop()
{
    Serial.println("Hello, World"); // Writes Hello, World to the serial port
}
```

Or...

```
void loop()
{
    Serial.println(potVal); // Writes potVal to the serial port
}
```

- `Serial.read()`: This reads in a value from the serial port. For example, you could use this to read something from your computer that you'd then want to write to an LCD on the Arduino.

```
void loop()
{
    char var = Serial.read(); // Read incoming byte from serial port
}
```

- `Serial.write()`: Use this to write binary data to a serial port, for example:

```
void loop()
{
    while(Serial.available() > 0)
    {
        char var = Serial.read(); // Reads incoming byte from serial
        Serial.write(var); // Writes binary data to serial
    }
}
```

■ **Note** In this book, most of the time you will use `Serial.println()` because we will be writing `int` or `string` values to the serial monitor. The `Serial.write()` function is used to send binary data to the serial monitor or any other serial port program..

- `Serial.available()`: This function checks to see if there are any incoming bytes at the serial port, for example:

```

void loop()
{
    while(Serial.available() > 0) // This makes sure there is at least one byte at the
                                // serial port.
    {
        // Put code here
    }
}

```

- `Serial.end()`: This disables serial communication.

Now that you have seen the command set for serial communication, we can use them in our programs. Listing 2-2 illustrates most of the functions we have been discussing.

Listing 2-2. Serial Communication

```

int incomingByte;
const int ledPin = 13;
void setup() {

    Serial.begin(9600);      // Opens serial port, sets data rate to 9600 bps

    pinMode(ledPin, OUTPUT);

}

void loop()
{

    while(Serial.available() > 0)
    {

        incomingByte = Serial.read();      // Reads incoming byte
        Serial.println(incomingByte, BYTE); // Prints incoming byte to serial port
        digitalWrite(ledPin, incomingByte); // Write to LED pin
    }

}

```

This program is the foundation of serial communication: it initializes `incomingByte` and `ledPin`. Next in the setup structure, the baud rate is set to 9,600. When we get inside the loop structure, the `while` loop is checking to see if anything is at the serial port. If there is, it assigns the information on the serial port to `incomingByte`. Finally, the program prints the data to the serial port and writes data to `ledPin` (in this case 1 or 0).

Using Arduino Libraries

Now that you know the basics of Arduino programming, we are going to go over the libraries that will be used in the chapters to come. The primary libraries we'll use are the `NewSoftwareSerial`, `TinyGPS`, and `LCD` libraries, and a few others will be explained in later chapters. To use these libraries, you will need to

download them and unzip them into the `Libraries` folder in the `Arduino-022` directory. After you do that, you should be able to use any library that is compatible with the Arduino.

NewSoftwareSerial

The `NewSoftwareSerial` library allows us to write to multiple software serials (we need to thank Mikal Hart). The only caveat is that you cannot read and write at the same time; these actions must be preformed sequentially. To use this library, the first thing you have to do is obtain the `NewSoftwareSerial` library from the Internet and add it to the `Libraries` folder in the Arduino IDE folder. You can download `NewSoftwareSerial` at <http://arduiniiana.org/libraries/newsoftserial>.

Once you have downloaded and added the `NewSoftwareSerial` to the Arduino IDE directory all you have to do is call `NewSoftwareSerial` in the program where you want to use it in, like this:

```
#include <NewSoftwareSerial.h>
```

```
NewSoftwareSerial gps(2,3); // This creates a new instance of NewSoftwareSerial under the
                             // name gps.
                             // the format in the brackets is (rx,tx)
```

This will include a `gps` serial to pins 2 and 3. In the setup structure, you have to set the new serial's baud rate, like so:

```
void setup()
{
    gps.begin(4800);
}
```

After that, instead of using the `Serial` library, you will be calling functions from the `NewSoftwareSerial` library, and you can just call those functions from the `gps` object we created earlier, for example:

```
Serial.read();
Becomes...
gps.read();
```

TinyGPS

This library parses NMEA data, such as longitude, latitude, elevation, and speed, into a user-friendly style. We will go into further detail on the `TinyGPS` library in Chapter 5. All you need to do now is download the `TinyGPS` library from <http://arduiniiana.org/libraries/tinygps> (we need to thank Mikal Hart).

ColorLCDShield Library

The `LCD` library will be used to read and write to an LCD screen (we need to thank Peter Davenport, Coleman Sellers, and Sparkfun). We will go over the `LCD` library in Chapter 4. For now, just download the `LCD` library from <http://www.apress.com/9781430238850>

Putting Together the Arduino Language Basics

You should now know how to create the most basic Arduino program, so let's take a moment to recap some of the key programming points just discussed. You can use this recap to help you program throughout this book, and through creating your own projects. Here is an example of that program:

```
void setup()
{
    // Setup I/Os here
}
void loop()
{
    // Put code here
}
```

We also went over declaring variables and using them globally, as in the following example:

```
char ch = 'A';
int pin = 13;
```

These values have types of character and integer.

You also learned about if and if-else statements and how to use them:

```
if (condition)
{
    // Put code here
}
else
    // Put code here
```

Also you can now add elseif statements to create if-elseif statements to add more conditions, if you need them.

```
else if(condition)
{
    // Put code here
}
```

We then went over the switch statement, another type of conditional statement that is used sometimes to clean up larger if statements; it has this format:

```
switch(value)
{
    case value:
        // Put code here
        break;
    case: value:
        // Put code here
        break;
    default:
        // Put code here
        break;
}
```

After the conditional statements were explained, we went over the various loop structures you can use to parse or iterate through code. The first loop we discussed was the for loop:

```
for(initialization;condition;Variable Manipulation
{
    // Put code here
}
```

We then went over the while loop and its functions:

```
while(condition)
{
    // Put code here
}
```

After that, you learned about a close relative of the while loop called the do. . .while loop; it has the following format:

```
do
{
    // Put code here
}while(condition);
```

Next, you needed to learn about different ways to communicate with sensors and other peripherals, so we discussed the `digitalRead()` and `digitalWrite()` functions, which follow:

```
digitalRead();
digitalWrite(pin,state);
```

We then discussed communicating with the analog pins on the Arduino. The analog pins use these commands:

```
analogRead();
analogWrite(pin,value);
```

After learning about the different ways to communicate with sensors, we needed a way to communicate with serial communication. Here are the commands for serial communication:

```
Serial.begin(baud);
Serial.println(value);
Serial.read();
Serial.write(value);
Serial.available();
Serial.end();
```

Finally, you learned a little about the different libraries covered in this book, primarily `NewSoftwareSerial`, `TinyGPS`, and `LCD`.

Summary

In this chapter, you learned about the Arduino language. Specifically, you learned how get your programs set up and how to use conditional statements and loops to refine them. You also learned how to communicate with different types hardware pins using digital, analog, and serial communication. Finally, we discussed several libraries we will use in later chapters.

Robot Engineering Requirements: Controlling Motion

Well, you have finally made it to the good stuff. In this chapter, we will be discussing motor control, a very important part of robotic design that will be used throughout this book. Motor control allows us to give a robot life, but we will not simply be pushing a button to control our robot. Instead, per a hypothetical customer's request, we will be controlling the motor speed and direction through serial communication.

We can accomplish this using an H-bridge and a hex inverter. The H-bridge will be used to drive two motors and the hex inverter will be used to convert a signal from 0 to 1 (1 to 0). The company wants each of the comma-separated parameters to go to these specific pins. Another requirement the company has is that the final prototype needs to be configured on a chassis—but not until the hardware and software have been tested.

To get the most out of this project, you will need a basic understanding of the engineering process and the Arduino programming language as discussed in the first two chapters. If you have not read them yet, please do so before proceeding. Before tackling our client's request, though, you'll need some fundamental information to give you the necessary foundation to create the final project. To start things off, we'll discuss the H-bridge and its various uses followed by a review of the hardware that will be used in this chapter. We will then create four small projects that will teach you how to turn on and off a motor with a button. Next, you'll learn how to use a potentiometer to control the speed on a motor. After that, you'll learn how to control multiple motors with the H-bridge and how to control both speed and direction with a potentiometer and a switch. Armed with your newfound knowledge, you'll create a robot that uses the H-bridge to control its movements.

Hardware Explained: The H-bridge

H-bridges are very useful integrated circuits (ICs); they allow for higher voltages to be connected to motors without using the Arduino's weak 5V supply. With some H-bridges, you can use up to 30V. Figure 3-1 shows three examples of an H-bridge; for this chapter, we will be using the motor shield from SparkFun shown on the far right.

■ **Note** For all of the shields that we use in this book, you will need to buy not only the shield itself but also two 6-pin female headers and two 8-pin female headers.

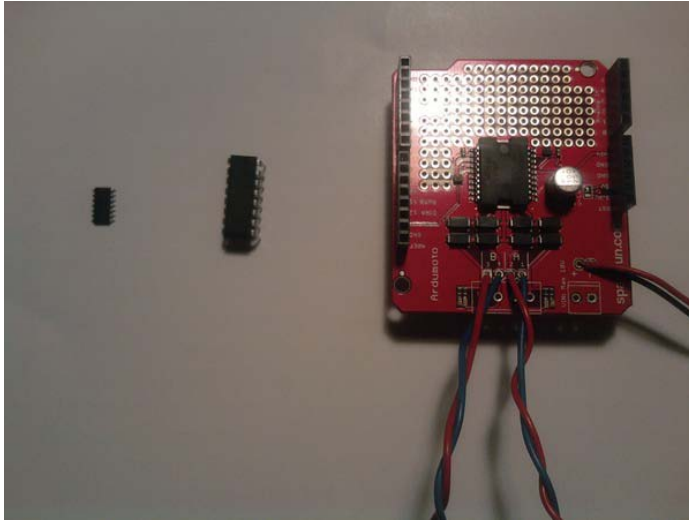


Figure 3-1. A few H-bridges (from left to right: surface mount H-bridge, through hole H-bridge, and motor shield from SparkFun)

The SparkFun motor shield has two motor ports, port A and port B; these ports correspond to four pins on the motor shield from SparkFun. They are digital pin 3: PWMA, which controls the speed of motor A, digital pin 12: DIRA, which controls the direction of motor A, digital pin 11: PWMB, which controls the speed of motor B, and digital pin 13: DIRB, which controls the direction of motor B. We will be using an H-bridge to control the speed and direction of motors. The motor shield from SparkFun has a built-in L298P H-bridge, which can handle up to 18V and uses digital pins 3, 11, 12, and 13 to control speed and direction. You can also use a stand-alone H-bridge such as the L293D, which has an input voltage of up to 37V. In this chapter, we will use the motor driver to control motors with a potentiometer and a switch, control speed, and control direction among other things.

Gathering the Hardware for this Chapter

We will be using the Dagou 2WD Beginner Robot Chassis V2 from Robotshop.com. Later in this chapter, the requirements document presented will have a specification for this chassis. This piece of hardware is going to be used to hold the circuitry, such as the Arduino and motor shield. You can find this chassis at www.robotshop.com/dagu-2wd-beginner-robot-chassis-v2-2.html.

I chose this chassis for its two-wheel design and the generous space on which it has to prototype. If you already have a chassis or you want to buy a different one, just make sure it is a two-wheel-drive chassis, as we will be using the two-wheel system throughout the rest of this book.

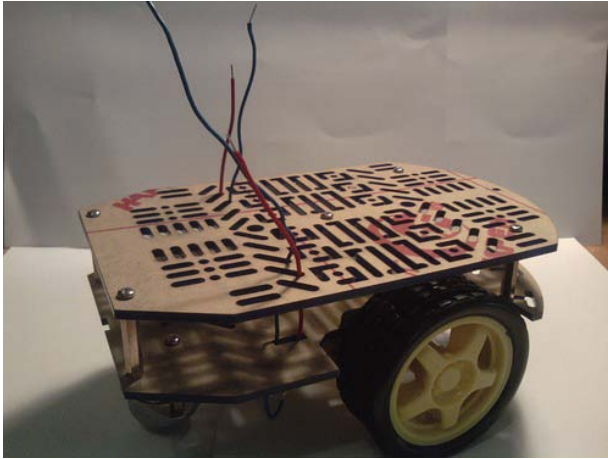


Figure 3-2. In this chapter, we will use Dagu 2WD Beginner Robot Chassis V2.

Understanding the Basics of Motor Control

In this section, we will be discussing the basics of motor control in four separate projects: turning on a motor with a button, controlling the speed of a motor with a potentiometer, controlling multiple motors with the Arduino, and finally, controlling speed and direction. These will form the foundation we use to build our project when we get the requirements document for a robot later in this chapter.

■ **Note** For this section, we will be using a solderless breadboard instead of a chassis (we use the chassis only for the last project in this chapter).

Project 3-1: Turning on a Motor with a Switch

This project will mostly focus on the digital pins and will use an H-bridge so we can use higher voltages to run the motors with the Arduino. Project 3-1 will allow us to turn on a motor on that moves in a clockwise direction.

Gathering the Hardware

Figure 3-3 shows the hardware for this project, minus the 9V battery, 9V connector, and solderless breadboard:

- Arduino Duemilanove (or UNO)
- Motor shield from SparkFun
- On/off switch

- 6V motor
- Solderless bread board
- 9V battery
- 9V battery connector
- Two terminal blocks from SparkFun
- Extra wire

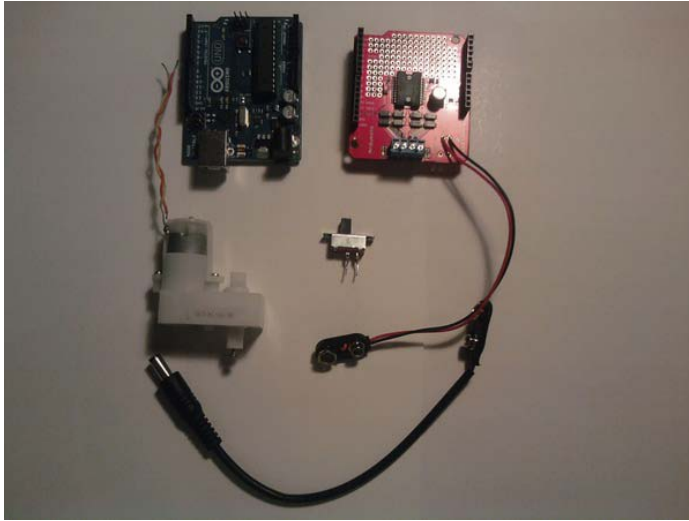


Figure 3-3. Hardware for this project

Configuring the Hardware

First, solder the female headers onto the motor shield and connect the motor shield to the female headers on the Arduino Duemilanove. Next, connect the A terminal on the motor shield to the 6V motor by soldering a terminal block to the motor shield and connecting your motor to that terminal block (Figure 3-4 shows this configuration). Finally, connect your on/off switch from ground to digital pin 10 on the Arduino, as Figure 3-5 shows. Figures 3-6 and 3-7 illustrate the hardware configuration for this project.

■ **Note** Both the 9V battery and the computer should be connected to get the best results, or you can hook a 9V battery to the power of the motor shield and the Arduino.

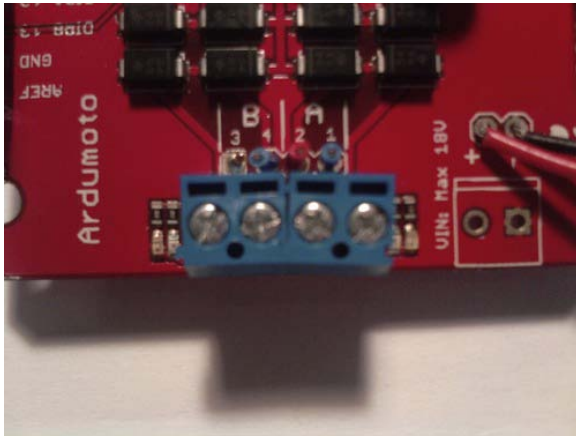


Figure 3-4. Terminal blocks added to the motor shield

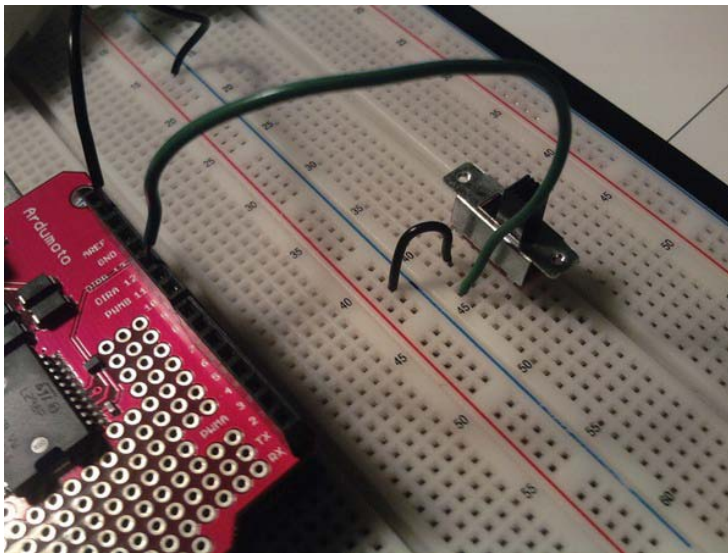


Figure 3-5. The switch is connected to digital pin 10 and ground on the Arduino.

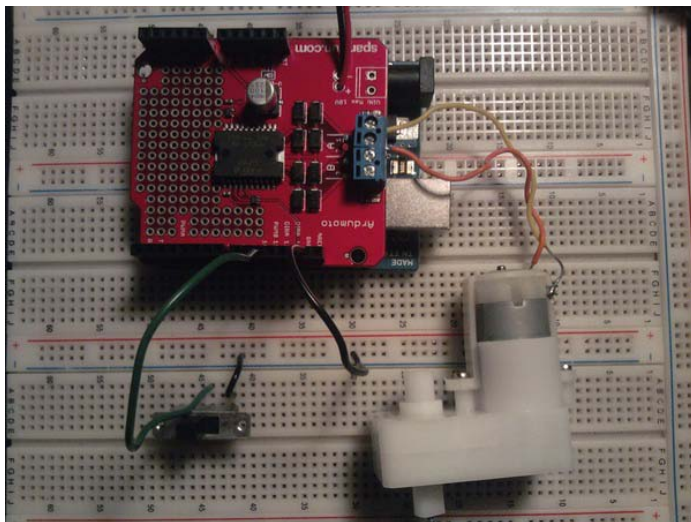


Figure 3-6. The motor is connected to the motor port A on the motor shield, and the switch is connected to digital pin 10 and ground on the Arduino.

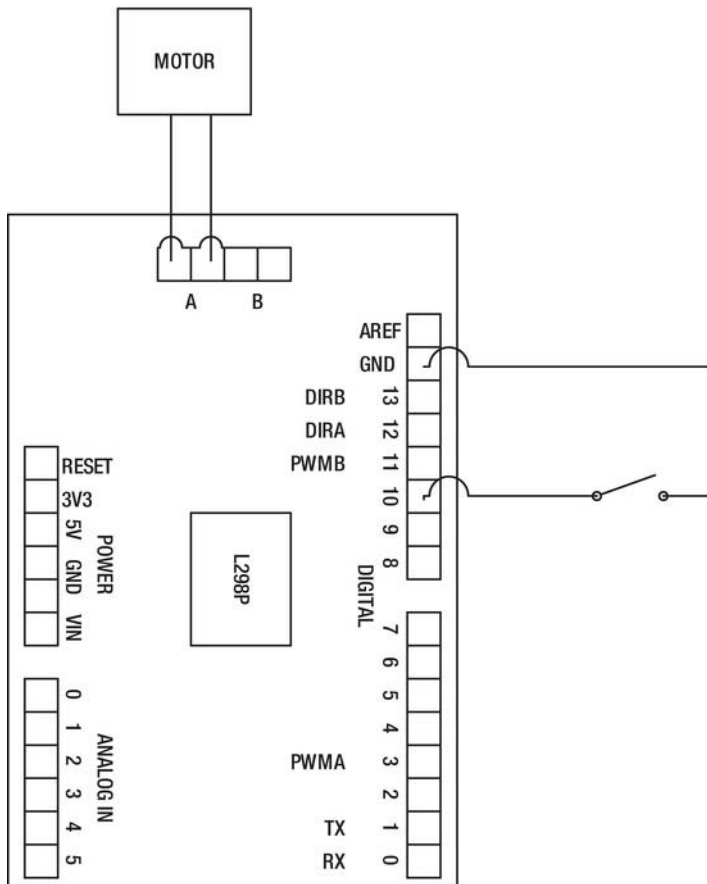


Figure 3-7. The schematic for this project

Now that you have connected the push button and motor, you need to connect your Arduino to your computer so that the software can be uploaded to the Arduino. We will write the program for this project in the next section.

Writing the Software

The software for this project will resemble the Blinking LED project that we created in Chapter 1. It will have a digital input, a digital output, and a conditional statement controlling that output. Listing 3-1 provides the code for this project.

Listing 3-1. Turning on a Motor

```
const int buttonPin = 10; // Sets buttonPin to digital pin 10
const int motorPin = 3; // Sets motorPin to digital pin 3
```

```

int buttonVal = 0; // Will pass button value here

void setup()
{
  pinMode(buttonPin, INPUT); // Makes buttonPin an input
  pinMode(motorPin, OUTPUT); // Makes motorPin an output
  digitalWrite(buttonPin, HIGH); // Activates the pull up resistor
                                // If we did not have this, the switch
                                // would not work correctly.
}

void loop()
{
  buttonVal = digitalRead(buttonPin); // Whatever is at buttonPin is
                                      // sent to buttonVal

  if (buttonVal == 1)
  {
    digitalWrite(motorPin, HIGH); // Turn motor on if buttonVal equals one
  }
  else
  {
    digitalWrite(motorPin, LOW); // Turns motor off for all other values
  }
}

```

This code first initializes `buttonPin` to pin 10, `motorPin` to pin 3, and `buttonVal` to 0 (not pin 0, just the integer value 0). Next in the `setup` structure, `buttonPin` is made an input; `motorPin` is made an output, and most importantly, we activate pin 10's *pull-up resistor*; this is an onboard resistor that can be called in the software by using a `digitalWrite()` function. For example, putting `digitalWrite(10, HIGH)` inside the `setup` structure activates pin 10's pull-up resistor (we could use an external resistor instead, but this way is easier). Finally, when we get inside the `loop` structure, we pass the `buttonPin` value to `buttonVal`. After that, we use an `if` statement with the condition (`buttonVal == 1`). If is the value does equal 1, write `HIGH` to `motorPin`, and for any other value, write `LOW` on `motorPin`.

Now, we can move on to another project that will help further your knowledge of motor control. The next section will discuss controlling the speed of a motor with a potentiometer.

Project 3-2: Controlling the Speed of a Motor with a Potentiometer

Using a potentiometer is a basic skill of motor control, and it will help you understand how to control speed in the final project of this chapter. This example project will also explain a function we have not gone over yet—the `map` function. We control the speed of a motor using analog inputs on the Arduino and scaling the values of a potentiometer (0 to 1024) to be used with the pulse width modulation (PWM) pins that the motor shield uses to control the speed of the motor. The next section will discuss the hardware we will need.

Gathering the Hardware

Figure 3-8 shows the hardware being used for this project, minus the 9V battery, 9V connector, and solderless breadboard:

- Arduino Duemilanove (or UNO)
- Motor shield
- 6V motor
- 10Kohm potentiometer
- 9V battery
- 9V battery connector
- Solderless bread board
- Extra wire (if needed)

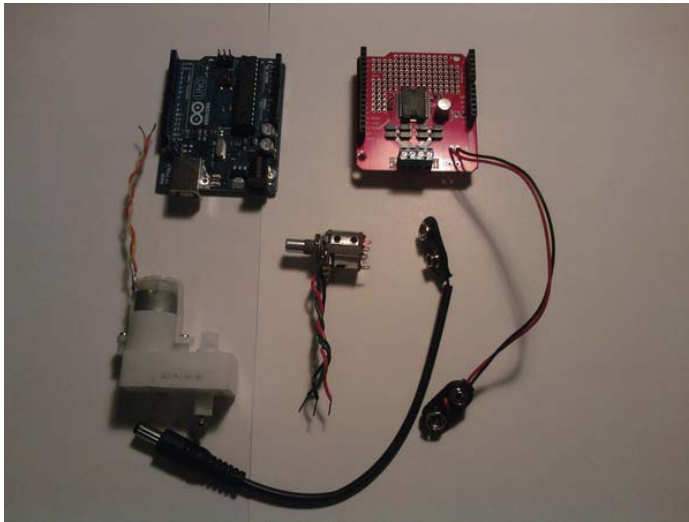


Figure 3-8. The hardware for this project

Configuring the Hardware

■ **Note** This book will sometimes refer to “potentiometer” as “pot.”

First, you will want to connect the motor shield to the top of the Arduino. Next, we need to connect the potentiometer to the Arduino. The middle wire of the potentiometer will connect to analog pin 0, and the first and last wires will be connected to the +5V pin and ground respectively. After you have the potentiometer connected, connect the motor to port A on the Motor Shield. Finally, connect the USB cable from the computer to the Arduino. See Figures 3-9 and 3-10 to ensure you have everything connected properly. If so, you're ready to move on to creating the software for this project.

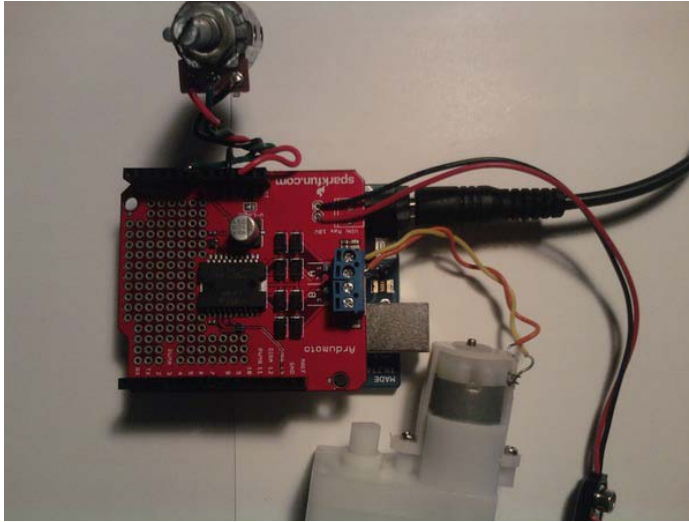


Figure 3-9. The motor is connected to motor port A on the motor shield, and the potentiometer is connected to analog pin 0, the power (+5V) pin, and ground on the Arduino.

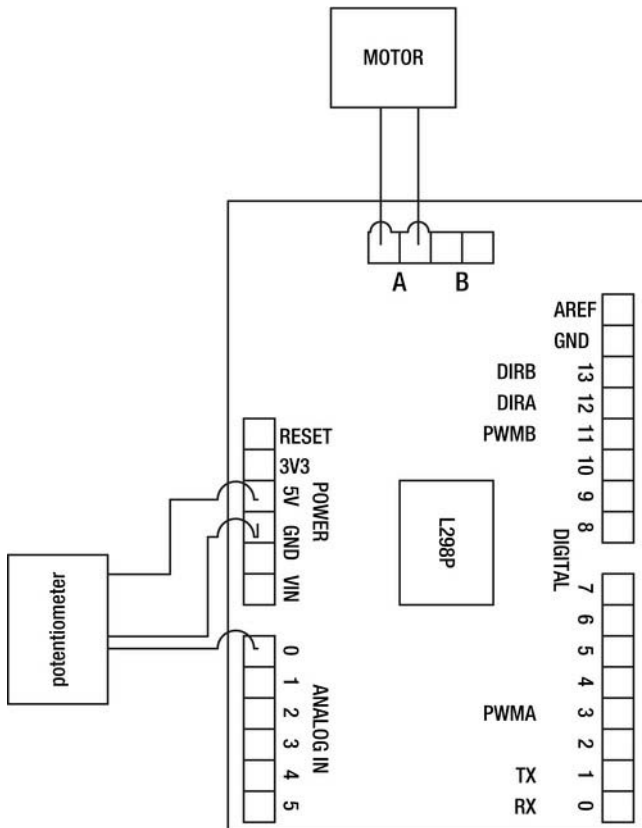


Figure 3-10. The schematic for this project

Writing the Software

We will be using analog pins and digital PWM pins for this project, so we will use `analogRead()` and `analogWrite()` functions to allow communication from the pot to the motor. Listing 3-2 provides the code for this project.

Listing 3-2. Controlling the Speed of the Motor

```
const int potPin = A0;    // A0 refers to analog pin 0
const int motorPin = 3;

int potVal = 0;
int mappedPotVal = 0;

void setup()
{
  pinMode(potPin, INPUT);
```

```

    pinMode(motorPin, OUTPUT);
}

void loop()
{
    potVal = analogRead(potPin);

    mappedPotVal = map(potVal, 0, 1023, 0, 255);

    analogWrite(motorPin, mappedPotVal);
}

```

This code starts off with the declarations `potPin = A0`, `motorPin = 3`, `potVal = 0`, and `mappedPotVal = 0`. Next, we move on to the setup structure, where `potPin` is set to an input, and `motorPin` is set to an output. After that, in the loop structure, `potVal` is set to equal the analog pin `potPin`. Finally, we scale the potentiometer data using the `map()` function:

```
map(potVal, 0, 1023, 0, 255);
```

This function scales the potentiometer value from 0 to 255, which is what the PWM uses as its value to control the speed of the motor.

■ **Note** The `map()` functions signature looks like this:

```
map(sensor Value, min sensor scale, max sensor scale, min scale to, max scale to);
```

After the code finishes scaling the potentiometer values, the scaled values are sent to the PWM pin 3, which was initialized as `motorPin`.

The next section will add to the first project you created in this chapter. We will be controlling multiple motors with buttons.

Project 3-3: Controlling Multiple Motors with the Arduino

In this section, we will discuss the application of controlling two motors with the Arduino and the motor shield. This will help us understand how the Arduino communicates with multiple motors using switches (or buttons). We will be using multiple motors in the final project of this chapter; the only difference is that we will use buttons to control the motors in this project.

Gathering the Hardware

Figure 3-11 shows the hardware being used for this project, minus the 9V battery, 9V connector, and solderless breadboard:

- Arduino Duemilanove (or UNO)

- Motor shield
- Two 6V motors
- Two buttons or switches
- Solderless bread board
- 9V battery
- 9V battery connector
- Extra wire

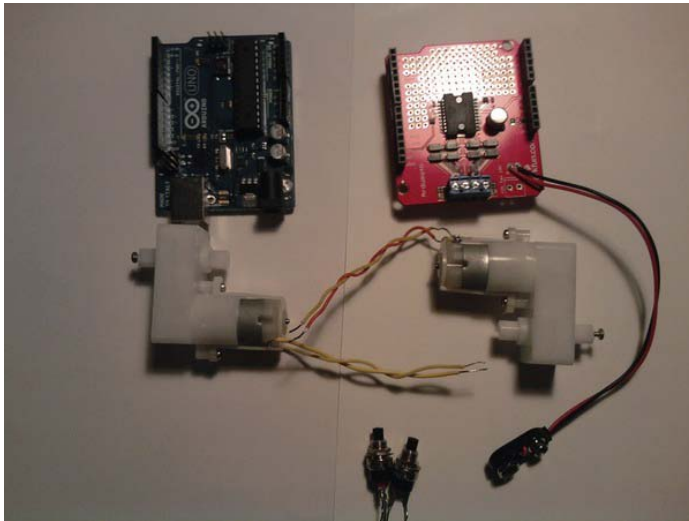


Figure 3-11. The hardware for this project

Configuring the Hardware

As in the first two examples, first connect the motor shield to the Arduino. In this example, there are two push buttons. The first of your buttons should have one wire connected to digital pin 9 and the other to ground. Next, connect one of the wires from the second button or switch to digital pin 10 on the Arduino; the other wire should be connected to ground. The motors are connected to digital pins 3 and 11 to take advantage of the PWM pins (we will not be using the PWM pins in this section; all you need to know is that the motors are connected to ports A and B on the motor shield). Figures 3-12, 3-13, and 3-14 illustrate the hardware setup.

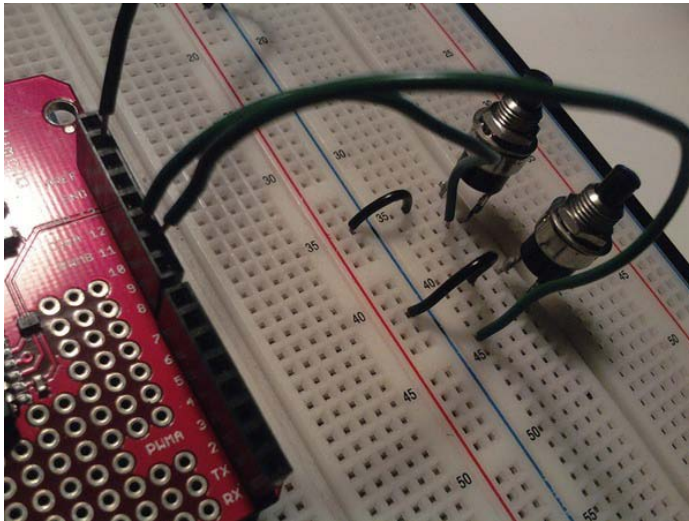


Figure 3-12. The push buttons are connected to digital pins 9 and 10; then, they are connected to ground on the Arduino.

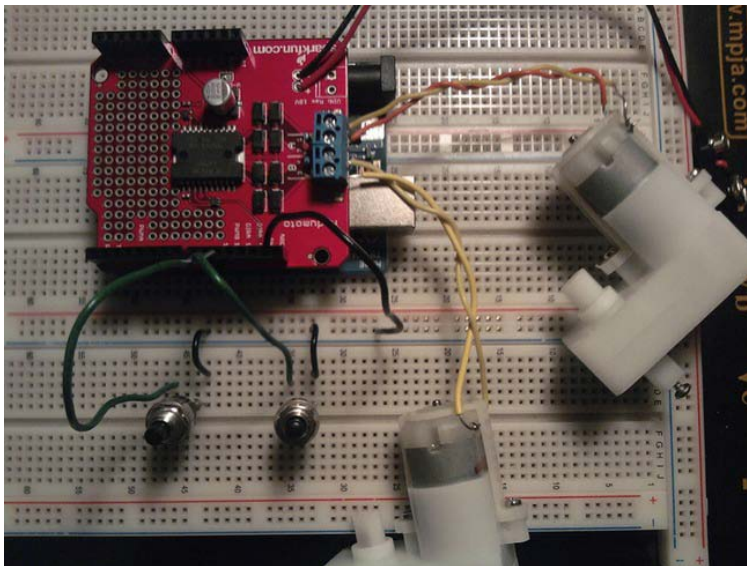


Figure 3-13. The motors are connected to motor ports A and B, and the switches are connected to digital pins 9 and 10 and ground.

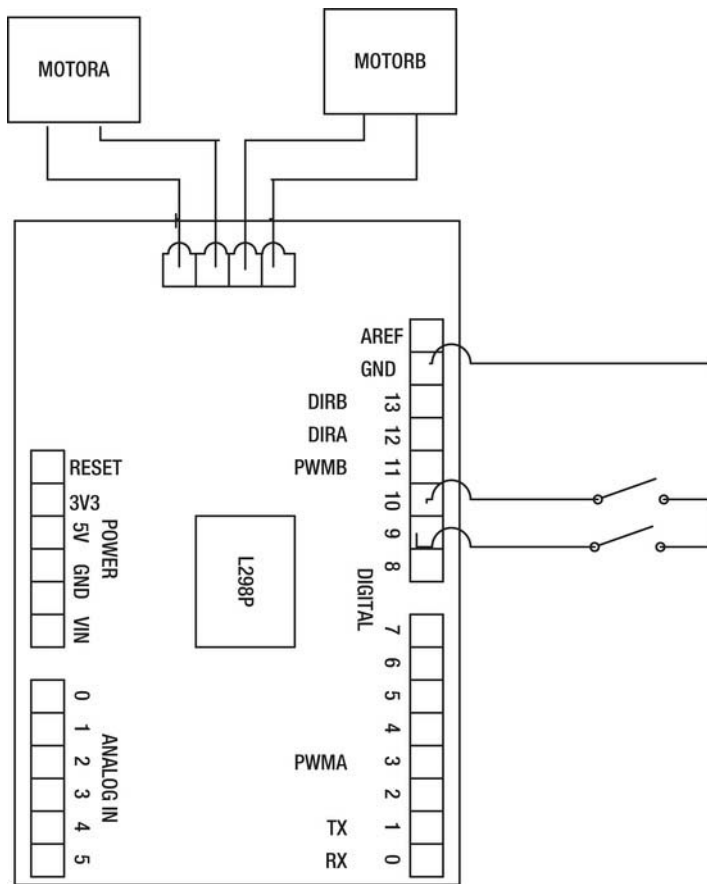


Figure 3-14. The schematic for this project

Writing the Software

This software will need to work with the two buttons we connected in the previous sections; we will use the `digitalWrite()` and `digitalRead()` functions to communicate with the Arduino. One button will turn on the motor at port A on the motor shield, and the other button will turn on the motor on port B of that shield. Listing 3-3 provides the code for this project.

Listing 3-3. Controlling Multiple Motors

```
const int button1Pin = 10; // Sets buttonPin to digital pin 10
const int motor1Pin1 = 3; // Sets motorPin to digital pin 3
const int button2Pin = 9; // Sets buttonPin to digital pin 9
const int motor2Pin = 11; // Sets motorPin to digital pin 11

int buttonVal1 = 0; // Will pass button value here
```

```

int buttonVal2 = 0; // Will pass button value here

void setup()
{
  pinMode(button1Pin, INPUT); // Makes buttonPin an input
  pinMode(motor1Pin, OUTPUT); // Makes motorPin an output
  pinMode(button2Pin, INPUT); // Makes buttonPin an input
  pinMode(motor2Pin, OUTPUT); // Makes motorPin an output

  digitalWrite(button1Pin, HIGH); // Activates the pull-up resistor
                                   // If we did not have this, the switch would not work
                                   // correctly.
  digitalWrite(button2Pin, HIGH); // Activates the pull-up resistor
                                   // If we did not have this, the switch would not work
                                   // correctly.
}

void loop()
{
  buttonVal1 = digitalRead(button1Pin); // Whatever is at buttonPin is sent to buttonVal1
  buttonVal2 = digitalRead(button2Pin); // Whatever is at buttonPin is sent to buttonVal2

  if (buttonVal1 == 1)
  {
    digitalWrite(motor1Pin, HIGH); // Turn motor on if buttonVal equals one
  }
  else
  {
    digitalWrite(motor1Pin, LOW); // Turns motor off for all other values
  }

  if (buttonVal2 == 1)
  {
    digitalWrite(motor2Pin, HIGH); // Turns motor off for all other values
  }
  else
  {
    digitalWrite(motor2Pin, LOW); // Turns motor off for all other values
  }
}

```

This code initializes all of the pins and `buttonVal1`, `buttonVal2` to 0. In the setup structure, we set each pin as an input if it is a button and an output if it is a motor. After we set up the pins, we need to activate the two pull up resistors for the buttons on the Arduino:

```

digitalWrite(buttonPin1, HIGH);
digitalWrite(buttonPin1, HIGH);

```

Next, we enter a loop structure where we set `buttonVal1` to the value on `buttonPin1` and `buttonVal2` to the value on `buttonPin2`. We then create two conditional statements that will send a high value to `motorPin1` if `buttonVal1` is equal to 1. Finally, in the other if statement, we say that if `buttonVal2` is equal

to one, the motorPin2 will turn on. That's it; this project should allow us to control multiple motors with the help of the H-bridge.

Now that we have completed this project, we can create another project that will help you understand how to control both speed and direction of a motor.

Project 3-4: Controlling Speed and Direction

Among the reasons an H-bridge is so useful are that it allows us to switch the direction of a motor and to control speed of the motor. This project will show you how to control a motor's speed and direction with a potentiometer and a switch.

Gathering the Hardware

Figure 3-11 shows the hardware being used for this project, minus the 9V battery, 9V connector, and solderless breadboard:

- Arduino Duemilanove (or UNO)
- Motor shield
- 10kohm potentiometer
- Switch (single-pole-single-throw switch)
- 6V motor
- 9V battery
- 9V battery connector
- Extra wire (if needed)

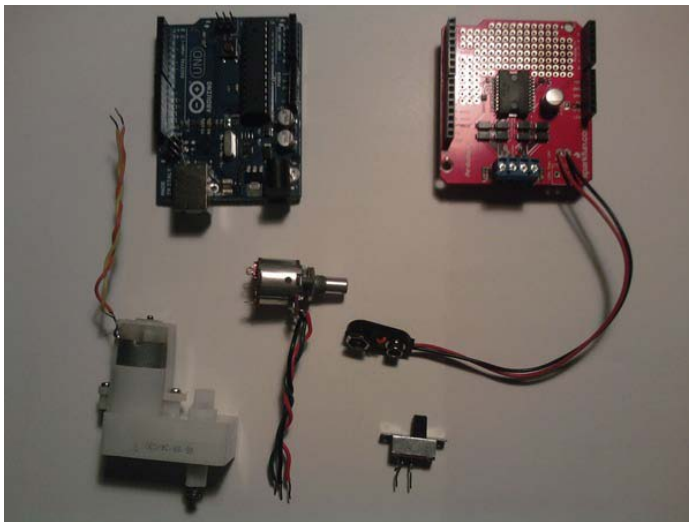


Figure 3-15. The hardware for this project

Configuring the Hardware

The first thing you need to do is connect the motor shield to the Arduino. Second, you will need to connect the potentiometer to analog pin 0, the +5V pin, and ground, like we did in the second project of this chapter. Next, you will need to connect the on/off switch from digital pin 10 to ground as you did in Project 3-1; this switch should be a single-pole-single-throw (SPST) switch. Finally, connect the USB from the computer to the Arduino. Figures 3-16, 3-17, and 3-18 show the configuration for this project.

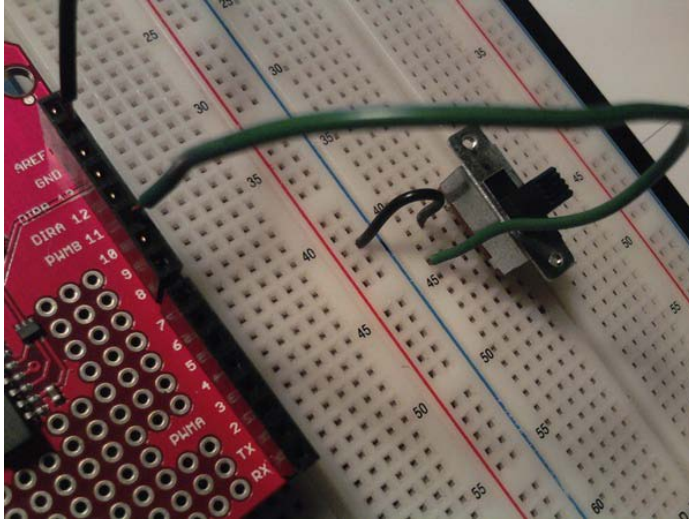


Figure 3-16. The switch is connected to digital pin 10 and ground on the Arduino.

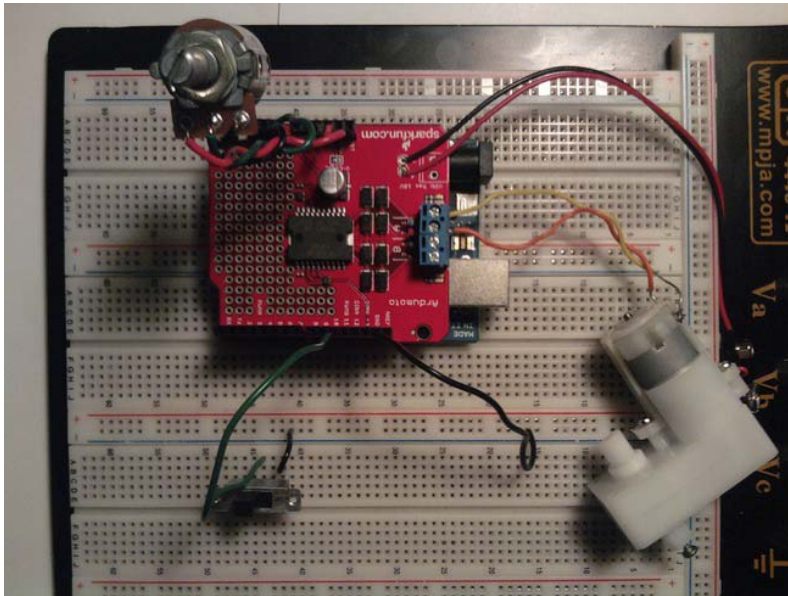


Figure 3-17. The motor is connected to motor port A, and the potentiometer is connected to analog pin 0, the +5V pin, and ground. Also, the switch is connected to digital pin 10 and ground.

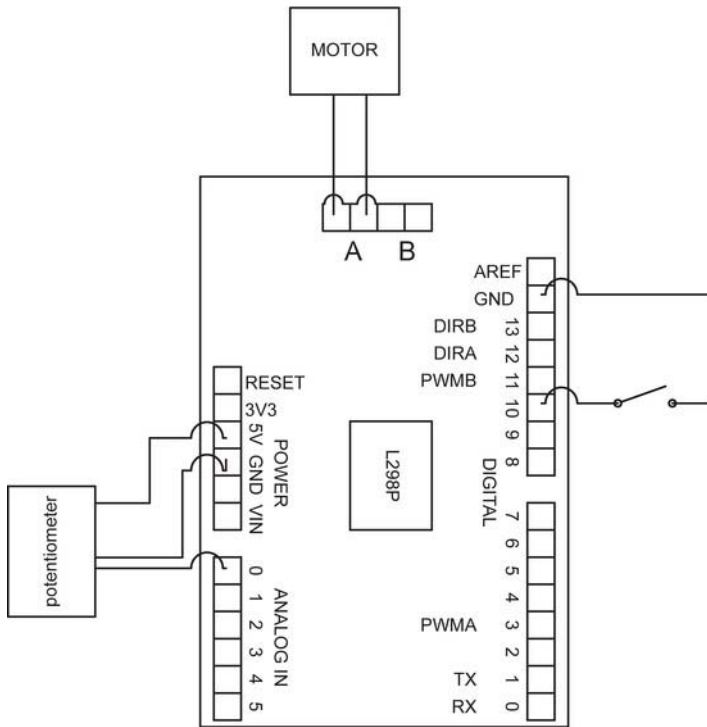


Figure 3-18. The schematic for this project

Writing the Software

In this project, we need to use both analog and digital pins. The analog pin will be used to control the speed of the motor, and the digital pin to control the direction of the motor (via the H-bridge and the hex inverter). The hex inverter inverts the signal of one of the pins, which will switch the polarity of the motor. For example, if the H-bridge is sent a value of 1, the hex inverter reads in a 0 and sends that to the other input pin on the H-bridge causing the motor to turn in a clockwise direction (for more information on this process, please see http://en.wikipedia.org/wiki/H_bridge). Listing 3-4 provides the code for this project.

Listing 3-4. Controlling Speed and Direction

```
const int switchPin = 10;
const int potPin = A0;
const int motorPin = 3;

int switchVal = 0;
int potVal = 0;
int mappedPotVal = 0;
```

```

void setup()
{
  pinMode(switchPin, INPUT);
  pinMode(potPin, INPUT);
  pinMode(motorPin, OUTPUT);
  digitalWrite(switchPin, HIGH);
}

void loop()
{
  switchVal = digitalRead(switchPin);
  potVal = analogRead(potPin);

  mappedPotVal = map(potVal, 0, 1023, 0, 255);

  analogWrite(motorPin, mappedPotVal);

  if (switchVal == 1)
  {
    digitalWrite(12, HIGH);  // direction (clockwise) of motor A port
  }
  else
  {
    digitalWrite(12, LOW);   // direction (counter-clockwise) of motor A port
  }
}

```

First, this code initializes `switchPin`, `potPin`, `motorPin`, `switchVal`, `potVal`, and `mappedPotVal`. Then, in the setup structure, we set `switchPin` and `potPin` to inputs and `motorPin` to an output. Next, we activate the pull-up resistor with the `digitalWrite()` function. After that, in the loop structure, we set `switchVal` equal to the reading on `switchPin` and set the value of `potVal` to the reading on `potPin`. Next, we set `mappedPotVal` equal to the scaled values 0 to 255 using the `map()` function; `mappedPotPin` is then sent to `motorPin`. Finally, we control the motor with a conditional `if` statement. If `switchVal` is equal to 1, the motor will turn clockwise; otherwise, the motor will turn counterclockwise.

Project 3-5: Controlling Motors with Serial Commands

Now that you understand the basics of motor control, we can visit our example company to see if it has any projects for us to complete that require using the Arduino to control motors. It does! So our first steps are gathering the requirements and creating the requirements document.

Requirements Gathering

The customer has set up a meeting and has several requirements for a robot controlled by the Arduino using serial communication; the Arduino will drive two motors with the help of a motor shield. The client's project also requires that the user send motor's parameters in a comma-separated format to the serial monitor (shown in Figure 3-19) as follows:

```
1,255,1,255
```

In this format, the first parameter is the direction of motor A; the second parameter is the speed of motor A; the third is the direction of motor B, and the forth is the speed of motor B. The serial monitor displays the information in this format:

```
Motor A
1
255
Motor B
1
255
```

■ **Note** Comma-separated format is a very common way for data to be passed and collected from the Arduino and many other peripherals.

The company wants each of the comma-separated parameters to go to these specific pins. The pins are: 12, 3, 13, and 11. Another requirement is that the final prototype needs to be configured on a chassis once the hardware and software have been tested.

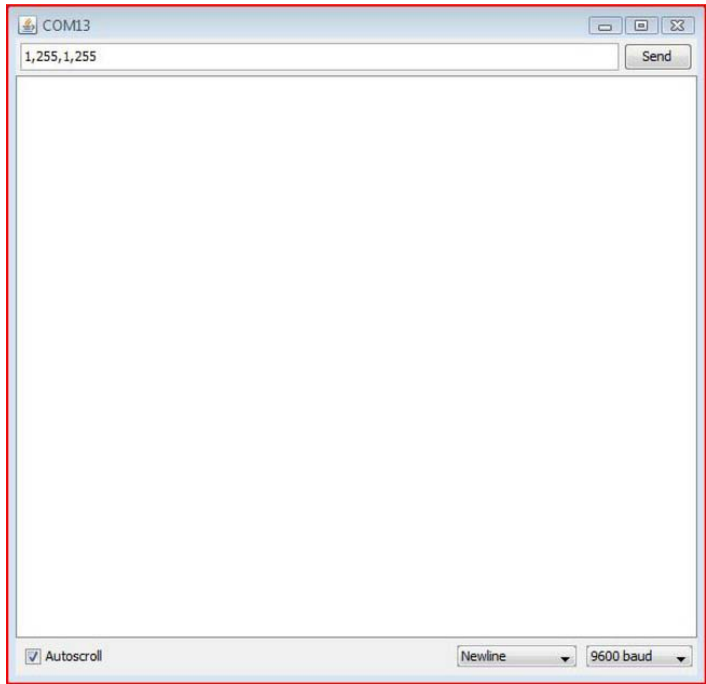


Figure 3-19. The user will type values for the direction and speed of the motors into the serial monitor shown here.

■ **Note** When using the serial monitor, make sure that newline is selected in the Line Ending parameter.

Now that you have notes for Project 3-5, we can configure them into a requirements document.

Gathering the Hardware

Figure 3-20 shows the following hardware, which you'll need for this project:

- Arduino Duemilanove (or UNO)
- Motor shield
- Two 6V motors
- Solderless breadboard
- Robot chassis (for use only after the hardware and software has been tested)
- USB cable
- 9V battery

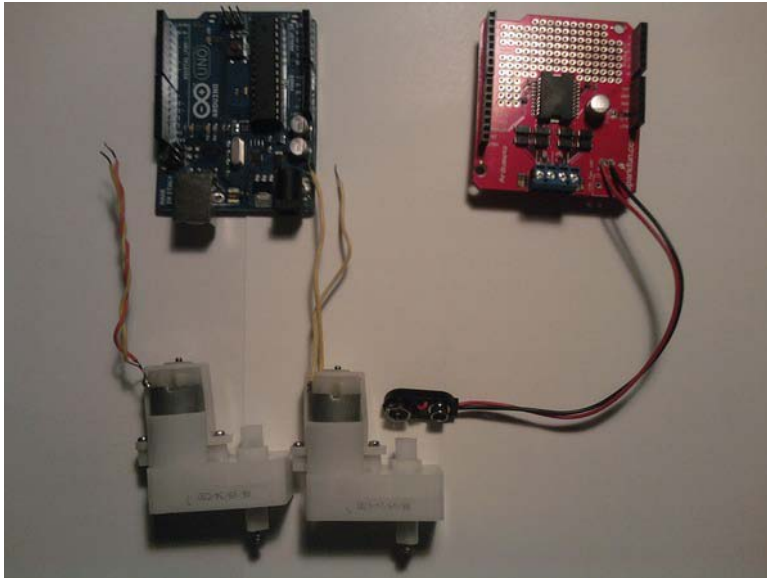


Figure 3-20. Hardware for this project

Outlining the Software Requirements

Following are the software requirements:

- Create a program that sends comma-separated data to the Arduino to control the speed and direction of two motors. The user should enter the data in this format:

1,255,1,255

- The first and third parameters in the comma-separated values are the direction of motors A and B, and the second and forth parameters are the speeds of motors A and B.
- The serial monitor should output the data in this format:

Motor A

1

255

Motor B

1

255

- The overall purpose of this program is to control the speed and direction of two 6V motors.

Now that we have the hardware and software requirements, we can create the software's flowchart. Figure 3-21 shows the flowchart for this project.

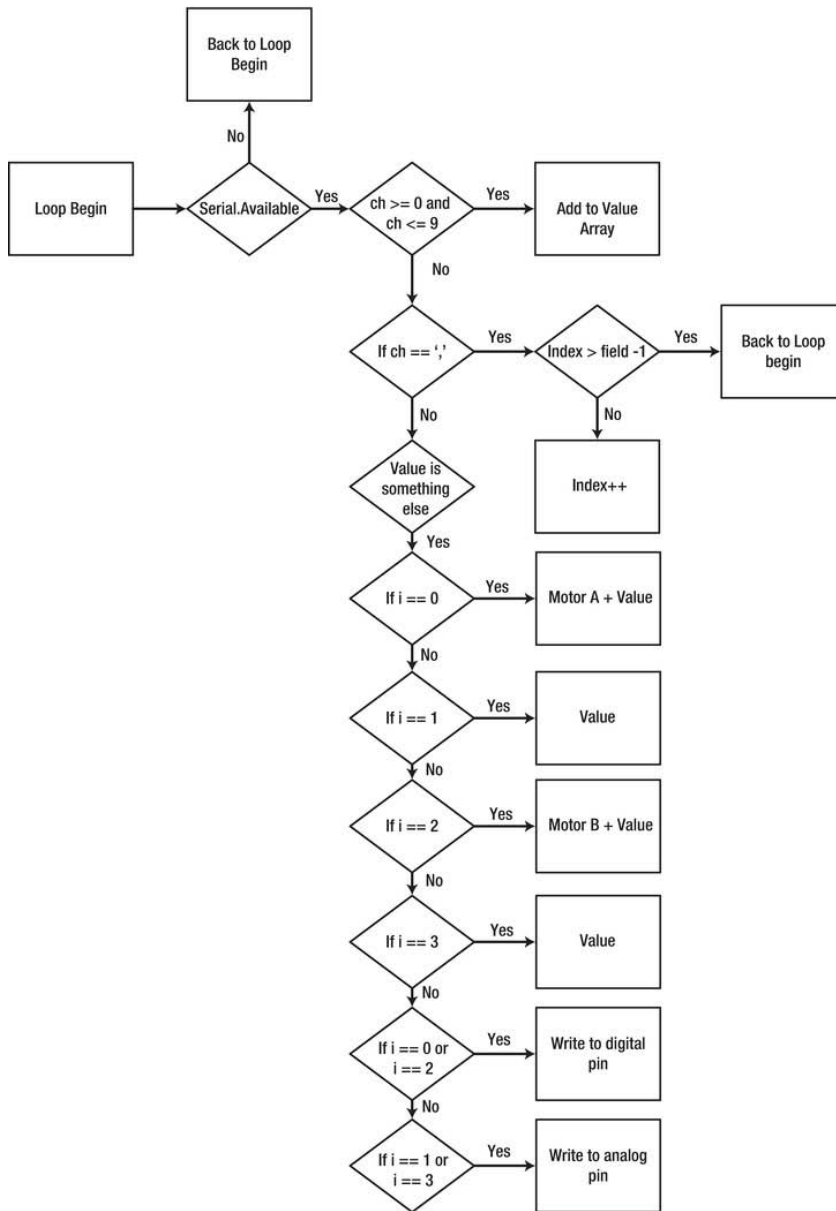


Figure 3-21. Flowchart for this project

Configuring the Hardware

The configuration of this project is to first connect the Arduino to the motor shield; next, connect the two motors to ports A and B on the motor shield. Finally, connect the USB to the Arduino and computer. Figures 3-22 and 3-23 illustrate the hardware configuration.

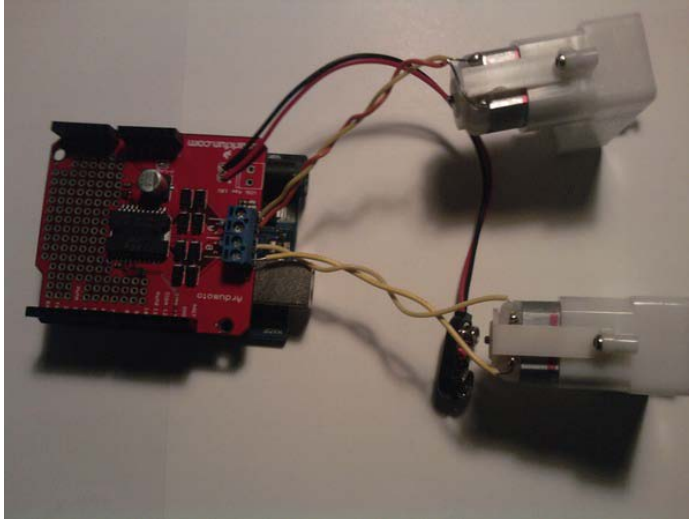


Figure 3-22. The motors are connected to motor ports A and B on the motor shield.

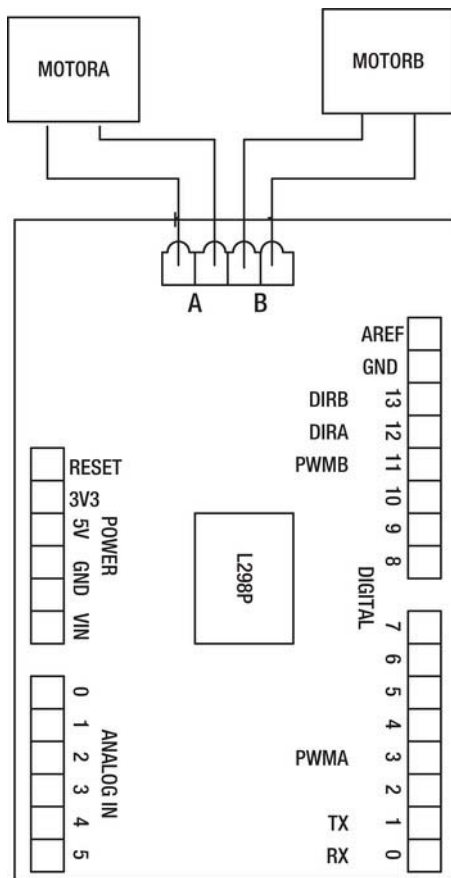


Figure 3-23. The schematic for this project.

Writing the Software

Now, we will move on to the software for this project. We need to communicate with both digital and analog pins. Unlike in our previous projects, here we will be interfacing data by means of serial communication, so we have to send in multiple sets of data, specifically the direction of motor A, speed of motor A, direction of motor B, and speed of motor B. We need to use comma-separated format to parse the data to their respective digital or analog pins. After that, we need to display the data on the serial monitor in this format:

```
Motor A
1
255
Motor B
1
255
```

Listing 3-5 shows the code.

Listing 3-5. *Code for the Client's Project*

```
const int fields = 4; // amount of values excluding the commas
int motorPins[] = {12,13,3,11}; // Motor Pins
int index = 0; // the current field being received
int values[fields]; // array holding values for all the fields

void setup()
{
    Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud

    for (int i; i <= 3; i++) // set LED pinMode to output
    {
        pinMode(motorPins[i], OUTPUT);
    }

    Serial.println("The Format is: MotoADir,MotoASpe,MotorBDir,MotoBSpe\n"); // \n is a new
                                                                    // line constant that will output a new line
}

void loop()
{

    if( Serial.available())
    {
        char ch = Serial.read();
        if(ch >= '0' && ch <= '9') // If it is a number 0 to 9
        {
            // add to the value array and convert the character to an integer
            values[index] = (values[index] * 10) + (ch - '0');
        }
        else if (ch == ',') // if it is a comma increment index
        {
            if(index < fields -1)
                index++; // increment index
        }
        else
        {
            for(int i=0; i <= index; i++)
            {
                if (i == 0)
                {
                    Serial.println("Motor A");
                    Serial.println(values[i]);
                }
                else if (i == 1)
                {
                    Serial.println(values[i]);
                }
            }
        }
    }
}
```

```

    }
    if (i == 2)
    {
        Serial.println("Motor B");
        Serial.println(values[i]);
    }
    else if (i == 3)
    {
        Serial.println(values[i]);
    }

    if (i == 0 || i == 1) // If the index is equal to 0 or 2
    {
        digitalWrite(motorPins[i], values[i]); // Here we see a logical error
    }
    if (i == 2 || i == 3) // If the index is equal to 1 or 3
    {
        analogWrite(motorPins[i], values[i]); // Here we see a logical error
    }

    values[i] = 0; // set values equal to 0
}
index = 0;
}
}
}

```

Notice that the code will run—with unexpected results. Look at the initialization of `motorPins` and you'll see that the array is out of order with the format we were given: motor A direction, motor A speed, motor B direction, motor B speed. This is one of those pesky logical errors, and it brings us to the next section, debugging the Arduino software.

Debugging the Arduino Software

Now that we have discovered the logical error, we need to fix it. Listing 3-6 contains the corrected array in bold.

Listing 3-6. Corrected Code for Project 3-5

```

const int fields = 4; // amount of values excluding the commas.
int motorPins[] = {12,3,13,11}; // Motor Pins
int index = 0; // the current field being received
int values[fields]; // array holding values for all the fields

void setup()
{
    Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud

    for (int i; i <= 3; i++) // set LED pinMode to output
    {
        pinMode(motorPins[i], OUTPUT);
    }
}

```

```

    }

    Serial.println("The Format is: MotoADir,MotoASpe,MotorBDir,MotoBSpe\n");
}

void loop()
{

if( Serial.available())
{
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9') // If the value is a number 0 to 9
    {
        // add to the value array
        values[index] = (values[index] * 10) + (ch - '0');
    }
    else if (ch == ',') // if it is a comma
    {
        if(index < fields -1) // If index is less than 4 - 1
            index++; // increment index
    }
    else
    {

        for(int i=0; i <= index; i++)
        {
            if (i == 0)
            {
                Serial.println("Motor A");
                Serial.println(values[i]);
            }
            else if (i == 1)
            {
                Serial.println(values[i]);
            }
            if (i == 2)
            {
                Serial.println("Motor B");
                Serial.println(values[i]);
            }
            else if (i == 3)
            {
                Serial.println(values[i]);
            }

            if (i == 0 || i == 2) // If the index is equal to 0 or 2
            {
                digitalWrite(motorPins[i], values[i]); // Write to the digital pin 1 or 0
                                                         // depending what is sent to the Arduino.
            }
            if (i == 1 || i == 3) // If the index is equal to 1 or 3

```

```

        {
            analogWrite(motorPins[i], values[i]); // Write to the PWM pins a number between
                                                    // 0 and 255 or what the person entered
                                                    // in the serial monitor.
        }

        values[i] = 0; // set values equal to 0
    }
    index = 0;
}
}
}

```

At this point, I want to discuss the finer details of this code, as we have gone over similar code before. The first thing I want to point out is where we parse the data to be sent to the correct pins:

```

if(ch >= '0' && ch <= '9') // If the value is a number 0 to 9
{
    // add to the value array
    values[index] = (values[index] * 10) + (ch - '0');
}
else if (ch == ',') // if it is a comma
{
    if(index < fields - 1) // If index is less than 4 - 1
        index++; // increment index
}
else
    // This is where the data is passed to the digital and analog pins

```

This part of the code first checks to see if an input character from 0 to 9 exists. If so, it converts the character type to an integer type by subtracting by 48, which has an integer value of 48, and tells the microcontroller to see this value as an integer instead of a character. Next, it checks to see if the character is a comma. If so, it will check to see if the index is greater than or equal to 3. If the value is less than 3, it will increment the index value. The if-elseif statement handles any other values such as numerical values, which is what the characters are converted to.

Next, I would like to discuss the parsing of the data to the digital and analog pins and how we formatted the data on the serial monitor. The code looks like this:

```

for(int i=0; i <= index; i++)
{
    if (i == 0)
    {
        Serial.println("Motor A");
        Serial.println(values[i]);
    }
    else if (i == 1)
    {
        Serial.println(values[i]);
    }
    if (i == 2)
    {
        Serial.println("Motor B");
        Serial.println(values[i]);
    }
}

```

```

    }
    else if (i == 3)
    {
        Serial.println(values[i]);
    }

    if (i == 0 || i == 2) // If the index is equal to 0 or 2
    {
        digitalWrite(motorPins[i], values[i]); // Write to the digital pin 1 or 0
                                                // depending what is sent to the Arduino.
    }
    if (i == 1 || i == 3) // If the index is equal to 1 or 3
    {
        analogWrite(motorPins[i], values[i]); // Write to the PWM pins a number between
                                                // 0 and 255 or what the person entered
                                                // in the serial monitor.
    }
}

```

The for loop iterates through all of the indexes (in this case, 0–3). The first and second if statements and if-else-if statements are printing the data to the serial monitor, which is where we get the format:

```

Motor A
1
255
Motor B
1
255

```

Do you see an easier way of programming this format? (Hint: use switch.) After those if statements, we come to the code that separates the data to its appropriate pin, which is what the company asked for, so the format this code accepts is motor A direction (0 or 1), motor A speed (0 to 255), motor B direction (0 or 1), and motor B speed (0 to 255). Now that we have the software sorted out, we can focus on testing the hardware.

Troubleshooting the Hardware

The beautiful thing about using a prototype shield is that it normally has no problems, but in the small chance you do have issues, such as if your motors turn the incorrect way or move at incorrect speeds, here's what you should check. For the first case, make sure your motors are connected correctly. If they are not, switch the terminals of the motors and see if they move in the correct direction. If the motors are still moving in the wrong direction, make sure your code is correct.

■ **Note** If you need to, copy and paste the code from the “Debugging the Arduino Software section” to the Arduino IDE, to make sure everything is correct.

Finished Prototype

The company's requirements document says we need to have a chassis for this project to be complete. Since we have tested both the hardware and software, we can now mount the hardware on a chassis so the robot can move. Figures 3-24 and 3-25 illustrate the finished prototype.



Figure 3-24. The finished prototype

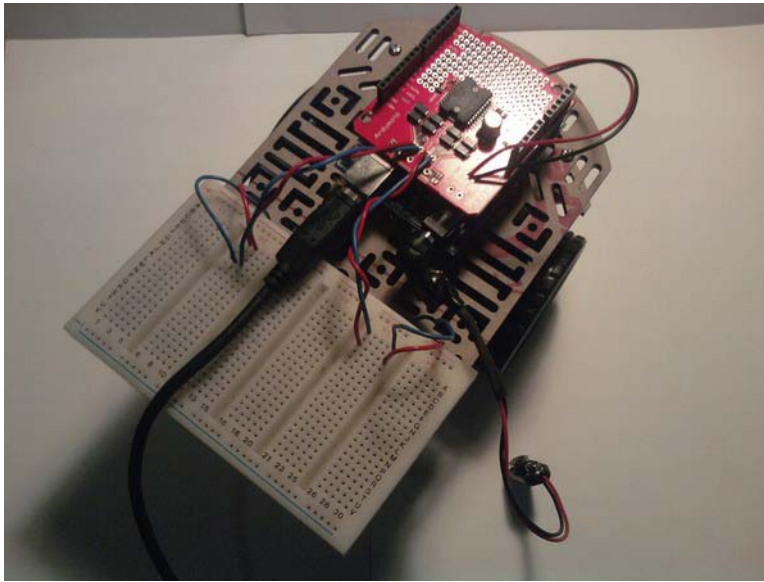


Figure 3-25. Top view of the finished prototype

■ **Note** Try to keep pressure off of the motors' connectors while you build the chassis and attach the wires to the Arduino; if you put too much pressure on the connectors, you may break them.

Summary

In this chapter, we discussed motor control and how it can be used to give a robot life. To do this, you first had to understand what an H-bridge is and how it can be used to drive motors. Next, we went over the basics of motor control by building several projects. After that, we focused on using the engineering process to complete a project to a client's specifications. In this project, we controlled multiple motors using a serial monitor to update the motor settings (i.e., motor speed and direction). You also learned about the comma-separated data format and how it is used to parse data. Now that this project is done, you can go out in the world and use motors on whatever project you deem worthy.

Adding Complexity to the Robot: Working with LCDs

If motors give your robot life, LCDs give your robot character. In this chapter, we will be working toward the goal of adding another piece of hardware to the previous chapter's robot. In a hypothetical professional scenario, the customer might like to have a color LCD that displays in plain English which direction the robot is moving. Let's assume that customer wants you to use the SparkFun color LCD shield for the prototype. But because the motor and color LCD shields share some of the same pins (a conflict that arises often), we'll need to create a workaround for that issue.

To do so, we'll first need to take a look at the LiquidCrystal library and the ColorLCDShield library. We will also take a look at the setup of a monochrome LCD. After that, we will go through four monochrome and color LCD projects: displaying multiple sensor values on a monochrome LCD, creating a menu on a monochrome LCD, creating a slot machine on the color LCD from SparkFun, using a keypad to communicate with the SparkFun color LCD.

We'll then be able to add to the robot we created in the previous chapter to satisfy our customer's request. But before we get to the projects, let's take a look at the configuration for the color LCD from SparkFun.

Configuring a Color LCD Shield

SparkFun ships its color LCD shield with two types of LCDs mounted on it: Phillips and Epson. You can tell them apart by the color of the small square right next to the color LCD shield (it is connected to the LCD as well). Once you determine which LCD you're using, you need to download the source code for that shield. You can find the code in the Source Code/Download area of the Apress web site at www.apress.com. Next, add the PhillipColorLCD or EpsonColorLCD folder (thanks to SparkFun, Peter Davenport, and Coleman Sellers for creating these) to the Libraries directory in the Arduino-022 folder. After that, you should be ready to use the color LCD shield from SparkFun. If you are having difficulty, Listing 4-1 should help you figure out which LCD you have.

Listing 4-1. Indicates Whether You Have an Epson or a Phillips Color LCD

```
#include <ColorLCDShield.h>

ColorLCDShield lcd;

void setup()
{
```

```

lcd.init(EPSON);
lcd.contrast(40);
lcd.clear(WHITE);

lcd.setStr("Epson", 50, 40, BLACK, WHITE);
}
void loop()
{
}

```

This code will display “Epson” in the middle of the screen if your color LCD is an Epson LCD. If it is, you need to add the `EpsonColorLCDShield` folder to the `Libraries` directory in the `Arduino-022` folder. If your LCD is an Epson, the code will display nothing or bad data. If it displays nothing or bad data, you have a Phillips color LCD, and you need to add the `PhillipsColorLCDShield` folder to the `Libraries` directory in the `Arduino-022` folder.

Now that we have the library added for the color LCD shield, let’s move on to discussing the new hardware for this chapter.

Introducing the Monochrome and Color LCD Shields

Projects 4-1 and 4-2 will be using the monochrome LCD. This LCD is great if you want to display sensor values or a small menu. Figure 4-1 shows the monochrome LCD we will be using. This particular Monochrome LCD has a resolution of 16×2 pixels, which is plenty of space for us to work with. It contains the Hitachi HD44780 driver, which works with the `LiquidCrystal` library.

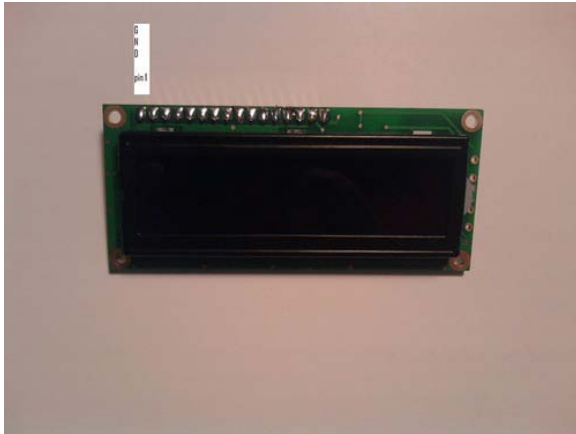


Figure 4-1. An example of a monochrome LCD with pin 1 labeled as ground

The best configuration for a monochrome LCD is an inline setup, because inline monochrome LCD works well with a solderless breadboard. Figure 4-2 shows the schematic for a monochrome LCD.

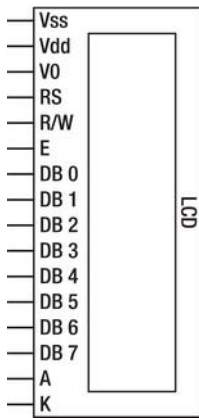


Figure 4-2. Example of an inline monochrome LCDs schematic

■ **Note** It is always a good idea to use a potentiometer on the V0 pin to adjust the contrast on the monochrome LCD.

This monochrome LCD has the HD44780 driver built into it. This driver is very common, so if you have a monochrome LCD, you may be in luck and already have the monochrome LCD you will need for this chapter; if not, you can purchase a monochrome LCD on most electronic web sites such as SparkFun (www.sparkfun.com).

Now, we need to discuss the SparkFun color LCD shield. This shield has a built-in Nokia 6100 LCD with a resolution of 128×128 pixels, and it has three buttons on the LCD. The buttons on this shield are connected to digital pins 3, 4, and 5 on the Arduino. As explained in the previous section, SparkFun ships these shields with either a Phillips or Epson color LCD. The color LCD shield will use digital pins 13, 11, 9, and 8 on the Arduino. This shield will be used in Projects 4-3 through 4-5. Figure 4-3 shows the color LCD shield from SparkFun.

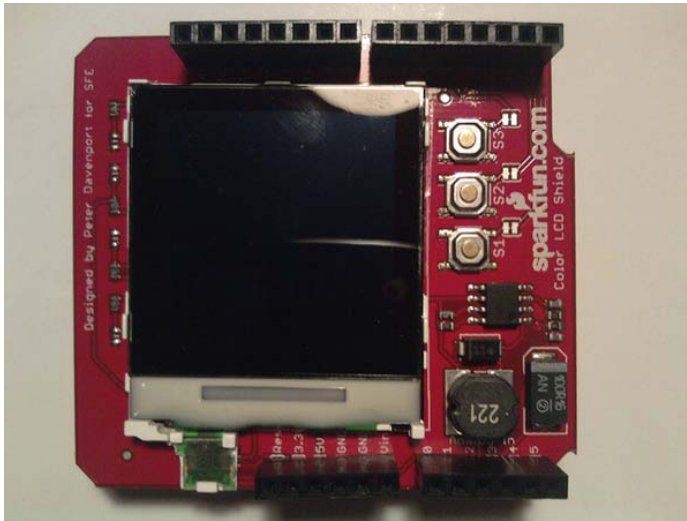


Figure 4-3. ColorLCD Shield

The next section will explain the two libraries needed to communicate with monochrome and color LCDs.

Working with the LiquidCrystal and ColorLCDShield (Epson or Phillips) Libraries

The LiquidCrystal library is used in conjunction with a monochrome LCD, while the ColorLCDShield library allows us to display images and text on the color LCD. The following two sections, we'll look at the two new libraries that we will use to create several projects that display information such as a sensor value or menu.

Using the LiquidCrystal Library

The LiquidCrystal library commands that we will discuss are `LiquidCrystal()`, `begin()`, `clear()`, `home()`, `setCursor()`, `write()`, `print()`, `blink()`, and `noBlink()`.

`LiquidCrystal()` creates a variable of type `LiquidCrystal` so that the LCD can be used in the program. The format looks like this:

```
LiquidCrystal name(rs,enable,D7,D6,D5,D4);
```

And this is how this function is used:

```
#include <LiquidCrystal.h>
```

```
LiquidCrystal lcd(13,12,11,10,5,4);
```

```
void setup()
{
```

```

}
void loop()
{
}

```

LiquidCrystal `begin()` is used just like `Serial.begin()` except you don't put a baud rate within the parentheses; instead, you put the number of columns and rows. For example the Monochrome LCD that is used in this chapter is a 16×2 monochrome LCD, so the `lcd.begin()` function would look like this:

```
lcd.begin(16,2);
```

And here is how it is used:

```

#include <LiquidCrystal.h>

LiquidCrystal lcd(13,12,11,10,5,4);

void setup()
{
    lcd.begin(16,2);
}
void loop()
{
}

```

LiquidCrystal `clear()` is used when you want to clear the monochrome LCD. Here is the format of the `clear()` function:

```
lcd.clear();
```

■ **Note** It is usually a good idea to put a `clear()` function inside the `setup` structure, because it will clear the screen before any new data is added to the monochrome LCD so data is not erased or written on top of other data.

LiquidCrystal `home()` is used to set the cursor to the top-left corner:

```
lcd.home();
```

LiquidCrystal `setCursor()` will set the cursor wherever you want:

```
lcd.setCursor(col,row);
```

LiquidCrystal `write()` writes a single characters to the monochrome LCD:

```
lcd.write(char data);
```

LiquidCrystal `print()` prints text to a monochrome LCD. The `data` argument is required argument and takes a data type as its value (`string`, `int`, `char`, etc.), but the `base` argument is an optional argument that takes the value `BIN`, `DEC`, `HEX`, or `OCT`:

```
lcd.print(string data, [base]); // Base is optional
```

For example, if you wanted to print “A” in hexadecimal, the function would read like this:

```
lcd.print("A", HEX);
```

LiquidCrystal blink() causes the cursor to blink:

```
lcd.blink();
```

LiquidCrystal noBlink() is the inverse of the blink() function, meaning it will stop the cursor from blinking:

```
lcd.noBlink();
```

Now that you have the basics of the LiquidCrystal library, we can take a look at the ColorLCDShield library, which is used in conjunction with the color LCD shield that we will use later in this chapter.

ColorLCDShield Library

We will be using the ColorLCDShield library in Projects 4-3 and 4-4 and in the final project, so let's take a look at how it works too. Some of the functions we will discuss in this section are init(), setPixel(), setLine(), setRect(), setCircle(), setStr(), and contrast(). Also, note that for every program in which you use the ColorLCDShield library, you need to include a few files at the beginning of the program, like this:

```
#include <ColorLCDShield.h> // for either library you will need to include this library.
```

```
LCDSshield lcd;
```

```
void setup()
{
```

```
    lcd.init(EPSON);
```

```
}
```

lcd.init() tells the library which color LCD is present on the color LCD shield. To use it, you need to supply EPSON in the parenthesis if you have an Epson color LCD or PHILLIPS if you have a Phillips. All of these functions will begin with lcd as that is the name we gave it in the previous code:

```
    LCDSshield lcd;
```

```
lcd.init(PHILLIPS)
```

lcd.setPixel() allows us to put a single pixel on the color LCD and to set the color of that:

```
lcd.setPixel(color,x,y);
```

lcd.setLine() draws a line on to the color LCD and allows us to set the color of that line:

```
lcd.setLine(x0,y0,x,y,color);
```

lcd.setRect() draws a rectangle onto the color LCD and allows the rectangle to be filled:

```
lcd.setRect(x0,y0,x,y,fill,color);
```

lcd.setCircle() draws a circle on the color LCD and allows you to set the circle's color:

```
lcd.setCircle(xcenter,ycenter,radius,color);
```

`lcd.setStr()` writes text, which can be colored, to the color LCD:

```
lcd.setStr(String,x,y,fontcolor,backgroundcolor);
```

`lcd.contrast()` controls the contrast of the color LCD and expects a value from 0 to 63:

```
lcd.contrast(contrast);
```

Now that we have gone over both of the libraries that we will use in this chapter, the next section will discuss the basics of LCD control using both the monochrome LCD and the color LCD shields from SparkFun.

Exploring the Basics of LCD Control

This section teaches you the basic functions of an LCD. After you've learned some of the basics, we will add on to the robot we created in Chapter 3. The projects in this section cover displaying multiple sensor values on a monochrome LCD, creating a menu on the monochrome LCD, creating a slot machine on the color LCD, and using a keypad to communicate with the color LCD.

Project 4-1: Displaying Multiple Sensor Values

This project will be using both digital and analog pins to write sensor values to a monochrome LCD, we will be using the LiquidCrystal Library in conjunction with a 16×2 monochrome LCD. Then we will program the Arduino to send sensor values to the monochrome LCD.

Gathering the Hardware

Here's a list of the hardware in this project (see Figure 4-4):

- Arduino Demilanove (or UNO)
- Monochrome LCD
- Two 1Kohm to 10Kohm potentiometers (or sensors)
- 10Kohm potentiometer
- USB cable
- Extra wire
- Solderless breadboard



Figure 4-4. Hardware for this project (Not pictured: the USB cable, extra wire, and solderless breadboard)

Configuring the Hardware

Before configuring the hardware, we will need to go over some basic terminology for the LCD: RS, E, R/W, and D0–D7. The RS pin is used to tell the LCD what will be written to it. The E pin is used to tell the LCD that the data is ready to be written. The R/W pin is the read/write pin; this pin is normally configured to ground (because we are writing to the LCD). Pins D0–D7 are the bus lines for the Monochrome LCD; this is where your data is being passed from the Arduino to the LCD.

Now, we can connect it to the solderless breadboard:

1. Connect the 10Kohm potentiometer (this potentiometer is used to control the contrast of the LCD). The wiper (middle pin of potentiometer) of the potentiometer is connected to pin 3 (V0) on the LCD, and the other two wires are connected to +5V and ground.
2. Connect the backlight; it is controlled by pins 15 and 16. Connect pin 15 to power (+5V), and connect pin 16 to ground.
3. Connect the data bus lines (you can use all eight bus lines, but only four are necessary) as follows: D7 is connected to digital pin 12. D6 is connected to digital pin 11. D5 is connected to digital pin 10, and D4 is connected to digital pin 9.
4. Connect the RS pin on the LCD to digital pin 7 on the Arduino.
5. Connect the R/W pin to ground.
6. Connect the E pin on the LCD to digital pin 8 on the Arduino.
7. Next, connect +5V to pin 2 on the LCD and ground to pin 1 on the LCD.

Now that the LCD is connected, we need to connect the two potentiometers (or sensors) to the Arduino.

8. First, connect the first potentiometer wiper to analog pin 1, and connect the other two wires to power (+5V) and ground.
9. Then, connect the second potentiometers wiper to analog pin 2 and the other two to power (+5V) and ground. Figures 4-5 and 4-6 illustrate the hardware configuration.

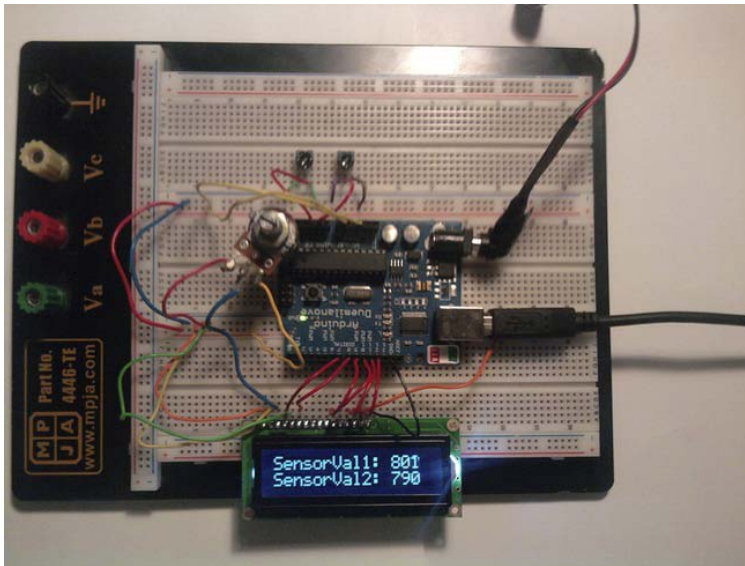


Figure 4-5. Hardware configuration for this project

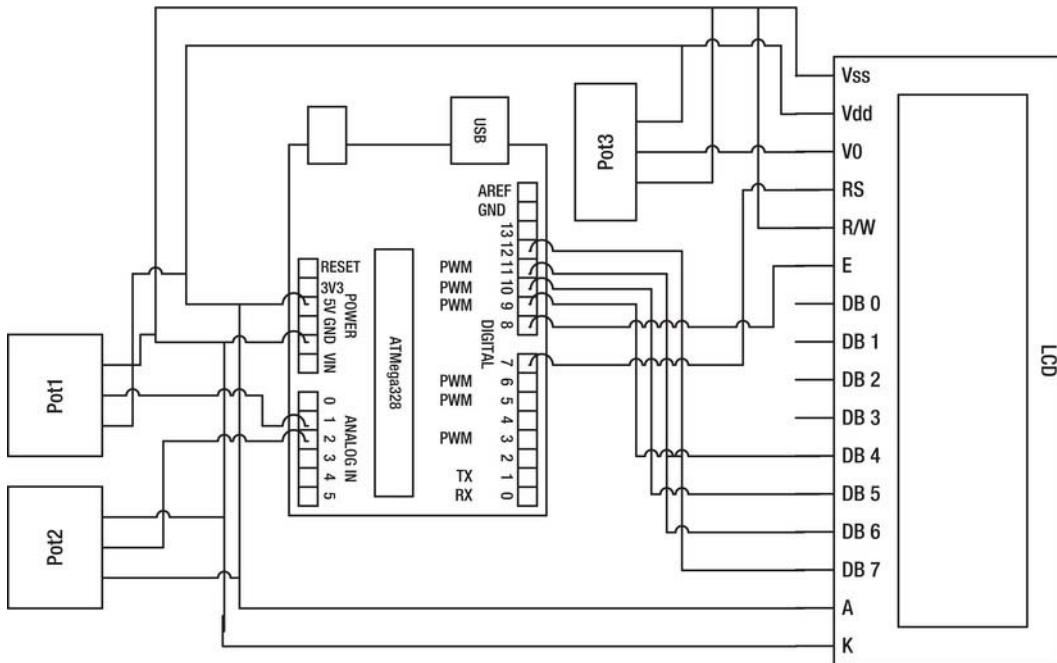


Figure 4-6. Schematic for this project

Now that we have finished the hardware configuration, we need to write the software.

Writing the Software

The software will need to communicate with the analog pins and the LCD. Because we are using the LiquidCrystal library, we don't have to directly communicate with the digital pins. Listing 4-2 provides the code for this project

Listing 4-2. Displaying Multiple Sensor Values on a Monochrome LCD

```
// include the library code:
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7,8,9,10,11,12);

int potPin1 = A1;
int potPin2 = A2;

void setup()
{
    // set up the LCD's number of columns and rows:
```

```

lcd.begin(16, 2);
lcd.clear();

pinMode(potPin1, INPUT);
pinMode(potPin2, INPUT);
}

void loop()
{
    lcd.setCursor(0,0); // Sets the cursor to col 0 and row 0
    lcd.print("SensorVal1: "); // Prints Sensor Val: to LCD
    lcd.print(analogRead(potPin1)); // Prints value on Potpin1 to LCD
    lcd.setCursor(0,1); // Sets the cursor to col 1 and row 0
    lcd.print("SensorVal2: "); // Prints Sensor Val: to LCD
    lcd.print(analogRead(potPin2)); // Prints value on Potpin1 to LCD
}

```

The first thing this code does is include the Liquid Crystal library. Next, it creates an instance of the `LiquidCrystal` type by using this piece of code:

```
LiquidCrystal lcd(7,8,9,10,11,12);
```

After that, we initialize both of the potentiometers' analog pins. In the setup structure, we set up the LCD to be a 16×2 LCD and clear the LCD. After that, we set up the potentiometers to be inputs.

In the loop structure, we first set the cursor to be at the top left corner. Then we print "SensorVal1:" and the value from analog pin 1. Next, we set the cursor to the second row and print "SensorVal2 " and the value from analog pin 2. Now that you know how to send values to the LCD, let's make our LCD more organized by creating a menu.

■ **Note** Project 4-2 has the same monochrome LCD setup as this one, so if you do not take apart your project, you will not have to do as much hardware configuration in the next project.

Project 4-2: Creating a Menu on the Monochrome LCD

This project will use the LCD and potentiometers as the previous project did; the only difference is that we will also need to use a button to toggle through the potentiometer values.

Gathering the Hardware

Here's the hardware for this project (see Figure 4-7):

- Arduino Demilanove (or UNO)
- Monochrome LCD

- Two potentiometers (or sensors)
- A normally off push button
- 10Kohm potentiometer
- USB cable
- Solderless breadboard
- Extra wire

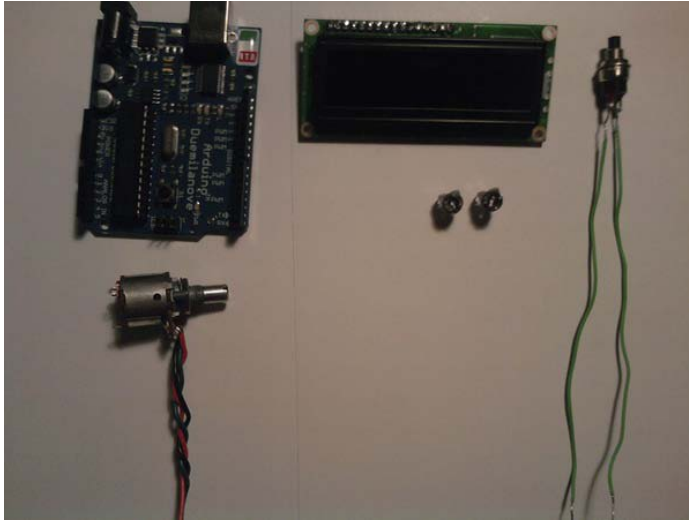


Figure 4-7. Hardware for this project (Not pictured: the USB cable, extra wire, and solderless breadboard)

Configuring the Hardware

To configure the hardware in Project 4-2, follow these steps:

1. Connect the monochrome LCD to the solderless breadboard.
2. Connect power (+5V) to pin 2 on the LCD and ground to pin 1 on the LCD.
3. Connect the 10Kohm potentiometer's wiper (middle wire on pot) to pin 3 on the LCD; the other two wires are connected to power (+5V) and ground.
4. Connect the RS, R/W, E, and bus lines from the Arduino to the LCD. The RS pin is connected to digital pin 7; the R/W pin is connected to ground; the E pin is connected to digital pin 8, and pins DB7–DB4 are connected to digital pins 12–9 on the Arduino.
5. Connect power (+5V) to pin 15 on the LCD and ground to pin 16 on the LCD.
6. Now that the LCD is connected, you need to connect the push button. The push button is connected to digital pin 2 and ground.

7. Then, connect the potentiometers' wipers to analog pins 1 and 2; connect the other wires on the potentiometers to power (+5V) and ground.

Figures 4-8 and 4-9 illustrate the hardware configuration for this project.

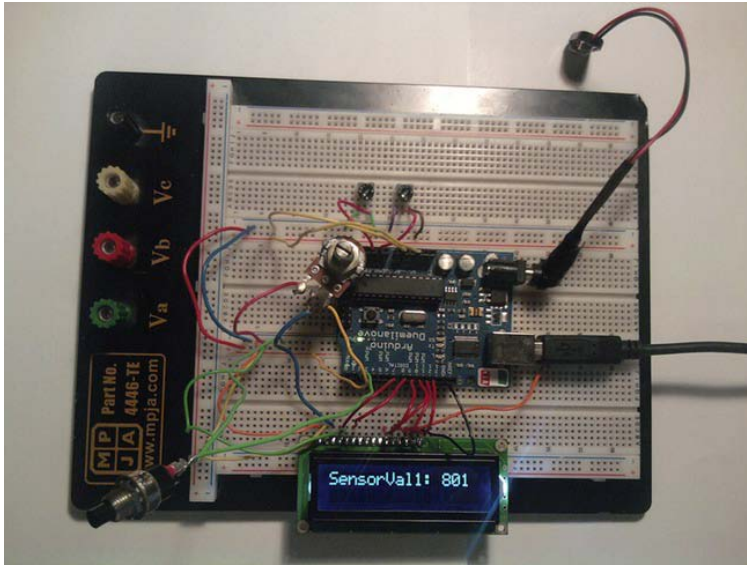


Figure 4-8. Hardware configuration for this project

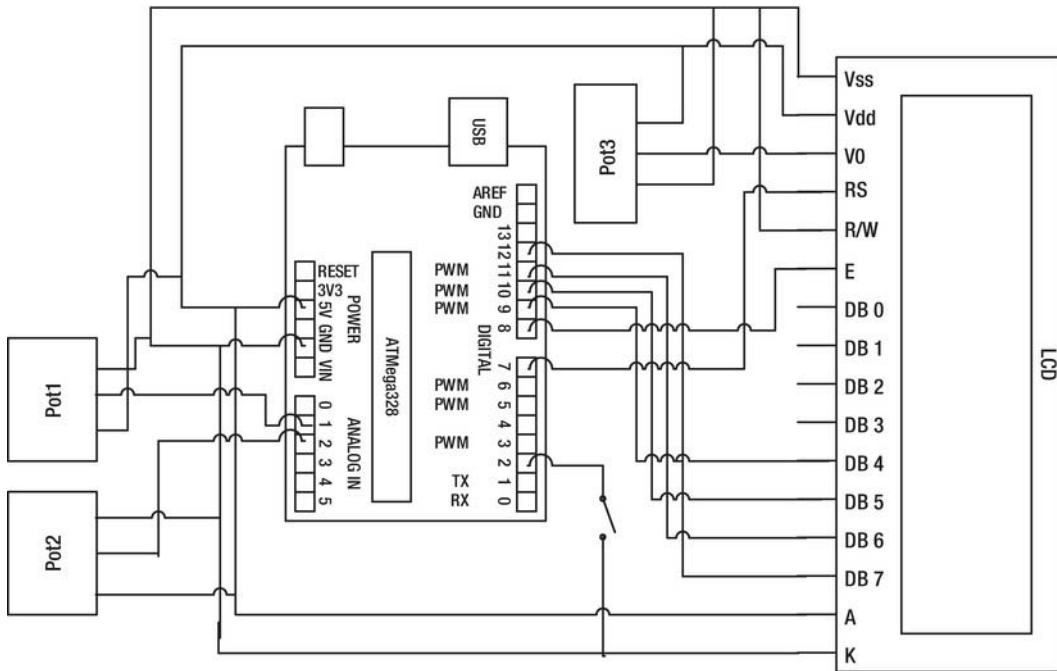


Figure 4-9. Schematic for this project

Now that we have the hardware configured, we need to write the software to make a menu on the LCD that is controlled by a push button.

Writing the Software

This project will need to communicate with both analog and digital pins. The analog pins will send the potentiometer values to the Arduino and eventually the monochrome LCD. The push button in this project will be used to control the menu of pot values. Listing 4-3 provides the code for this project.

Listing 4-3. Menu System for a Monochrome LCD

```
// include the library code:
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7,8,9,10,11,12);

int potPin1 = A1;
int potPin2 = A2;
int button1 = 2;
```

```

int buttonVal1 = 0;

int menuCount = 0;

void setup()
{
    // set up the LCD's number of columns and rows:
    lcd.begin(16, 2);
    lcd.clear();

    pinMode(potPin1, INPUT);
    pinMode(potPin2, INPUT);
    pinMode(button1, INPUT);

    digitalWrite(button1, HIGH);

    lcd.setCursor(0,0); // Sets the cursor to col 0 and row 0
    lcd.print("Press button "); // Prints to the LCD
    lcd.setCursor(0,1); // Sets the cursor to col 0 and row 0
    lcd.print("to view pot val");
}

void loop()
{
    buttonVal1 = digitalRead(button1);

    if (buttonVal1 == LOW)
    {
        menuCount++;
    }
    switch (menuCount)
    {
        case 1:
            lcd.setCursor(0,0); // Sets the cursor to col 0 and row 0
            lcd.print("SensorVal1: "); // Prints Sensor Val: to LCD
            lcd.print(analogRead(potPin1)); // Prints value on Potpin1 to LCD
            lcd.setCursor(0,1); // Sets the cursor to col 1 and row 0
            lcd.print(" "); // Prints Sensor Val: to LCD
            delay(250); // Increasing the delay here will increase the button delay
            break;
        case 2:
            lcd.setCursor(0,0); // Sets the cursor to col 0 and row 0
            lcd.print("SensorVal2: "); // Prints Sensor Val: to LCD
            lcd.print(analogRead(potPin2)); // Prints value on Potpin1 to LCD
            lcd.setCursor(0,1); // Sets the cursor to col 1 and row 0
            lcd.print(" "); // Prints Sensor Val: to LCD
            delay(250);
            break;
    }
}

```

```

    Case 3:
    lcd.setCursor(0,0); // Sets the cursor to col 0 and row 0
    lcd.print("Press button "); // Prints to the LCD
    lcd.setCursor(0,1); // Sets the cursor to col 0 and row 0
    lcd.print("to view pot val");
    delay(250);
    break;
    default:
    menuCount = 0;
    break;

}

}

```

The first thing this code does is include the `LiquidCrystal` library. After that, it creates an instance of `LiquidCrystal` and set it to `lcd(7,8,9,10,11,12)`. Next, it initializes both pots to analog pins 1 and 2. After that, the code initializes both `buttonVal1` and `menuCount` to 0.

In the setup structure, we set the LCD to be 16×2 pixels, and we clear the screen. Next, we set both of the potentiometers and push button to inputs, and we activate pin 2's pull up resistor. Then, we set the cursor to the top-left corner of the LCD and write "Press button " (the spaces are there to put spaces on the LCD). After that, we set the cursor to the bottom-left corner and write "to view pot val" to the LCD.

Next, we enter the loop structure. The first thing we do here is set `buttonVal1` to the digital read on pin 2. Then, we create an if statement that has the condition `buttonVal1 == LOW` (this is equal to LOW because we had to set up the pull up resistor in the setup structure). If this is true, `menuCount` will increment. If it is not equal to LOW (meaning that the button is not pressed), it will do nothing. After the if statement, we have a switch statement with three cases and a default. The first case displays the value of the first potentiometer if `menuCount` is equal to 1. The second case displays the value of the second potentiometer if `menuCount` is equal to 2. The third case displays "Press the button to see pot val" if `menuCount` is equal to 3. The default case sets `menuCount` to 0.

That's it for the monochrome LCD. From now on (at least in this chapter), we will be using the color LCD shield and the LCD library to complete projects.

Project 4-3: Creating a Slot Machine with the Color LCD Shield

In this project, we will use the color LCD shield rather than a monochrome LCD. This means we will also be using a different library to control the color LCD. The next section will discuss the hardware we will need for this project.

Gathering the Hardware

Here's the hardware for this project (see Figure 4-10):

- Arduino Demilanove (or UNO)
- Color LCD shield
- USB cable

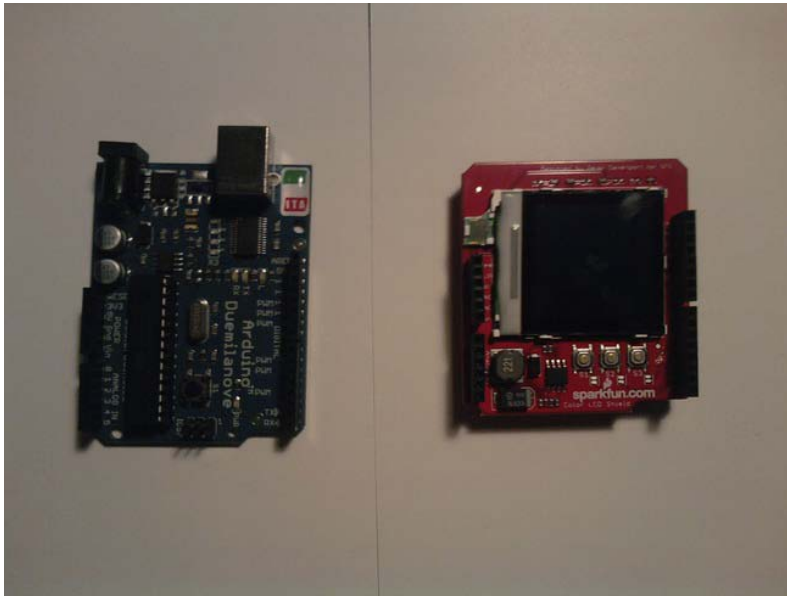


Figure 4-10. Hardware for this project (Not pictured: USB cable)

Configuring the Hardware

Configuring the hardware is the easy part of this project; all you need to do is connect the color LCD shield to the Arduino. Figure 4-11 illustrates the hardware configuration.

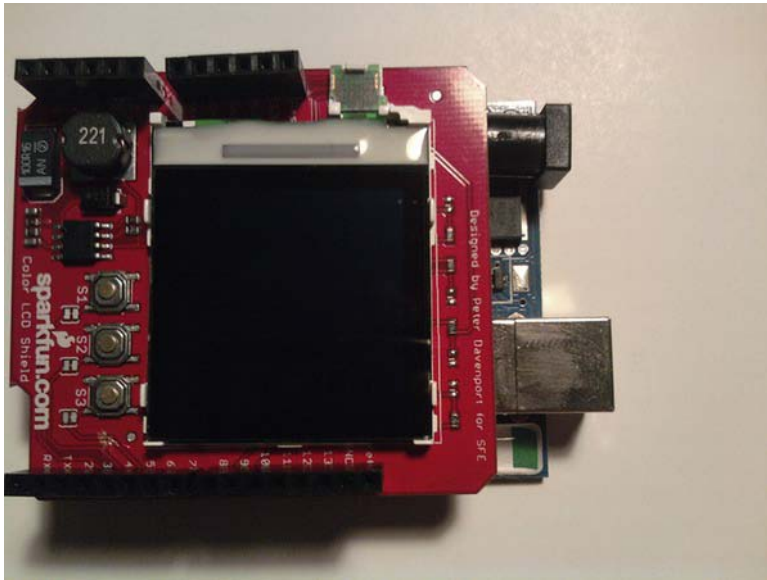


Figure 4-11. Hardware configuration for this project

Writing the Software

Well, if the hardware setup is simple, you know the software will probably be a bit more complicated. We will have to set up one of the buttons that is already on the shield. Listing 4-4 shows the code for this project.

■ **Note** If you are using a Phillips color LCD, you will need to change `lcd.init(EPSON);` to `lcd.init(PHILLIPS);` to allow you to use the Phillips color LCD.

Listing 4-4. Arduino Slot Machine

```
#include <ColorLCDShield.h>

LCDShield lcd;

int Money = 10000;
int button = 3;

void setup()
{
```

```

    pinMode(button, INPUT);
    digitalWrite(button, HIGH);

    lcd.init(EPSON); //Epson. Remember that if you are using a Phillips LCD
                     // you need to change the inside of the init() function to PHILLIPS.
    lcd.contrast(40);
    lcd.clear(WHITE);
    randomSeed(analogRead(0));
    lcd.setStr("Welcome to", 1, 13, BLACK, WHITE);
    lcd.setStr("Arslots", 20, 13, BLACK, WHITE);
}

void loop()
{
    char MoneyBuf [15];
    char val1 [10];
    char val2 [10];
    char val3 [10];

    int random1 = random (0, 9);
    int random2 = random (0, 9);
    int random3 = random (0, 9);

    sprintf (MoneyBuf, "%5i", Money);
    lcd.setStr("$", 100, 1, BLACK, WHITE);
    lcd.setStr(MoneyBuf, 100, 10, BLACK, WHITE);

    if (!digitalRead(button))
    {
        sprintf (val1, "%i", random1);
        lcd.setStr(val1, 64, 44, BLACK, WHITE);
        delay(250);
        sprintf (val2, "%i", random2);
        lcd.setStr(val2, 64, 64, BLACK, WHITE);
        delay(250);
        sprintf (val3, "%i", random3);
        lcd.setStr(val3, 64, 84, BLACK, WHITE);

        if (random1 == random2 == random3)
        {
            Money = 10 + Money;
        }
        else if (random1 == 0 && random2 == 0)
        {
            Money = 1000 + Money;
        }
        else if (random1 == random2)

```

```

    {
        Money = 5 + Money;
    }
    else if (random2 == random3)
    {
        Money = 5 + Money;
    }
    else if (random1 == random3)
    {
        Money = 5 + Money;
    }
    else
    {
        Money = Money - 5;
    }
}

delay(200);
}

```

This code first includes the header files that the `ColorLCDShield` library requires. Then, it declares `Money` as an integer. After that, it sets up the buttons in an array.

Next, we enter the setup structure where we first initialize the LCD to use the Epson color LCD and set the buttons to be inputs. Also, we set the contrast to 40, clear the LCD, and write “Welcome to Ardslots” to the LCD.

After that, we create three random numbers, three character arrays for the random numbers, and a character array for the `Money` value. We use a new function called `sprintf()` to convert values to strings:

```
sprintf (MoneyBuf, "%5i", Money);
```

This function works by converting the `Money` value into a string with a length of five characters. It then stores those characters in the `MoneyBuf` character array. Next, we print the dollar sign (\$) and the `moneyBuf` variable to the LCD. After that, we enter an if statement that has the condition `if(!digitalRead(button));` this means that if button one is pressed, the random numbers will be converted to strings `Val1`, `Val2`, and `Val3` and written to the LCD in a horizontal line.

Then, we enter an if statement that controls the “winning.” The first condition is met if all three values are equal to one another (`random1 == random2 == random3`). The second condition is met if `random1` equals 0 and `random2` equals 0.

■ **Note** We use integer values to have better precision. For instance, we use the value `random1` instead of `Val1`.

The third condition is met if `random1` equals `random2`. The forth condition is met if `random2` equals `random3`; the fifth condition is met if `random1` equals `random3`. The final condition is met if none of the previous conditions were true. Finally, we set `s1` to 0 and delay for 200ms. That’s it for this project. Now that you know how to put values on a color LCD, the next project will show you how to make digital LEDs onto an LCD and turn them on and off with a keypad.

Project 4-4: Using a Keypad to Communicate with the Color LCD

In this project, we are going to use a keypad to communicate with the color LCD. A 16-button keypad will be used for this project, but we will be using only four of the buttons. These buttons will be used to turn on and off the LEDs on the color LCD, for which we will have to use a couple of commands we have not used in previous projects. Let's take a look at the hardware you will need for this project.

Gathering the Hardware

Here's the list of hardware to gather for this project (see Figure 4-12):

- Arduino Demilanove (or UNO)
- Color LCD shield
- 16-button keypad
- USB cable
- Solderless breadboard
- Extra wire

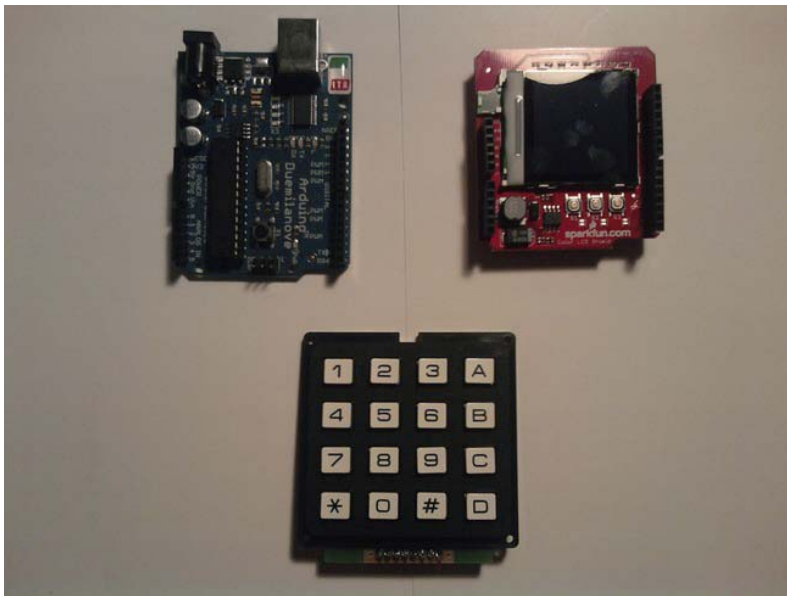


Figure 4-12. Hardware for this project (Not pictured: the USB cable, extra wire, and solderless breadboard))

Configuring the Hardware

These steps will guide you through the hardware configuration for this project:

1. Put the color LCD shield on to the Arduino.
2. Connect the 16-button keypad to the solderless breadboard.
3. Connect pin 4 on the keypad to ground on the Arduino. This will allow us to use column 4 of the keypad.
4. Connect the positive lines up to the Arduino. Pin 5 on the keypad is connected to digital pin 7; pin 6 on the keypad is connected to digital pin 6; pin 7 on the keypad is connected to digital pin 5, and pin 8 on the keypad is connected to digital pin 4.

Figure 4-13 illustrates the hardware configuration for this project.

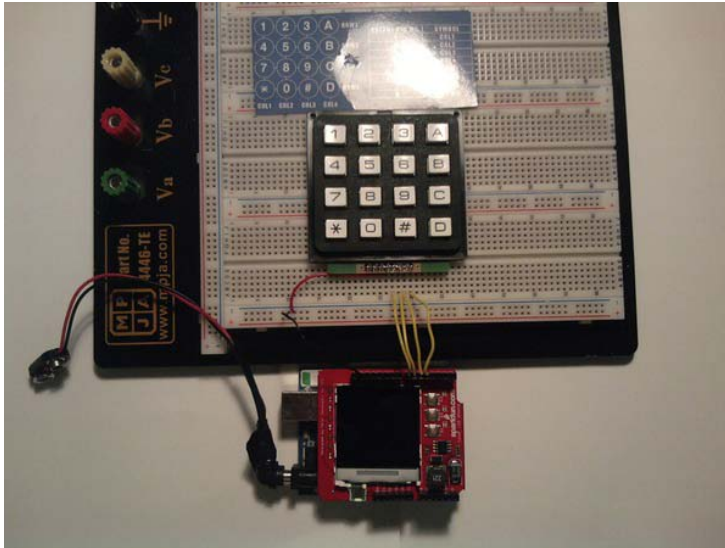


Figure 4-13. Hardware configuration for this project

The next section will discuss the software we will need to communicate with the LCD via the keypad.

Writing the Software

This project's software needs to use the keypad as an input (the LCD is the output). We then need to write some code to make an LED on the color LCD. Listing 4-5 provides the code for this project.

■ **Note** If you are using a Phillips color LCD, you will need to change `lcd.init(EPSON);` to `lcd.init(PHILLIPS);` to allow you to use the Phillips color LCD.

Listing 4-5. Using a Keypad with the Arduino to Turn On and Off Digital LEDs

```
#include <ColorLCDShield.h>

LCDShield lcd;

int button[4] = {7,6,5,4};

int buttonState[4] = {0,0,0,0};

void setup()
{
    lcd.init(EPSON); //Epson. Remember that if you are using a Phillips LCD
                    // you need to change the inside of the init() function to PHILLIPS
                    // instead of EPSON.
    lcd.contrast(40);
    lcd.clear(WHITE);
    for (int i = 0; i <= 3; i++)
    {
        pinMode(button[i], INPUT);
        digitalWrite(button[i], HIGH);
    }
}

void loop()
{
    for( int j = 0; j <= 3; j++)
    {
        buttonState[j] = digitalRead(button[j]);
    }

    lcd.setRect(9, 9,35 , 36, 0, BLACK);
    lcd.setRect(9, 39,35 , 66, 0, BLACK);
    lcd.setRect(9, 69,35 , 96, 0, BLACK);
    lcd.setRect(9, 99,35 , 126, 0, BLACK);

    if(buttonState[0] == 0)
    {
        lcd.setRect(10, 10,35 , 35, 1, YELLOW);
    }
    else
    {

```

```

    lcd.setRect(10, 10,35 , 35, 1, WHITE);
  }
  if(buttonState[1] == 0)
  {
    lcd.setRect(10, 40,35 , 65, 1, YELLOW);
  }
  else
  {
    lcd.setRect(10, 40,35 , 65, 1, WHITE);
  }
  if(buttonState[2] == 0)
  {
    lcd.setRect(10, 70,35 , 95, 1, YELLOW);
  }
  else
  {
    lcd.setRect(10, 70,35 , 95, 1, WHITE);
  }
  if(buttonState[3] == 0)
  {
    lcd.setRect(10, 100,35 , 125, 1, YELLOW);
  }
  else
  {
    lcd.setRect(10, 100,35 , 125, 1, WHITE);
  }
}

```

The first thing this code does is include the header file that we need to use the `ColorLCDShield` library. After that, it initializes the `buttonArray` and the `buttonState`.

Next, we enter the `setup` structure where we initialize the Epson color LCD, clear the LCD screen, set the buttons to inputs, and activate the four pull up resistors.

After that, we enter the `loop` structure and set each of the buttons equal to the digital read on its `buttonState`. Next, we create the housing for the LED (we draw a rectangle on the LCD). Now, we have a series of `if-else` statements. The first condition is for pin 7: if it is equal to 0, the housing will fill with the color yellow; otherwise, it will fill with white. The rest of the `if-else` statements do the same thing for different buttons.

Project 4-5: Creating the Customer's Robot

Now that you have learned how to read and write to both monochrome and color LCDs, it is time to revisit our hypothetical company to see if it has any interesting projects for us to complete that use the skills in this chapter. The first step, as always, is gathering the requirements and creating the requirements document.

Requirements Gathering and Creating the Requirements Document

The customer has set up another meeting and has an additional piece of hardware to add to the previous chapter's robot. The robot needs to have a color LCD that displays, in plain English, in which direction

the robot is moving. The customer wants you to use a color LCD shield for the prototype and realizes that the motor shield and the color LCD shield share some of the same pins (a conflict that arises often), but they want you to figure out a way past that issue.

The display should show “Forward”, “Reverse”, “Left”, or “Right” to the center of the color LCD. Figure 4-14 illustrates the format of the display.



Figure 4-14. Format of the color LCD

Now that we have our notes from the meeting, we can compile them into a requirements document, but before we do that, let's brainstorm ideas to solve the hardware problem of using the motor shield and the color LCD shield. The motor shield and the color LCD shield share a couple of pins, specifically pins 13 and 11. This is a problem because we cannot use the color LCD shield with this motor shield, so we'll need to create our own motor driver so we don't use the same pins the color LCD shield uses. With that settled, let's take a look at the requirements document.

For the requirements document, we'd need an outline of the hardware in the “Gathering the Hardware” section. The software will be the same as in Chapter 3. The only change we will need is to display the direction the robot is moving at the center of the color LCD, as shown in Figure 4-14.

Gathering the Hardware

Here's a list of the hardware you'll need to gather for this project (see Figure 4-15):

- Arduino Demilanove (or UNO)
- Color LCD shield
- L293D H-bridge from STMicroelectronics
- 74F04 hex inverter
- Three terminal blocks
- 9V battery
- 9V battery connector
- .01microfarad ceramic capacitors
- The chassis from robot you built in Chapter 3

- USB cable
- Extra wire

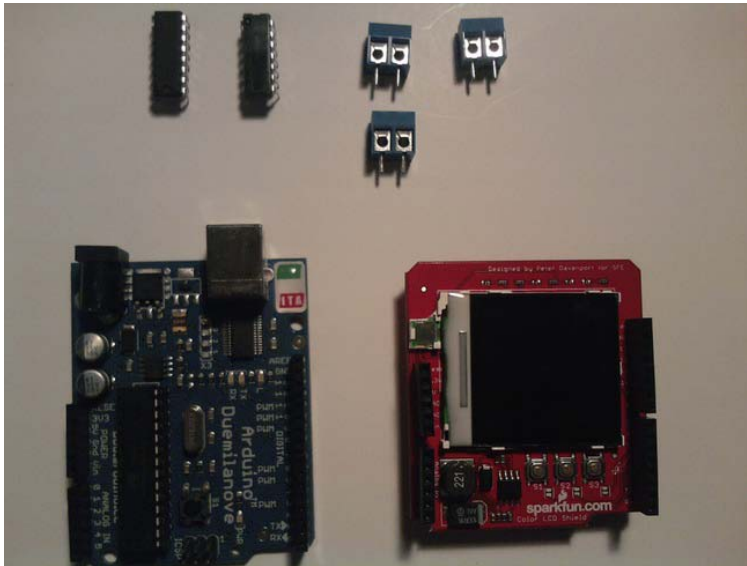


Figure 4-15. Hardware for this project (Not pictured: the 9V battery, 9V battery connector, .01microfarad ceramic capacitor, chassis from Chapter 3's robot, USB cable, and extra wire)

The software will be the same as in chapter 3. The only change we will need is to display the direction the robot is moving at the center of the color LCD. Figure 4-16 illustrates the format of the display.

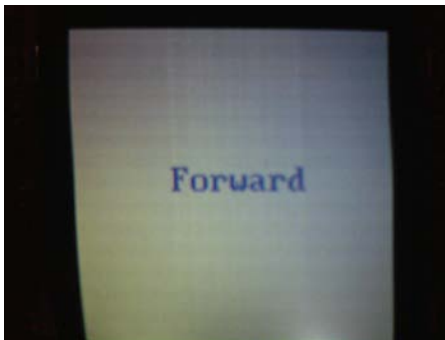


Figure 4-16. Format of the color LCD

Configuring the Hardware

Following are the steps required to configure the hardware:

1. Attach the color LCD shield to the Arduino (for now, put the motor shield in a safe place; we will not use it for this project).
2. Put the H-bridge and the hex inverter on the solderless breadboard, as shown in Figure 4-16.

■ **Note** The hex inverter is used to invert the signal we get from two of the four inputs; this frees up two I/O pins!

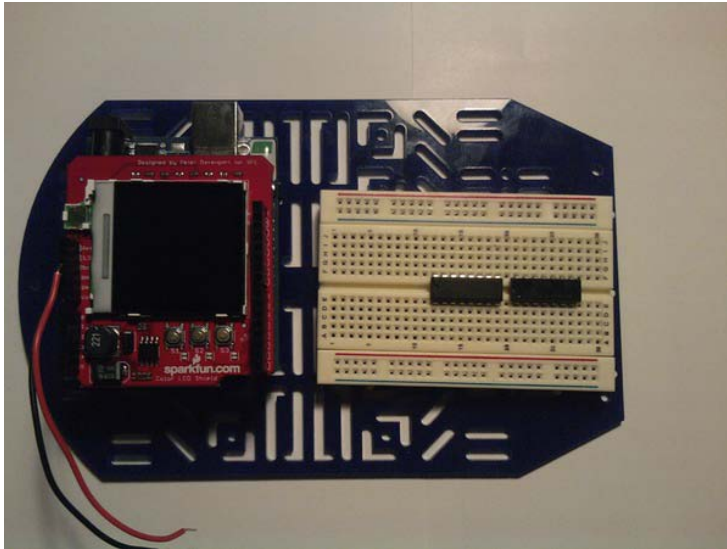


Figure 4-17. The H-bridge is connected first, and then the hex inverter is placed behind it.

■ **Caution** Make sure the H-bridge's and hex inverter's notches are facing toward the Arduino. If you put these on the breadboard incorrectly, when you power this circuit, you could harm the electronics as well as yourself if you touch the circuit (it will be very hot!). If you are holding the H-bridge with the notch up, pin 1 is to the left of the notch. The same goes for the hex inverter.

3. Add the three terminal blocks to the breadboard. See Figure 4-17.

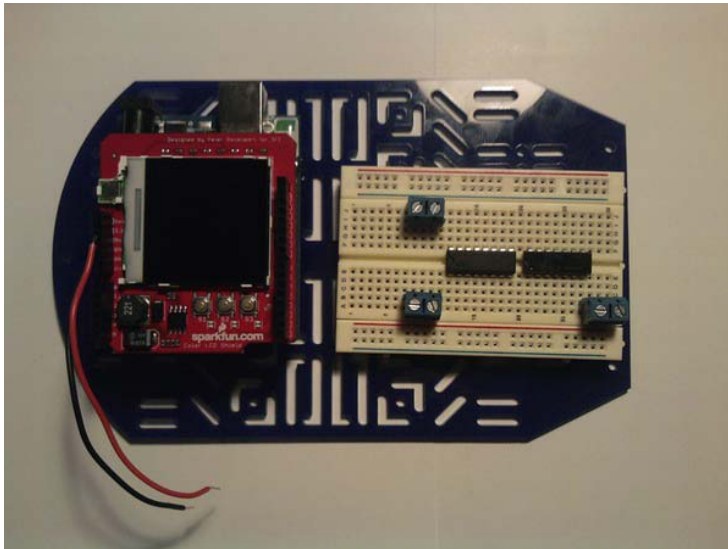


Figure 4-18. The three connected terminal blocks

4. Add the Vss, Vs, and the ground lines to the breadboard (Figure 4-18).

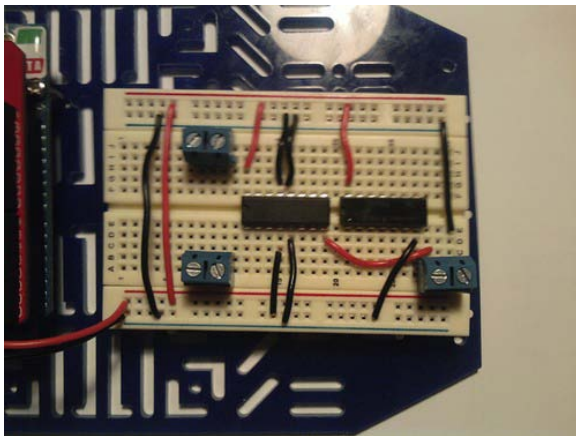


Figure 4-19. Vss, Vs, and ground connected

5. Connect H-bridge pins 3 and 6 to one of the terminal blocks on the breadboard.
6. Connect the H-bridge pins 11 and 14 (these are on the right side of the H-bridge) to the other terminal block (see Figure 4-19).

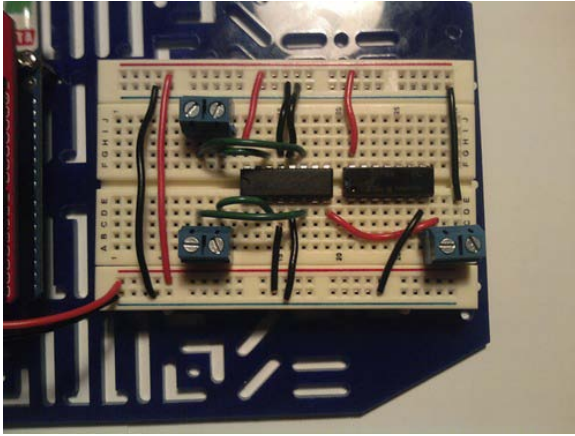


Figure 4-20. The output pins on the H-bridge are connected to the terminal blocks.

7. Attach the 9V connector to the last terminal block, as shown in Figure 4-20.

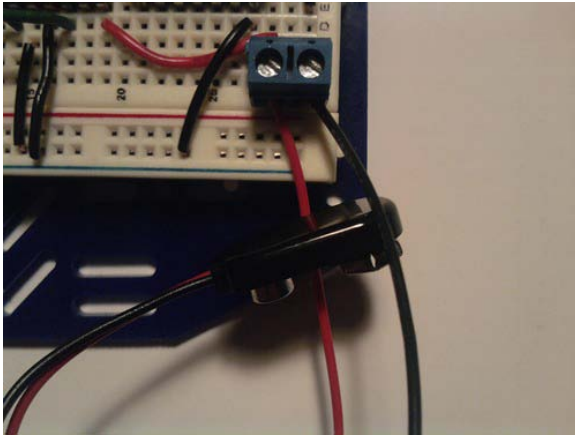


Figure 4-21. 9V connector attached to the terminal block

8. Now, connect pin 10 on the H-bridge to pin 13 on the hex inverter. Connect pin 15 on the H-bridge to pin 12 on the hex inverter, and connect pin 7 on the H-bridge to pin 1 on the hex inverter.
9. Connect pin 2 on the hex inverter to pin 2 on the H-bridge (see Figure 4-21).

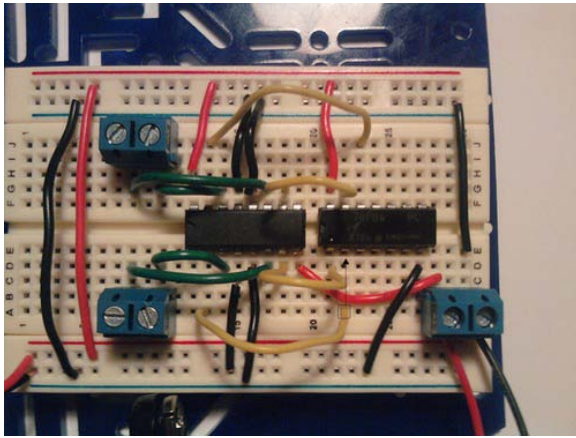


Figure 4-22. Configuration of the hex inverter

10. Connect digital pin 5 to pin 1 on the H-bridge, and digital pin 4 to pin 7 on the H-bridge. After that, connect digital pin 6 to pin 9 on the H-bridge, and digital pin 7 to pin 10 on the H-bridge (see Figure 4-22).

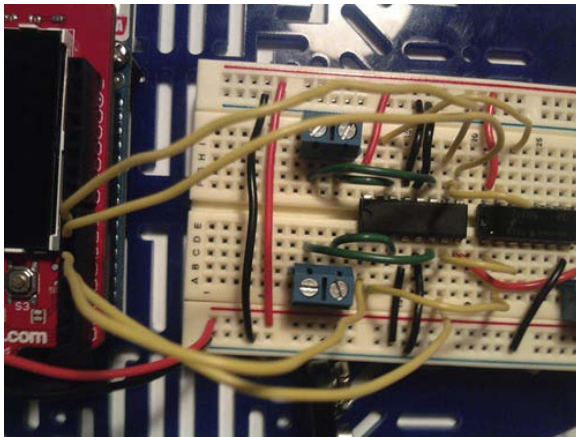


Figure 4-23. I/O connections: D4–7, D5–1, D6–9, D7–15

11. Connect the motors to the terminal blocks (see Figure 4-23).

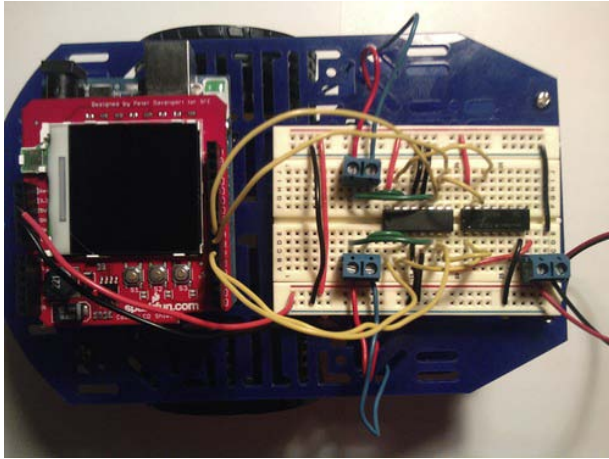


Figure 4-24. Motors attached to the terminals blocks

■ **Note** The Arduino is the front of this robot. Make sure the bottom wires of your motors are in the rightmost terminals and that the top wires of the motors are going into the left terminals.

Now that the hardware is configured, we need to fulfill the software requirements, as discussed in the next section.

Writing the Software

Now, we need to write the software. Lucky for us, we are able to use the code from the previous chapter, so we just need to add in the LCD components. We will need to filter through different values the program receives from the serial monitor. The best way to do this is with an `if` statement (see Listing 4-6).

■ **Note** If you are using a Phillips color LCD, you will need to change `lcd.init(EPSON);` to `lcd.init(PHILLIPS);` to allow you to use the Phillips color LCD.

Listing 4-6. Display the Direction the Robot Is Moving on a Color LCD

```
#include <ColorLCDShield.h>

LCDShield lcd;
```

```

const int fields = 4; // How many fields are there? Right now, 4
int motorPins[] = {4,5,7,6}; // Motor Pins
int index = 0; // the current field being received
int values[fields]; // array holding values for all the fields

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud

  lcd.init(EPSON); //Epson. If you have a Phillips Color LCD, then put PHILLIPS instead of
                  // EPSON.
  lcd.contrast(40);
  lcd.clear(WHITE);

  for (int i; i <= 3; i++)          // set LED pinMode to output
  {
    pinMode(motorPins[i], OUTPUT);
    digitalWrite(motorPins[i], LOW);
  }

  Serial.println("The Format is: MotoADir,MotoASpe,MotorBDir,MotoBSpe\n");
}

void loop()
{

  if( Serial.available())
  {
    char ch = Serial.read();
    if(ch >= '0' && ch <= '9') // If the value is a number 0 to 9
    {
      // add to the value array
      values[index] = (values[index] * 10) + (ch - '0');
    }
    else if (ch == ',') // if it is a comma
    {
      if(index < fields -1) // If index is less than 4 - 1...
        index++; // increment index
    }
    else
    {
      if (values[0] == 0 && values[1] == 0 && values[2] == 0 && values[3] == 0)
      {
        lcd.clear(WHITE);
        lcd.setStr("Motors have ", 1, 10, BLACK, WHITE);
        lcd.setStr("Stopped", 20, 10, BLACK, WHITE);
      }
      else if (values[0] == 0 && values[2] == 0)
      {

```

```

    lcd.clear(WHITE);
    lcd.setStr("Reverse", 50, 40, BLACK, WHITE);
}
else if (values[0] == 1 && values[2] == 1)
{
    lcd.clear(WHITE);
    lcd.setStr("Forward", 50, 40, BLACK, WHITE);
}
else if (values[0] == 0 && values[2] == 1)
{
    lcd.clear(WHITE);
    lcd.setStr("Left", 50, 50, BLACK, WHITE);
}
else if (values[0] == 1 && values[2] == 0)
{
    lcd.clear(WHITE);
    lcd.setStr("Right", 50, 50, BLACK, WHITE);
}
}

for(int i=0; i <= index; i++)
{
    if (i == 0)
    {
        Serial.println("Motor A");
        Serial.println(values[i]);
    }
    else if (i == 1)
    {
        Serial.println(values[i]);
    }
    if (i == 2)
    {
        Serial.println("Motor B");
        Serial.println(values[i]);
    }
    else if (i == 3)
    {
        Serial.println(values[i]);
    }

    if (i == 0 || i == 2) // If the index is equal to 0 or 2
    {
        digitalWrite(motorPins[i], values[i]); // Write to the digital pin 1 or 0
                                                // depending what is sent to the Arduino
    }

    if (i == 1 || i == 3) // If the index is equal to 1 or 3
    {
        analogWrite(motorPins[i], values[i]); // Write to the PWM pins a number between
                                                // 0 and 255 or what the person entered
    }
}

```

```

// in the serial monitor.

    }

    values[i] = 0; // set values equal to 0
}

    index = 0;
}
}
}

```

Because much of this code is from the last chapter, I will just explain the new code for this project. The new code begins with the include statements, specifically the header file that is necessary for us to use the `ColorLCDShield` library. Next, we need to change the I/O pins from the Arduino; this is accomplished by initializing the `motorPins` values to 4, 5, 7, and 6.

After that, we enter the setup structure, where we initialize the LCD to use the Epson color LCD. We also clear the LCD screen and set it to a white background, and we digitally write to the motors to make sure they are off. The next new piece of code is not until the last part of the program, when we get to this section of code:

```

if (values[0] == 0 && values[1] == 0 && values[2] == 0 && values[3] == 0)
{
    lcd.clear(WHITE);
    lcd.setStr("Motors have ", 1, 10, BLACK, WHITE);
    lcd.setStr("Stopped", 20, 10, BLACK, WHITE);
}
else if (values[0] == 0 && values[2] == 0)
{
    lcd.clear(WHITE);
    lcd.setStr("Reverse", 50, 40, BLACK, WHITE);
}
else if (values[0] == 1 && values[2] == 1)
{
    lcd.clear(WHITE);
    lcd.setStr("Forward", 50, 40, BLACK, WHITE);
}
else if (values[0] == 0 && values[2] == 1)
{
    lcd.clear(WHITE);
    lcd.setStr("Left", 50, 50, BLACK, WHITE);
}
else if (values[0] == 1 && values[2] == 0)
{
    lcd.clear(WHITE);
    lcd.setStr("Right", 50, 50, BLACK, WHITE);
}
}

```

This if statement checks to see whether the motors are running and in what direction. For instance, if `value[0]` and `values[2]` are equal to 0, the LCD will display “Reverse”. The code works the same way (with different values, of course) for forward, left, and right.

Debugging the Arduino Software

If you are having problems displaying anything on your color LCD shield, you may be using the incorrect `lcd.init()`; call. Remember that Phillips color LCD's will use `lcd.init(0)`; and Epson Color LCD's will use `lcd.init(1)`;

Because this code has a lot of `if` statements, logical errors can happen, so double-check your `if` statements. If you are having problems getting the robot to display the correct direction, there may be a logical error with one the `if` statements discussed in the previous section. You may want to look at the code again. If it is not the code, you may have your motors attached in the opposite terminals.

Syntax errors happen all the time as well. If you run into a syntax error that says you need to add a semicolon, look around for a statement that does not have a semicolon. If you face any other type of error, make sure you have the right format of your code. For instance, you may have typed `for(int i, i <= 3, i++)`. This is a common syntax error; you need to do is replace the commas with semicolons.

Now that we have fixed the software, we can move on to troubleshooting the hardware.

Troubleshooting the Hardware

You may have noticed that your motors make a lot of electrical noise. A good practice when using motors is to add a .01microFarad ceramic capacitor to get rid of the extra noise. Figure 4-24 illustrates using a .01microfarad cap to remove the noise from the DC motors.

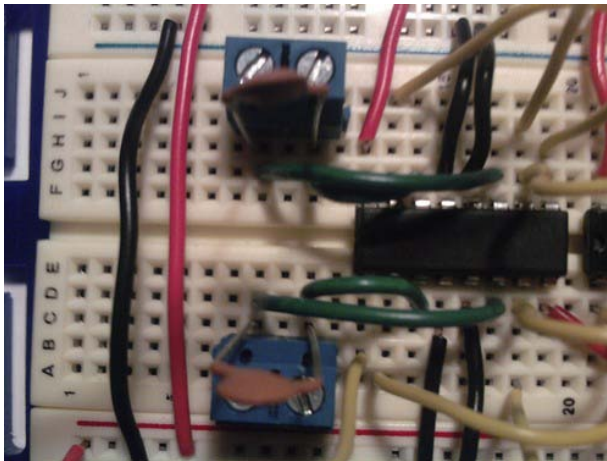


Figure 4-25. Capacitors added to remove noise from DC motors

Now if your H-bridge circuit did not work, the best thing to do is make sure all the wires are in the correct place. Go over the wiring 100 times if you must, because you do not want to hurt these fragile integrated circuits (ICs).

Finished Prototype

The robot should now be complete (well, at least for this chapter) and should display the direction of its momentum when it's moving (e.g., “Forward” when it is moving forward). Your circuit may get a little warm, but it won't harm the ICs. Figure 4-25 illustrates the final prototype for this chapter.

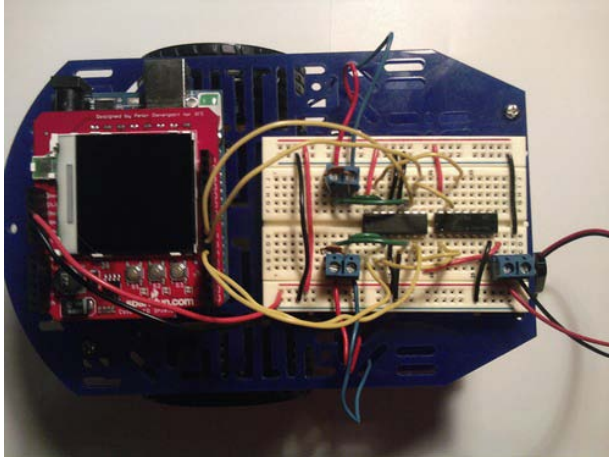


Figure 4-26. The finished prototype

Summary

Throughout this chapter, you have been learning about LCDs, both monochrome and color. You learned about the LCD libraries and practiced using them in several different projects. We also worked with a few new pieces of hardware, such as a keypad and LCDs. You learned that sometimes a company's requirements may lead to hardware problems you must solve; this was the case when we had to use both a color LCD and a motor shield. As part of that solution, we also covered building your own H-bridge driver. In creating this chapter's robot, we reviewed the engineering process and created a finished prototype. Now that you have finished the projects for this chapter, you should have a good foundation for working with LCDs in general.

Robot Integration Engineering a GPS Module with the Arduino

In this chapter, we will be discussing GPS and adding it to the customer's robot we built in the previous chapter. You can use GPS for many things, such as tracking a robot or selecting where you want your robot to be. The customer wants to do just that by sending the GPS location of the robot to a microSD card and logging it to a text file. They want to send a separate command to the robot in order to get the latitude and longitude data from GPS.

Throughout this chapter, we will be using a protocol called NMEA. This protocol is used to communicate important GPS-related information. We will also discuss two new libraries. They are TinyGPS (thanks to Mikal Hart: <http://arduiniiana.org/libraries/tinygps/>) and SdFat Library (thanks to Bill Greiman: <http://code.google.com/p/sdfatlib/downloads/list>). These libraries will help us both communicate with the GPS and create files to store our GPS data. I will also introduce two new shields; they are the microSD shield from Sparkfun. This shield will be used in conjunction with the SdFat library and the GPS shield from Sparkfun. This shield allows you to use a GPS module with your Arduino to get GPS location (but more on these shields later).

After we have discussed the new libraries and shields, we will walk through a few preliminary projects in preparation for the customer's new set of requests. These preliminary projects include writing raw GPS data to the serial monitor, creating a car finder, writing GPS data to a monochrome LCD, logging GPS data, and finally adding to the robot we created in the previous chapter.

Let's start off by reviewing microSD shield and then the NMEA protocol so that you understand what the GPS is displaying.

Hardware Explained: microSD Shield

In the GPS data logger project we will create in this chapter, we will use the microSD shield; this shield is used to store data. Because GPS data can be large and the Arduino does not have that much onboard memory, we need to outsource the GPS data to another memory source. This is where the microSD shield comes into play. This particular microSD shield will allow us to use up to a 1GB microSD card (1GB is not fully supported, but it should work). For this chapter, we will be using a 512MB microSD card. Figure 5-1 illustrates the new hardware for this chapter.

■ **Note** Make sure you format your microSD card to FAT32.

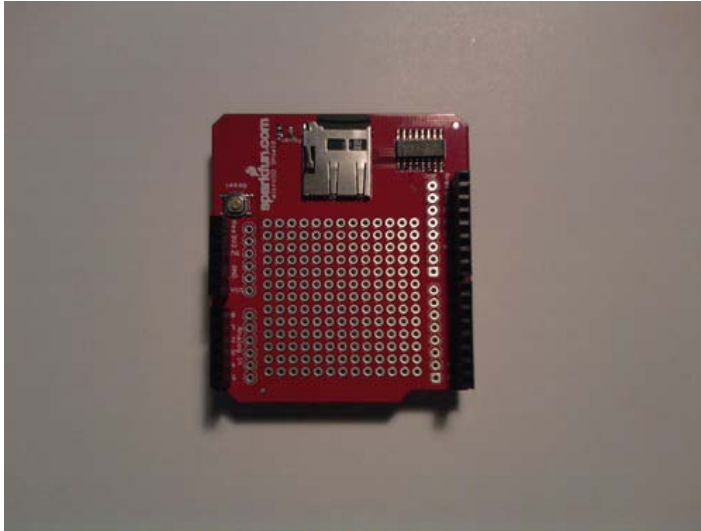


Figure 5-1. microSD shield

■ **Note** The microSD shield uses the same pins as the color LCD shield, so we will not be using the color LCD shield in the final project.

Understanding NMEA Protocol

NMEA stands for “National Marine Electronics Association.” In this section, we discuss three specific NMEA sentences: GPGGA, GPGSV, and GPRMC. We need to understand these NMEA sentences for the first project of this chapter to make sense. Each of these sentences offers different data. The next few sections will discuss the uses and formats of GPGGA, GPGSV, and GPRMC.

- *GPGGA*: This sentence holds the data for longitude and latitude and other important information about the GPS’s fix. Here is the format of the GPGGA command:

```
$GPGGA, TimeofFix, Lat, dir, Long, dir, FixQual, NumofSats, Altit, unit, HghtofGeoid, unit, emp, emp, CS
```

The information looks something like this:

```
GPGGA, 105026, 23456.061, N, 43214.056, W, 1, 04, 0.9, 100.56, M, 56.5, M,,*47
```

- *GPGSV*: This sentence holds the satellite information such as signal-to-noise ratios and a few other parameters. Here is the format of the GPGSV command:

```
$GPGSV, NumofSent, SentNum, NumofSat, PRNnum, Elevation, Azimuth, SNR, CS
```

The information looks like this:

\$GPGSV, 2, 1, 08, 01, 50, 083, 55, (The last 5 pieces of data can happen up to 4 times each sentence), CS

- *GPRMC*: This sentence holds data that will tell us speed, longitude, and latitude. Here is the format of the GPRMC command:

\$GPRMC, TimeofFix, Status, Lat, dir, Long, dir, SpeedOverGround, TrackAngle, date, MagVar, CS

The information looks like this:

\$GPRMC, 105019, V, 1234.345, N, 3456.067, W, 025.3, 100.4, 231286, 003.2, W, *6A

Now that you understand NMEA a bit better, we can discuss the TinyGPS library and how it is used in GPS applications.

Libraries Explained: TinyGPS and SdFat Libraries

In this section, you will learn about the TinyGPS library and the SdFat library that we will use throughout this chapter. The first section will cover the TinyGPS library; this library parses the raw GPS data that you will see in the first project. It is important to understand NMEA sentences so that in the first project you can identify whether you are receiving valid data. The next section will discuss the TinyGPS library.

TinyGPS

The first library that will be explained is the TinyGPS library (by Mikal Hart). You can download this library from <http://arduiniiana.org/libraries/tinygps/>. This library is used to parse through the GPS data and make it more understandable. The first thing we need to do in order to use the TinyGPS library is to include the header file:

```
#include "TinyGPS.h"
```

After that, we need to create an instance of the TinyGPS:

```
TinyGPS gps;
```

Next, we need to create a soft serial (aka software serial) to communicate with the GPS; this was discussed in Chapter 2. After that, we need to use the `encode()` function to get the GPS data. Here is a code snippet using the `encode()` function:

```
#include <NewSoftSerial.h>
#include <TinyGPS.h>

TinyGPS gps;
NewSoftSerial gpsSerial(2,3); // rx and tx
void setup()
{
    // setup code
}
void loop()
{
    while(gpsSerial.available() > 0)
    {
        int data = gpsSerial.read();
```

```

        if(gps.encode(data));
        {
            // Processed data
        }
    }
}

```

Now that you understand how to set up the TinyGPS library, here are a few functions that give us parsed NMEA data:

- `TinyGPS get_position()`: This function gives us the longitude and latitude direction; it also has the parameter that allows us to check how recent the data is. Here is the format of the `get_position()` function:

```
gps.get_position(&lat, &long, &fix_age);
```

- `TinyGPS get_datetime()`: This function returns the values for the date and time; it also has the parameter that allows us to check how recent the data is. Here is the format of the `get_datetime()` function:

```
gps.get_datetime(&date, &time, &fix_time);
```

- `TinyGPS speed()`: This function displays the current speed of the GPS module. Here is the format of the speed function:

```
unsigned long speed = gps.speed();
```

Now that you are familiar with the TinyGPS library, we can focus our attention on the SdFat library. The next section will discuss several functions that will allow us to read and write data to a microSD card.

SdFat Library

The SdFat library (by Bill Greiman) allows us to open, create, and send data to a file. You can find this library at <http://code.google.com/p/sdfatlib/downloads/list>. As usual, we need to add the include files in order to use the SdFat library. They are as follows:

```

#include <SdFat.h>
#include <SdFatUtil.h>
#include <ctype.h>

```

After that, we need to declare a few variables that the SdFat libraries use in order to communicate with a file.

```

Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

```

Next, we need to initialize a few of the variables we declared in the previous step. They are card and volume. Also, the root directory of the microSD card is opened. This is all done within the setup structure.

```

void setup()
{
    pinMode(10, OUTPUT);    // this pin needs to be set to an output in order to work.

```

```

    card.init();
    volume.init(card);
    root.openRoot(volume);
}

```

Now that we have set up the SdFat library, you need to learn a few commands that will allow us to open a file, create a file, write to a file, and close a file.

- **SdFat open():** This function will allow us to open or create a file if needed. Here is the format of the open() function:

```
file.open(SDFile *dirFile, const char *path, uint8_t oflag);
```

- **SdFat print():** This function writes data to the SD card. Here is the format of the print() function:

```
file.print("string");
```

- **SdFat close():** This function closes the file. Here is the format of the close() function:

```
file.close();
```

Now that you understand some of the functions of the TinyGPS library and the SdFat library, we can move on to the first project of this chapter.

The Basics of GPS Communication with the Arduino

In this section, we will be working through a few projects to get us ready for the final project. The projects are writing raw GPS data to the serial monitor, writing GPS data to a monochrome LCD, creating a car finder, and creating a GPS data logger. These projects will help us understand the TinyGPS and the SdFat libraries in a practical approach.

Project 5-1: Writing Raw GPS Data to the Serial Monitor

This project will make sure everything is up and running smoothly. We will use the GPS shield to send raw GPS commands to the serial monitor. That is, we will be sending GPGGA, GPGSV, and GPRMC data with no formatting to the serial monitor. The first thing we need to do is configure the hardware.

Hardware for This Project

Figure 5-2 shows some of the hardware being used in this project. (Not pictured: USB cable.)

- Arduino Duemilanove (or UNO)
- GPS shield
- USB cable

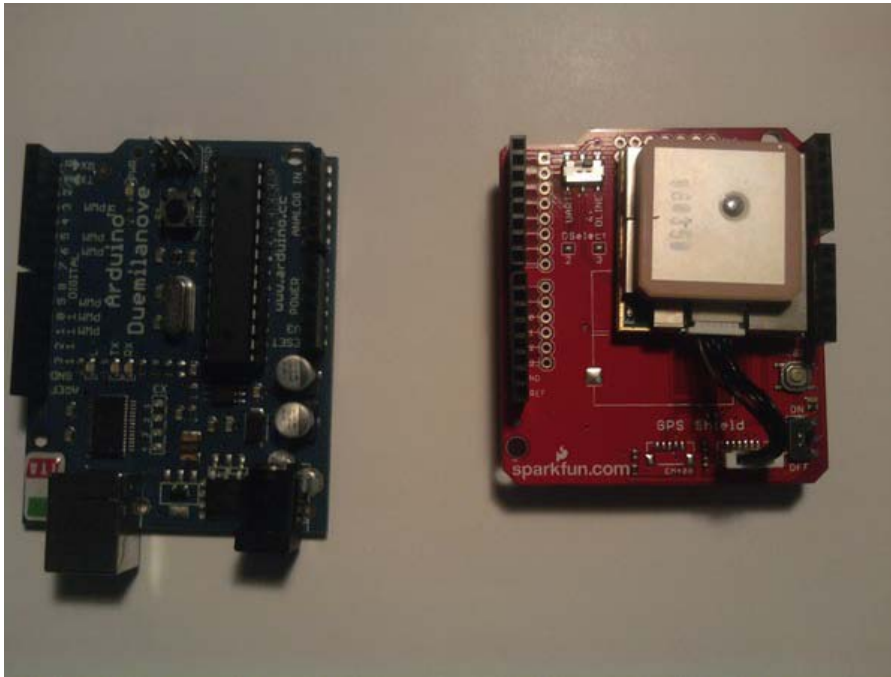


Figure 5-2. Hardware for this project

Configuring the Hardware

Follow these steps to configure the hardware in Project 5-1:

1. Connect the GPS shield to the Arduino.
2. Make sure the switch that controls the pins the GPS shield uses is switched to DLINE (see Figure 5-12 for an illustration of this process). We use DLINE because it allows us to use the software serial rather than the hardware serial; this is the switch closest to the tx and rx pins on the GPS shield.
3. Connect the USB from the Arduino to a computer.

Figure 5-3 illustrates the hardware configuration. In the next section, we will be discussing the software for this project.

■ **Note** A hardware serial uses some form of microchip to transfer data bit by bit. A software serial uses interrupts and other software controls to make a virtual serial port; this is useful because the Arduino Duemilanove and UNO only have one serial port, so essentially we can create a couple of serial ports rather than just one.



Figure 5-3. Hardware configuration for this project

Writing the Software

We will need to use the NewSoftwareSerial (Thanks Mikal Hart) you learned about in Chapter 2. After that, all we have to do is send the GPS data to the serial monitor. Listing 5-1 provides the code for this project.

Listing 5-1. Writes Raw GPS Data to a Serial Port

```
#include <NewSoftSerial.h>

NewSoftSerial serial_gps(2,3);

void setup()
{
    serial_gps.begin(4800);

    Serial.begin(9600);

    Serial.print("\nRAW GPS DATA\n");
}

void loop()
{
    while (serial_gps.available() > 0)
    {
        int c = serial_gps.read();
        Serial.print(c, BYTE);
    }
}
```

The first thing this code does is include the `NewSoftwareSerial` header file so that we can use the `NewSoftwareSerial` library. After that, we create an instance of the `NewSoftwareSerial` with this code:

```
NewSoftSerial serial_gps(2,3);
```

Next, we enter the setup structure where we begin both the software serial and the hardware serial, and then we write “Raw GPS Data.” Then we enter the loop structure, where we enter a `while` loop with the condition `serial_gps.available > 0`. The `while` loop will run as long as there are bytes at the serial port. Within the `while` loop, `int c` is getting the values from the serial read command. Then the data is written to the serial monitor. Now that we have configured the GPS shield to send data to the serial monitor, we can make more complicated applications. In the next project, we will be working with the GPS shield and a monochrome LCD to send latitude and longitude values to the LCD.

Project 5-2: Writing GPS Data to a Monochrome LCD

In this project, we will need to communicate with a monochrome LCD and display latitude and longitude data. We will still need a software serial for this project, but you could use the hardware serial, and everything would work because we are no longer sending data to the serial monitor. You would just need to make sure you do not use the `DLIN`; instead, you would switch to `UART` on the GPS shield. In this project, we will be using three libraries: `NewSoftwareSerial`, `TinyGPS`, and `LiquidCrystal`. Now that you know a little about this project, we can see what hardware we will need for this project.

Hardware for This Project

Figure 5-4 shows some of the hardware being used in this project. (Not pictured: solderless breadboard, 9V battery, 9V battery connector, and extra wire.)

- Arduino Duemilanove (or UNO)
- GPS shield
- Monochrome LCD
- 10K potentiometer
- Solderless breadboard
- 9V battery
- 9V battery connector
- Extra wire

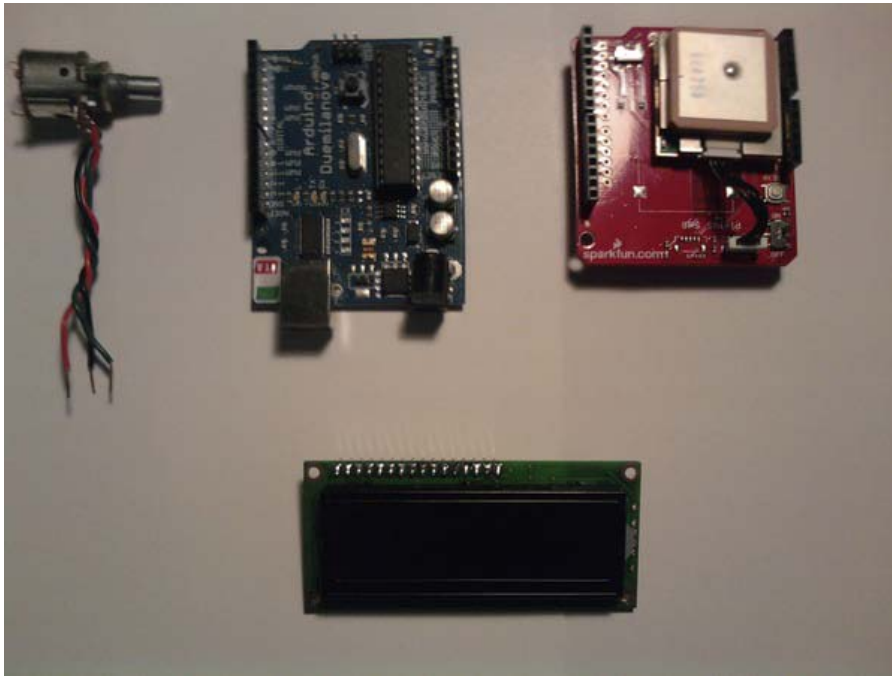


Figure 5-4. Hardware for this project

Configuring the Hardware

To configure the hardware in Project 5-2, follow these steps:

1. Connect the GPS shield to the Arduino.
2. Connect the Monochrome LCD to the solderless breadboard.
3. Connect power (+5V) to pin 2 on the LCD and ground to pin 1 on the LCD.
4. Connect the 10K potentiometer's middle wiper (middle wire on pot) to pin 3 on the LCD; the other two wires are connected to power (+5V) and ground.
5. Connect the RS, R/W, E, and bus lines from the Arduino to the LCD. We will start with the RS pin. The RS pin is connected to digital pin 7, the R/W pin is connected to ground, the E pin is connected to digital pin 8, and DB7–DB4 are connected to digital pins 12–9 on the Arduino.
6. Connect power (+5V) to pin 15 on the LCD and ground to pin 16 on the LCD.

Figure 5-5 illustrates the hardware configuration for this project.

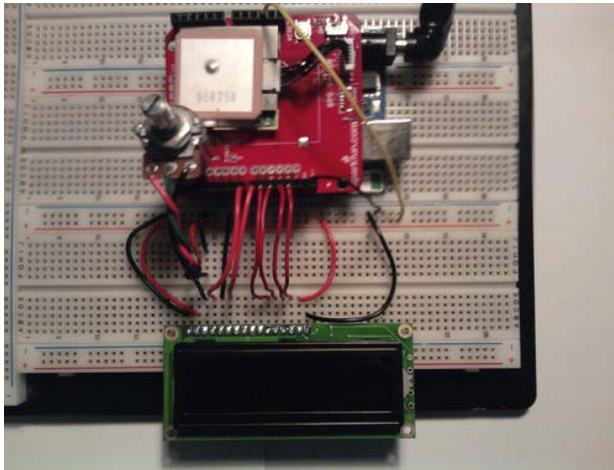


Figure 5-5. Hardware configuration for this project

Writing the Software

The software will need to communicate with the software serial and the monochrome LCD. It will then have to write longitude and latitude to the display in a user-friendly fashion. Listing 5-2 introduces the code for this project.

Listing 5-2. Write latitude and longitude data to monochrome LCD

```
// include the library code:
#include <LiquidCrystal.h>
// includes the NewSoftwareSerial Library
#include <NewSoftSerial.h>
// include TinyGPS library
#include <TinyGPS.h>

// create an instance of the TinyGPS object
TinyGPS gps;

// initializes the soft serial port
NewSoftSerial nss(2,3);

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7,8,9,10,11,12);

void gpsData(TinyGPS &gps);

void setup()
{
    nss.begin(4800);
```

```

    // set up the LCD's number of columns and rows:
    lcd.begin(16, 2);
    lcd.clear();
}

void loop()
{
    while(nss.available()) // Makes sure data is at the Serial port
    {
        int c = nss.read();
        if(gps.encode(c)) // New valid sentence?
        {
            gpsData(gps); // Write data to the LCD
        }
    }
}

void gpsData(TinyGPS &gps)
{
    // Initialize Longitude and Latitude to floating-point numbers
    float latitude, longitude;

    // Get longitude and latitude
    gps.f_get_position(&latitude,&longitude);

    // Set cursor to home (0,0)
    lcd.home();
    // Print "lat: " to LCD screen
    lcd.print("lat: ");
    // Prints latitude with 5 decimal places to LCD screen
    lcd.print(latitude,5);

    // Sets the cursor to the second row
    lcd.setCursor(0,1);
    // Print "long: " to LCD screen
    lcd.print("long: ");
    // Prints longitude with 5 decimal places to LCD screen
    lcd.print(longitude,5);
}

```

The first thing this code does is include all of the libraries that the code needs to use. They are the `LyquidCrystal` library, `NewSoftSerial` library, and `TinyGPS` library. After that, we create an instance of `TinyGPS`, `NewSoftSerial`, and `LyquidCrystal`. Then we create a functional prototype called `gpsData` that has the arguments for the GPS data. We then enter the setup structure, which begins the software serial and the LCD; it also clears the LCD. Next we enter the loop structure. In the loop structure, we first see whether there is any data on the software serial port; if there is, it enters an `If` statement that has the argument `gps.encode(c)`; . This argument makes sure we have a valid GPS sentence. Finally, we enter the subroutine `gpsData`. In `gpsData()` we write the longitude and latitude data to the monochrome LCD.

■ **Note** Make sure that the GPS shield is set to DLINE; otherwise, this code will not work.

Now that we can display GPS data on a monochrome LCD, we can add onto our existing hardware and code to make a car finder with the GPS shield and the monochrome LCD.

Project 5-3: Creating a Car Finder

While this project might not find you a new car, it can find your car if you left it in a massive parking lot. This project will need to use some extra digital pins for a push button and a switch; we will use these to interface with the menu we create on the LCD. Let's first take a look at the hardware for this project.

Hardware for This Project

Figure 5-6 shows some of the hardware being used in this project. (Not pictured: solderless breadboard, 9V battery, 9V battery connector, and extra wire.)

- Arduino Duemilanove (or UNO)
- GPS shield
- Monochrome LCD
- Toggle switch
- Normally off push button
- 10K potentiometer
- Solderless bread board
- Extra wire
- 9V battery
- 9V battery connector

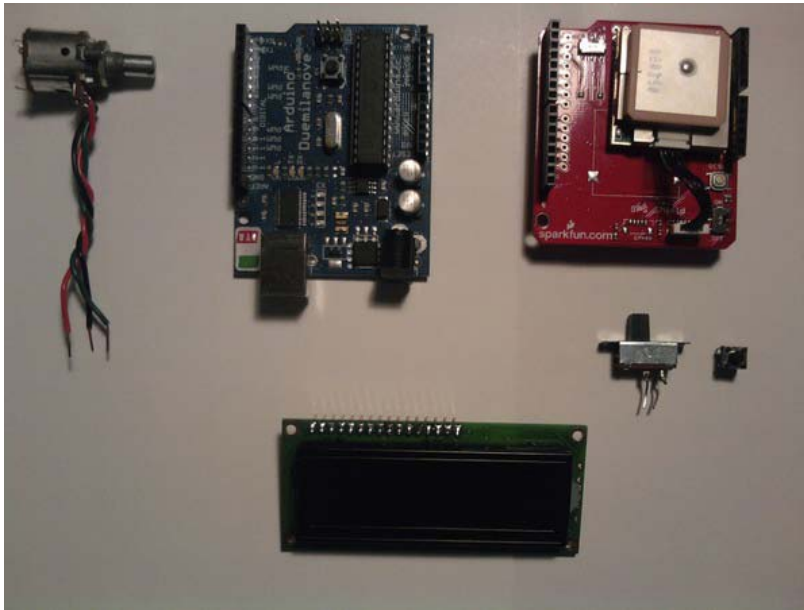


Figure 5-6. Hardware for this project

Configuring the Hardware

To configure the hardware in Project 5-3, follow these steps:

1. Connect the GPS shield to the Arduino.
2. Connect the Monochrome LCD to the solderless breadboard.
3. Connect power (+5V) to pin 2 on the LCD and ground to pin 1 on the LCD.
4. Connect the 10K potentiometer's middle wiper (middle wire on pot) to pin 3 on the LCD; the other two wires are connected to power (+5V) and ground.
5. Connect the RS, R/W, E, and bus lines from the Arduino to the LCD. We will start with the RS pin. The RS pin is connected to digital pin 7, the R/W pin is connected to ground, the E pin is connected to digital pin 8, and DB7–DB4 are connected to digital pins 12–9 on the Arduino.
6. Connect power (+5V) to pin 15 on the LCD and ground to pin 16 on the LCD.
7. Now that the LCD is connected, we need to connect the switch and normally off push button:
8. Connect the switch and push button to the solderless breadboard.
9. Connect one pin of the switch and push button to ground.
10. Connect the other pin on the switch to digital pin 5 on the Arduino.

11. Connect the other pin on the push button to digital pin 6 on the Arduino.

Figure 5-7 illustrates the hardware configuration for this project.

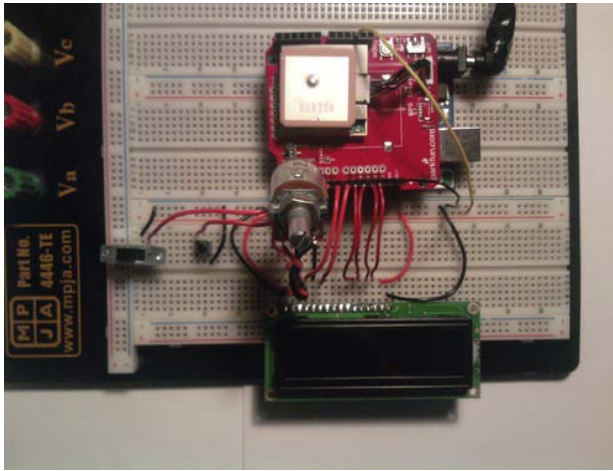


Figure 5-7. Hardware configuration for this project

Writing the Software

The software for this project is a lot like the software from the previous project. The only difference is that we filter a lot of the data; we are using a push button and a switch. The switch is used to control a menu, and the push button is used to store the GPS position of a car in a parking lot. We also do limit testing to tell the person that their car is near or that they need to keep looking. Listing 5-3 is the code for this project.

Listing 5-3. ArduinoCarFinder.pde

```
// include the library code:
#include <LiquidCrystal.h>
// includes the NewSoftwareSerial Library
#include <NewSoftSerial.h>
// include TinyGPS library
#include <TinyGPS.h>

// create an instance of the TinyGPS object
TinyGPS gps;

// initializes the soft serial port
NewSoftSerial nss(2,3);

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7,8,9,10,11,12);

// set up buttons
```

```

int button1 = 5;
int button2 = 6;

float latVal = 0;
float longVal= 0;

int buttonVal1 = 0;
int buttonVal2 = 0;

int buttonCount = 0;
int numCount = 0;

void gpsData(TinyGPS &gps);

void setup()
{
    nss.begin(4800);

    pinMode(button1, INPUT);
    pinMode(button2, INPUT);

    digitalWrite(button1, HIGH);
    digitalWrite(button2, HIGH);

    // set up the LCD's number of columns and rows:
    lcd.begin(16, 2);
    lcd.clear();
    lcd.home();
    lcd.print("Arduino");
    lcd.setCursor(0,1);
    lcd.print("Car Finder");
    delay(3000);
    lcd.clear();
}

void loop()
{
    while(nss.available())    // Makes sure data is at the Serial port
    {
        int c = nss.read();
        if(gps.encode(c))    // New valid sentence?
        {
            gpsData(gps);    // Write data to the LCD
        }
    }
}

```

```

void gpsData(TinyGPS &gps)
{
    // Initialize Longitude and Latitude to floating-point numbers
    float latitude, longitude;

    // Get longitude and latitude
    gps.f_get_position(&latitude,&longitude);

    buttonVal1 = digitalRead(button1);
    buttonVal2 = digitalRead(button2);

    if (buttonVal2 == LOW)
    {
        buttonCount++;
    }

    if(buttonVal1 == LOW)
    {
        switch (buttonCount)
        {
            case 1:
                if (numCount <= 0)
                {
                    gps.f_get_position(&latitude,&longitude);
                    latVal = latitude;
                    longVal = longitude;
                    delay(250);
                    numCount++;
                }
                else if (numCount > 0)
                {
                    lcd.home();
                    lcd.print("car is here:   ");
                    delay(2000);
                    lcd.clear();
                    lcd.home();
                    lcd.print("Lat:   ");
                    lcd.print(latVal,5);
                    lcd.setCursor(0,1);
                    lcd.print("Long: ");
                    lcd.print(longVal,5);
                    delay(5000);
                    lcd.clear();
                }
            }
            break;
            case 2:
                lcd.clear();
                lcd.home();
                lcd.print("Reset Car loc");
                lcd.setCursor(0,1);

```

```

        lcd.print("keep holding");
        delay(5000);
        lcd.clear();
        break;
        default:
        buttonCount = 0;
        numCount = 0;
        break;
    }
}

if(buttonVal1 == HIGH)
{
    lcd.home();
    lcd.print("You are here:  ");
    delay(2000);
    lcd.clear();
    // Set cursor to home (0,0)
    lcd.home();
    // Print "lat: " to LCD screen
    lcd.print("lat:  ");
    // Prints latitude with 5 decimal places to LCD screen
    lcd.print(latitude,5);

    // Sets the cursor to the second row
    lcd.setCursor(0,1);
    // Print "long: " to LCD screen
    lcd.print("long: ");
    // Prints longitude with 5 decimal places to LCD screen
    lcd.print(longitude,5);
    delay(5000);
    lcd.clear();
    if (longitude <= longVal + 0.00010 && longitude >= longVal - 0.00010 && latitude <= latVal
+ 0.00010 && latitude >= latVal - 0.00010)
    {
        lcd.clear();
        lcd.home();
        lcd.print("You should see ");
        lcd.setCursor(0,1);
        lcd.print("your car");
        delay(2000);
        lcd.clear();
    }
}
else
{
    lcd.clear();
    lcd.home();
    lcd.print("Keep Looking  ");
}

```

```

        delay(1000);
        lcd.clear();
    }

}

}

```

This code has a lot of software we have already discussed, so I am going to go over the new pieces that may look a little different. The first piece of code is at the very beginning of the program. It looks like this:

```
void gpsData(TinyGPS &gps);
```

This is a functional prototype; we have to declare it at the beginning of the program because the compiler needs to know it is there. After that, we don't see new code until we get to the function `gpsData`; in this function we initialize the longitude and latitude variables and set them as arguments in the `f_get_position()` function. Then we set `buttonVal1` and `buttonVal2` to the digital reads on pins 5 and 6. Next we create an if statement that controls the `buttonCount` value. After that, we check whether `buttonVal1` is "LOW." If it is, we enter the switch statement that controls a car's GPS position in a parking lot. The first case writes the value of the GPS location into `latVal` and `LongVal`. The second case is there to notify the user that they need to keep holding the push button in order to reset the car's GPS location. The third case sets `numCount` to 0 and `buttonCount` to 0. The next if statement checks whether `buttonVal1` is "HIGH." If it is, we write the current latitude and longitude values to the monochrome LCD. Also within this if statement is a limit test (the nested if statement). This if statement is checking whether you are close to your car. If you are within 0.00010 of your car, the LCD will display "You should see your car." If it is not within 0.00010, then the LCD will display "Keep looking."

To use this project, you will need to switch to car mode (off) you should then see on the LCD the longitude and latitude values. Then press the button, and the LCD should display "Car is here" and then the location of your car. Next flip the switch to the locator mode (On). The LCD should display "You are here" and then the longitude and latitude position of where you are. To reset your car's location, switch back to car mode (off) and hold the push button until a blank LCD screen appears. Then you should be able to save a new GPS location.

Now that we have worked extensively with the TinyGPS library, we are going to work with the SdFat library to log GPS data to file. This will be the discussion of the next project.

■ **Note** When resetting a car's GPS location, you need to hold the push button down until "Car is here" comes back onto the LCD.

Project 5-4: GPS Data Logger

In this project, we will need to utilize the SdFat library and the NewSoftSerial library. You will also learn a new function that will allow us to use floating-point numbers. We will need all of this to create a stand-alone GPS data logger. The next section will discuss the hardware we will need for this project.

Hardware for This Project

Figure 5-8 shows some of the hardware being used in this project. (Not pictured: USB cable, 9V battery, and 9V battery connector.)

- Arduino Duemilanove (ATMega328 version) (or UNO)
- GPS shield
- microSD shield
- USB cable (or 9V battery)
- 9V battery connector if 9V is used

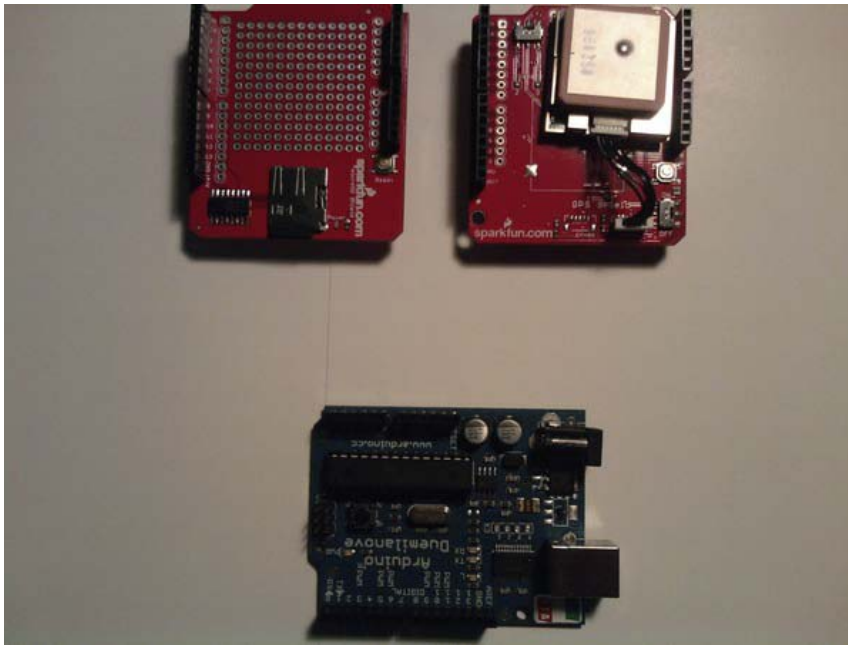


Figure 5-8. Hardware for this project

Configuring the Hardware

The first thing we will do for this project is attach the microSD shield to the Arduino. Next attach the GPS shield to the microSD shield. Figure 5-9 illustrates the hardware configuration for this project.

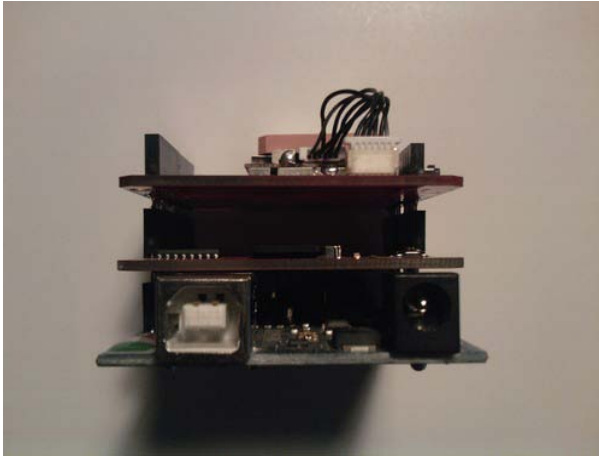


Figure 5-9. Hardware configuration for this project

Writing the Software

For this project to work correctly, we need to utilize the SdFat library, the NewSoftSerial library, and the TinyGPS library. We also need to use a function that will allow us to write the longitude and latitude data to the microSD card; this function is called the `printFloat()` function. This function will convert a floating-point variable to a string, which will allow us to send the longitude and latitude data to a microSD card. Listing 5-4 shows the code for this project.

Listing 5-4. Send Longitude and Latitude Data to a microSD Card

```
//Add the SdFat Libraries
#include <SdFat.h>
#include <SdFatUtil.h>
#include <ctype.h>
#include <NewSoftSerial.h>
#include <TinyGPS.h>

NewSoftSerial nss(2,3);
TinyGPS gps;

void printFloat(double number, int digits); // function prototype for printFloat function

//Create the variables to be used by SdFat Library
Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

char name[] = "GPSData.txt"; // holds the name of the new file
char LatData[50];           // data buffer for Latitude
char LongData[50];          // data buffer for longitude
```

```

void setup(void)
{
  Serial.begin(9600);          // Start a serial connection.
  nss.begin(4800);             // Start soft serial communication
  pinMode(10, OUTPUT);         // Pin 10 must be set as an output for the SD communication to
                                // work.
  card.init();                 // Initialize the SD card and configure the I/O pins.
  volume.init(card);           // Initialize a volume on the SD card.
  root.openRoot(volume);       // Open the root directory in the volume.
}

void loop(void)
{
  while (nss.available() > 0)
  {
    int c = nss.read();

    // Initialize Longitude and Latitude to floating-point numbers
    if(gps.encode(c))          // New valid sentence?
    {
      float latitude, longitude;

      // Get longitude and latitude
      gps.f_get_position(&latitude,&longitude);

      // Print "lat: " to LCD screen
      Serial.print("lat: ");
      // Prints latitude with 5 decimal places to the serial monitor
      Serial.println(latitude,5);

      // Print "long: " to the serial monitor
      Serial.print("long: ");
      // Prints longitude with 5 decimal places to the serial monitor
      Serial.println(longitude,5);

      delay(500);

      file.open(root, name, O_CREAT | O_APPEND | O_WRITE); // Open or create the file 'name'
                                                            // in 'root' for writing to the end of the file.
      file.print("Latitude: ");
      printFloat(latitude, 6);
      file.println("");
      file.print("Longitude: ");
      printFloat(longitude, 6);
      file.println("");
      file.close(); // Close the file.
      delay(1000); // Wait 1 second
    }
  }
}

```

```

}

void printFloat(double number, int digits)
{
    // Handle negative numbers
    if (number < 0.0)
    {
        file.print('-');
        number = -number;
    }

    // Round correctly so that print(1.999, 2) prints as "2.00"
    double rounding = 0.5;
    for (uint8_t i=0; i<digits; ++i)
        rounding /= 10.0;

    number += rounding;

    // Extract the integer part of the number and print it
    unsigned long int_part = (unsigned long)number;
    double remainder = number - (double)int_part;
    file.print(int_part);

    // Print the decimal point, but only if there are digits beyond
    if (digits > 0)
        file.print(".");

    // Extract digits from the remainder one at a time
    while (digits-- > 0)
    {
        remainder *= 10.0;
        int toPrint = int(remainder);
        file.print(toPrint);
        remainder -= toPrint;
    }
}

```

The first thing this code does is include all of the headers so that we can use the libraries that the program needs. Next we create a function prototype so that we can use the `printFloat()` function. After that, we create instances of `Sd2Card`, `SdVolume`, and `SdFile`. Then we enter the setup structure where we begin serial communication and soft serial communication; we also set digital pin 10 to an output. Then we initialize the microSD card. Next we enter the loop structure. In the loop structure, we first create a while loop that checks whether the software serial has data. If it does, we add the data from the software serial to `int c`. We then use an if statement to make sure `int c` is a valid GPS sentence. After that, we send the parsed GPS data to the serial monitor (serial port). The next bit of code opens or creates a file and appends data to the end of the file. Finally, we use the `printFloat()` function to convert the `LatData` and `LongData` to strings and then write `LatData` and `LongData` to the microSD card in this format:

```

Lat:  ** *****
Long: ** *****

```

You can use this GPS data logger stand-alone so that you can take it for a ride in a car or while you are riding your bike.

Now that we understand the basics of GPS communication, we can return to the company and see whether they have any project that needs our newfound knowledge. It just so happens that they need us to add GPS communication to the previous chapter's robot. The first thing we need to do is gather the requirements and create the requirements document.

Requirements Gathering and Creating the Requirements Document

The customer wants to add an accessory onto the robot from Chapter 4. They want a field robot that has the ability to send the GPS location of the robot and log it to a text file. They will no longer need the color LCD because it is an accessory to the robot as well. They want to send a separate command to the robot in order to get the latitude and longitude data from GPS (this robot adds on to the robot from Chapter 3 and Chapter 4). The latitude and longitude should have this format:

```
Lat:   **.*****
Long:  **.*****
```

They still want the motors controlled by serial communication (USB wire).

Now that we have our notes from the meeting, we can compile them into a requirements document.

Hardware

Figure 5-10 shows some of the hardware being used in this project. (Not pictured: USB cable and 9V battery.)

- Arduino Duemilanove (ATMega328 version)(or UNO)
- GPS shield
- microSD shield
- 512MB microSD card and card reader
- Chassis from previous robot
- USB cable
- 9V battery

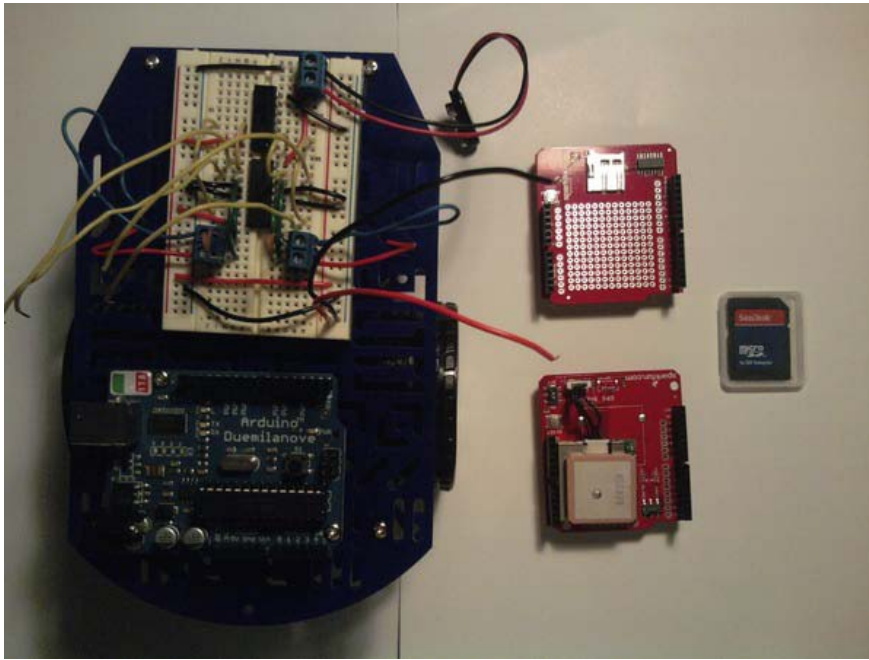


Figure 5-10. Hardware for this project

Software

The following requirements need to be met for the software of this project:

- Display latitude and longitude GPS data in this format:

Lat: **.******

Long: **.******

- Write latitude and longitude to text file.
- Use NewSoftSerial library.
- Use TinyGPS library.
- Use SdFat library.
- Remove color LCD code .

Now that we have our hardware and software requirements, we can add to the flowchart we created in Chapter 3. Figure 5-11 shows the flowchart for this project.

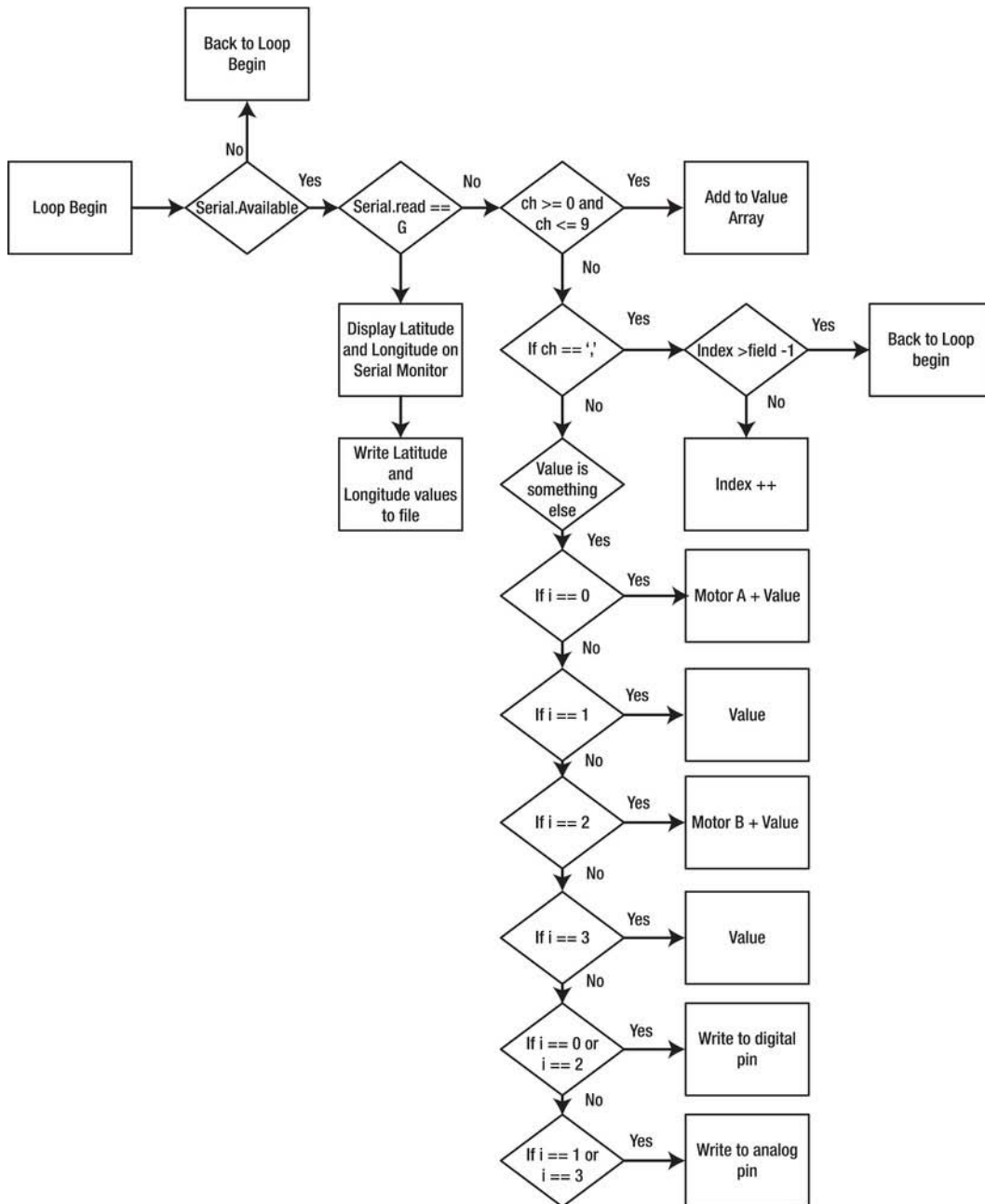


Figure 5-11. Flowchart for this project

Configuring the Hardware

The first thing we want to do is configure the GPS shield so that it uses DLINE and not UART (DLINE will use digital pins 2 and 3 for a software serial, and UART will use digital pins 0 and 1 for a hardware serial). Figure 5-12 illustrates this process.



Figure 5-12. Select DLINE on GPS shield

To configure the hardware in this project, follow these steps:

1. Connect the microSD shield to the Arduino.
2. Connect the GPS Shield to the microSD shield.
3. Attach the H-bridge pin 1 to digital pin 5 on the Arduino, attach pin 7 on the H-bridge to digital pin 4 on the Arduino, attach pin 9 on the H-bridge to digital pin 6 on the Arduino, and attach pin 10 on the H-bridge to digital pin 7 on the Arduino.
4. Connect H-bridge to power (+5V) and ground on the Arduino.

Figure 5-13 illustrates the hardware configuration for this project.

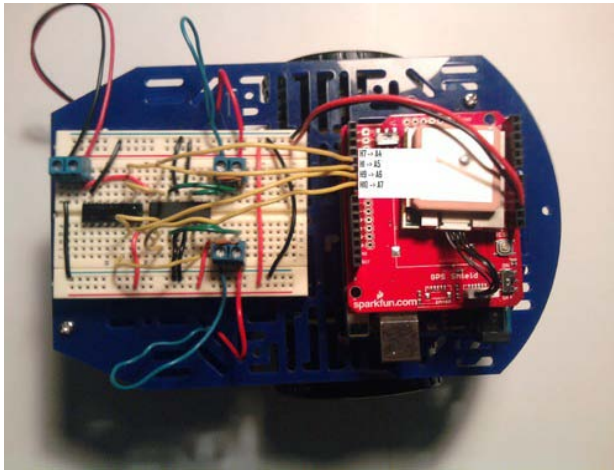


Figure 5-13. Hardware configuration for this project (“H” stands for H-bridge pin and “A” stands for Arduino pin)

The next section will discuss the software we will need to write in order to meet the requirements the company has given us.

Writing the Software

Now that the hardware is configured, we need to write some software so that the GPS data will be parsed into latitude and longitude, then printed to the serial monitor, and finally written to a text file whenever the user enters “G” into the serial monitor. The latitude and longitude data also has to be in a format that the user can understand. Here is the format:

```
Lat:   **.*****
Long:  **.*****
```

Listing 5-5 presents the code for this project.

Listing 5-5. Send GPS location to a microSD card and control motors through the Serial Monitor

```
#include <SdFat.h>
#include <SdFatUtil.h>
#include <ctype.h>

#include <TinyGPS.h>

#include <NewSoftSerial.h>

TinyGPS gps;

NewSoftSerial nss(2,3);
```

```

void printFloat(double number, int digits); // function prototype for printFloat function

int numCount = 0;

const int fields = 4; // how many fields are there? right now 4
int motorPins[] = {4,5,7,6}; // Motor Pins
int index = 0;           // the current field being received
int values[fields];      // array holding values for all the fields

Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

char name[] = "GPSData.txt";    // holds the name of the new file
char LatData[50];              // data buffer for Latitude
char LongData[50];

void setup()
{
  Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud
  nss.begin(4800);   // Begins software serial communication

  pinMode(10, OUTPUT); // Pin 10 must be set as an output for the SD communication to
                        // work.
  card.init();         // Initialize the SD card and configure the I/O pins.
  volume.init(card);   // Initialize a volume on the SD card.
  root.openRoot(volume);

  for (int i; i <= 3; i++) // set LED pinMode to output
  {
    pinMode(motorPins[i], OUTPUT);
    digitalWrite(motorPins[i], LOW);
  }

  Serial.println("The Format is: MotoADir,MotoASpe,MotorBDir,MotoBSpe\n");
}

void loop()
{
  if( Serial.available())
  {
    char ch = Serial.read();

    if (ch == 'G') // if Serial reads G
    {
      digitalWrite(motorPins[0], LOW);
      digitalWrite(motorPins[2], LOW);
      analogWrite(motorPins[1], 0);
      analogWrite(motorPins[3], 0);
    }
  }
}

```

```

while (numCount == 0)
{
  if (nss.available() > 0) // now gps device is active
  {
    int c = nss.read();
    if(gps.encode(c))      // New valid sentence?
    {

      // Initialize Longitude and Latitude to floating-point numbers
      float latitude, longitude;

      // Get longitude and latitude
      gps.f_get_position(&latitude,&longitude);

      Serial.print("Lat: ");
      // Prints latitude with 5 decimal places to the serial monitor
      Serial.println(latitude,7);

      Serial.print("long: ");
      // Prints longitude with 5 decimal places to the serial monitor
      Serial.println(longitude,7);

      file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file
                                                                // 'name'
                                                                // in 'root' for writing
                                                                // to the
                                                                // end of the file.

      file.print("Latitude: ");
      printFloat(latitude, 6);
      file.println("");
      file.print("Longitude: ");
      printFloat(longitude, 6);
      file.println("");
      file.close();      // Close the file.
      delay(1000);       // Wait 1 second

      numCount++;
    }
  }
}
}
else if(ch >= '0' && ch <= '9') // If the value is a number 0 to 9
{
  // add to the value array
  values[index] = (values[index] * 10) + (ch - '0');
}
else if (ch == ',') // if it is a comma
{
  if(index < fields -1) // If index is less than 4 - 1...
    index++; // increment index
}
else

```

```

{
  for(int i=0; i <= index; i++)
  {
    if (i == 0 && numCount == 0)
    {
      Serial.println("Motor A");
      Serial.println(values[i]);
    }
    else if (i == 1)
    {
      Serial.println(values[i]);
    }
    if (i == 2)
    {
      Serial.println("Motor B");
      Serial.println(values[i]);
    }
    else if (i == 3)
    {
      Serial.println(values[i]);
    }

    if (i == 0 || i == 2) // If the index is equal to 0 or 2
    {
      digitalWrite(motorPins[i], values[i]); // Write to the digital pin 1 or 0
      // depending what is sent to the arduino.
    }

    if (i == 1 || i == 3) // If the index is equal to 1 or 3
    {
      analogWrite(motorPins[i], values[i]); // Write to the PWM pins a number between
      // 0 and 255 or what the person has enter
      // in the serial monitor.
    }

    values[i] = 0; // set values equal to 0
  }

  index = 0;
  numCount = 0;
}

}

void printFloat(double number, int digits)
{

```

```

// Handle negative numbers
if (number < 0.0)
{
    file.print('-');
    number = -number;
}

// Round correctly so that print(1.999, 2) prints as "2.00"
double rounding = 0.5;
for (uint8_t i=0; i<digits; ++i)
    rounding /= 10.0;

number += rounding;

// Extract the integer part of the number and print it
unsigned long int_part = (unsigned long)number;
double remainder = number - (double)int_part;
file.print(int_part);

// Print the decimal point, but only if there are digits beyond
if (digits > 0)
    file.print(".");

// Extract digits from the remainder one at a time
while (digits-- > 0)
{
    remainder *= 10.0;
    int toPrint = int(remainder);
    file.print(toPrint);
    remainder -= toPrint;
}
}

```

Most of this code is from the previous robot, so I will go over only the new pieces of code that this program has. At the beginning of this program, we include the header files for the NewSoftSerial library, TinyGPS library, and SdFat library. Then we create instances of those libraries with these pieces of code:

```

TinyGPS gps;

NewSoftSerial nss(2,3);

Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

```

Then we create the file name we will use to store the latitude and longitude data and create the character arrays that will hold the latitude and longitude values. The next piece of new code is not encountered until we get inside the setup structure, where we begin soft serial communication. After that in the loop structure, we see this code at the beginning:

```

if (ch == 'G') // if Serial reads G
{

```

```

digitalWrite(motorPins[0], LOW);
digitalWrite(motorPins[2], LOW);
analogWrite(motorPins[1], 0);
analogWrite(motorPins[3], 0);

while (numCount == 0)
{
  if (nss.available() > 0) // now gps device is active
  {
    int c = nss.read();
    if(gps.encode(c))      // New valid sentence?
    {

      // Initialize Longitude and Latitude to floating-point numbers
      float latitude, longitude;

      // Get longitude and latitude
      gps.f_get_position(&latitude,&longitude);

      Serial.print("Lat: ");
      // Prints latitude with 5 decimal places to the serial monitor
      Serial.println(latitude,7);

      Serial.print("long: ");
      // Prints longitude with 5 decimal places to the serial monitor
      Serial.println(longitude,7);

      file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file
                                                                // 'name'
                                                                // in 'root' for writing
                                                                // to the
                                                                // end of the file.

      file.print("Latitude: ");
      printFloat(latitude, 6);
      file.println("");
      file.print("Longitude: ");
      printFloat(longitude, 6);
      file.println("");
      file.close();      // Close the file.
      delay(1000);       // Wait 1 second

      numCount++;
    }
  }
}

```

This is the main code that we added. The if statement uses the condition that if the serial port reads G, then it needs to set all of the H-bridge pins that are connected to the Arduino to 0 or LOW. Next we use a while loop with the condition `numCount == 0` (`numCount` was initialized at the very beginning of the program; this variable is used to turn on and off the GPS data). If this condition is true, we make sure that there is information waiting on the software serial (`nss`). If this is true, then we set `int c` to the value

that is on nss. After that, we make sure that we are receiving valid GPS data. If we are receiving valid data, then we parse through the GPS data and print latitude and longitude data to the serial monitor. Then we send latitude and longitude values to a text file and increment the numCount variable. Now that we have written the software, we may have encountered a few bugs. The next section will discuss debugging this project's software.

Debugging the Arduino Software

First, if you copied and pasted the code from the previous project, you may have run into problems that involve the color LCD. Make sure you removed all of the color LCD code, and also if you used some of the code from the monochrome LCD project in the gpsData() function, make sure you change these values from lcd.write() to Serial.write() functions. I also want to go over a good technique to find errors within your code. You can use Serial.write() functions to find out where your code is at a certain time or if it is entering a certain conditional statement or even a loop. Listing 5-6 is an example of using this to make sure that your program is entering a conditional statement before you implement code that may take you hours to fix (probably not hours for this example, but think about how long it would take for thousands of lines).

Listing 5-6. Debugging your code with serial commands

```
char ch;

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

  Serial.println("Entered Setup Structure");
  delay(1000);
}

void loop()
{
  ch = Serial.read();

  if (ch == 'g')
  {
    Serial.println("First If Statement entered");
  }
  else if (ch == 's')
  {
    Serial.println("Second If Statement entered");
  }
  else
  {
    Serial.println("Else Statement entered");
  }
  delay(1000);
}
```

This type of debugging can be used at the beginning or end of your project. I like to use this debugging technique when trying to figure out whether my architecture is correct. Now that we have debugged the software for this project and gone over a new technique, we can troubleshoot the hardware.

Troubleshooting the Hardware

Well, you also may have run into a few hardware issues with this project. If you did, we will run through a few tips to make this project work correctly. First, if your GPS is not giving you any data, you may want to make sure it is on (we all have done this before) and also make sure you have selected DLINE. Next, make sure that the GPS's LED is blinking. If it is a solid green light, that means the GPS has not found a fix yet; if it is blinking, then it has a fix. Also, you may want to make sure your serial monitor is set to the correct baud rate; in this case, 9600 will do for this project. If you are having problems getting the microSD shield working, make sure you are using a microSD that is no larger than 2GB and that it is formatted to FAT16 or FAT32. If problems persist with the microSD shield, you should contact the shop you bought it from because you may have a soldering problem (this actually happens but not often, though). If your motors are not going in the correct direction, you may need to switch your power and ground on each of the motors (just make sure you are consistent on the H-bridge's motor terminals). Also make sure that all of your wires are connected properly on the solderless breadboard (see Chapter 4's final project). If everything is working and you are having no problems, then move to the next step, which is the finished prototype.

Finished Prototype

Well, this project is now at a stage where everything is working and has been tested thoroughly. Figure 5-14 illustrates the finished prototype.

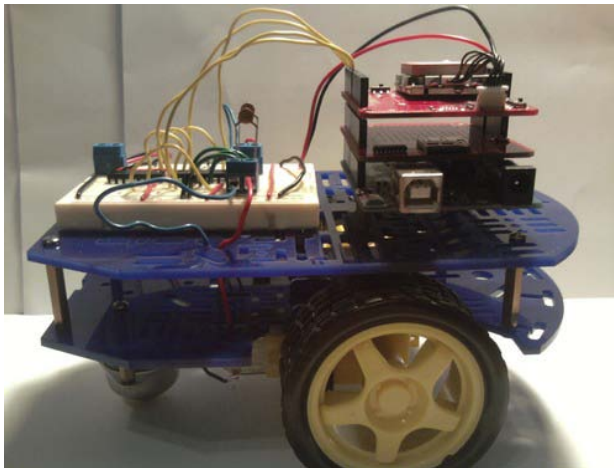


Figure 5-14. Finished prototype

Summary

This chapter has been aimed at getting you ready to experiment with GPS communication. We first ventured into NMEA commands, and I summarized their uses and values. After that we talked about the TinyGPS library and the SdFat library. Also, throughout this chapter, we went over various projects that implement GPS in various ways. They were writing raw GPS data to the serial monitor, writing GPS data to a monochrome LCD, creating a car finder, and creating a GPS data logger. Then we worked on another engineering project that utilized the engineering process and GPS communication. Now that you understand GPS communication, you can make your own projects with the GPS shield.

Interlude: Home Engineering from Requirements to Implementation

In this chapter, we are going to take a little break from the usual scheme of things so you can learn about various sensors and how to use them in your projects. We will use several new pieces of hardware: photoresistors and tilt, flex, FSR (force sensitive resistor), digital temperature and humidity, and temperature sensors. We will also use two new libraries to communicate with the temperature and humidity sensors: DHT22 (thanks to nethoncho) and the Wire library (thanks to the Arduino development team). These are all key parts in completing our hypothetical customer's request for a wireless temperature monitor that uses Bluetooth communication.

■ **Note** We will be using two different temperature sensors in this chapter so that you can learn more about I2C circuits and how to use them.

We'll start by discussing some of the math we need to use a voltage divider circuit. We'll then turn our attention to each sensor and both libraries. After we have a better understanding of the sensors, we will be doing several projects that utilize each one of the sensors.

Understanding the Voltage Divider

When using a voltage divider, you may have to figure out what values you need for the resistors. Say we want to find voltage on a 6V battery; this can be done with a voltage divider. See Figure 6-1.

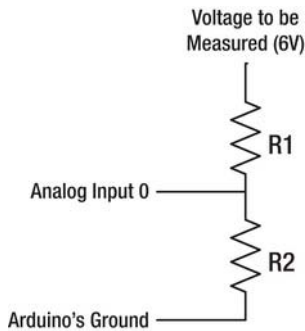


Figure 6-1. Voltage divider circuit

The first step is to make sure you do not use a ton of amps from the battery. We will use 100uA, and the voltage out will be 4V. Now that we have these values, we can figure out the total resistance of our voltage divider by using Ohm's Law.

$$RT = 6V / 100\mu A$$

$$RT = 60\text{kohms}$$

Next, we need to solve for R2 as follows:

$$R2 = (\text{voltage out} / \text{voltage in}) * \text{Total Resistance}$$

$$R2 = (4V / 6V) * 60\text{kohms}$$

$$R2 = 40\text{kohms}$$

Now that we have the value of R2, we can find R1 using this equation:

$$R1 = RT - R2$$

$$R1 = 60\text{kohms} - 40\text{kohms}$$

$$R1 = 20\text{kohms}$$

This should get you started on using a voltage divider. We will be using voltage dividers extensively in this chapter to scale sensors to the analog input pins. For additional information on voltage dividers, please see http://en.wikipedia.org/wiki/Voltage_Divider. The next section will discuss the new hardware in this chapter.

Hardware Explained: Sensors

Throughout this chapter, we will be using sensors on the Arduino so that it can recognize its surroundings. Many sensors are available today, but we will focus on only a few of them: photoresistors, and the tilt, flex, force sensitive resistor (FSR), digital temperature and humidity, and digital temperature sensors. The next few sections will discuss each of these sensors and how they achieve their abilities to understand their surroundings.

Photoresistor

A photoresistor (see Figure 6-2) gets its resistance from the sun. If it is dark, the sensor will read up in the millions of Ohms, but as light is shined at it, it will have a lower resistance. This sensor requires a voltage divider to get a correct signal.



Figure 6-2. Photoresistor

Tilt Sensor

The tilt sensor (see Figure 6-3) is used to check whether or not something is level. It has a small metal ball inside the cap; when it is tilted, the ball connects with the outer cylinder and sends HIGH to the Arduino (or any other microcontroller). This sensor acts like a switch and requires a pull-up resistor.

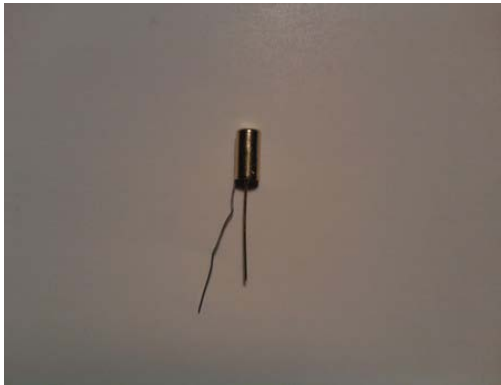


Figure 6-3. Tilt sensor

Flex Sensor

The flex sensor, as the name says, will tell you how much something is bending. This sensor is a large resistor, and when it is flat, it is at 25kohms, and when it is fully bent, it will go up to 125kohms. You need

to use a voltage divider with this sensor to get a correct reading from the analog inputs on the Arduino. Figure 6-4 shows the flex sensor.

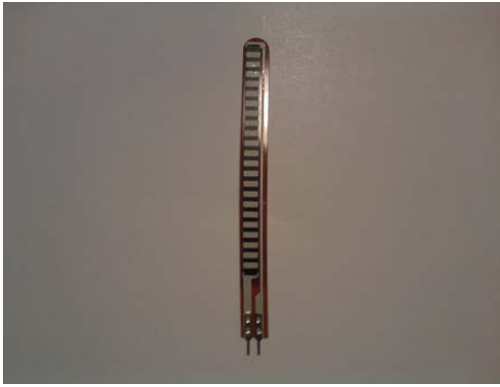


Figure 6-4. Flex sensor

Force Sensitive Resistor (FSR)

The FSR sensor (shown in Figure 6-5) can detect force applied to it. It also is a large resistor and will go past 1Mohm when it is not being pressed. The FSR I am using has limits of 100g and 10kg. You need a voltage divider to get connect data from the analog inputs on the Arduino for this sensor as well.



Figure 6-5. Force sensitive resistor (FSR)

Digital Temperature and Humidity Sensor

The DHT22 sensor shown in Figure 6-6 can detect temperature (C and F) and humidity. It has everything it requires built into it, so it will work very well with the Arduino. This sensor is used in conjunction with the DHT22 Library (which will be discussed later in this chapter).

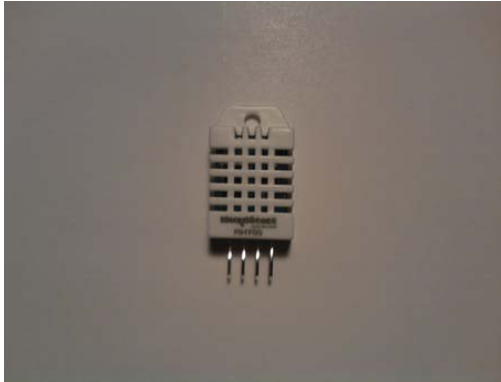


Figure 6-6. Temperature and humidity sensor

Digital Temperature Sensor (I2C)

We will use the inter-integrated circuits (I2C) digital temperature sensor in the final project; this sensor has a very high resolution, so it is very accurate. It sensor requires us to use the Wire library (which will be discussed later in this chapter) to communicate with it. Figure 6-7 illustrates the digital temperature sensor. Since this sensor is a bit more complicated than the others, here is a link to the data sheet if you'd like more information: <http://www.sparkfun.com/datasheets/Sensors/Temperature/tmp102.pdf>.



Figure 6-7. Digital temperature sensor

Libraries Explained: Wire Library and DHT22 Library

Now that we have discussed the new hardware, we need to go over some new libraries that will allow us to communicate with both of the temperature sensors: the Wire library and the DHT22 library.

Wire Library

This library will allow us to communicate with I2C with the Arduino. The Wire library comes with the Arduino IDE, so there is no need to download it. To use the Wire library, you will need to include the `wire.h` header file, like this:

```
#include <Wire.h>
```

This section will discuss how to use the library and its commands: `Wire.begin()`, `Wire.requestFrom()`, and `Wire.receive()` (these are just a few of the commands, but they will be used in the last part of this chapter when we assemble our company's project):

- `Wire.begin()`: This function initiates the Wire library. To use it, add it in the setup structure just like the `Serial.begin()` function.
- `Wire.requestFrom()`: This function requests data from the I2C device. The data can then be retrieved using the `Wire.receive()` function.
- `Wire.receive()`: This function gathers a byte of data from the I2C device.

That is it for the Wire library; while it is not the most verbose library, it can still be tricky to understand. We will use this library in the last project to communicate with the TMP102 digital temperature sensor.

DHT22 Library

The DHT22 library is used in conjunction with the DHT22 sensor. To download this library, go to the following link: <http://github.com/nethoncho/Arduino-DHT22>. This sensor will allow us to receive temperature data and humidity data. The first thing you will need to do is include the header for the DHT22 library, like this:

```
#include <DHT22.h>
```

Here are a few functions from the DHT22 library:

- `readData()`: This function reads in data from the DHT22 sensor just like the `analogRead()` function reads in data from the analog inputs.
- `getTemperatureC()`: This function will return the temperature in Celsius. It can later be converted to Fahrenheit with a simple formula we will discuss later in this chapter.
- `getHumidity`: This function will return the humidity in the area.

Now that we have gone over both of the libraries we will use for this chapter, we can move on to the projects for this chapter. We will be using all of the sensors we discussed in the previous section, including the TMP102 sensor.

Understanding the Basics of Sensors

In this section, Projects 6-1 through 6-5 will help you understand how to use sensors. Some of these projects will incorporate a couple of pieces of hardware we have used in the past (microSD shield and the monochrome LCD), but for the most part, you will be learning how to configure the sensors to work

with the Arduino. The projects are flower pot analyzer, using a FSR sensor, using a flex sensor, digital level using a tilt sensor, and using a DHT22 sensor with a monochrome LCD.

Project 6-1: Flower Pot Analyzer

In this project, we will be using the photoresistor to tell us whether we are putting flowers in a sunny spot. We will need to use an analog input and store the data to a microSD card.

Gathering the Hardware

Figure 6-8 shows some of the hardware being used in this project:

- Arduino Duemilanove (or UNO)
- microSD shield
- Photoresistor
- 1-to-10kohm resistor
- Small solderless breadboard
- Enclosure
- Extra wire

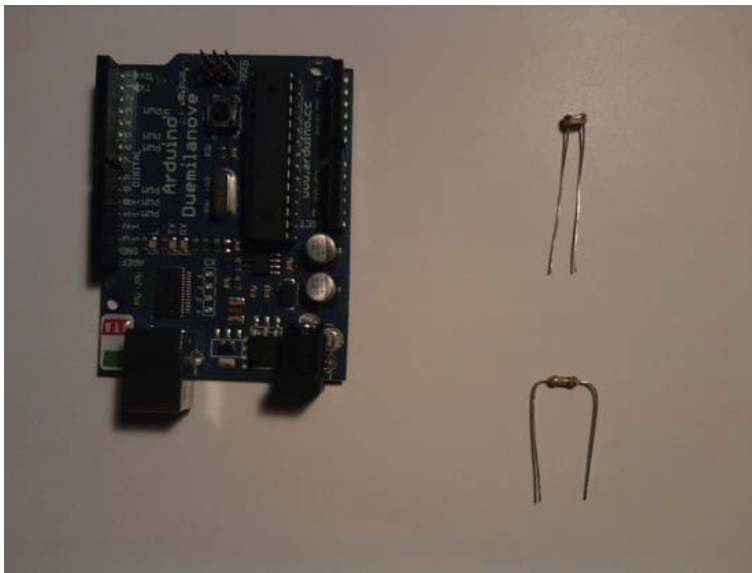


Figure 6-8. Hardware for this project (Not pictured: small solderless breadboard, enclosure, and extra wire)

Configuring the Hardware

The following steps will guide you through the hardware configuration for this project:

1. Create the enclosure for the flower pot analyzer. We will start with cutting a square in the lid of the enclosure (see Figure 6-9).



Figure 6-9. Square cut out of the lid of the enclosure

2. Put double-sided tape around the square we just cut out.
3. Take a piece of clear plastic, and cut it to a size that will fit on the hole on the plastic lid (see Figure 6-10).



Figure 6-10. Clear screen attached with double-sided tape

4. Now that the enclosure has a sunroof, we need to add the small breadboard into the enclosure (see Figure 6-11).

5. Put the microSD shield on top of the Arduino, and set it inside the enclosure.
6. Connect the photoresistor to the Arduino, and connect ground to one end of the photoresistor.
7. Connect the 1kohm resistor to the other end of the photoresistor.



Figure 6-11. Add bread board to project

8. Connect analog pin 0 to the power end of the photoresistor.
9. Connect the power (+5V) to the other end of the 1kohm resistor.
10. Connect a 9V battery connector to the Arduino. You should now have the hardware configured. Figures 6-12 and 6-13 illustrate the final hardware configuration.

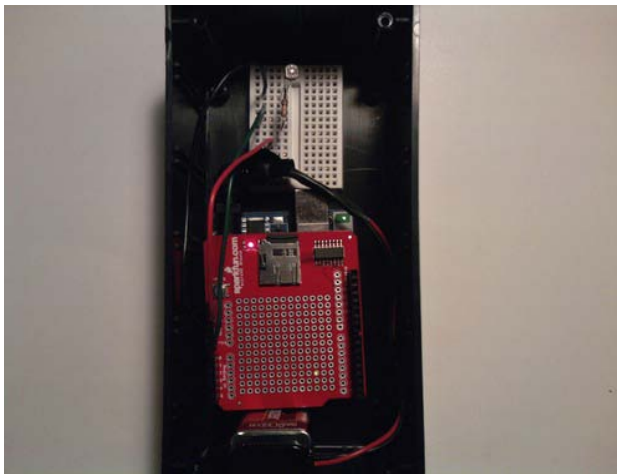


Figure 6-12. Hardware configuration, without the cover



Figure 6-13. *Hardware Configuration, with the sunroof*

You may have noticed that the photoresistor uses a special circuit called a voltage divider; this circuit is useful, because it will split the voltage into different parts allowing us to scale a sensor from 0 (When it is bright and 1023 when it is dark) to 1023.

■ **Note** I do not recommend using this project in a rain storm, as the enclosure is not waterproof.

Writing the Software

We will need to use the SdFat library from the previous chapter. Also, we will need to use analog pin 0. The microSD shield will use the SPI pins to send weather data to a microSD card. Listing 6-1 presents the code for this project.

Listing 6-1. *Flower Pot Analyzer*

```
#include <SdFat.h>
#include <SdFatUtil.h>

//Create the variables to be used by SdFat Library
Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

char name[] = "weather.txt";    // holds the name of the new file
int photoPin = A0;
int weatherVal = 0;
```

```

void setup()
{
  pinMode(photoPin, INPUT);
  pinMode(10, OUTPUT);    // Pin 10 must be set as an output for the SD communication to
                          // work.
  card.init();            // Initialize the SD card and configure the I/O pins.
  volume.init(card);      // Initialize a volume on the SD card.
  root.openRoot(volume);  // Open the root directory in the volume.
}

void loop()
{
  weatherVal = analogRead(photoPin);

  if (weatherVal >= 800)
  {
    file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file 'name'
                                                             // in 'root' for writing
                                                             // to the end of the file.

    file.println("Dark");
    file.close();      // Close the file.
    delay(1000);       // Wait 1 second
  }
  else if (weatherVal >= 300 && weatherVal <= 500)
  {
    file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file 'name'
                                                             // in 'root' for writing
                                                             // to the end of the file.

    file.println("Cloudy");
    file.close();      // Close the file.
    delay(1000);       // Wait 1 second
  }
  else if (weatherVal < 300 && weatherVal >= 0)
  {
    file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file 'name'
                                                             // in 'root' for writing
                                                             // to the end of the file.

    file.println("Sunny");
    file.close();      // Close the file.
    delay(1000);       // Wait 1 second
  }
}

```

Let's review the code.

The first thing we do in this program is include the header files so that we can use the SdFat library. Next, we create instances of the SdFat library that we will use throughout the rest of the program. After that, we create the file the data will be stored in, set `photoPin` to analog pin 0, and create the `weatherVal` variable. We then enter the `setup` structure, where we set the analog pin 0 to an input and initialize the SD card and I/Os. Next, we enter the `loop` structure where we set the `weatherVal` equal to the reading on analog pin 0. We then go through a few `if` statements and `else-if` statements. The `if` statements will

only be true if weatherVal is greater-than-or-equal-to 800. If this is true, then "Dark" will be written to the microSD card. The first else-if statement will only be true if weatherVal is greater-than-or-equal-to 300, and weatherVal is less-than-or-equal-to 500. If this is true, then "Cloudy" will be sent to the microSD card. The last else-if statement will only be true if weatherVal is less-than 300, and weatherVal is greater-than-or-equal-to 0. If this is true, then "Sunny" will be sent to the microSD card. When you run the program, depending on the light in the flower pot, data will be sent to the microSD card explaining how much sun the flowers are receiving every second. You can make this project more robust by adding in a real time clock to make a fully featured data logger.

Project 6-2: Using a FSR Sensor

This sensor will allow us to detect weight by using the `map()` function as we have in the previous chapters. We will then display weight on the serial monitor.

Gathering the Hardware

Figure 6-14 shows some of the hardware being used in this project:

- Arduino Duemilanove (or UNO)
- 10kohm resistor
- Force Sensitive Resistor (FSR)
- Solderless breadboard
- Extra wire

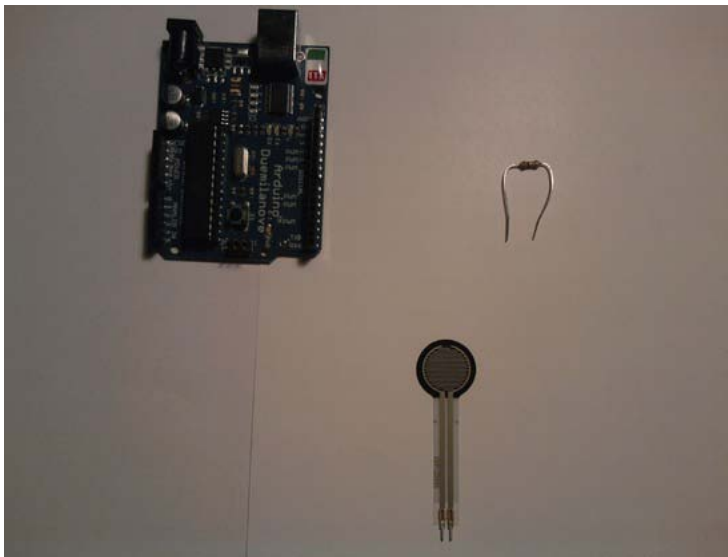


Figure 6-14. Hardware for this project (Not pictured: small solderless breadboard, enclosure, and extra wire)

Configuring the Hardware

The following steps will guide you through the hardware configuration for this project.

1. Connect the FSR to the breadboard.
2. Connect the 10kohm resistor to one of the FSR's pins.
3. Connect ground to the other pin of the FSR.
4. Connect analog pin 0 to the same pin to which the resistor is connected.
5. Connect power (+5V) to the other end of the 10kohm resistor.
6. Finally, put some double-sided tape on the back of the FSR and connect it to the breadboard. Figure 6-15 illustrates the hardware configuration for this project.

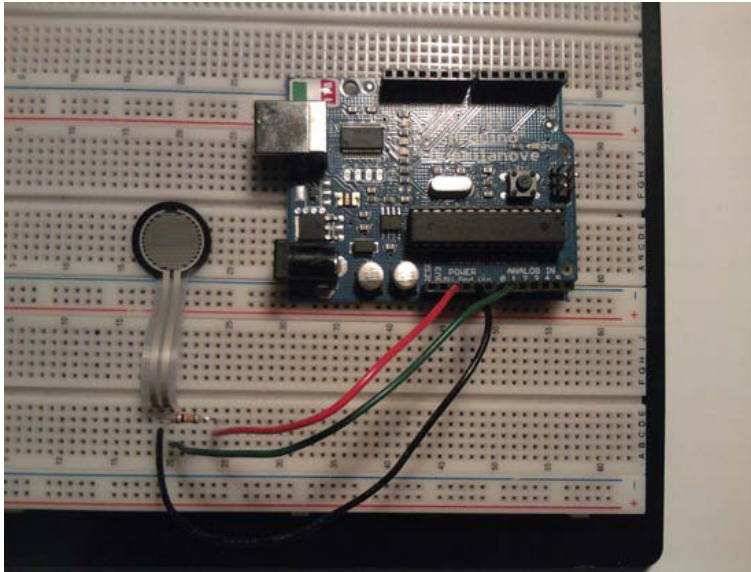


Figure 6-15. Hardware configuration for this project

Writing the Software

We will need to use an analog pin for the FSR and write the weight information in a user-friendly way to the serial monitor. Listing 6-2 provides the code for this project.

Listing 6-2. Write Weight to the Serial Monitor

```
int forcePin = A0;
int forceVal = 0;
int scaleVal = 0;
```

```

void setup()
{
  Serial.begin(9600);
  pinMode(forcePin, INPUT);
}
void loop()
{
  forceVal = analogRead(forcePin);
  scaleVal = map(forceVal, 1023, 0, 0, 20); // This is reverse mapping. The reason
                                           // we are doing reverse mapping is because
                                           // the FSR reads a high resistance when no
                                           // weight is applied to the FSR and a lower
                                           // resistance when weight is sensed.

  Serial.print("Pounds ");
  Serial.println(scaleVal);
  delay(1000);
}

```

This code is very basic; all it is doing is mapping the FSR reading from 0 to 20lbs. These sensors are not very accurate, but they get the job done. Now that you understand how to use an FSR, we can move on to learning about the Flex sensor.

■ **Note** You may have to adjust your `map()` function (see Chapter 3 for more information on the `map()` function) depending on your FSR sensor.

Project 6-3: Using a Flex Sensor

This is another short project just to get us used to using the Flex sensor (we will be using this sensor again in Chapter 8). This sensor is handy if you need to do limit testing for flexibility of a material or sensing if something has moved (like a window).

Hardware for this Project

Figure 6-16 shows some of the hardware being used in this project:

- Arduino Duemilanove (or UNO)
- Flex sensor
- 100kohm resistor
- Solderless breadboard
- Extra wire

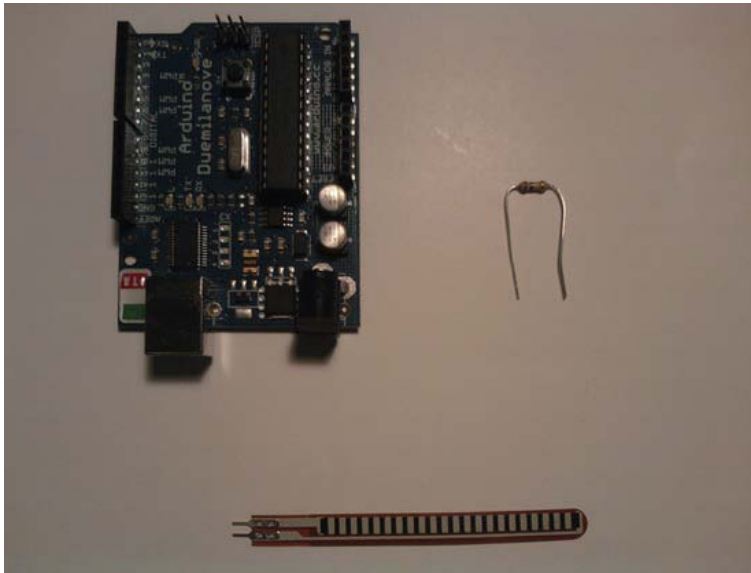


Figure 6-16. Hardware for this project (not pictured: solderless breadboard and extra wire)

Configuring the Hardware

The following steps will guide you through the hardware configuration for this project.

1. Connect the Flex sensor to the solderless breadboard.
2. Connect ground from the Arduino to one of the Flex sensor's pins.
3. Connect the 100kohm resistor to the other pin on the Flex sensor.
4. Connect power (+5V) to the other end of the 100kohm resistor, and connect analog pin 0 to the same pin on the Flex sensor that has the 100kohm resistor attached to it. Figure 6-17 illustrates the hardware configuration for this project.

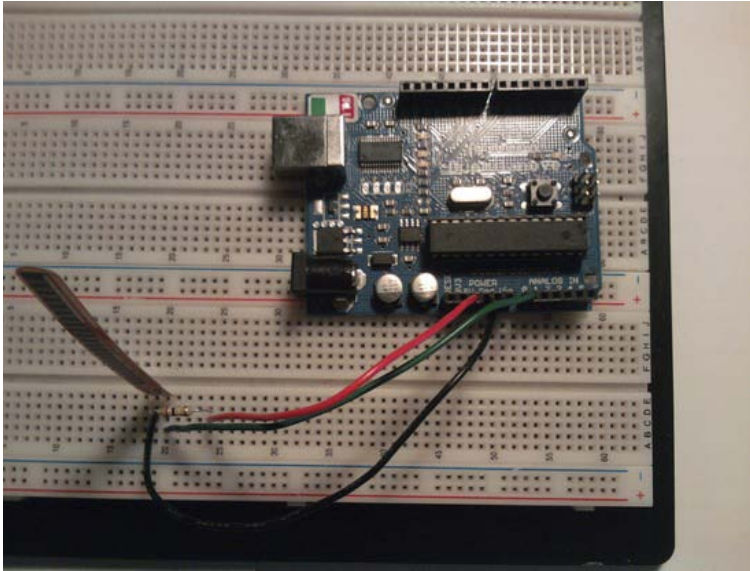


Figure 6-17. Hardware configuration for this project

Writing the Software

This program will be set up just like the previous project, except we will have to scale the values differently because we are using a 100kohm resistor. See Listing 6-3.

Listing 6-3. Send Flex Information to the Serial Monitor

```
int flexPin = A0;
int flexVal = 0;

void setup()
{
  Serial.begin(9600);
  pinMode(flexPin, INPUT);
}
void loop()
{
  flexVal = analogRead(flexPin);
  flexVal = map(flexVal, 200, 1023, 0, 1023);
  Serial.print("Flex: ");
  Serial.println(flexVal);
  delay(1000);
}
```

This program is just like the previous project except we use different parameters to scale the flex sensor properly. Without any scaling, you may see numbers from 200 to 1023, but because we scale the

values, the serial monitor will display values from 0 to 1023. In the next project, we will be creating a stand-alone level with an LCD.

Project 6-4: Digital Level

While our digital level may not tell you how many degrees you are off on that kitchen counter, this project can tell you that it is not level. We will be using a monochrome LCD for this project so that we can bring the digital level anywhere.

Gathering the Hardware

Figure 6-18 shows some of the hardware being used in this project:

- Arduino Duemilanove (or UNO)
- Tilt sensor
- Monochrome LCD
- 10K potentiometer
- 1kohm resistor
- Solderless bread board
- Extra wire

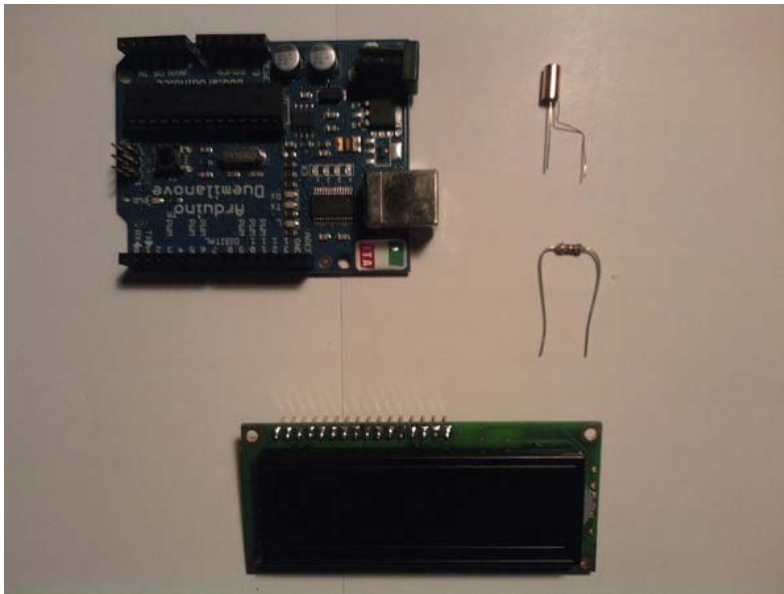


Figure 6-18. Hardware for this project (Not pictured: solderless breadboard, 10K potentiometer, and extra wire)

Configuring the Hardware

To configure this project, we first need to attach the monochrome LCD to the solderless breadboard.

1. Attach the 10k potentiometer to the solderless breadboard.
2. Connect pins 1, 5, and 16 on the LCD to ground.
3. Connect pins 2 and 15 on the LCD to power (+5V).
4. Connect ground and power up to the potentiometer.
5. Connect the potentiometer's wiper to pin 3 on the LCD.

Now that the potentiometer and the monochrome LCD's power and ground lines are connected, we need to connect the RS, E, and data bus lines from the monochrome LCD to the Arduino.

6. Connect pin 4 on the LCD to digital pin 7 on the Arduino.
7. Connect pin 6 on the LCD to digital pin 8 on the Arduino.
8. Connect pins 11, 12, 13, and 14 to digital pins 9, 10, 11, 12 on the Arduino. The monochrome LCD should now be connected to the Arduino.

Now, we need to connect the tilt sensor.

9. Connect the tilt sensor to the solderless breadboard.
10. Connect ground to one end of the tilt sensor. After that, connect a 1kohm resistor to the other end of the tilt sensor.
11. Connect power (+5V) to the other end of the 1kohm resistor.
12. Connect digital pin 4 to the same lead on the tilt sensor that has the 1kohm resistor.

That should do it for the hardware configuration, which is illustrated in Figure 6-19.

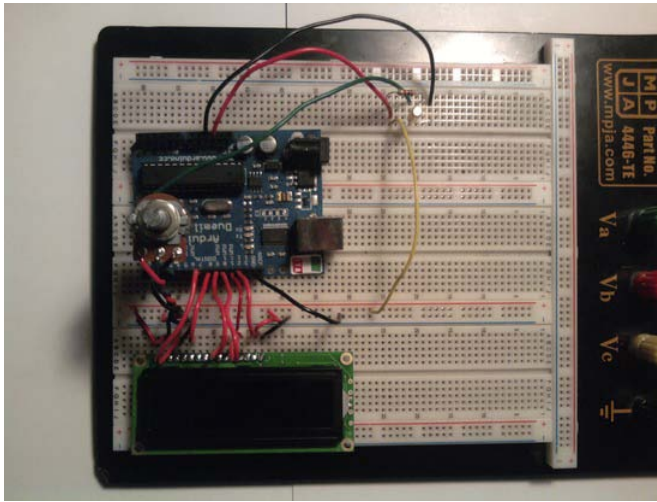


Figure 6-19. Hardware configuration for this project

Writing the Software

The program for this project will need to communicate with the monochrome LCD, so we will need to use the LiquidCrystal library as we have for the past couple of chapters. We will also need to use a digital pin for the tilt sensor (because we are using a pull-up resistor, we do not have to use a `digitalWrite()` function in the setup structure). See Listing 6-4.

Listing 6-4. Digital Level

```
// include the library code:
#include <LiquidCrystal.h>

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7,8,9,10,11,12);

int tiltPin = 4;
int tiltState = 0;

void setup()
{
  pinMode(tiltPin, INPUT);
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  lcd.clear();
}

void loop()
{
  tiltState = digitalRead(tiltPin);
```

```

    if(tiltState == HIGH)
    {
        lcd.home();
        lcd.print("Not Level");
        delay(1000);
        lcd.clear();
    }
    if(tiltState == LOW)
    {
        lcd.setCursor(0,1);
        lcd.print("Level");
        delay(1000);
        lcd.clear();
    }
}

```

In this code, we first include the header files so that we can use the LiquidCrystal library. Then, we create an instance of the LiquidCrystal class. After that, we initialize tiltPin and tiltState. Next, we enter the setup structure, where we set tiltPin to be an input and set up the monochrome LCD. After that, we enter the loop structure to set tiltState equal to the digital read on pin 4 (tiltPin). Next, we have a few conditional statements that will write “Not Level” to the LCD if the tilt sensor is HIGH, or “Level” if the tilt sensor is LOW. The next project we will be using one of the new libraries we discussed at the beginning of this chapter. We will also be working with a new sensor that can detect humidity and temperature.

Project 6-5: Using a DHT22 Sensor with a Monochrome LCD

In this project, we will be using a new library called the DHT22 library, which will allow us to use the DHT22 sensor. Also, we will need to use a monochrome LCD to display the humidity and temperature in degrees Fahrenheit.

Gathering the Hardware

Figure 6-20 shows some of the hardware being used in this project:

- Arduino Duemilanove (or UNO)
- Monochrome LCD
- DHT22 sensor
- 10k trimpot (or 10k potentiometer)
- 10kohms resistor
- Solderless breadboard
- Extra wire



Figure 6-20. Hardware for this project (Not pictured: solderless breadboard, 10kohm resistor, and extra wire)

Configuring the Hardware

Just like the previous project, we will be using the monochrome LCD and the Arduino to display humidity and temperature data:

1. Attach the monochrome LCD to the solderless breadboard.
2. Attach the 10k trimpot to the solderless breadboard.
3. Connect pins 1, 5, and 16 on the LCD to ground.
4. Then connect pins 2 and 15 on the LCD to power (+5V).
5. Connect ground to the trimpot. After that, connect the trimpot's wiper to pin 3 on the LCD.

Now that the trimpot and the monochrome LCD's power and ground lines are connected, we need to connect the RS, E, and data bus lines from the monochrome LCD to the Arduino:

6. Connect pin 4 on the LCD to digital pin 7 on the Arduino.
7. Connect pin 6 on the LCD to digital pin 8 on the Arduino.
8. Connect pins 11, 12, 13, and 14 to digital pins 9, 10, 11, 12 on the Arduino.

The monochrome LCD should now be connected to the Arduino. Now, we need to connect the DHT22 sensor to the Arduino:

9. Connect the DHT22 to the solderless breadboard.
10. Connect the 10kohm resistor from pin 1 on the DHT22 and pin 2 on the DHT22.
11. Connect pin 1 to power (+5V) and pin 2 to digital pin 2 on the Arduino.
12. Connect pin 4 on the DHT22 to ground on the Arduino. Figure 6-21 illustrates the hardware configuration for this project.

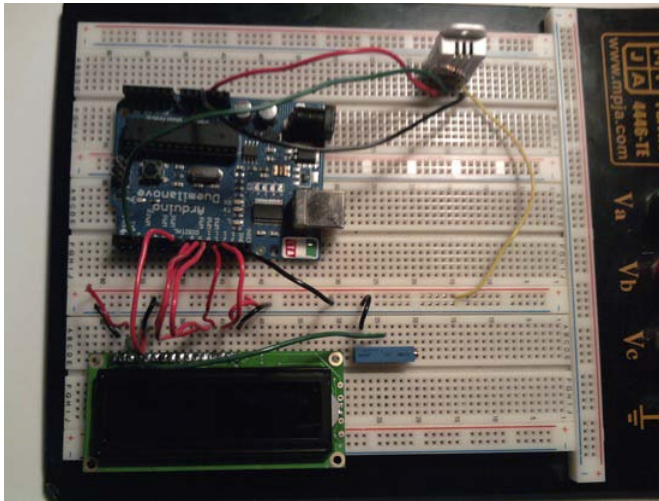


Figure 6-21. Hardware configuration for this project

Writing the Software

We will need to utilize both the LiquidCrystal and DHT22 libraries to write humidity and temperature data to the LCD. We will also need to convert the temperature data from Celsius to Fahrenheit. We will use this formula to convert that data:

T_f = temperature in Fahrenheit

T_c = temperature in Celsius

$T_f = (9/5) * T_c + 32$

See Listing 6-5.

Listing 6-5. Display Temperature and Humidity on a Monochrome LCD

```
// include the library code:
#include <LiquidCrystal.h>
#include <DHT22.h>

int tempPin = 2;
```

```

// Create instance
DHT22 DHT22Sen(tempPin);

// initialize the library with the numbers of the interface pins
LiquidCrystal lcd(7,8,9,10,11,12);

float tempF = 0.00;
float tempC = 0.00;

void setup()
{
    // set up the LCD's number of columns and rows:
    lcd.begin(16, 2);
    lcd.clear();
}

void loop()
{
    delay(2000);
    DHT22Sen.readData();

    lcd.home();
    lcd.print("Temp: ");
    tempC = DHT22Sen.getTemperatureC();
    tempF = (tempC * 9.0 / 5.0) + 32;
    lcd.print(tempF);
    lcd.print("F ");
    lcd.setCursor(0,1);
    lcd.print("Hum: ");
    lcd.print(DHT22Sen.getHumidity());
    lcd.println("%");
}

```

First, we include the header files necessary to use the `LiquidCrystal` and `DHT22` libraries. After that, we initialize `tempPin` to digital pin 2 and create an instance of the `DHT22` type. Next, we create an instance of the `LiquidCrystal` type and initialize `tempC` and `tempF` to 0. We then enter the `setup` structure and begin communicating with the monochrome LCD. Then, we enter the `loop` structure, where we first delay for 2 seconds. After that, we read from the `DHT22` sensor and then print “Temp: ” to the LCD. After that, we set `tempC` equal to the temperature reading on the `DHT22` sensor, and we convert the Celsius data to Fahrenheit data using this code:

```
tempF = (tempC * 9.0 / 5.0) + 32;
```

Notice how the `tempC` variable is inside the parentheses instead of on the outside, because the compiler does not discriminate based on the type of arithmetic performed; it just reads from left to right. After the conversion is completed, we write `tempF`’s value to the monochrome LCD. Next, we send the humidity data to the monochrome LCD.

Now that you have the knowledge to use sensors, we can check in with the company and proceed with a project using a different type of digital temperature sensor and requirements document.

Project 6-6: Wireless Temperature Monitor

Now that you understand how to use the sensors and libraries necessary, we're ready to jump into this chapter's final project. This project will extend your knowledge of setting up sensors, and we will work with the Wire Library for the first time.

Requirements Gathering and Creating the Requirements Document

The customer has set up a meeting and has several requirements for a wireless temperature monitor. The customer would like to use the TMP102 temperature sensor with the Arduino UNO (or Duemilanove). The sensor will need to operation within a 10% margin of error limit. It needs to have a range of 30 feet in the open air and display temperature in Fahrenheit on the serial monitor in this format: "Temperature: XX.XXF".

We know that the Arduino is a 5V system, which means that we cannot use this sensor unless we bring the voltage down to a 3.3V level in order to interface with the TMP102. We will have to use a logic level converter so that we can use a 3.3V system, rather than the 5V system. The company only wants a prototype of the wireless temperature system, so we can use a solderless breadboard for this project. Now that we have our notes from the meeting, we can create a requirements document.

Gathering the Hardware

Figure 6-22 shows some of the hardware being used in this project:

- Arduino UNO (or Duemilanove)
- Bluetooth Mate Silver
- Logic level converter
- TMP102 sensor
- 9V battery
- 9V battery connector
- Solderless breadboard
- Extra wire

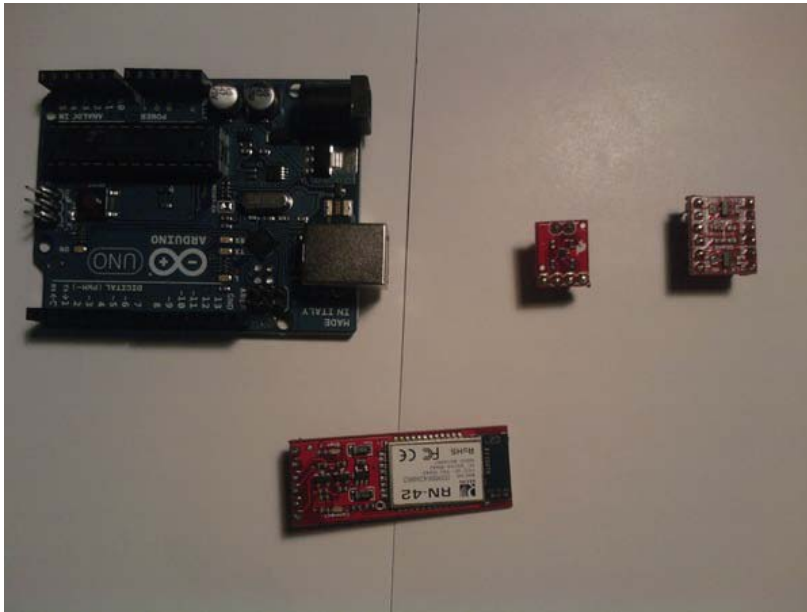


Figure 6-22. Hardware for this project (Not pictured: solderless breadboard, 9V battery, 9V battery connector, and extra wire)

Outlining the Software Requirements

These are the basic software requirements for our project:

- Use the Wire library to communicate with the TMP102 sensor (I2C)
- Convert from Celsius to Fahrenheit using the formula $(9/5) * T_c + 32$ (modify this equation to work on the Arduino).
- The sensor will need to be within a 10% margin of error, so add any values to make this sensor accurate
- Write the temperature to the serial monitor in this format: “Temperature: XX.XXF”.

Now that we have the hardware and software requirements, we can make a flowchart that the software will follow. This will help us later in the debugging stages if we have any problems. Figure 6-23 illustrates this project’s flowchart.

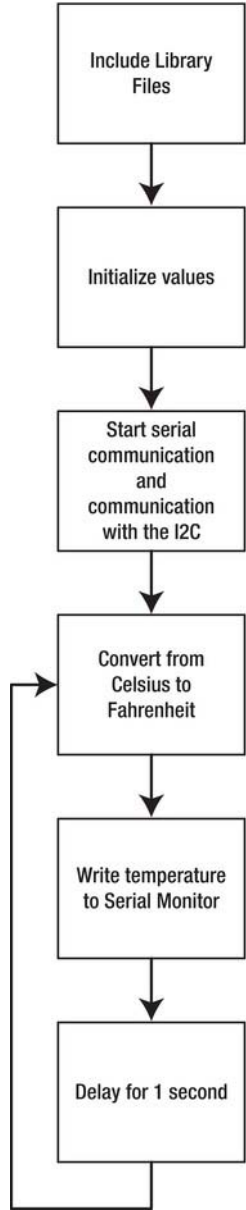


Figure 6-23. Flowchart for this project

Configuring the Hardware

As with Projects 6-4 and 6-5, our customer's project involves three main tasks. In this case, they are adding the Bluetooth Mate Silver, adding the logic level converter, and adding the sensor. This section outlines the steps involved in each.

First, we'll connect the Bluetooth Mate Silver, logic level converter, and the TMP102 sensor to the solderless bread board and configure the Bluetooth Mate Silver. Figure 6-24 illustrates this process.

1. Connect the RX pin on the Arduino to the pin 3 on the Bluetooth Mate Silver.
2. Connect the TX pin on the Arduino to pin 2 on the Bluetooth Mate Silver.
3. Connect pin 1 on the Bluetooth Mate Silver to pin 5 on the Bluetooth Mate Silver.
4. Connect pin 4 to +5V and pin 6 to ground on the Bluetooth Mate Silver.

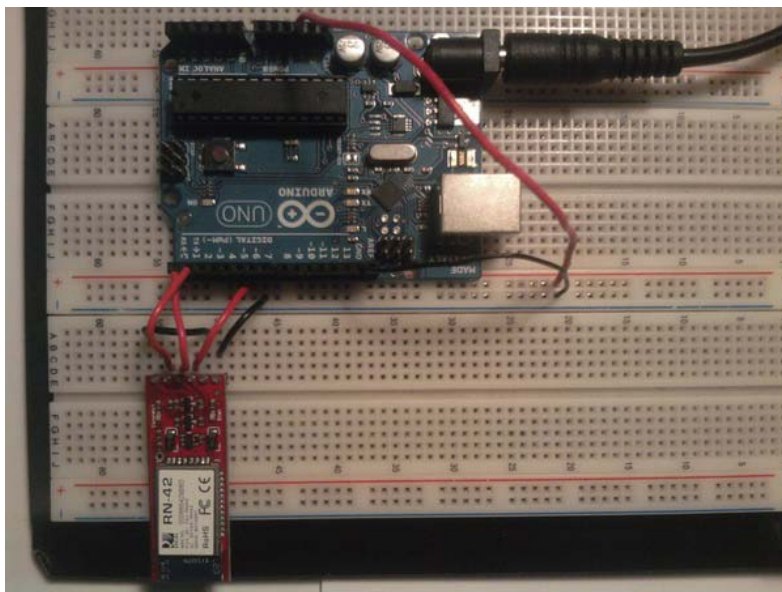


Figure 6-24. *Configuring the Blue Tooth Mate Silver*

Now that the Bluetooth Mate Silver is connected, we need to configure the logic level converter and the TMP102 sensor. Figures 6-25 and 6-26 illustrate this process.

5. Connect the high voltage pin on the logic level converter to +5V on the Arduino.
6. Connect the low voltage pin on the logic level converter to +3.3V on the Arduino.
7. Connect Channel 2 High Voltage TX0 line to analog pin 5 on the Arduino.
8. Connect Channel 1 High Voltage TX0 line to analog pin 4 on the Arduino.

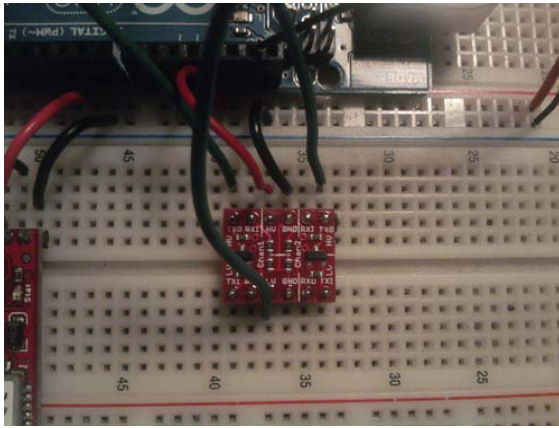


Figure 6-25. Close up of the logical converter

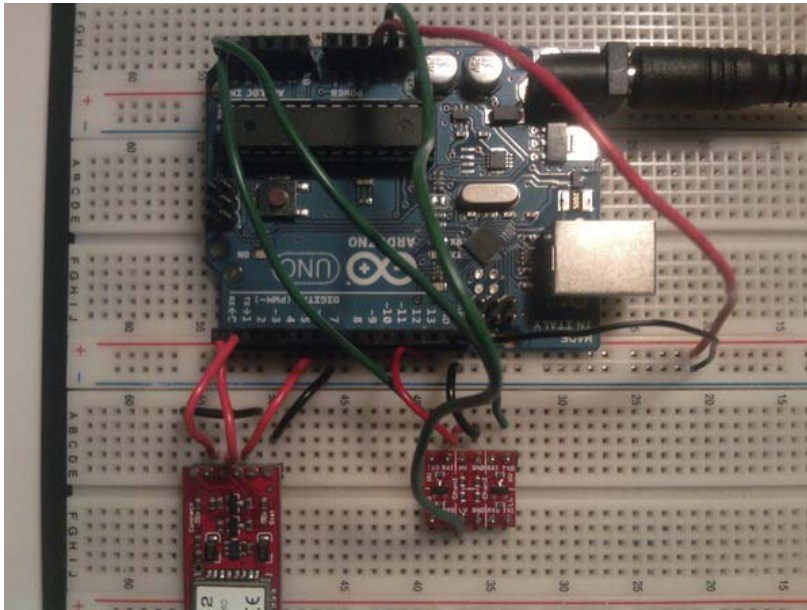


Figure 6-26. Configuration of logical converter

Now, we need to connect the logic level converter to the TMP102 sensor. Figures 6-27 and 6-28 illustrate this process.

9. Connect Channel 1 Low Voltage TXI line to the SDA pin on the TMP102 sensor.
10. Connect Channel 2 Low Voltage TXI line to the SCL pin on the TMP102 sensor.

11. Connect 3.3V to the TMP102's V+ pin and ground to the TMP102's GND pin and ADD0 pin.

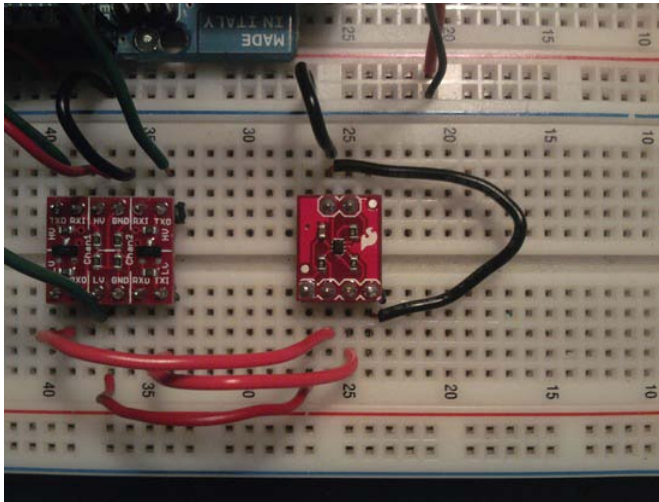


Figure 6-27. Configuration of TMP102 temperature sensor

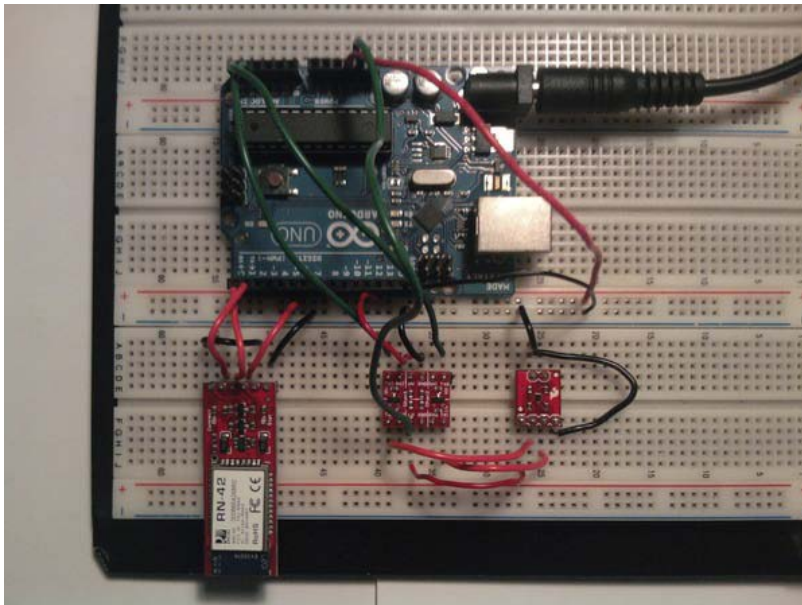


Figure 6-28. Hardware configuration for this project

Writing the Software

For this project, we will need to communicate with the TMP102 sensor using the Wire library. We will also need to convert the temperature data from Celsius to Fahrenheit. After that, we will need to write the temperature data to the serial monitor. Listing 6-6 provides the code for this project (this code is provided by the Arduino development team).

Listing 6-6. Send Temperature Data to the Serial Monitor

```
#include <Wire.h>
byte res;
byte firstByte;
byte lastByte;
int val;
float tempC = 0.00;
float tempF = 0.00;

void setup()
{
  Serial.begin(115200);
  Wire.begin();
}

void loop()
{
  res = Wire.requestFrom(72,2); // requests address and quantity to request bytes from slave
                                // device, for example the TMP102 (slave device)

  if (res == 2) {
    firstByte = Wire.receive(); // Read on byte from TMP102
    lastByte = Wire.receive(); // read least significant data
    val = ((firstByte) << 4);   // firstByte
    val |= (lastByte >> 4);     // lastByte
    tempC = val*0.0625; // Converted to celsius
    tempF = (tempC * 9 / 5) + 32; // Convert to Fahrenheit
    Serial.print("Temperature: ");
    Serial.print(tempF - 5); // this is due to the +/- 10% error that may occur
    Serial.println("F");
    delay(1000);
  }
}
```

■ **Note** To upload this code to the Arduino, you will need to disconnect the TX and RX lines that the Bluetooth Mate Silver is connected to. Then, connect your USB to the Arduino and upload the code.

We first include the header files that will allow us to communicate with the TMP102 sensor. Then, we initialize all of the variables to store the TMP102's information. After that, we enter the setup

structure and begin serial communication. Also, we start the communication between the Arduino and the TMP102. Next, we enter the setup structure and request 2 bytes from the TMP102 sensor. After that, we receive the significant bits from the TMP102 and receive the least significant values from the TMP102 sensor. We convert the val variable to Celsius and the temperature value from Celsius to Fahrenheit. Finally, we write the temperature data to the serial monitor in this format: “Temperature: XX.XXF”.

For more information on I2C, you can visit <http://en.wikipedia.org/wiki/I2c>.

Debugging the Arduino Software

This program will not have many logical errors, because we do not use any conditional statements or loops. You may have run into some syntax errors if you forgot to put a semicolon after a statement. You also may not have configured the Arduino IDE properly. Make sure that you use the Bluetooth’s COM port, and not the USB COM port, to see the data on the serial monitor. Also, make sure the baud rate is set to 115200 on the serial monitor; otherwise, you will not see any data.

Troubleshooting the Hardware

The main problem you may run into is that your wiring is incorrect. Make sure that you followed the instructions in the “Configuring the Hardware” section, and you should not have this problem. Also, you may need to reconnect your RX and TX lines from the Arduino to the Bluetooth Mate Silver, if you had to disconnect them to upload the code to your Arduino. The other concern is that you may have connected 5V to the TMP102 sensor instead of 3.3V.

Final Prototype

If everything is working, you should have a finished prototype like the one shown in Figure 6-29 to deliver to the customer.

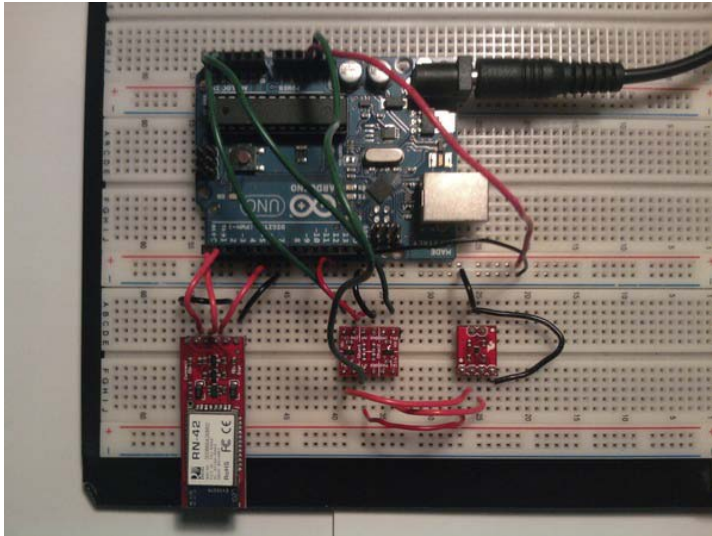


Figure 6-29. Finished prototype

Summary

In this chapter, we first went over the voltage divider circuit and how we use it to scale voltage levels that work with various sensors. After that, you learned about the various sensors that we would use throughout this chapter. Then, we went over the DHT22 and Wire libraries. The DHT22 library was used to communicate with the DHT22 sensor, and the Wire library is used to communicate with a variety of I2Cs. Next, we began to use sensors to create various projects that would help us understand how to use sensors with the Arduino. This chapter's final project gave us a break from the robot that we have been working in the previous chapters, but it still gave us a challenge using the engineering process. We created a wireless temperature system that has a margin of error less than 10%.

Robot Perception: Object Detection with the Arduino

Automation allows your robot to think for itself. In this chapter, we will be discussing a few new pieces of hardware that will make automating our robot simple and affective. We will be using a Ping))) ultrasonic sensor from Parallax and a servo to make an automated robot. First, we will learn more details about the hardware we will use in this chapter. Then we will take a look at the Servo Library (thank you, Arduino team). We will then do a few basic projects that will make you more confident using these new pieces of hardware: digital ruler, object alarm, and solar control. Finally, we will be creating a new robot that will think for itself.

After you have learned more about the ultrasonic sensor and servo control, we will be working on a final project that will incorporate all of what we reviewed. The customer will be asking us to create a robot that can detect objects in front of it. We will need to use the ultrasonic sensor to accomplish this. Also, we will need to understand servo control, as the robot will need to have 80 degrees of vision—but first, we need to take a look at the hardware used in this chapter.

Hardware Explained: Ultrasonic Sensor, Servo, and Buzzer

This chapter has a few new pieces of hardware that we will be using in order to learn more about automation and control. We will be using a Parallax Ping))) ultrasonic sensor, servo, and a buzzer to create several projects—but we first need to explain what these components do, how they do it, and why we use these components in automated projects.

Ultrasonic Sensor

The ultrasonic sensor is used to sense distance from objects. The Ping))) Parallax Ultrasonic Sensor is a very good example of this, which is why we are using it. This particular ultrasonic sensor connects to a digital pin on the Arduino. This sensor works by sending a sound wave out and calculating the time it takes for the sound wave to get back to the ultrasonic sensor. By doing this, it can tell us how far away objects are relative to the ultrasonic sensor. This helps us automate our robot because it allows the robot to “see” objects in its way. Figure 7-1 illustrates a Ping))) RangeFinder from Parallax.



Figure 7-1. Ultrasonic sensor

We will also be using the Ping))) bracket kit from Parallax to mount the Ping))) Ultrasonic Sensor onto our robot. There are other ways to mount this sensor; I chose this method because it works very well with this particular sensor. Figure 7-2 illustrates the Ping))) bracket kit from Parallax.



Figure 7-2. Ping))) bracket kit

Servo

Servos come in many shapes and sizes, from small to large. We will be using a medium size servo for this chapter; it comes with the Ping))) bracket kit from Parallax. The reason we are using a servo for this chapter and not a standard DC motor is that a servo can keep track of its orientation allowing us to control the position of the servo. A DC motor, as you know, will spin, making it vary hard to control the orientation of a DC motor. The servo also has its controller built into it, so there's no need for an H-bridge, but if you use more than one servo or have a servo that has a great load on it then you may need to use a separate power source as the Arduino cannot supply enough amps. To use a servo, all you need to do is connect a digital pin to the signal wire on the servo and connect power (+5V) and ground. A servo will allow you to sweep an area in front of our robot that will give our robot a more robust field of view. Figure 7-3 illustrates a few examples of servos.

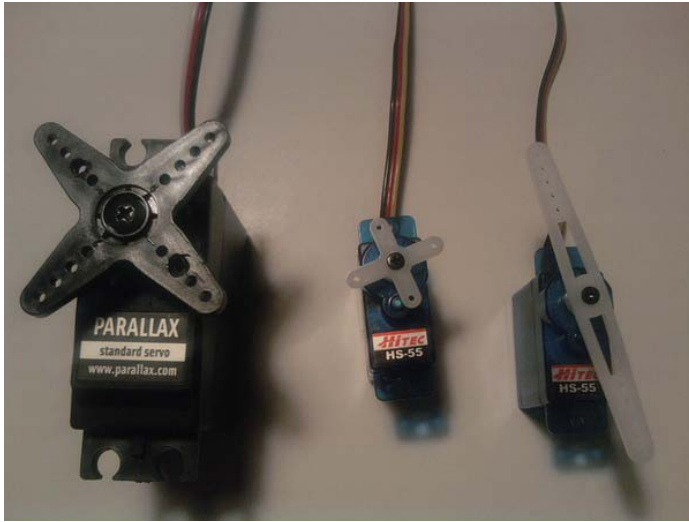


Figure 7-3. Servos

Buzzer

A buzzer can be used whenever you want to make some noise. We will use this in the second project when we make an object alarm. I am using a piezoelectric buzzer for this chapter. When we use a buzzer, we need to connect it to a PWM pin on the Arduino; this allows us to send various frequencies out of the buzzer. The buzzer works by applying a current to a metal disk that vibrates and thus produces sound waves. Figure 7-4 illustrates a buzzer.



Figure 7-4. Buzzer

Now we will go over a library that will allow us to use servos with the Arduino. The library is cleverly called the Servo Library, and is discussed in the next section of this chapter.

Libraries Explained: The Servo Library

This library is used in conjunction with a servo. There are two functions that deserve our attention: `attach()`, and `write()`. While this is not the largest library we have used, it certainly is very useful. As in the previous libraries we have used, in order to use this library, we first need to include the library and create an instance of the Servo type. Like this:

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

The first function in the Servo Library we will explain is the `attach()` function.

- `Servo attach()`: This function attaches a servo to a PWM pin and is mostly configured in the Setup Structure.

■ **Note** You must use a PWM pin. The pins you can use are 9 and 10 for the Servo Library to work. If you use any other pin you will run into issues when using this library.

- `Servo write()`: This function writes an angle to the servo. For instance, if you wanted to set the servo to its middle point; you would put `myServo.write(90);`.

That is all for this library for now, but there are a couple of functions you may want to check out on the reference page for this library at www.Arduino.cc/en/Reference/Servo (thanks again, Arduino Team). In the next section, we will be starting the projects for this chapter, so I hope you're ready!

Basics of the Ultrasonic Sensor and the Servo

In this section, we will be going over a few projects to get us ready for the final project of this chapter. The projects are: digital ruler, object alarm, and solar controller. Finally, we will revisit the company to see if there is a new project we can work on. We will be using an ultrasonic sensor and a servo throughout this section to do these various projects.

Project 7-1: Digital Ruler

In this project, we will be using an ultrasonic sensor in order to find distance from 0" to 12" and display it on the Serial Monitor. The ultrasonic sensor will need to be connected to a digital pin, and we will be using serial communication to display our values.

Hardware for this Project

Figure 7-5 shows some of the hardware being used in this project. (Not pictured: enclosure, extra wire.)

- Arduino Deumilanove (or UNO)
- Ultrasonic sensor (Ping))) from Parrallax)
- Enclosure
- Solderless breadboard
- Extra wire

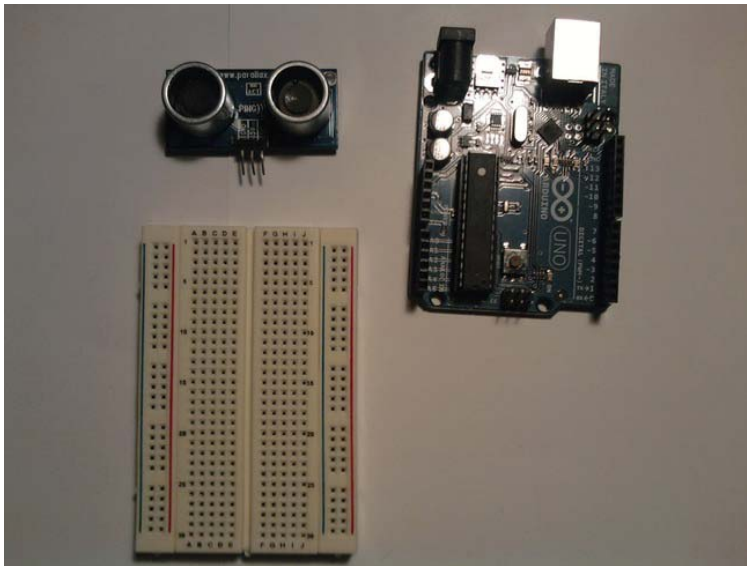


Figure 7-5. Hardware for this project

Configuring the Hardware

Follow these steps to configure the hardware for Project 7-1:

1. Connect the ultrasonic sensor to the solderless bread board.
2. Connect power (+5V) and ground from the Arduino and connect it to the ultrasonic sensor.
3. Connect the signal pin on the ultrasonic sensor to digital pin 9 on the Arduino.

Figure 7-6 illustrates the hardware configuration for this project.

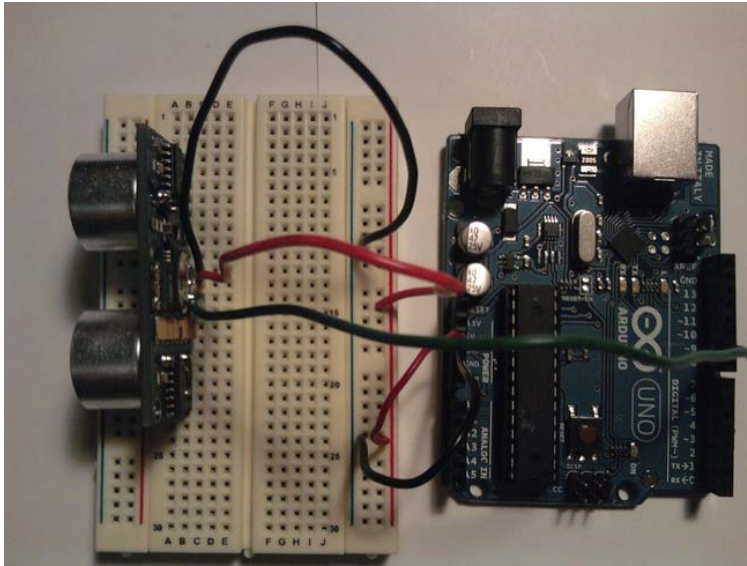


Figure 7-6. Hardware configuration for this project

After you have set up the ultrasonic sensor, you will need to connect the USB to the Arduino. We will be using the enclosure as an object in front of the ultrasonic sensor so you can control the distance read by the ultrasonic sensor.

The next section will explain the software that we will use to make the digital ruler.

Writing the Software

We will need to set up a digital pin to read in the sensor's values and send the data to the Serial Monitor in a user friendly fashion. Listing 7-1 provides the code for this project.

Listing 7-1. Digital Ruler

```

const int pingPin = 9; // Ultrasonic sensor pin

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

void loop()
{
  float duration, inches;
  // output a signal and then receive that signal back
  // and analyze duration
  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW); // send low pulse from the ultrasonic sensor
  delayMicroseconds(2);      // wait 2 microseconds
  digitalWrite(pingPin, HIGH); // send high pulse from the ultrasonic sensor
  delayMicroseconds(5);      // wait 5 microseconds
  digitalWrite(pingPin, LOW); // send low pulse from ultrasonic sensor

  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH); // waits for a high pulse then waits for a low pulse
                                     // and calculates the time it took from the high to
                                     // low pulse.

  // convert the time into a distance
  inches = duration / 74 / 2;

  Serial.print(inches);
  Serial.println("in");

  delay(1000);
}

```

First, we create a constant that will hold the value of the ultrasonic sensor pin. After that, we enter the Setup Structure where we initialize communication with serial communication. Next, we enter the Loop Structure. Here, we initialize duration and inches to be floats. Next up, we send a pulse out, and then we collect it and calculate the distance by dividing the duration by 74 and then dividing that by 2 (so we get the distance to get to the object). After that, we send the data to the Serial Monitor.

■ **Note** Remember to add a delay to this program; otherwise, you will not be able to read the information on the Serial Monitor.

To use this project put the enclosure in front of the ultrasonic sensor and open the Serial Monitor. Then move the enclosure back and forth and see that the distance has changed on the Serial Monitor.

Now that we have finished this project, we can work on a project that will help us detect when something has been moved.

Project 7-2: Object Alarm

Have you ever wandered into your office after a long day and noticed that something on your desk has been moved? Well, I have, and I always wonder whether I am imagining things. This project will help us solve this mystery by using a buzzer and an ultrasonic sensor. So that being said, here is the hardware for this project.

Hardware for This Project

Figure 7-7 shows some of the hardware being used in this project. (Not pictured: extra wire.)

- Arduino Duemilanove (or UNO)
- Ultrasonic sensor (Ping))) by Parallax
- Buzzer
- Solderless bread board
- Extra wire

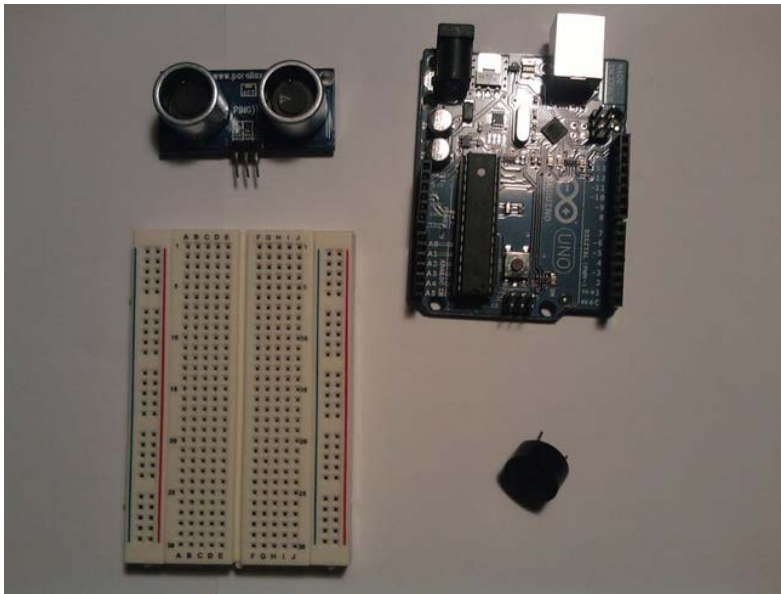


Figure 7-7. Hardware for this project

Configuring the Hardware

Follow these steps to configure the hardware for Project 7-2:

1. Connect the ultrasonic sensor and the buzzer to a solderless bread board.
2. Connect digital pin 4 to the positive side of the buzzer and ground to the negative side of the buzzer.
3. Connect power (+5V) to the 5V pin on the ultrasonic sensor and ground to the GND pin on the ultrasonic sensor.
4. Connect the signal pin on the ultrasonic sensor to digital pin 9 on the Arduino.

Figure 7-8 illustrates the hardware configuration for this project.

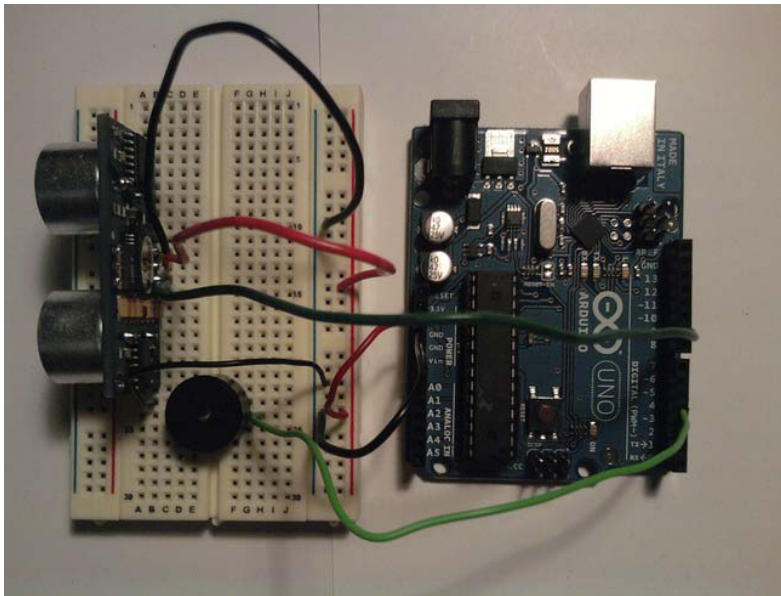


Figure 7-8. Hardware configuration for this project

Now that we have the hardware configured, we need to focus on writing the software for this project. The next section will discuss this process.

Writing the Software

This program will have to communicate with two digital pins and a serial port. We will also be using a new function called the `tone(pin, frequency, duration)` function. This function sends a tone to a digital pin; in this case, digital pin 4 will work with the `tone()` function. This function will run for a specified duration; if a duration is not specified then the tone will run until a `noTone()` function is called. Listing 7-2 provides the code for this project.

Listing 7-2. Sounds an alarm when an item has been moved

```

const int pingPin = 9; // Ultrasonic pin
const int buzzer = 4; // buzzer pin

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

void loop()
{
  float duration, inches;

  pinMode(pingPin, OUTPUT);
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(5);
  digitalWrite(pingPin, LOW);

  pinMode(pingPin, INPUT);
  duration = pulseIn(pingPin, HIGH);

  // convert the time into a distance
  inches = duration / 74 / 2;

  if (inches >= 4)
  {
    tone(buzzer, 5000, 500);
  }
  else
  {
    digitalWrite(buzzer, LOW);
  }

  Serial.print(inches);
  Serial.println("in");

  delay(1000);
}

```

First we initialize both pins; we will use on the Arduino. They are digital pins 4 and 9; one for the buzzer (4) and one for the ultrasonic sensor (9). After that, we enter the Setup Structure; here, we initialize serial communication and set the buzzer to be an output. Next, we enter the Loop Structure; in here we first initialize duration and inches. Then we send a ping out and receive it. After that, we convert duration into inches using this equation:

```
Inches = duration / 74/ 2;
```

Next, we have a conditional If Statement that has the comparison inches ≥ 4 . If this is true, the buzzer will sound, and you will know whether your favorite coffee mug has been moved. But if it is false, the buzzer will not sound, and you will know your office is safe. After that, we send the distance information to the Serial Monitor to be viewed later.

We have worked with the ultrasonic sensor and understand how it works with the Arduino. We need to work with a servo so we can incorporate both of them in the final project. The next project will satisfy this. We will be working with a servo to make a solar controller.

Project 7-3: Solar Controller

In this project, we will be using a servo to control anything with light; I suggest a small solar panel as it will move to the area that has the most light, but anything you want to move due to light, is fine. This project will be using the Parallax Ping))) bracket kit to mount whatever needs to be mounted. Also this kit comes with the servo we will use. The following section describes is the hardware for this project.

Hardware for This Project

Figure 7-9 shows some of the hardware being used in this project. (Not pictured: extra wire and solderless breadboard.)

- Arduino Duemilanove (or UNO)
- Ping))) bracket kit from Parallax (servo and mount)
- 2 photoresistors
- 1kohm resistors
- Solderless bread board
- 3-pin male header
- Extra wire

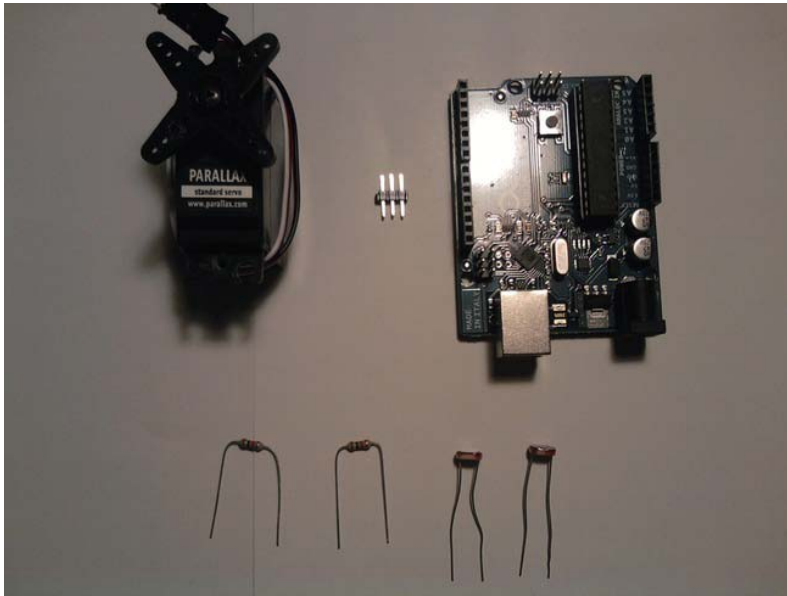


Figure 7-9. Hardware for this project

Configuring the Hardware

Follow these steps to configure the hardware for Project 7-3:

1. Connect the 3-pin male header to the servo and connect the other end of the header to the solderless bread board.
2. Connect both of the photoresistors to the solderless bread board.
3. Connect both of the 1k ohm resistors to each of the photoresistors.

Figure 7-10 illustrates the voltage divider set up.

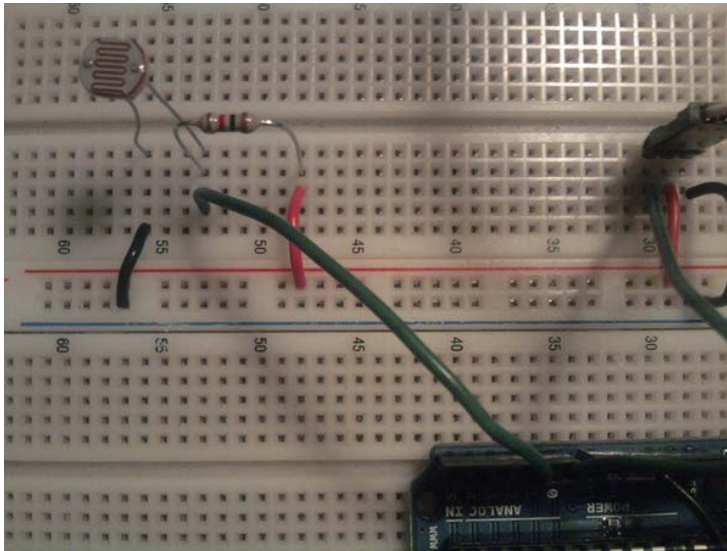


Figure 7-10. Voltage divider for one of the photoresistors

Next connect power (+5V) to the other end of the resistor and connect ground to the other pin of the photoresistor. Then, connect analog pin 0 and analog pin 1 to each of the photoresistors on the power pin. Now that the photoresistors are set up, we can work on setting up the servo. First connect power (+5V) to the power on the servo, then connect ground to the ground pin on the servo, and connect digital pin 9 to the signal of the servo. Figure 7-11 illustrates the hardware configuration for this project.

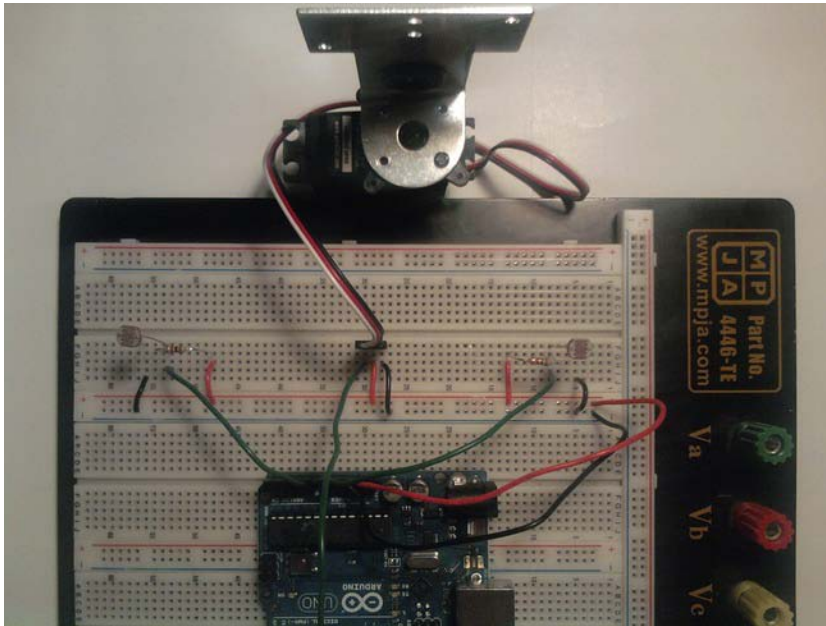


Figure 7-11. Hardware configuration for this project

Writing the Software

This program will need to talk to both the digital and analog pins in order to create the correct outcome. We will use a new library to control the servo and write the data to the Serial Monitor so that we can make sure the code is correct. Listing 7-3 provides the code for this project.

Listing 7-3. Controls the position of a servo by the amount of light that a photoresistor senses.

```
#include <Servo.h>

Servo myServo;

int photo1 = A0; // right photoresistor
int photo2 = A1; // left photoresistor
int servoPin = 9; // servo pin

int val1 = 0; // value of right photoresistor
int val2 = 0; // value of left photoresistor

void setup()
{
  Serial.begin(9600);
  myServo.attach(servoPin);
  myServo.write(90);
}
```

```

pinMode(photo1, INPUT);
pinMode(photo2, INPUT);
Serial.println("Solar Controller v1.0");
}

void loop()
{
  delay(500);
  val1 = analogRead(photo1);
  val2 = analogRead(photo2);

  Serial.print("Photo1 Value: ");
  Serial.println(val1);
  Serial.print("Photo2 Value: ");
  Serial.println(val2);

  if (val1 >= 800 && val2 >= 800)
  {
    myServo.write(90);
    Serial.println("To Dark");
  }
  else if(val1 < val2)
  {
    myServo.write(115);
    Serial.println("Right Photoresistor has more light");
  }
  else if(val2 < val1)
  {
    myServo.write(40);
    Serial.println("Left Photoresistor has more light");
  }
}

```

Listing 7-3 first includes the header file so that we can use `sevos` in our program. After that, we create and instance of the servo type. Next, we set up the pin for the photoresistors and create values that will hold the data from the photoresistors. Then we enter the Setup Structure where we begin serial communication, set the servo's pin to digital pin 9, set the servo to 90 degrees, set the photoresistors to inputs, and finally write "Solar Controller v1.0" to the Serial Monitor. Then we enter the Loop Structure here we set the analog inputs equal to `val1` and `val2`. After that, we write the values of the photoresistors to the Serial Monitor. Next we enter a series of If Statements. The first If Statement checks whether the photoresistors are greater-than-or-equal-to 800; if they are, the servo goes to 90 degrees. Then we enter an Else-If Statement that has the condition `val1 < val2`. If this is true, the sensor will move to the "right" photoresistor. If the first Else-If Statement is false, then we will move to the final condition that states `val2 < val1`. If this is true, the servo will move to the "left" photoresistor.

■ **Note** The higher the resistance on a photoresistor, the darker it is.

Now that we have gained considerable knowledge on both the ultrasonic sensor and servos, we can check with the company to see if they have any interesting projects that we can do to better our understanding of the engineering process and how it can be used to create autonomous robots. It just so happens that the customer does have a project that we can complete. They have set up a meeting for us to get familiar with the requirements so that we can create a requirements document.

Requirements Gathering and Creating the Requirements Document

The company has requirements to create another robot that will be autonomous using an Arduino. It will need to be able to avoid obstacles in its way. The company wants a completed prototype (meaning it does not have to be a completely robust system) that will have a chassis and a servo to move a ultrasonic sensor from 40 degrees to 115 degrees. Any error handling should be displayed on the Serial Monitor. Because this will be the first wireless system, the company also would like two 9V battery connectors to be connected to the chassis so that the batteries do not fall out of the chassis. The company also wants an efficient mounting solution for the ultrasonic sensor. Now that we have our notes, we can create a requirements document that will guide us through the rest of the engineering process.

Hardware

Figure 7-12 shows some of the hardware being used in this project. (Not pictured: extra wire, chassis from chapter 4, 2X 9V connectors, 2X 9V batteries, 2X 3 pin male headers, and Velcro.)

- Arduino Duemilanove (or UNO)
- Ultrasonic sensor (Ping))) by parallax
- Chassis from Chapter 4's robot (with motors)
- Ping))) bracket kit from Parrallax
- 2X 9V connectors
- 2X 3 pin male header
- Velcro
- Extra wire



Figure 7-12. Hardware for this project

Software

These are the software requirements for this project:

- Write software to avoid obstacles using an ultrasonic sensor
- Use the Serial Monitor to track any errors that may occur
- Use the Servo Library to control a servo to go from 40 degrees to 115 degrees

Now that we have the hardware and software requirements, we can create a flowchart shown in Figure 7-13 that the software will follow.

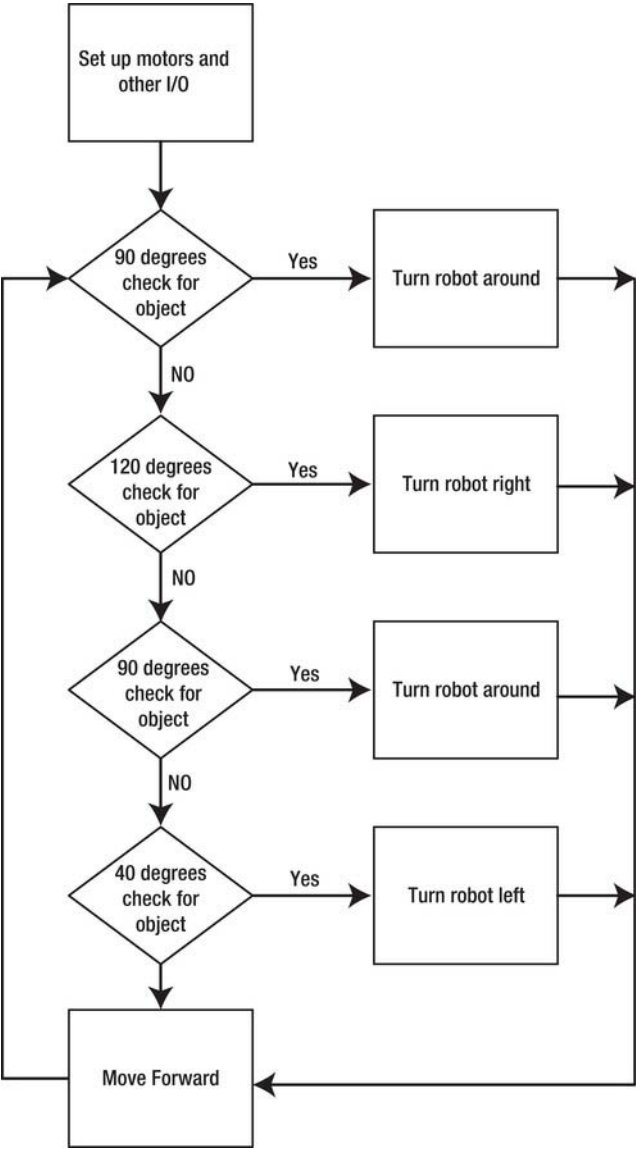


Figure 7-13. Flowchart for this project

Configuring the Hardware

First, we need to add the 9V connectors to an area on the chassis that does not impede any of the functions.

1. Add the Arduino's 9V connector to the front of the chassis. You can attach the 9V connector by using Velcro or screwing it into the chassis. If you are using Velcro, add one side to the back of the 9V connector.
2. Add a piece of Velcro to the front of the chassis.
3. For the motor drivers 9V connector we need to add the 9V connector to the back of the chassis. Put Velcro on the back of the 9V connector.
4. Then add a piece of Velcro onto the back of the chassis and attach the Arduinos' 9V connector to the front piece of Velcro and attach the motor drivers 9V connector to the piece of Velcro you put on the back of the chassis.

Figure 7-14 illustrates the 9V connector's set up.

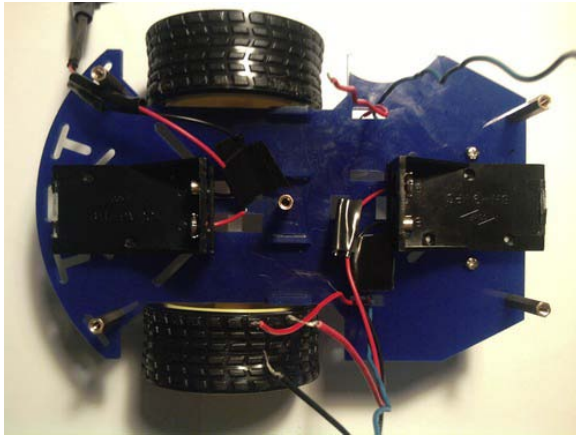


Figure 7-14. 9V connectors attached using Velcro

Now that the 9V connectors are added, we need to add the Ping))) bracket kit from Parrallax to the chassis.

First you will need to construct the bracket kit. You can find out how to do this by reading the instructions that came with the bracket kit. After you have constructed the bracket kit, you can mount it onto the chassis. Figure 7-15 and 7-16 illustrate the Ping))) bracket kit mounted to the chassis.

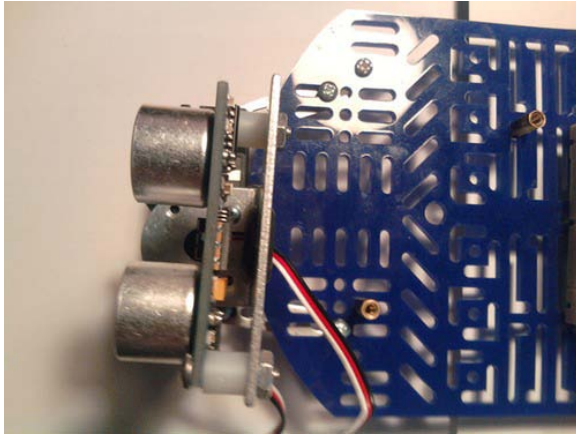


Figure 7-15. Bracket kit attached (top view)

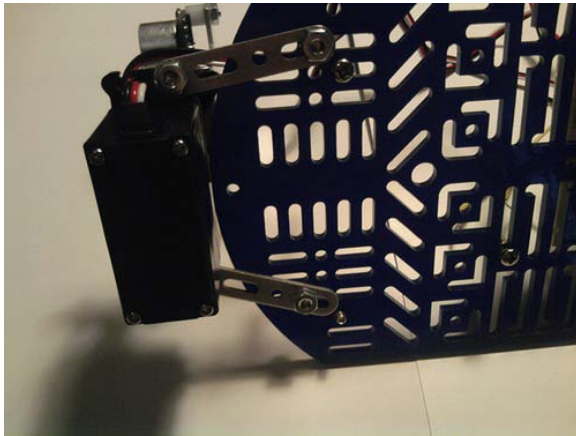


Figure 7-16. Bracket kit attached (bottom view)

Now we need to connect the servo and the ultrasonic sensor to the Arduino. Follow these steps:

4. Connect the power (+5V) and ground to one side of the solderless breadboard (you should have left space at the front of the motor driver that you built in Chapter 4).
5. After that you need to add the power (+5V) and ground to the other side of the breadboard. Next add the signal connections to the Arduino; the servo will be connected to digital pin 9, and the ultrasonic sensor will be connected to pin 12.
6. Next connect servo to the leftmost part of the breadboard, and connect the ultrasonic sensor to the rightmost part of the breadboard.

Figure 7-17 and 7-18 illustrate the set up of the ultrasonic sensor and the servo.

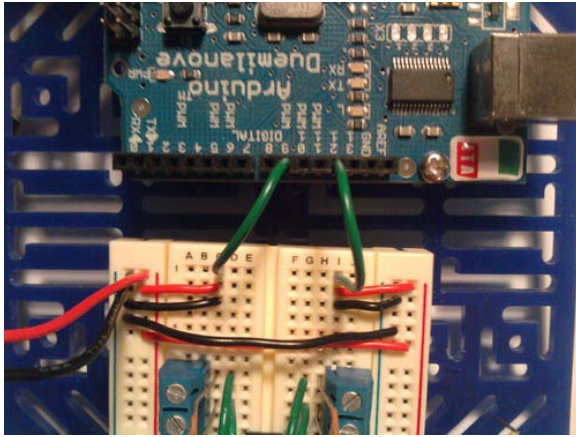


Figure 7-17. Power, ground, and signal connected to the solderless bread board

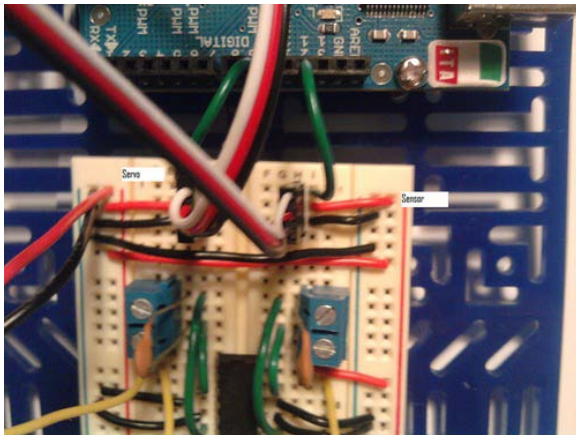


Figure 7-18. Servo and ultrasonic sensor connected to solderless breadboard (Right Ultrasonic sensor, Left Servo)

The hardware configuration should be completed. Figure 7-19 illustrates the hardware configuration for this project.

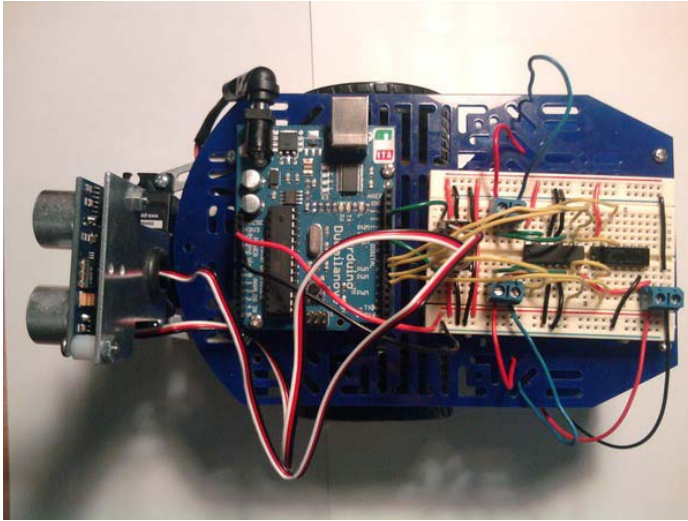


Figure 7-19. Hardware configuration for this project

In the next section we will be writing the software for this robot. We will not be adding on the previous robot as this is a different project.

Writing the Software

We will be communicating with the digital pins for this project. We may also want to do error handling for this project, but that will be in the next section. We will also need to use the Servo Library in order to use the servo that will sweep the area. Here is the code for this project.

Listing 7-4. Automated Robot Software

```
#include <Servo.h> // includes Servo Library

Servo myservo; // creates an instance of Servo

const int pingPin = 12; // ultrasonic pin
int motorPins[] = {4,5,7,6}; // Motor Pins
int servoPin = 9; // servo pin
int pos = 0; // servo position
const int range = 12;

long inoutPing(); // function prototype inoutPing()
void moveRight(); // function prototype moveRight()
void moveLeft(); // function prototype moveLeft()
void turnAround(); // function prototype turnAround()
void moveForward(); // function prototype moveForward()
```

```

void setup()
{
  Serial.begin(9600);
  myservo.attach(servoPin);
  for (int i; i <= 3; i++)
  {
    pinMode(motorPins[i], OUTPUT); // makes motorPins[i] an output
    digitalWrite(motorPins[i], LOW); // sets motorPins[i] to high
  }
}

void loop()
{
  long duration, inches;

  moveForward();

  for(pos = 40; pos < 120; pos += 1) // moves servo from 40 to 119
  {
    myservo.write(pos);
    if(pos == 90)
    {
      inches = inoutPing();

      if (inches <= range) // sees how far away an object is
      {
        turnAround();
        moveForward();
      }
      else
      {
        moveForward();
      }
      delay(50);
    }
    delay(15);
  }

  inches = inoutPing();

  if (inches <= range)
  {
    moveRight();
    moveForward();
  }
  else
  {
    moveForward();
  }
}

```

```

    delay(50);

    for(pos = 120; pos >= 40; pos-=1) // moves servo from 120 to 40
    {
        myservo.write(pos);
        if(pos == 90)
        {
            inches = inoutPing();

            if (inches <= range)
            {
                turnAround();
                moveForward();
            }
            else
            {
                moveForward();
            }
            delay(50);
        }
        delay(15);
    }

    inches = inoutPing();

    if (inches <= range)
    {
        moveLeft();
        moveForward();
    }
    else
    {
        moveForward();
    }

    delay(50);
}

long inoutPing() // calculates distance
{
    long time, distance;
    pinMode(pingPin, OUTPUT);
    digitalWrite(pingPin, LOW);
    delayMicroseconds(2);
    digitalWrite(pingPin, HIGH);
    delayMicroseconds(5);
    digitalWrite(pingPin, LOW);

    pinMode(pingPin, INPUT);
    time = pulseIn(pingPin, HIGH);

```

```

    distance = time / 74 / 2;

    return distance;
}

void moveRight() // turns robot right
{
    digitalWrite(motorPins[2], LOW);
    analogWrite(motorPins[3], 110);
    delay(350);
}

void moveLeft() // turns robot left
{
    digitalWrite(motorPins[0], LOW);
    analogWrite(motorPins[1], 110);
    delay(350);
}

void turnAround() // turns robot around
{
    digitalWrite(motorPins[0], LOW);
    analogWrite(motorPins[1], 110);
    delay(750);
}

void moveForward() // moves robot forward
{
    digitalWrite(motorPins[0], HIGH);
    analogWrite(motorPins[1], 150);
    digitalWrite(motorPins[2], HIGH);
    analogWrite(motorPins[3], 130);
}

```

We first include the header file for the Servo Library so that we can use a servo to sweep the area in front of the robot. Next, we create an instance of the Servo class and initialize the motor pins and the position of the servo. After that, we have a few function prototypes; the first returns a value for duration. The next four function prototypes are used to move the robot forward, turn left, turn right, and turn around. Then we enter the Setup Structure; in here we attach the servo to digital pin 9 and set up the pins for the robot's motors. After that, we enter the Loop Structure; in here we initialize duration and inches. Next, we tell the robot to move forward. After that, we use the ultrasonic sensor to check if anything is in front of the robot. If there is, it will turn around; otherwise, the robot will continue forward. Next the ultrasonic sensor will check if anything is to the left of the robot; if there is, the robot will turn to the right and move forward. Then the robot will check if anything is in front of the robot again and turn around if there is an object to the in front of it. After that, the robot checks if there is anything to the right of it and turns left if there is an object to the right of it. The next function is used to send out a signal from the ultrasonic sensor and receive the information back so that it can be converted to distance. Then the next four functions are used to move the robot forward, turn left, turn right, and turn around.

■ **Note** You may have to adjust the `moveForward()`, `moveRight()`, `moveLeft()`, and `turnAround()` delays in order for the robot to move correctly. For example, the `moveLeft` function has a delay of 350 milliseconds. If your robot is only doing a quarter left turn, then you may need to have a 750-millisecond delay. You may also need to adjust how close you want an object to be in order for the robot to move. Here is an example of the code you would change:

```
const int range = 10; // instead of this being 12 it has been changed to 10
```

Now that the software is written, we can debug it using serial communication. The next section will discuss debugging the software.

Debugging the Arduino Software

Now if we are having issues with the software, we can debug it by writing out the process to the Serial Monitor. Here is the code with error handling.

Listing 7-5. Automated Robot with Error Handling

```
#include <Servo.h>

Servo myservo;

const int pingPin = 12;
int motorPins[] = {4,5,7,6}; // Motor Pins
int servoPin = 9;
int pos = 0;
const int range = 12;

long inoutPing();
void moveRight();
void moveLeft();
void turnAround();
void moveForward();

void setup()
{
  Serial.begin(9600);
  myservo.attach(servoPin);
  for (int i; i <= 3; i++)
  {
    pinMode(motorPins[i], OUTPUT);
    digitalWrite(motorPins[i], LOW);
  }
}
```

```

void loop()
{
  long inches;
  moveForward();

  for(pos = 40; pos < 120; pos += 1)
  {
    myservo.write(pos);
    if(pos == 90)
    {
      inches = inoutPing();

      if (inches <= range)
      {
        turnAround();
        moveForward();
      }
      else
      {
        moveForward();
      }

      Serial.println("1st Measurement Check: ");
      Serial.print(inches);
      Serial.println("in");
      delay(50);
    }
    delay(15);
  }

  inches = inoutPing();

  if (inches <= range)
  {
    moveRight();
    moveForward();
  }
  else
  {
    moveForward();
  }

  Serial.println("2nd Measurement Check: ");
  Serial.print(inches);
  Serial.println("in");
  delay(50);

  for(pos = 120; pos >= 40; pos-=1)
  {

```

```

myservo.write(pos);
if(pos == 90)
{
    inches = inoutPing();

    if (inches <= range)
    {
        turnAround();
        moveForward();
    }
    else
    {
        moveForward();
    }
    Serial.println("3rd Measurement Check: ");
    Serial.print(inches);
    Serial.println("in");
    delay(50);
}
delay(15);
}

inches = inoutPing();

if (inches <= range)
{
    moveLeft();
    moveForward();
}
else
{
    moveForward();
}

Serial.println("4th Measurement Check: ");
Serial.print(inches);
Serial.println("in");
delay(50);
}

long inoutPing()
{
    long time, distance;
    pinMode(pingPin, OUTPUT);
    digitalWrite(pingPin, LOW);
    delayMicroseconds(2);
    digitalWrite(pingPin, HIGH);
    delayMicroseconds(5);
    digitalWrite(pingPin, LOW);

    pinMode(pingPin, INPUT);

```

```

    time = pulseIn(pingPin, HIGH);

    distance = time / 74 / 2;

    return distance;
}

void moveRight()
{
    Serial.println("Turning Right"); // writes to the serial port "Turning Right"
    digitalWrite(motorPins[2], LOW);
    analogWrite(motorPins[3], 110);
    delay(350);
}

void moveLeft()
{
    Serial.println("Turning Left"); // writes to the serial port "Turning Left"
    digitalWrite(motorPins[0], LOW);
    analogWrite(motorPins[1], 110);
    delay(350);
}

void turnAround()
{
    Serial.println("turning around"); // writes to the serial port "Turning Around"
    digitalWrite(motorPins[0], LOW);
    analogWrite(motorPins[1], 110);
    delay(750);
}

void moveForward()
{
    Serial.println("moving forward"); // writes to the serial port "moving forward"
    digitalWrite(motorPins[0], HIGH);
    analogWrite(motorPins[1], 150);
    digitalWrite(motorPins[2], HIGH);
    analogWrite(motorPins[3], 130);
}

```

As you can see, we have added a few `Serial.println()` functions that will print to the Serial Monitor every time a process has occurred—such as when the servo turns and the ultrasonic sensor checks whether anything is in front of the robot. This can be very useful if your robot is turning in the wrong direction or you think the ultrasonic sensor is not receiving the correct distances. After everything is working correctly with your robot you should use the code that we created in the “Writing the Software” section, because we no longer need to test our robot. This will save battery life and make the robot more responsive (because it will have extra power to run). Now that we have resolved some of the issues with the software, we can work on figuring out any problems that may have occurred during the hardware configuration.

Troubleshooting the Hardware

You will want to make sure that you have the servo and ultrasonic sensor connected correctly to the Arduino. Also make sure that 5V power and ground are connected to those pieces of hardware. If the 9V connector is getting in the way of the chassis, move it around just make sure everything fits properly. If you are using the same chassis as in this book, you can also stand the robot on its back and test whether the robot is moving in the correct directions; just make sure your Arduino is connected to the computer.

■ **Note** Use brand new 9V batteries or fully charged rechargeable batteries; if you use used 9V batteries, you may experience some problems with the ultrasonic sensor.

Finished Prototype

Well, if you made it to this section, you have a finished prototype to deliver to the company. Try and make this autonomous robot even better if you wish. You can do this by experimenting with different delays in the subroutines or by changing the “range” variable to see if there is a more accurate range that the ultrasonic sensor can work with. Figure 7-20 illustrates the final prototype.

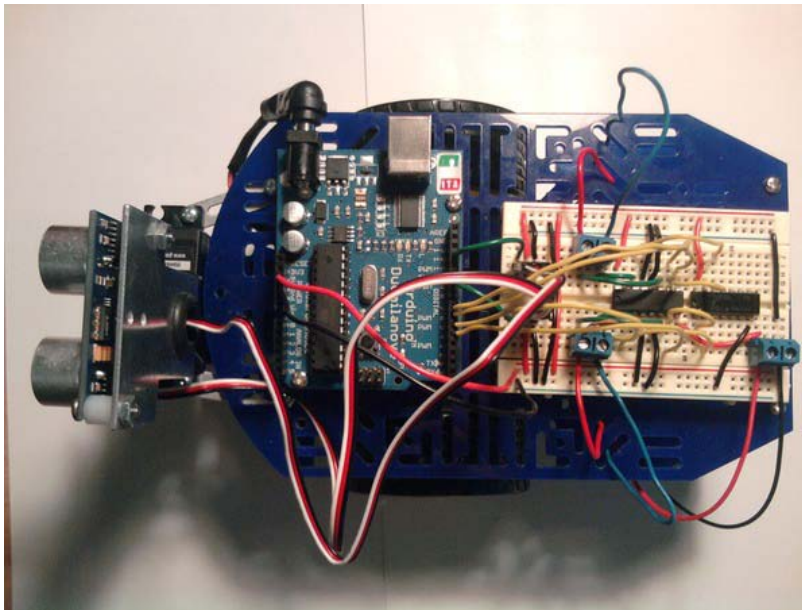


Figure 7-20. Finished prototype

Summary

We first took a look at the new hardware for this chapter. Then we learned how the Servo Library works. After that, we completed a few projects to prepare us for the final project of this chapter. The projects were: digital ruler, object alarm, and solar controller. Next we created an autonomous robot that avoids objects. It is also the first stand-alone robot we have created in this book. We also gained further knowledge of debugging our robots by using serial communication. The final project of this chapter used the engineering process to create an autonomous robot.

Mature Arduino Engineering: Making an Alarm System Using the Arduino

In this chapter we will be creating various pieces of a motion detecting alarm system using the Arduino. To satisfy our example company's requirements, the alarm system will track any motion detected throughout the day and store it on a microSD card. The company also wants to have real-time tracking, so the information will need to be displayed on the Serial Monitor.

To do so, we will be learning about a new piece of hardware called the passive infrared (PIR). We will also be using an ultrasonic sensor in order to create a different type of security system. Our preliminary project uses an ultrasonic sensor as a door alarm. This will prep us for the final project, which is to be delivered to the customer. We will use the PIR in order to detect motion that has occurred in a room.

Before we get into the project, though, the next section will discuss the new hardware for this chapter.

■ **Note** I would not recommend using these projects as your sole security system. This chapter will introduce you to some prototypes for a security system.

Hardware Explained: PIR

The PIR will detect changes in infrared radiation; thus, we can use it to detect motion, because the human body does not have a constant temperature. The PIR cannot sense movement unless there is this change in temperature. To use a PIR sensor, connect the signal of the PIR to a digital pin on the Arduino, just as you did with the ultrasonic sensor. Figure 8-1 shows a PIR sensor.



Figure 8-1. PIR sensor

This particular PIR sensor comes from Parallax and has a three-pin configuration (i.e., power, ground, and signal).

That's it for the new hardware for this chapter—we can move on to a project that will help you get ready for the final project later in this chapter.

Basic Security System

Before we get to the final project, we need to go over a project that will give some basics of a security system. In this project, we will be using an ultrasonic sensor to detect whether a door has been opened. This project will require a large enclosure to mount the ultrasonic sensor. We will also be using the Bluetooth Mate Silver from Sparkfun to send alarm information to the Serial Monitor.

Project 8-1: Door Alarm

In this project, we will be creating a door alarm system. While you can use many different approaches to do this, we are going to use an ultrasonic sensor to make this project sense that a door has been opened. We will also use the microSD shield to datalog how many times the door has been opened. The next section will discuss the hardware for this project.

■ **Note** We will be expanding on this project in the next chapter.

Hardware for This Project

Figure 8-2 shows some of the following hardware being used in this project. (Not pictured: 9V adapter, AC Adapter 9V 650mAh, 9V connector, Velcro, breadboard, and scrap piece of plastic.)

- Arduino Duemilanove (or UNO)
- microSD shield
- 512mb–1GB microSD card

- Ultrasonic sensor (Ping))) from Parallax
- Ping))) bracket kit from Parrallax
- Bluetooth Mate Silver or BluSmirf
- 9V or AC adapter (9V 650mAh)
- 9V connector (if needed)
- Scrap piece of plastic orenclousure to make the barrier
- Velcro
- Solderless breadboard
- Enclosure to house circuit
- Extra wire



Figure 8-2. Hardware for this project

■ **Note** I recommend using an AC adapter for this project, as it will be on for extended periods of time.

Configuring the Hardware

There are several steps required to configure the hardware for Project 8-1. We will use a dremmel to drill holes into the plastic enclosure. First, we need to configure the enclosure for this project. To do so, follow these steps:

1. Using a drill or dramel, drill two holes for screws; these screws will connect the ultrasonic sensor bracket to the top of the enclosure. To figure out where the holes need to go, use a marker and hold the ultrasonic mount to the lid of the enclosure and mark where the screw holes need to be. Figure 8-3 illustrates this process.



Figure 8-3. Drill holes for the ultrasonic bracket.

2. Next, drill a hole in the top so that the ultrasonic sensor's wire can reach the solderless breadboard inside the enclosure. Figure 8-4 illustrates this process.



Figure 8-4. Drill a hole so the ultrasonic wires can be connected to solderless breadboard.

3. Now, connect the ultrasonic bracket to the top of the enclosure using the two holes you drilled in the first step. Figure 8-5 illustrates this process.



Figure 8-5. Connect ultrasonic bracket to lid of the enclosure.

4. Attach the ultrasonic sensor to the ultrasonic bracket. Figure 8-6 illustrates this process.



Figure 8-6. Attach ultrasonic sensor to bracket.

5. Finally, use a dremel to drill a hole for the power supply. I chose the right back corner for the Arduino, so the right back corner will be where we drill this hole. Figure 8-7 illustrates this process.



Figure 8-7. Drill a hole so that power can be connected to the Arduino.

Now that the enclosure is set up, we need to configure the circuit.

6. Connect the microSD Shield to the Arduino.
7. Connect the Bluetooth Mate Silver to the solderless breadboard.
8. Connect the CTS pin and the RTS pin on the Bluetooth Mate Silver. After that, connect power and ground to the Bluetooth Mate Silver.
9. Connect wires to the TX and RX pins on the Bluetooth Mate Silver, as shown in Figure 8-8. (Note that the Bluetooth mate silvers pins are referenced by name, as they have no pin numbers.)

■ **Note** Do not connect the TX and RX lines of the Bluetooth Mate Silver until after you have uploaded the code to your Arduino. If you attach the TX and RX pins and attempt to send the sketch (code) to the Arduino you will run into an error and the code will not be uploaded to the Arduino.

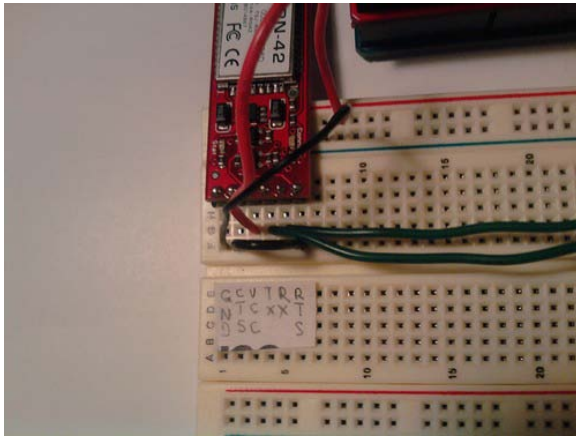


Figure 8-8. Configuration of the Bluetooth Mate Silver

10. Connect the ultrasonic sensor to the solderless breadboard.
11. Connect power (+5V) and ground to the power and ground pins on the ultrasonic sensor.
12. Connect digital pin 7 on the Arduino to the signal pin on the ultrasonic sensor. Figure 8-9 illustrates this process.

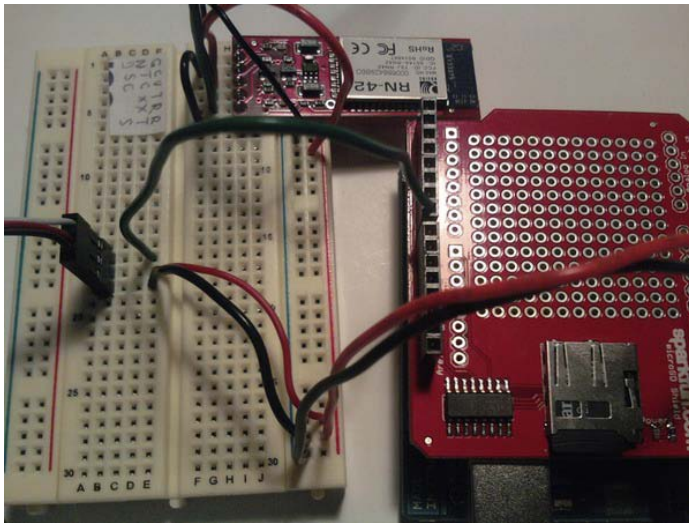


Figure 8-9. Configuration of the ultrasonic sensor

■ **Note** All that is left to do with the hardware configuration is to connect the TX pin on the Bluetooth Mate Silver to the RX pin on the Arduino and the RX pin on the Bluetooth Mate Silver to the TX pin on the Arduino, but we will do that after we have uploaded this project's code to the Arduino. In the next section, we will be going over the software for this project.

Writing the Software

To write this software, we are going to use the SdFat Library in order to write data to a microSD card. Also, we will use serial communication to test that the device is working properly, and to monitor the security system. The ultrasonic sensor will need to check how far away the door is relative to the ultrasonic sensor. Listing 8-1 provides the program for this project.

Listing 8-1. Door Alarm

```
//Add the SdFat Libraries
#include <SdFat.h>
#include <SdFatUtil.h>

//Create the variables to be used by SdFat Library
Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

char name[] = "Data.txt";    // holds the name of the new file
const int pingPin = 7;

void setup()
{
    Serial.begin(115200);    // Start a serial connection.
    pinMode(10, OUTPUT);    // Pin 10 must be set as an output for the SD communication to
                           // work.
    card.init();            // Initialize the SD card and configure the I/O pins.
    volume.init(card);      // Initialize a volume on the SD card.
    root.openRoot(volume);  // Open the root directory in the volume.
}

void loop()
{
    float duration, inches;
    // send low and high pulse out of the ultrasonic sensor to calculate distance from
    // an object.
    pinMode(pingPin, OUTPUT);
    digitalWrite(pingPin, LOW);
    delayMicroseconds(2);
    digitalWrite(pingPin, HIGH);
    delayMicroseconds(5);
```

```

digitalWrite(pingPin, LOW);

pinMode(pingPin, INPUT);
duration = pulseIn(pingPin, HIGH);

// convert the time into a distance
inches = duration / 74 / 2;
delay(500);

if (inches >= 5)
{
  Serial.println("Door Open!!"); // send "Door Opened" if distance is reached to the
                                // Serial Monitor.

  delay(500);
  file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file 'name'
                                                           // in 'root' for writing to the
                                                           // end of the file.

  file.println("Door Open!!");
  file.close();
}
else
{
  Serial.println("Nothing to Report");
}

delay(1000);
}

```

We first include the header files for the SdFat Library so that we can use the microSD Shield to send data. After that, we create instances of the Sd2Card, SdVolume, and SdFile. Then we create the name of the file that will be written to a microSD card. Next we create a variable that will hold the pin number for the ultrasonic sensor. After that, we enter the Setup Structure; here, we initialize serial communication. We set pin 10 to an output and initialize the card and volume in order to use the microSD Shield. Next, we enter the Loop Structure; here, we initialize duration and inches to be floating point numbers. Then we send a pulse out of the ultrasonic sensor and receive the pulse back, and next we calculate duration and convert it to inches. After that, we have a conditional If-Else statement that will print to the Serial Monitor “Door Open!!” If the ultrasonic sensor detects a distance greater than five inches, it will also write “Door Open!!” to the microSD card. If the door has not been opened, the Serial Monitor will display “Nothing to report.”

Once you upload this program to the Arduino, connect the Bluetooth Mate Silver’s TX and RX pins to the Arduino, as described in the Note at the end of the “Configuring the Hardware” section.

Figure 8-10 illustrates the Bluetooth Mate silver connected to the Arduino.

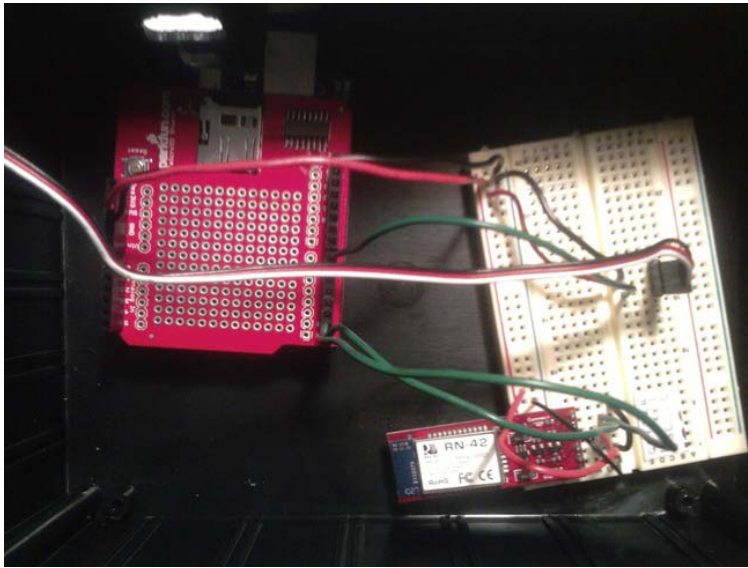


Figure 8-10. Bluetooth Mate Silver's TX and RX pins connected to the Arduino's TX and RX pins

To use this device, you need to put it to the side of a door. When the door is opened, it will read that the door has been opened. This project can be tested by simply putting your hand less than five inches away from the ultrasonic sensor and moving it at a distance greater than five inches and you should see "Door Opened!!" on the Serial Monitor. To make the distance great all you need to do is change the If Statement `if (inches >= 5)` to a different number, for example: `if (inches >= 9)`. Figure 8-11 illustrates the door alarm in action.

■ **Note** Make sure the microSD card is not write protected.



Figure 8-11. Project in action

Now that we have completed this project, we can contact the company again and see if it has a need for any security systems. It just so happens that it does, and we have set up a meeting to go in and hear the requirements. Like with every final project, we need to start with requirements gathering and creating the requirements document.

Requirements Gathering and Creating the Requirements Document

The company has several requirements for a motion detecting alarm system controlled by the Arduino. The motion detector will need to keep track of motion detected throughout the day and stored on a microSD card. The company only wants to write to the microSD card when motion has been detected. The company also wants to have real-time tracking, so the information will need to be displayed on the Serial Monitor. If motion is detected, this will be printed to the Serial Monitor and the microSD card: “Motion Detected!!” If no motion has been detected, this will be printed to the Serial Monitor: “Nothing to Report.” Now that we have gathered our notes, we can create a requirements document that will guide us through the rest of this project.

Hardware

Figure 8-12 shows some of the following hardware being used in this project. (Not pictured: AC adapter, 9V battery, 9V battery connector.)

- Arduino Duemilanove (or UNO)
- microSD shield
- 512mb –1GB microSD card
- PIR sensor
- Bluetooth Mate Silver
- Three terminal extender (included with the Ping))) bracket kit from Parallax)

- Medium size solderless breadboard
- AC adapter (9V 650mAh) or 9V battery
- 9V battery connector if you are using a 9V battery
- Extra wire

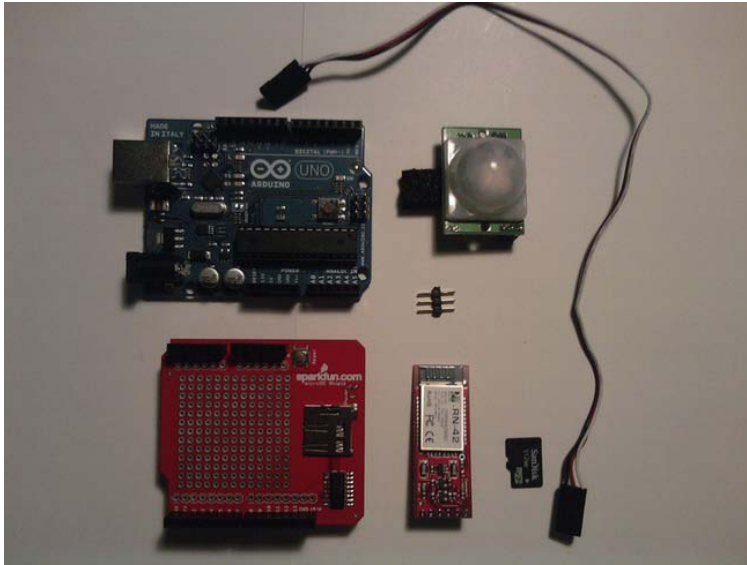


Figure 8-12. Hardware for this project

Software

These are the software requirements for this project:

- Use digital IO to check whether motion has been detected.
- Send data to microSD card when motion has been detected.
- Write to the Serial Monitor when motion is detected, and if motion is not detected, write “Nothing to Report” to the Serial Monitor.

Now that we have the requirements for this project laid out in an understandable fashion, we can create a flowchart that will help us create the software for this project. Figure 8-13 illustrates the flowchart for this project.

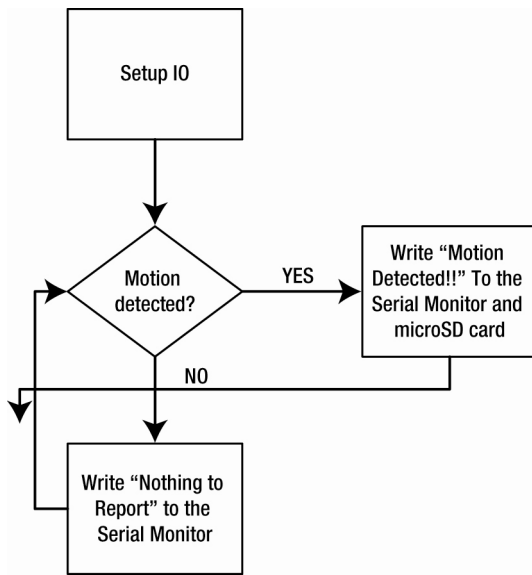


Figure 8-13. Flowchart for this project

The next section will discuss the hardware configuration for this project.

Configuring the Hardware

The following steps will guide you through the hardware configuration for this project.

1. Connect the Bluetooth Mate Silver to the solderless breadboard.
2. Connect the PIR to the three-pin terminal extender.
3. Connect the other end of the extender to a three-pin male header.
4. Connect the PIR to the solderless breadboard.

Figure 8-14 illustrates these processes.

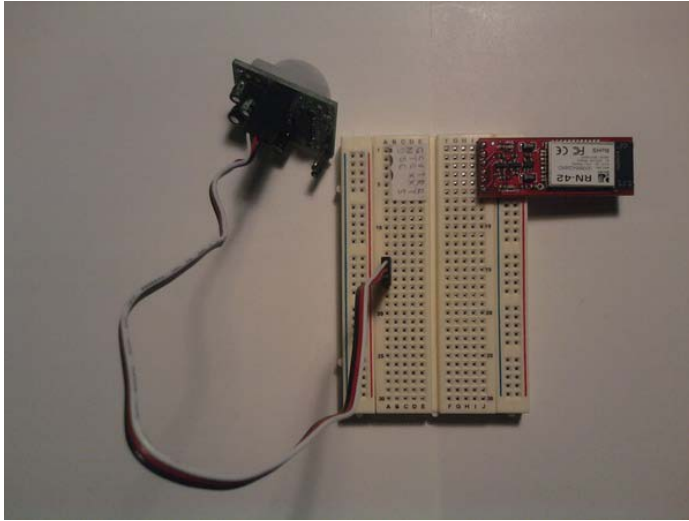


Figure 8-14. Attach the Bluetooth Mate Silver and the PIR sensor to the solderless breadboard.

Now we need to connect power and ground to our circuits. Follow these steps:

5. Connect power and ground to the solderless breadboard.
6. Connect power (+5V) to the Bluetooth Mate Silver and to the PIR.
7. Connect ground to the Bluetooth Mate Silver and to the PIR. Figure 8-15 illustrates these processes.

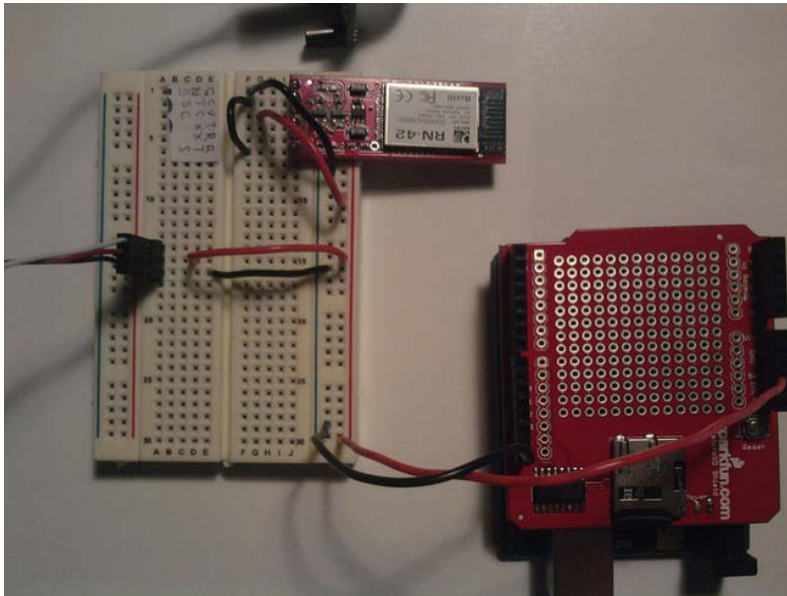


Figure 8-15. Connect power (+5V) and ground to Bluetooth Mate Silver and PIR sensor.

8. Connect digital pin 6 from the Arduino to the PIR's signal pin.
9. Connect the RTS pin and the CTS pin on the Bluetooth Mate Silver together. We will not connect the RX and TX pin on the Bluetooth Mate Silver to the Arduino until we upload the Arduino sketch to the Arduino.

Figure 8-16 shows the complete hardware configuration for this project.

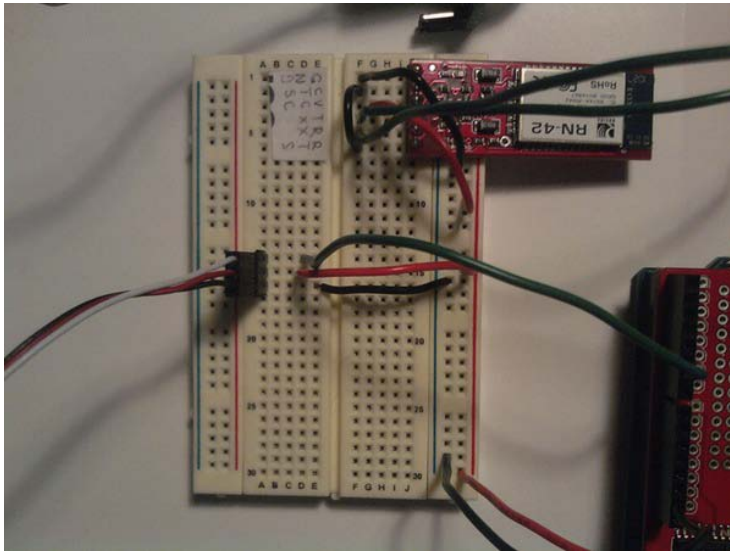


Figure 8-16. Connect PIR sensor to digital pin 6 on the Arduino.

Now that the hardware is configured, we can work on the software requirements for this project. The next section will discuss the software necessary to complete this project.

Writing the Arduino Software

The software will need to communicate with the PIR sensor, and because it acts like a switch, it will be very simple to implement it into our code. Also, we will need to write data to a microSD card and to the Serial Monitor. Listing 8-2 provides the code for this project.

Listing 8-2. *Using a PIR to Detect Motion— If Motion Is Detected, It Will Write to the Serial Monitor.*

```
// header files to use microSD shield
#include <SdFat.h>
#include <SdFatUtil.h>
//Create the variables to be used by SdFat Library
Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

char name[] = "Data.txt";

int PIRpin = 6;
int PIRval = 0;

void setup()
{
```

```

Serial.begin(115200);    // initialize serial communication
pinMode(PIRpin, INPUT);
pinMode(10, OUTPUT);    // Pin 10 must be set as an output for the SD communication to
                        // work.
card.init();            // Initialize the SD card and configure the I/O pins.
volume.init(card);      // Initialize a volume on the SD card.
root.openRoot(volume);  // Open the root directory in the volume.
}

void loop()
{
  PIRval = digitalRead(PIRpin);

  if(PIRval == HIGH) // if motion is detected
  {
    Serial.println("Motion Detected!!");
    delay(500);
    file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file 'name'
                                                            // in 'root' for writing to the
                                                            // end of the file.

    file.println("Motion Detected!!");
    file.close();                                           // Close the file.
    delay(500);
  }
  else
  {
    Serial.println("Nothing to Report"); // sends "Nothing to Report" to the Serial
                                        // Monitor.
    delay(500);
  }
}

```

First we include the headers so that we can send data to the microSD card. After that, we create instances of the variables necessary for the SdFat Library. Next we initialize the PIR pin on the Arduino. Then we enter the Setup Structure. Here, we start serial communication, set up the PIRpin variable to an output, and start communicating with the microSD shield. After that, we enter the Loop Structure. Here, we set the PIRval equal to the reading on digital pin 6. Then we have a conditional If-Else-Statement that will control what data is written to the Serial Monitor or the microSD card.

If the PIRval is “HIGH,” “Motion Detected!!” will be written to the Serial Monitor and the microSD card. Otherwise, “Nothing to Report” will be written to the Serial Monitor. Now that the software is completed, we can upload it to the Arduino. After we have uploaded the software to the Arduino, we can connect the TX pin on the Bluetooth Mate Silver to the RX pin on the Arduino, and the RX pin on the Bluetooth Mate Silver to the TX pin on the Arduino.

Now that the software requirements have been completed, we are ready to figure out any errors that we may have encountered. The next sections will discuss debugging the software and troubleshooting the hardware. Figure 8-17 illustrates connecting the RX and TX wires.

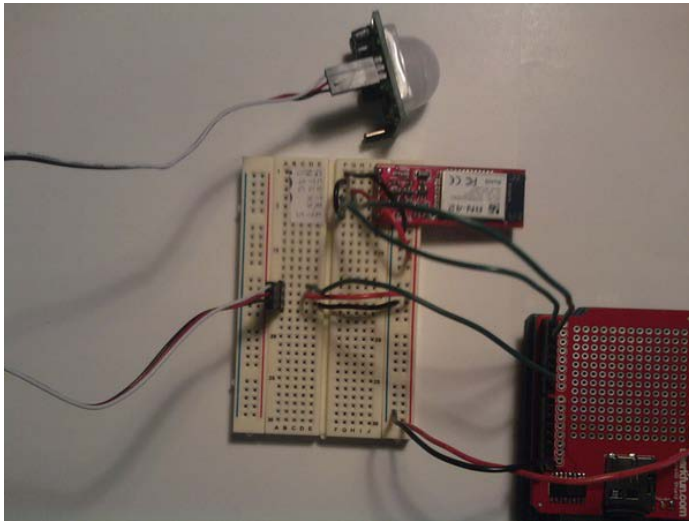


Figure 8-17. TX and RX pins on Bluetooth Mate Silver connected to TX and RX pins on Arduino.

Debugging the Arduino Software

This program was pretty straightforward. If you did run into an error, it may have been a syntax error; make sure you have all variables declared properly, and make sure that all of the statements in the program have a semicolon at the end of them. If you are still having trouble with the code, copy and paste my code into the Arduino IDE. If you are still having issues, it may be due to a library not being installed correctly. Make sure that the library files are all stored in the Arduino-022->Libraries folder. Now that we have debugged our software, we can focus on troubleshooting the hardware.

Troubleshooting the Hardware

You may have had some trouble configuring the hardware. One problem may be that you did not connect the RTS and the CTS pins together. Also make sure you only connect the RX and TX pins on the Bluetooth Mate Silver to the Arduino after you have uploaded the code to the Arduino through a USB cord.

Finished Prototype

Now that we have successfully met all the requirements for this project, we can submit the finished prototype to the company. Figure 8-18 illustrates the finished prototype.

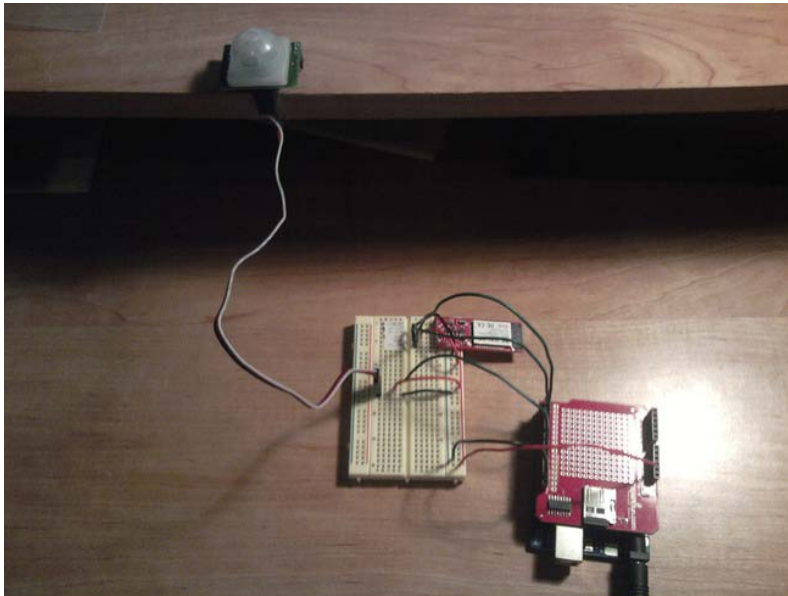


Figure 8-18. Project in action

Summary

We have gone over a few things in this chapter. First, we went over the new hardware we would be using in this chapter. After that, we introduced the first project of this chapter to get us ready for the final project. We then built a door alarm that senses whether a door is open or closed. Next we went through the engineering process to create a motion detection system for a company. There are always ways of making projects better and this project is no different, for example: you could add a real time clock to the project to get the exact time in which a motion was detected, or you may want to take a picture when motion has been detected, so there are many ways to make this project more advanced.

Error Messages and Commands: Using GSM Technology with Your Arduino

In this chapter, we'll be using attention (AT) commands to send text messages to your phone when a particular condition is met. The first two projects will give you an understanding of the GSM protocol and how to configure the Cellular Shield to send text messages. The AT commands that we will be using are AT+CMGF and AT+CMGS, which will always begin with AT and end with a carriage return. (For more information on the AT command set go to

www.sparkfun.com/datasheets/CellularShield/SM5100B%20AT%20Command%20Set.pdf.)

These preliminary projects cover sending a simple text message and a door alarm with text alerts. Once these are completed, we will move on to the final project of creating a GPS tracker for our customer; this GPS tracker will need to send both latitude and longitude coordinates to a cell phone number of the customers choosing.

But before we can do any of these projects, we need to first learn a bit more about the new hardware for this chapter. The next section will introduce the Cellular Shield from Sparkfun.

■ **Caution** Enter this chapter at your own risk. Sending text messages costs real money, so make sure your code is exactly the same as my code. Otherwise, you may find you have sent more text messages than you planned.

Hardware Explained: Cellular Shield

The Cellular Shield from Sparkfun will allow us to send text messages to other phones or any device that is GSM and can handle text messages. This shield has a GSM module attached and has a SMA connector attached to the GSM module for easy antenna attachment. You will also need to acquire a SIM card for this shield. I am using a T-Mobile GO Phone's SIM card; it works great and is relatively cheap. Also, you will need to send your country code to the Cellular Shield (more on that during the first project of this chapter). Figure 9-1 illustrates the Cellular Shield with a SIM card. This shield is already configured to work with pin 2 and 3 on the Arduino. You can reconfigure this if you need to by desoldering the two solder spots at the top right corner of the Cellular Shield.

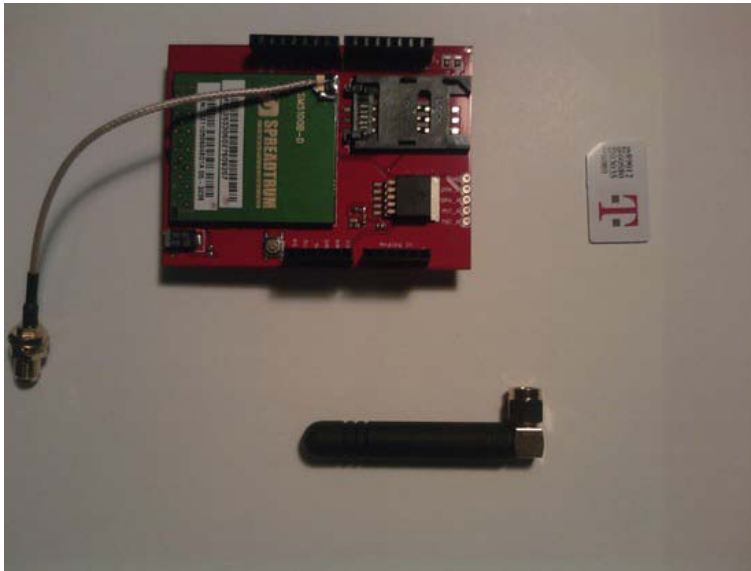


Figure 9-1. Cellular Shield with duck antenna and T-mobile SIM card

When using the Cellular Shield, we will also need to use the NewSoftSerial Library that we used in Chapter 5. Now that we know a bit more about the hardware, let's move on to the AT commands that we will be using to send text messages to the Arduino.

Understanding the AT Command Set

We will be using a few AT commands to send text messages to other phones. This protocol can be used directly with the Arduino and the Cellular Shield with a little extra code. The AT commands we will be using are AT+SBAND = x, AT+CMGF = 1, and AT-CMGS = "Phone Number." Some of these commands need to be manipulated in order to work with the Arduino, but I will go over that when we are going through the projects. For now, here is a brief description of each:

- AT+SBAND = x Command: This command will be used in the first project to set the frequency of the GSM module. For example:

AT+SBAND = 7 : This will set the frequency of the GSM module to the United States. If you wanted to use this Shield with a different country you would need to figure out the frequency your country uses.

- AT+CMGF = 1 Command: This command is used to set the GSM module to text messaging mode. To use this command with the Arduino you would input this code:

```
#include <NewSoftSerial.h>
NewSoftSerial cell(2,3);

cell.println("AT+CMGF=1"); // Sets the GSM module to text mode
```

- **AT-CMGS = “Phone Number” Command:** This command is used to configure the phone number that the text message will be sent to. When putting this into you Arduino program, it will look like this: (“1234567890”). Note that no country is code necessary. After you input the phone number, you need to give the GSM module a little bit of thinking time—normally about half a second then the module will expect the message. It has a very different setup when using it with the Arduino. It might look something like this:

```
#include <NewSoftSerial.h>
NewSoftSerial cell(2,3);

cell.print("AT+CMGS="); // start our message
cell.print(34,BYTE); // ASCII equivalent of "
cell.print("xxxxxxxxxx"); // Phone number
cell.println(34,BYTE); // ASCII equivalent of "
delay(500); // give the module some thinking time
cell.print("Text Message"); // message set to phone number
cell.println(26,BYTE); // ASCII equivalent of Ctrl-Z
```

Now that you have a basic understanding of the AT command set, we can work through a couple of projects that will get us ready for the final project.

The Basics of GSM Communication

Now we can work on a couple of projects that will get you ready for the final project of this chapter. As mentioned, the projects are sending a text message and creating a door alarm with SMS messaging. These projects will go over special configurations for your Cellular Shield, so make sure you read through these projects before you move on to the final project.

Project 9-1: Sending a Text Message

In this project we will set up our Cellular Shield so that it is able to send text messages. In order to do this, I am going to introduce a free terminal program that will allow us to send messages to the cellular module. After that, we can focus of sending text messages to another cell phone.

Hardware for This Project

Figure 9-2 shows some of the hardware being used in this project. (Not pictured: 9V battery, AC adapter, 9V connector.)

- Arduino Duemilanove (or UNO)
- Cellular Shield from Sparkfun
- SIM card (T-Mobile)
- Duck Antenna from Sparkfun
- 9V battery or AC adapter
- 9V battery connector (if 9V battery is used)



Figure 9-2. Hardware for this project

Configuring the Hardware

The following are the steps required to ensure we are sending and receiving information from the Cellular Shield:

1. We will first need to configure the Cellular Shield to work in the country in which you are located.
2. To do this, we first need to download the terminal program from <https://sites.google.com/site/terminalbpp/>. This is a free terminal program, but if you are feeling generous, feel free to give a contribution (thanks, Vlado Brajer). If you are using Linux or OS X, you can use SyncTERM, although it is not covered in this book. Here is the web site for SyncTERM: syncterm.bbsdev.net.
3. Unzip the terminal program to your desktop. You don't have to install this software after you are done unzipping it; double-clicking the icon will prompt the program to start. Figure 9-3 shows the terminal program.

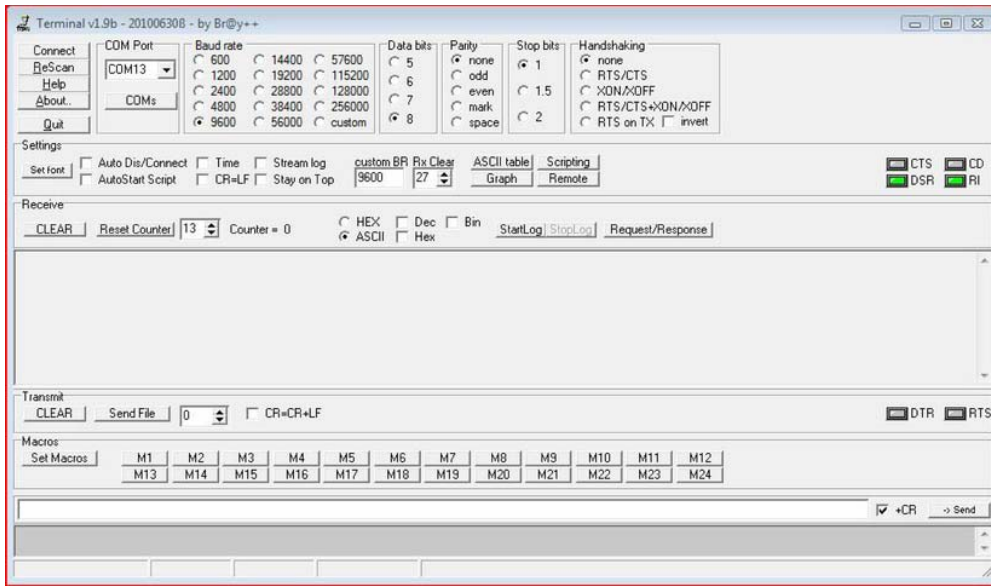


Figure 9-3. Terminal program

4. Connect the antenna to the Cellular Shield and connect the Cellular Shield to the Arduino.

Now that we have the software we need in order to configure the Cellular Shield, we need to run software on the Arduino to make sure we are receiving information from the Cellular Shield. To do so, follow these steps:

5. Copy and paste the code in Listing 9-1 into your Arduino IDE and connect your Arduino to your computer.
6. Upload Listing 9-1 to the Arduino.

Listing 9-1. Pass-Through Sample Sketch

`/* SparkFun Cellular Shield - Pass-Through Sample Sketch`

`SparkFun Electronics Written by Ryan Owens CC by v3.0 3/8/10`

`Thanks to Ryan Owens and Sparkfun for sketch */`

`#include <NewSoftSerial.h> //Include the NewSoftSerial library to send serial commands to the cellular module.`

`#include <string.h> //Used for string manipulations`

`char incoming_char=0; //Will hold the incoming character from the Serial Port.`

`NewSoftSerial cell(2,3); //Create a 'fake' serial port. Pin 2 is the Rx pin, pin 3 is the Tx`

```

        //pin.

void setup()
{
    //Initialize serial ports for communication.
    Serial.begin(9600);
    cell.begin(9600);
    Serial.println("Starting SM5100B Communication...");
}

void loop()
{
    //If a character comes in from the cellular module...
    if(cell.available() >0)
    {
        incoming_char=cell.read();    //Get the character from the cellular serial port.
        Serial.print(incoming_char); //Print the incoming character to the terminal.
    }

    //If a character is coming from the terminal to the Arduino...
    if(Serial.available() >0)
    {
        incoming_char=Serial.read(); //Get the character coming from the terminal
        cell.print(incoming_char);   //Send the character to the cellular module.
    }
}

```

Your Serial Monitor will have the following commands in it:

```

+SIND: 1
+SIND: 10,"SM",1,"FD",1,"MC",1,"RC",1,"ME",1
+SIND: 0
+SIND: 11

```

```
+SIND: 3
+SIND: 4
+SIND: 8
```

It may take up to 30 seconds for these commands to display on the Serial Monitor. This is due to the GSM module on the Cellular Shield.

We need to get rid of that +SIND: 8 (which means network lost), because it is preventing us from communicating via SMS. To do so, we need to send a command to the cellular module to allow it to configure with the appropriate country code. We need to send this command to the Cellular Shield from the terminal program, but first we need to make sure we select the correct options on the terminal program. You will need to use the same port number that you used with the Serial Monitor—for example, COM13. Figure 9-4 illustrates the correct options to select to run the GSM pass-through example with the terminal program.

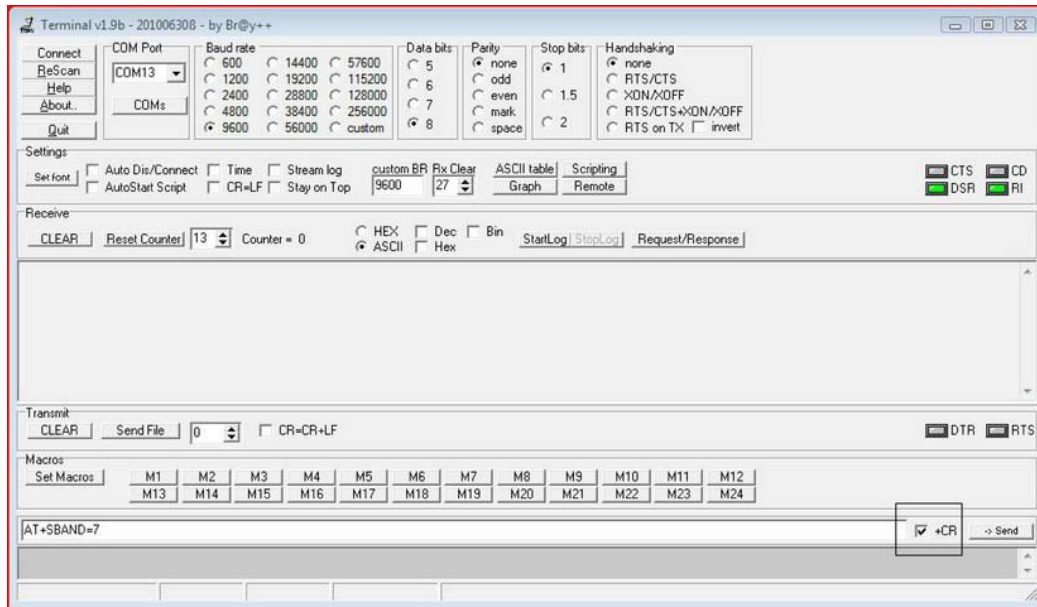


Figure 9-4. Terminal program with correct settings

Now that we are sending data to the terminal program, we can send data to the Cellular Shield. Follow these steps:

7. First type AT+SBAND=7 and check the +CR box (this is very important) at the bottom right-hand of the terminal program.
8. After that, hit the Send button at the bottom right-hand of the terminal program.
9. Hit the Reset button on the Cellular Shield. You should receive the following commands from the Cellular Shield:

```
+SIND: 1
+SIND: 10,"SM",1,"FD",1,"MC",1,"RC",1,"ME",1
```

```
+SIND: 0
+SIND: 11
+SIND: 3
+SIND: 4
```

These commands mean that we are ready to send text messages with the Cellular Shield. First, make sure you have your antenna connected to the shield, and connect the shield to the Arduino if it is not already attached. That's it! Now that the configuration of this project is done, we can move on to writing the software for this project.

Writing the Software

We will be focusing on using AT commands in order to send text messages to other devices such as cell phones. Listing 9-2 provides the code for this project.

Listing 9-2: Sending a Text Message

```
#include <NewSoftSerial.h>

NewSoftSerial cell(2,3); // Soft Serial Pins

char phoneNumber[] = "xxxxxxxxx"; // Replace xxxxxxxx with the recipient's mobile number

void setup()
{
    cell.begin(9600);
    delay(35000); // Allow GSM to initialize
}

void loop()
{
    cell.println("AT+CMGF=1"); // set SMS mode to text
    cell.print("AT+CMGS="); // now send message...
    cell.print(34,BYTE); // ASCII equivalent of "
    cell.print(phoneNumber);
    cell.println(34,BYTE); // ASCII equivalent of "
    delay(500);
    cell.print("Hello, This is your Arduino"); // our message to send
    cell.println(26,BYTE); // ASCII equivalent of Ctrl-Z
```

```

        // this will send the following to the GSM module
        // on the Cellular Shield: AT+CMGS="phonenumber"<CR>
        // message<CTRL+Z><CR>

delay(15000); // The GSM module needs to return to an OK status

{

    delay(1);

}

while (1>0); // if you remove this you will get a text message every 30 seconds or so.

```

Let's examine this code more closely. First, we include the header for the NewSoftSerial Library. This will allow us to use the Cellular Shield, as it is preconfigured to use on pins 2 and 3. After that, we create an instance of the NewSoftSerial class and set the pins to 2 and 3. Next, we create a character array to hold the recipient's phone number. Then, we enter the Setup Structure; here, we start serial communication with the Cellular Shield. After that, we enter the Loop Structure; here, we first send the AT command AT+CMGF=1 to the Cellular Shield. This is used to set the GSM module to accept text messages. Next, we send the AT command that will send the text message to a specific phone number. Then, we send the text we would like to send to that phone number. We next set up an infinite While Loop. This is used because we do not want to send continuous text messages to a particular phone number. If you do want to send continuous messages, you could take out this While Loop, and you would receive a text message every 30 seconds or so.

Project 9-2: Door Alarm with SMS Messaging

This project will extend our previous chapter's project to have it send a text message to a phone, so we will need to incorporate the previous door alarm project's hardware configuration and the software for that project. The next section will discuss the hardware that we will need in order to create the door alarm with SMS messaging.

Hardware for This Project

Figure 9-5 shows some of the hardware being used in this project. (Not pictured: large enclosure, AC Adapter.)

- Arduino Duemilanove (or UNO)
- Cellular Shield from Sparkfun
- Antenna (duck antenna)
- SIM card (T-mobile)
- Ultrasonic rangefinder (Ping))) from Chapter 8)
- Large enclosure (from Chapter 8)
- AC adapter (9V 650mAh)

- 3-pin female extension (Ping))) Bracket Kit)
- 3-pin male to male adapter
- Medium solderless breadboard



Figure 9-5. Hardware for this project

Configuring the Hardware

First, I recommend that you go back to Chapter 8 and review Project 8-1 for the full configuration of that project, which we will be adding on to here. Then follow these steps:

1. Drill a hole for the duck antenna with a drill or dremel; just make sure that the hole can be reached by the SMA cable on the Cellular Shield (see Figure 9-6).



Figure 9-6. Modify enclosure so that it can hold the duck antenna.

2. Attach the duck antenna to the Cellular Shield through the hole you just drilled into the enclosure, as shown in Figure 9-7.



Figure 9-7. Attach duck antenna to SMA connector on Cellular Shield.

3. By now, the Arduino and the Cellular Shield should be attached. Next connect power (+5V) and ground to the solderless breadboard.
4. Connect the Ping))) ultrasonic sensor to the solderless breadboard where you connected your power (+5V) and ground wires from the Arduino.
5. Connect digital pin 7 to the signal pin on the ultrasonic sensor (see project 8-1 for the setup of the ultrasonic sensor). Figures 9-8 and 9-9 illustrate the hardware configuration for this project.



Figure 9-8. Hardware configuration (1 of 2)

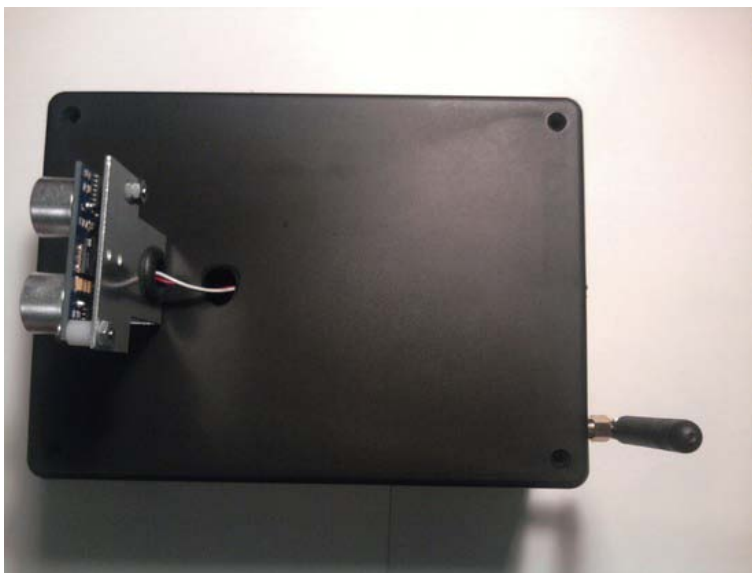


Figure 9-9. Hardware configuration (2 of 2)

■ **Note** In this project I do not use the microSD Shield or the Bluetooth Mate Silver.

Writing the Software

We only need to use the NewSoftSerial Library in order to use the Cellular Shield for this project. We will also have to use a few AT commands to send text messages when a door has been opened. As I said in the hardware configuration section, a lot of this project is from the first project in Chapter 8. Listing 9-3 provides the code for this project.

Listing 9-3: Send a Text Message When a Door Has Been Opened

```
#include <NewSoftSerial.h>

NewSoftSerial cell(2,3); // Soft Serial Pins

const int pingPin = 7;

char phoneNumber[] = "xxxxxxxx"; // Replace xxxxxxxx with the recipient's mobile number

void setup()
{
    cell.begin(9600);

    delay(35000); // Allow GSM to process
}

void loop()
{
    float duration, inches;

    pinMode(pingPin, OUTPUT);
    digitalWrite(pingPin, LOW);
    delayMicroseconds(2);
    digitalWrite(pingPin, HIGH);
    delayMicroseconds(5);
    digitalWrite(pingPin, LOW);

    pinMode(pingPin, INPUT);
    duration = pulseIn(pingPin, HIGH);

    // convert the time into a distance
    inches = duration / 74 / 2;

    if (inches >= 5)
    {
        cell.println("AT+CMGF=1"); // set SMS mode to text
```

```

    cell.print("AT+CMGS="); // now send message...
    cell.print(34,BYTE); // ASCII equivalent of "
    cell.print(phoneNumber);
    cell.println(34,BYTE); // ASCII equivalent of "
    delay(500); // Wait for the modem to respond
    cell.print("Door Has Been Opened!!"); // our message to send
    cell.println(26,BYTE); // ASCII equivalent of Ctrl-Z
    delay(15000);

    while (1>0); // if you remove this you will get a text message every 30 seconds or so.
  }
  else
  {
    // you may add additional code here.  }
}

```

Now let's review the code.

The first new code lies at the beginning of the program where we set up the phone number. Make sure you add the recipient's phone number; otherwise, this code will not do anything. The next new code for this project lies in the If statement inside the Loop Structure; this will send the text message "Door has Been Opened!!" the ultrasonic sensor reads any value less than 5 inches. This whole process will take around 35 to 50 seconds. The Else statement does nothing—it is there in case you want to add something to it later.

Now that we have gone over a few basic projects to get us familiar with cellular communication, we can revisit the company to see if they have any interesting projects for us to complete. It just so happens that they do have a project for us to complete that involves creating a GPS tracker. The company has set up a meeting for us to gather the requirements.

Requirements Gathering and Creating the Requirements Document

The company has several requirements for a GPS tracking system. This system will need to send longitude and latitude data to a cell phone every couple of minutes. The system will need to use the Arduino and the Cellular Shield that we have been using for this chapter. The text message that the GPS tracker will send needs to have this format:

Lat: xx.xxxxx , Long: xx.xxxxx

where the x represents the latitude and longitude location of the GPS tracker

This system will also need to be able to use a 9V battery as its power source. The company also wants us to use the NewSoftSerial Library and the TinyGPS Library for this project. Now that we have the requirements, we can compile them into a requirements document so that we have a better understanding of the project.

Hardware

Figure 9-10 shows some of the hardware being used in this project. (Not pictured: 9V battery and 9V battery connector.)

- Arduino Duemilanove (or UNO)
- Cellular Shield from Sparkfun
- SIM card
- GPS Shield
- GPS module
- Duck antenna
- 9V battery
- 9V battery connector

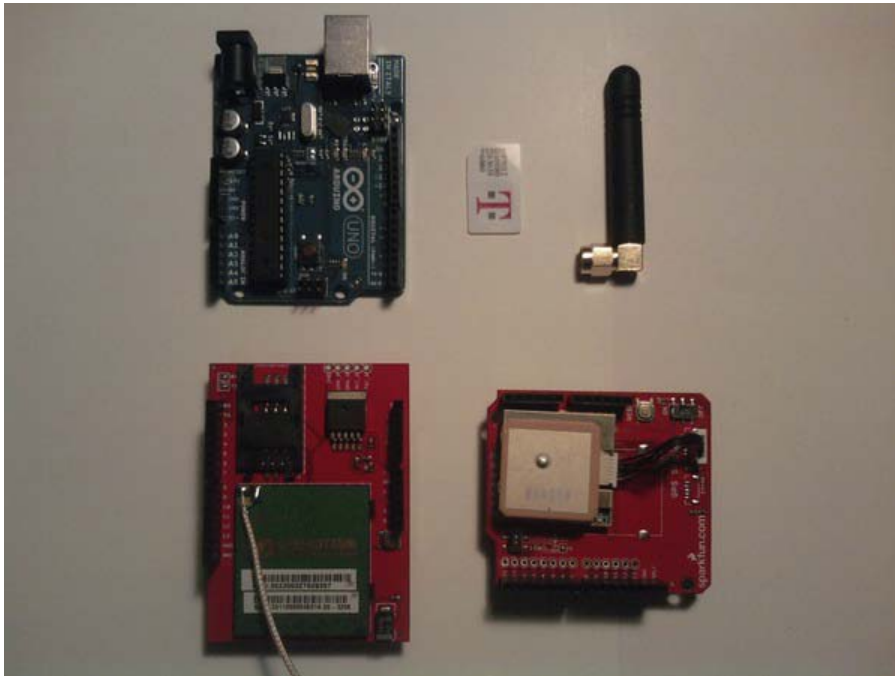


Figure 9-10. Hardware for this project

Software

The software requirements are as follows:

- Send latitude and longitude location in a text message to cell phone every couple of minutes (this is a loose requirement).
- Send a text message in this format:
Lat: xx.xxxxx , Long: xx.xxxxx
- Use NewSoftSerial Library and TinyGPS Library.

Now that we have the hardware and software requirements, we can create a flowchart to express the process the program will need to follow in order to meet all of the requirements (see Figure 9-11).

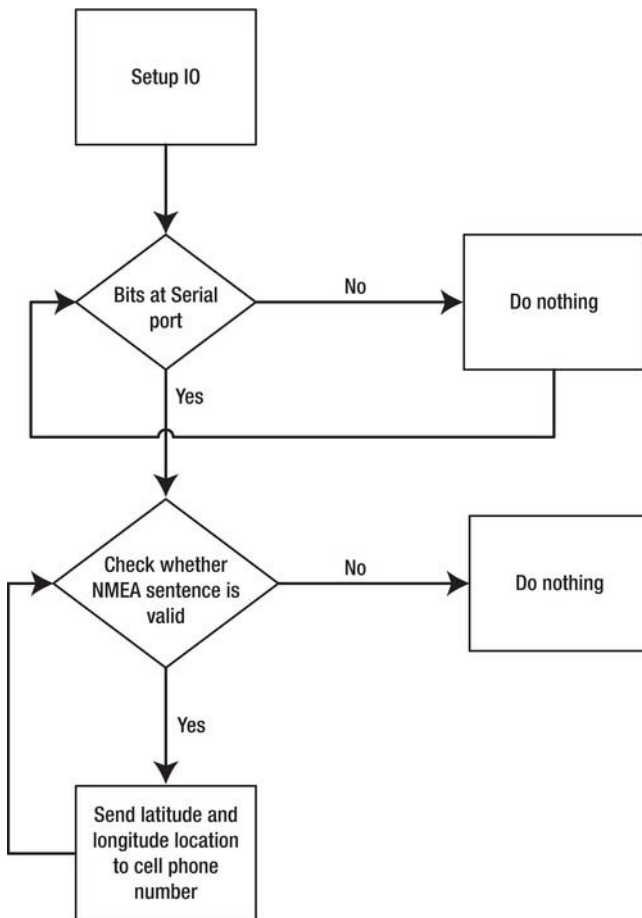


Figure 9-11. Flowchart for this project

Configuring the Hardware

The following steps will guide you through the hardware configuration for this project:

1. Attach the GPS Shield to the Cellular Shield.
2. Connect the duck antenna to the SMA connector that is attached to the GSM module.
3. Insert the SIM card into the Cellular Shield.

Because we will be using both the hardware serial and a software serial port, we cannot connect the Cellular and GPS Shields to the Arduino until we upload the software. You'll see how shortly, so stay tuned.

Figure 9-12 illustrates the partial hardware configuration.

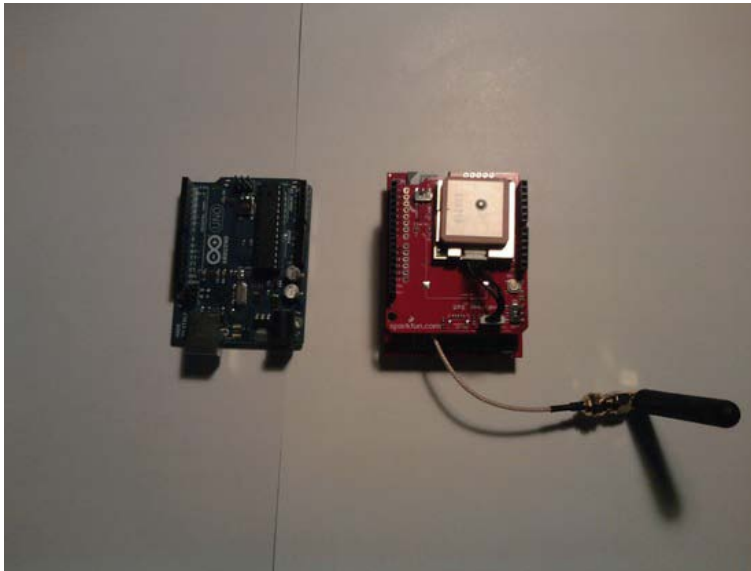


Figure 9-12. Partial hardware configuration

Now that the hardware is as configured as it can be, we need to work on creating the software to send latitude and longitude location to a phone using text messaging. The next section will discuss this software.

Writing the Software

The software will need to communicate with a hardware serial and software serial; thus, we will use the NewSoftSerial Library. We will also need to parse the NMEA commands that the GPS module will send to the hardware serial, so we will need to use the TinyGPS Library. Other than that, this piece of software will have the same structure as Chapter 5's GPS data logger project. Listing 9-4 provides the code for this project.

Listing 9-4: Send Latitude and Longitude Coordinates in a Text Message

```

#include <NewSoftSerial.h>
#include <TinyGPS.h>

NewSoftSerial cell(2,3); // Soft Serial Pins
TinyGPS gps;

char phoneNumber[] = "xxxxxxx"; // Replace xxxxxxxx with the recipient's mobile number

void printFloat(double number, int digits); // function prototype for printFloat() function

void setup(void)
{
  Serial.begin(4800); // GPS
  cell.begin(9600); // Cellular Shield
  delay(35000); // Allow GSM to initialize
}

void loop()
{
  while (Serial.available() > 0)
  {
    int c = Serial.read();

    // Initialize Longitude and Latitude to floating point numbers
    if(gps.encode(c)) // New valid sentence?
    {
      float latitude, longitude;

      // Get longitude and latitude
      gps.f_get_position(&latitude,&longitude);

      cell.println("AT+CMGF=1"); // set SMS mode to text

      cell.print("AT+CMGS="); // now send message...

      cell.print(34,BYTE); // ASCII equivalent of "

      cell.print(phoneNumber);

      cell.println(34,BYTE); // ASCII equivalent of "

      delay(500);

      cell.print("Lat: "); // our message to send
      printFloat(latitude, 6);
      cell.print(", ");
      cell.print("Long: ");
      printFloat(longitude, 6);
    }
  }
}

```

```

        cell.println(26,BYTE); // ASCII equivalent of Ctrl-Z
    }
    delay(60000);
}

}

void printFloat(double number, int digits)
{
    // Handle negative numbers
    if (number < 0.0)
    {
        cell.print('-');
        number = -number;
    }

    // Round correctly so that print(1.999, 2) prints as "2.00"
    double rounding = 0.5;
    for (uint8_t i=0; i<digits; ++i)
        rounding /= 10.0;

    number += rounding;

    // Extract the integer part of the number and print it
    unsigned long int_part = (unsigned long)number;
    double remainder = number - (double)int_part;
    cell.print(int_part);

    // Print the decimal point, but only if there are digits beyond
    if (digits > 0)
        cell.print(".");

    // Extract digits from the remainder one at a time
    while (digits-- > 0)
    {
        remainder *= 10.0;
        int toPrint = int(remainder);
        cell.print(toPrint);
        remainder -= toPrint;
    }
}

```

In reviewing the code, you'll see we include the header files so we can use the NewSoftSerial Library and the TinyGPS Library. Then we create instances of NewSoftSerial and TinyGPS. After that, we initialize `phoneNumber[]` to hold the phone number to which we will send our text message. Next, we create a function prototype for our `printFloat()` function, and then we enter the Setup Structure. Here, we start hardware serial communication and software serial communication; then we wait for 35 seconds to allow the GSM module to configure properly. After that, we enter the Loop Structure; here, we use a while loop to make sure data is at the serial port; if there is, we will read the data from the serial

port and enter a condition If Statement. This If Statement will make sure that the NMEA sentences are valid. If they are valid, longitude and latitude data will be sent in a text message to the phone number you indicated at the beginning of the program. This code is used to send the text message in the format specified by the requirements document.

```
cell.print("Lat: "); // our message to send
printFloat(latitude, 6);
cell.print(" , ");
cell.print("Long: ");
printFloat(longitude, 6);
// This will send a text message in the following format:
// Lat: xx.xxxxx , Long: xx.xxxxx
```

The `printFloat()` function has been used before in Chapter 5. We use `printFloat()` here for the same purpose to send floating-point numbers rather than doubles that do not have any precision. Now that the software is completed, you can upload the code through USB. After you have uploaded the code, you can attach the Cellular Shield and GPS Shield to the Arduino and connect power. In a minute or so you should receive a text message with the latitude and longitude location of the GPS tracking system. Figure 9-13 illustrates the completed configuration for this project.

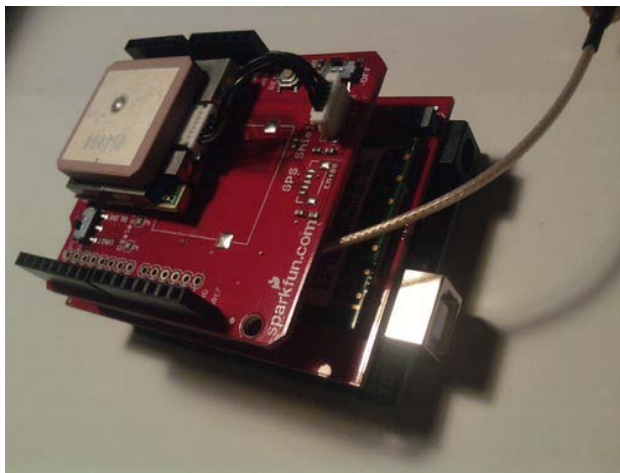


Figure 9-13. Completed hardware configuration for this project

Now that we have written the software and configured the hardware, we can fix any problems we had with the software or hardware. In the next section will discuss debugging the software.

Debugging the Arduino Software

You may have come across some errors while working on this project's software, so we need to go over a few common mistakes. First, make sure you update the phone number in the program. As it stands, it is xxxxxxxxxx. You need to change it to the number of the recipient phone; if you did not do this, nothing will happen, because there is no real phone number given. Also, you may have skipped the first project in this chapter; if you did, you may need to configure your GSM module to the correct frequency (see

project 9-1: sending a text message). If you have any other issues, I suggest you copy and paste this code into the Arduino IDE.

Troubleshooting the Hardware

Again, we see the beauty of using shields. You should not have any problems as long as you connected the shields correctly. You may still have had some problems with the SIM card. I can only say from experience that the T-mobile SIM cards work for this and all other projects in this chapter. I cannot say that other SIM cards will work, as I did not use them.

Finished Prototype

Well, if every couple of minutes you get a text message from your Arduino giving you the latitude and longitude location of the GPS tracker system, you will have a finished prototype ready for anything (well almost anything), as shown in Figure 9-13.

Summary

We started off this chapter by learning about the Cellular Shield. Then we moved on to understanding AT commands and how they are used with the Arduino. After that, we went through a couple of projects that helped us understand how to use the Cellular Shield with the Arduino. The projects showed us how to send a text message and how to send a text message when a sensor detects a certain value. Finally, we used the engineering process to create a GPS tracking system for a customer. This GPS tracking system sends latitude and longitude data to a cell phone of the customers choosing.

Control and Instrumentation: The Xbox Controller and the LabVIEW Process

In this chapter, you will learn about a new piece of software (LabVIEW) that will allow us to integrate our robot with a computer and an Xbox controller, per our customer's requirements document. LabVIEW is a very powerful programming language, as well as a powerful testing tool. In this chapter, we will use LabVIEW to interface with the Xbox controller so that we can control a robot's movements. We will not need to write any new Arduino code for this chapter, as we will be using the same code from Chapter 5's final project. We will not have any preliminary projects for this chapter, so we will first go over the basics of the LabVIEW environment and programming language so that you are more comfortable with the customer's project for this chapter.

■ **Note** I suggest visiting www.ni.com, as they will have a large selection of tutorials and videos.

Introduction to the LabVIEW Environment

We will first need to install the LabVIEW Student Edition onto a computer. You can get a great bundle from SparkFun at www.sparkfun.com/products/10812. If you don't want to buy the bundle, you can download a 30-day trial from www.ni.com/labview. This process is very straightforward. Simply put the LabVIEW CD into your DVD-ROM drive, and follow the onscreen instructions.

Now that we've installed the LabVIEW Student Edition, we can start using it for various projects, but first let's take a look at some of the fundamentals of LabVIEW. The next section will discuss the various parts of the LabVIEW environment that we will use in this chapter. They are the Front Panel, the Controls Palette, the Block Diagram, the Functions Palette, and the Tools Palette.

■ **Note** If you ever need help in LabVIEW, all you need to do is press Ctrl+H, and a help box will pop up. Anything you run your mouse over—a control, indicator, function, structure, and so on—the help box will give you information on.

The Front Panel

After you open LabVIEW, a screen will open. Click the “Blank VI” option on the screen, and two windows will open, one of them being the Front Panel. The Front Panel is where we will put all of the controls and indicators for our projects. When we are finished with the design of the Front Panel, we will have completed our GUI. You can also align the controls and indicators on the Front Panel by using Align Functions buttons at the top of the Front Panel. You will be starting your program from the Front Panel using the white arrow button in the upper-left corner of the window (you must click this white arrow in order for the program to start). Figure 10-1 shows the Front Panel.

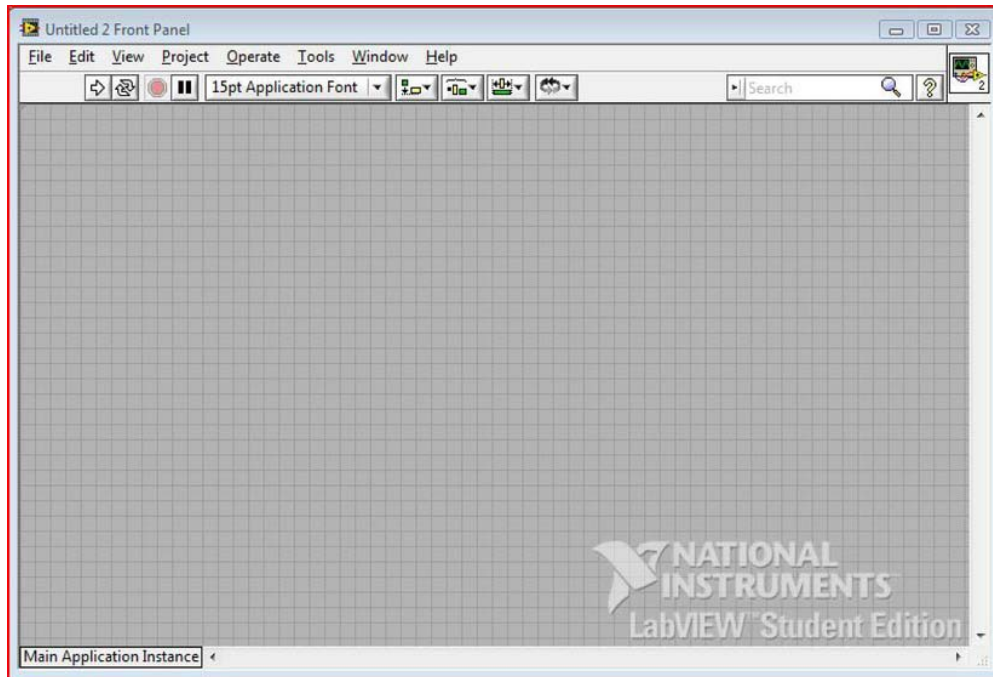


Figure 10-1. The Front Panel

■ **Note** If you have a broken arrow instead of a solid white arrow in the Front Panel, that means that your code has an error and will not run. If you click the broken arrow, an error dialog box will pop up and tell you what errors you have.

The Controls Palette

In this palette, you will find all of the controls and indicators that you will use to create your GUI. Some of these controls and indicators include toggle switches, numerical indicators, string controls and

indicators, and much more (I suggest playing around with this palette). To get to this palette, go to View ► Controls Palette. We will be using only a few controls and indicators in this chapter, but if you want to learn more, I suggest visiting www.ni.com, as they have a large selection of tutorials and videos. Figure 10-2 shows the Controls Palette.

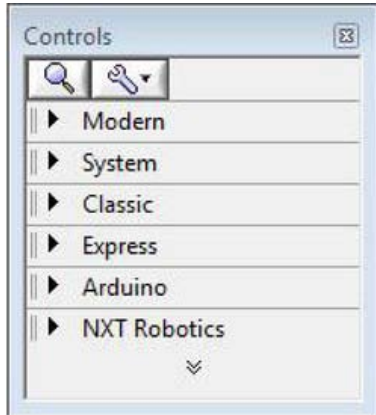


Figure 10-2. The Controls Palette

The Block Diagram

This is where all the magic happens. The Block Diagram is where we code the application and make the Front Panel do something (this can range from turning on an LED to GPS data analysis). It contains the white arrow button to run the program, and it also has a few debugging functions (we will talk about those later). The Block Diagram also has a palette that we will discuss in the next section. Figure 10-3 shows the Block Diagram.

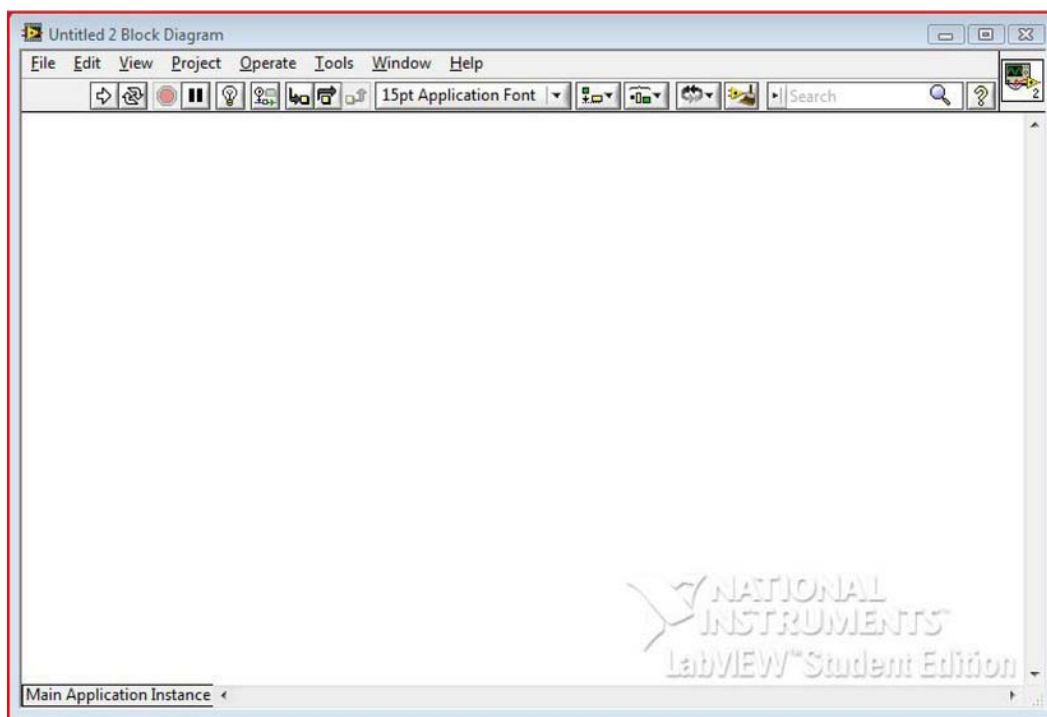


Figure 10-3. The Block Diagram

The Functions Palette

This palette has all of the various functions that you might or might not need. We will be going over only a few functions, but it is also a good idea to play around with this palette. You can find this palette by going to View ► Functions Palette. You will find functions for strings, numerical, Boolean, comparison, serial communication, and much more. Figure 10-4 shows the Functions Palette.

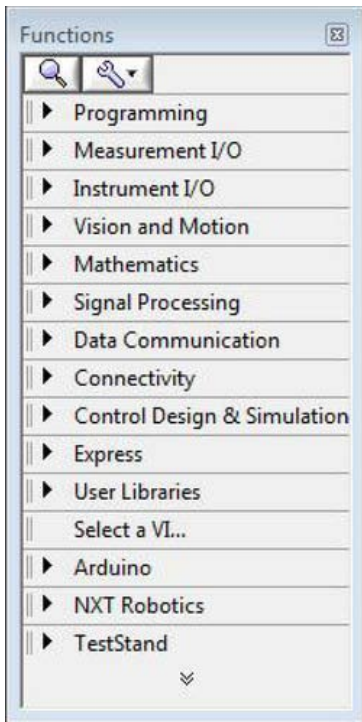


Figure 10-4. *The Functions Palette*

Next, we will discuss the Tools Palette, which is used to control what your mouse will do.

The Tools Palette

This palette can be used in either the Front Panel or the Block Diagram, although most of the options will work only in the Block Diagram. You can view this palette by clicking on the View ► Tools Palette. For the most part, we will not be using this palette because it defaults to Automatic Tool Selection, which means it will automatically select the best tool for what you are doing. Figure 10-5 shows the Tools Palette.



Figure 10-5. The Tools Palette

Now that you are a bit more familiar with the LabVIEW environment, let's go over some of the functions we'll be using in this chapter.

LabVIEW Functions Explained

LabVIEW uses a different approach to programming; it uses the Data Flow Process, which means data will “flow” from left to right on the screen. This makes code very easy to read and understand. The following functions will be used in the project for this chapter, as we will be creating software to scale values from the Xbox controller.

■ **Note** It is always a good idea to wire error clusters and error wires to keep data flow moving from left to right. We will see an example of this later in this chapter.

To find the first function that we will discuss, go to Block Diagram ► Functions Palette ► Programming ► Structures. Here you will see several types of loops and conditional structures. We will be using the While Loop, the Case Structure, and the Sequence Structure for this chapter. The next section will discuss the While Loop.

The While Loop

This loop operates like any other While Loop, except that it is a visual While Loop. In order to use it with other functions, you simply place the functions within the While Loop. The While Loop will run at least one time, and has many uses, just as in our Arduino programs. We can use a conditional terminal to stop the While Loop, and we can use an iteration terminal to check what iteration the While Loop is on. Figure 10-6 shows the While Loop.



Figure 10-6. *A While Loop*

In the next section, we will discuss the Case Structure and its functions.

The Case Structure

This is a conditional structure much like the Switch Statement or the If-Else-If Statements. You can have a true or false Case Structure, or you can use enumerated data to have multiple case statements, such as a State Machine; however, we will not go over State Machines in this book. To use a Case Structure, you will need to select the Case Structure from the Functions Palette ► Programming ► Structures, and drag the Case Structure to the appropriate size that you need. You can then switch from the true case to the false case by clicking on the arrow at the top-center of the Case Structure. The Case Structure uses the Selector Terminal to select the case that the Case Structure will call (we will see an example of this in the project for this chapter). Figure 10-7 shows a Case Structure.

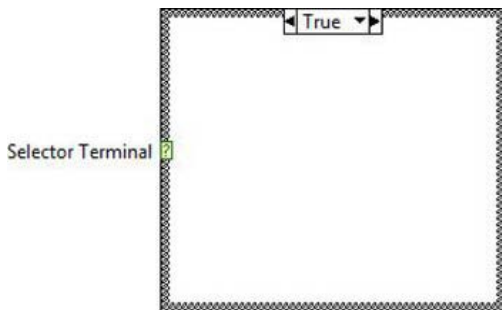


Figure 10-7. *A Case Structure*

The Sequence Structure

This structure is used to force code to flow in a sequential manner (as the name suggests). It is not good LabVIEW programming practice to have multiple Sequence Structures or Stacked Sequences, as they hide code and alter the Data Flow Process. However, a one-frame Sequence Structure used well, such as for initializing values, is not a bad practice. You can find the Sequence Structure by going to the Functions Palette ► Programming ► Structures. Figure 10-8 shows a Sequence Structure.

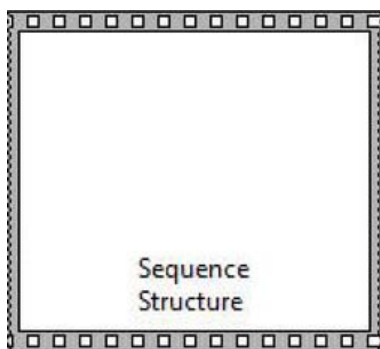


Figure 10-8. A Sequence Structure

Now that we have discussed the While Loops, Case Structures, and Sequence Structures, we can move on to the rest of the functions we will use for this chapter.

Numerical Functions

We will use a few Numerical Functions that will help us in the final project. To get to the Numerical Functions we need to first go to the Block Diagram, then go to the Functions Palette ► Programming ► Numeric. Figure 10-9 shows the Numeric Palette.

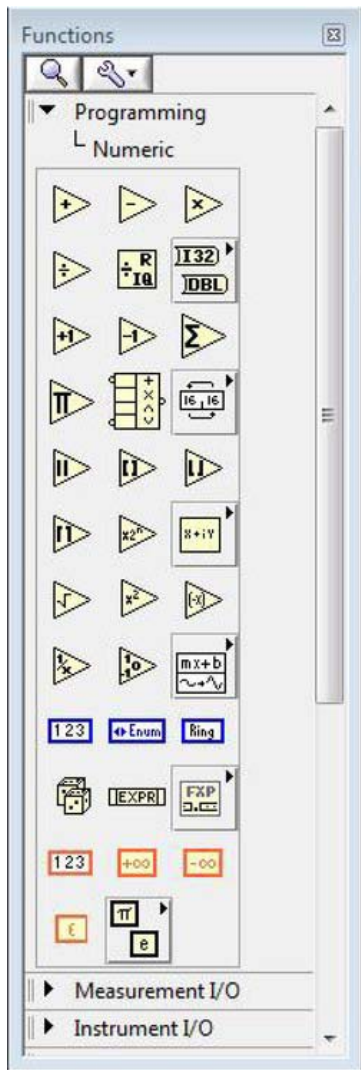


Figure 10-9. The Numeric Palette

In here, you will see functions ranging from decrementing to a random number function. We will use these functions to scale values from the Xbox controller to work with the Arduino. Now, we need to discuss a few functions from this palette that we will use in this chapter:

- *Divide*—This function is used to divide numerical values (see Figure 10-10(a)).
- *Multiply*—This function is used to multiply numerical values (see Figure 10-10(b)).

- *Decrement*—This function is used to subtract by one or decrement by one (see Figure 10-10(c)).

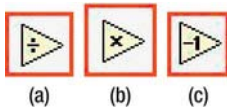


Figure 10-10. (a) The Divide Function, (b) the Multiply Function, and (c) the Decrement Function

String Functions

We use String Functions to manipulate strings. We will use a few of these functions to create the protocol so that the Xbox controller can communicate with the Arduino. You can find the String Functions by going to the Functions Palette ► Programming ► String (see Figure 10-11).

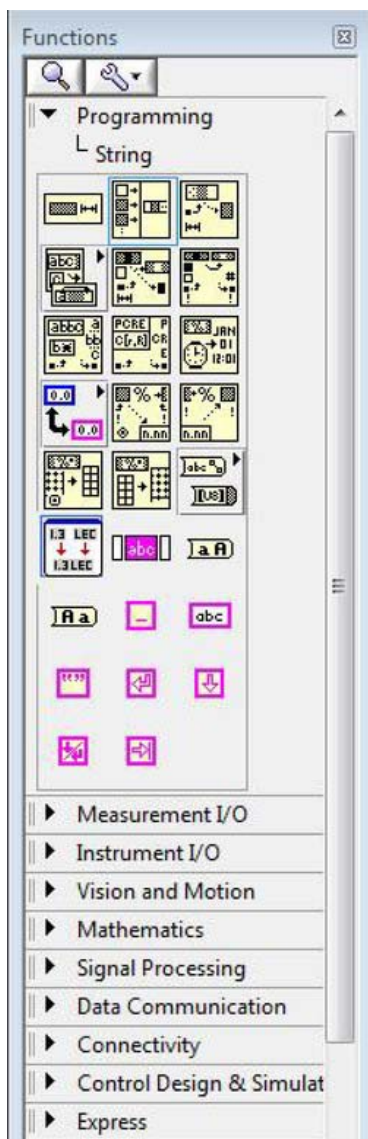


Figure 10-11. The String Palette

We'll be using the following String Functions for this chapter:

- *Concatenate String*—This function is used to combine two or more strings (see Figure 10-12(a)).

- **Number to Decimal String**—This function converts a numerical value to a string value (see Figure 10-12(b)).

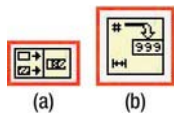


Figure 10-12. (a) The Concatenate String Function, and (b) the Number to Decimal String Function

Comparison Functions

We use Comparison Functions to compare types to one another; for example, whether $2 > 1$. You can find the Comparison Functions by going to the Functions Palette ► Programming ► Comparison (see Figure 10-13).

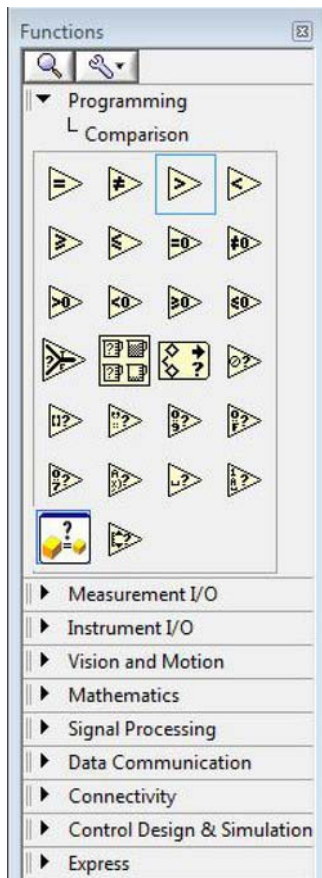


Figure 10-13. The Comparison Palette

We will be using the following Comparison Functions:

- *Less?*—This function compares a value x to a value y and tests whether x is less than the y value (see Figure 10-14(a)).
- *Greater?*—This function compares a value x to a value y and tests whether x is greater than y (see Figure 10-14(b)).
- *Less than 0?*— This function compares a value x to zero (see Figure 10-14(c)).

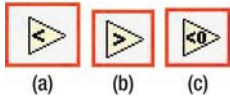


Figure 10-14. (a) *The Less? Function*, (b) *the Greater? Function*, and (c) *the Less than 0? Function*

Now that we have some of the fundamentals of LabVIEW covered, we can move on to Serial Functions and Input Device Control Functions.

Serial Functions

We can use these functions to communicate with USB devices. We will use these functions to write data from the Xbox controller to the Arduino. You can find the Serial Functions by going to the Functions Palette ► Instrument I/O ► Serial. Figure 10-15 shows the Serial Palette.

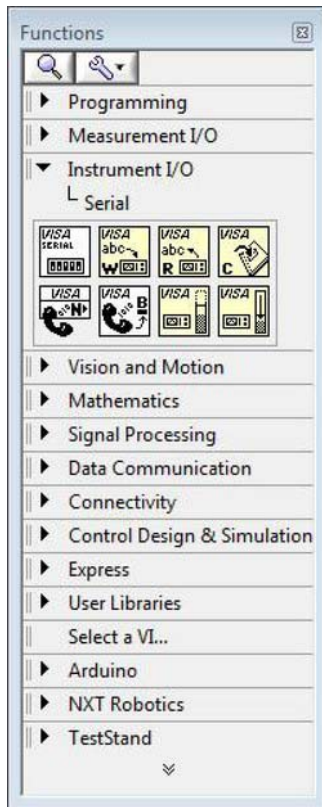


Figure 10-15. The Serial Palette

In this chapter, we will use the following Serial Functions:

- *Virtual Instrument Software Architecture (VISA) Configure Serial Port*—This function sets up the serial port's resource name, baud rate, parity, stop bits, flow control, and data bits (see Figure 10-16(a)).
- *VISA Flush I/O Buffer*—This function deletes the data that is stored on the buffer, allowing for more information to take its place (see Figure 10-16(b)).
- *VISA Write*—This function writes data to the serial port you specified in the VISA Configure Serial Port Function (see Figure 10-16(c)).
- *VISA Close*—This function closes out the serial communication session (see Figure 10-16(d)).

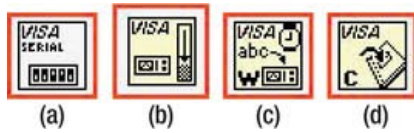


Figure 10-16. (a) The VISA Configure Serial Port Function, (b) the VISA Flush I/O Buffer Function, (c) the VISA Write Function, and (d) the VISA Close Function

Now that we understand the functions that we will need to communicate with a serial port, we can move on to understanding the functions that are necessary for the Xbox controller to work with the application for this chapter.

Input Device Control Functions

We can use these functions to communicate with HID (Human Interface Devices), such as a mouse, keyboard, or joystick. We will use these functions to communicate with the Xbox controller. You can find the Input Device Control Functions by going to the Functions Palette ► Connectivity ► Input Device Control. Figure 10-17 shows the Input Device Control Functions.

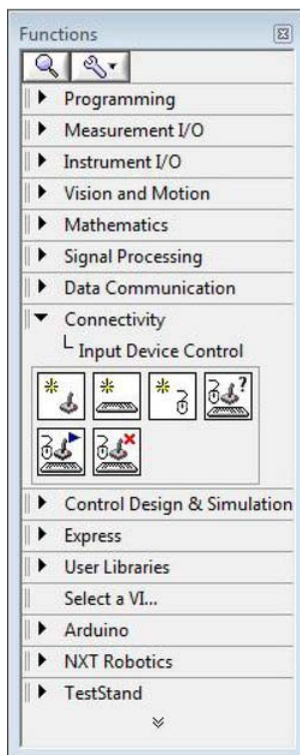


Figure 10-17. The Input Device Control Palette

In this chapter, we will use the following Input Device Control Functions:

- *Initialize Joystick*—This function starts the communication between the computer and the joystick that you selected with the device index (see Figure 10-18(a)).
- *Acquire Input Data*—This function gets the data from the joystick device, such as button information and axis information (see Figure 10-18(b)).
- *Close Input Device*—This function closes out the input device session (see Figure 10-18(c)).

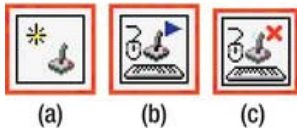


Figure 10-18. (a) The Initialize Joystick Function, (b) the Acquire Input Data Function, and (c) the Close Input Device Function

With a primer of the LabVIEW software environment under our belts, we can apply our newfound knowledge to our customer's project. Let's first gather the requirements, and then address hardware and software details.

Gathering Requirements and Creating the Requirements Document

The customer would like several additions on a previous final project that we created in Chapter 5. They need to be able to control the robot with an Xbox controller. They also need the speed to be variable from 0–255, using the controller's joystick. They want the program developed in LabVIEW, and they want the data displayed in a String Indicator in the following format, which makes the robot move forward:

1,255,1,255

The robot still needs to datalog longitude and latitude data to a microSD card. Now that we have our notes from the meeting, we can compile a requirements document for this project.

Hardware

Figure 10-19 shows some of the hardware being used in this project. (Not pictured: USB cable, 9V battery, 9V connector, and Chassis from Chapter 4.)

- Arduino Demilanove (or UNO)
- GPS shield
- GPS module
- microSD shield
- microSD card (512MB to 1GB)

- Xbox controller (wired controller)
- Chassis from Chapter 4 with H-bridge motor driver attached
- 9V battery connector
- 9V battery
- USB cable



Figure 10-19. Hardware for this project

Software

Here are the software requirements for this project:

- Write LabVIEW software that allows the Xbox controller to control the Arduino's motion, and control when longitude and latitude will be datalogged to a microSD card.
- Display scaled data from the joystick in a String Indicator in the following format: 1,255,1,255
- Use the same code from Chapter 5 for the Arduino.

We should now have everything we need to move forward with this project. The next section will discuss the hardware configuration for this project.

Configuring the Hardware

The following steps will guide you through the hardware configuration of this project:

1. Connect the Arduino to the Chassis that we used in Chapter 4.
2. Connect the microSD shield to the Arduino.
3. Connect the GPS shield to the microSD shield.
4. Connect digital pin 4 on the Arduino to pin 7 on the H-bridge.
5. Connect digital pin 5 on the Arduino to pin 1 on the H-bridge.
6. Connect digital pin 6 on the Arduino to pin 9 on the H-bridge.
7. Connect digital pin 7 on the Arduino to pin 10 on the H-bridge.
8. Connect power (+5V) and ground from the Arduino to the solderless breadboard.

■ **Note** Make sure that your motors are connected correctly and your components on the solderless breadboard are connected correctly.

Figure 10-20 shows the hardware configuration for this project.

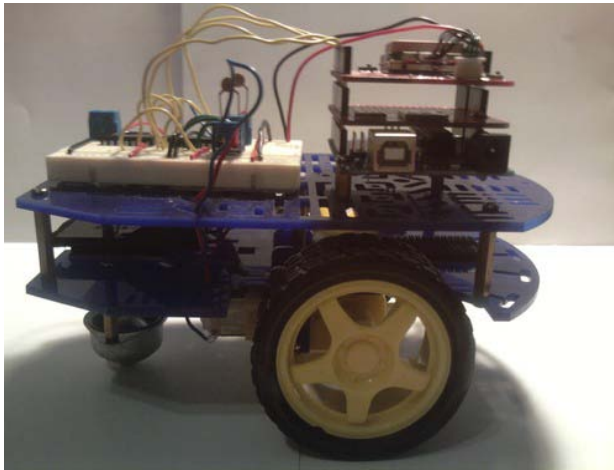


Figure 10-20. Hardware configuration for this project

■ **Note** Make sure your GPS shield is set to DLINE, and it is ON.

Now that we have finished configuring the hardware, we can move on to writing the software for this project. The next section will discuss the software that will be used in this project.

Writing the Software

This section is a bit different from what you normally see in this book; this is because we will not be writing any Arduino code. Instead, we will be writing code in LabVIEW, which will let us use the Xbox controller with the Arduino.

Getting Started

Use the following steps to get started writing the software in LabVIEW:

1. First, you will need to start LabVIEW by double-clicking the LabVIEW icon.
2. After LabVIEW starts, click the “Blank VI” option. Figure 10-21 shows this process.
3. A Front Panel and a Block Diagram should appear on the screen, ready to be added to. Figure 10-22 shows the Front Panel and Block Diagram.

■ **Note** A black-outlined box within a figure indicates where you should click with your mouse, or it denotes new code that has been added.



Figure 10-21. The start screen for LabVIEW (double-click “Blank VI”)

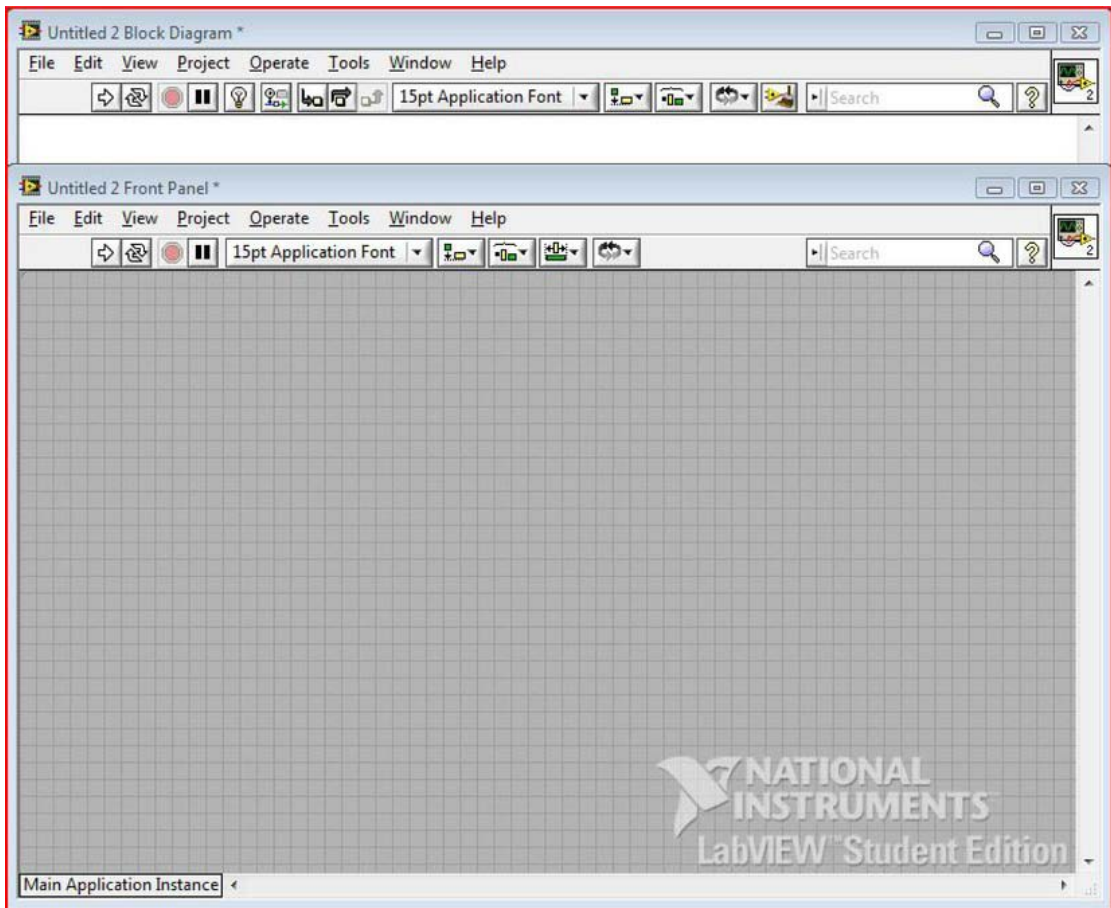


Figure 10-22. Block Diagram (top), and Front Panel (bottom)

Designing the GUI

Now we need to design the GUI for this project:

1. First, go to the Controls Palette ► Modern ► Boolean ► Stop Button, and click the Stop Button and drag it to the Front Panel.
2. After that, we need to add the String Indicator to the Front Panel. Go to the Controls Palette ► Modern ► String & Path ► String Indicator, and drag the String Indicator to the Front Panel. If you want to resize any controls or indicators, simply hover your mouse over the edge of the control or indicator and drag it out to the size that you want.

3. Next, add the Write Button to the Front Panel by going to the Controls Palette ► Modern ► Boolean ► OK Button, and dragging the OK Button to the Front Panel.
4. Then, you can rename the OK Button by clicking on the text and changing it to “Write.” After that, right-click the Write button, and a pop-up menu should appear; click Mechanical Action ► Switch when Pressed.

Programming the Application

For now, the Front Panel is complete, and we are going to move on to programming our LabVIEW application. Figure 10-23 shows the GUI for this project.

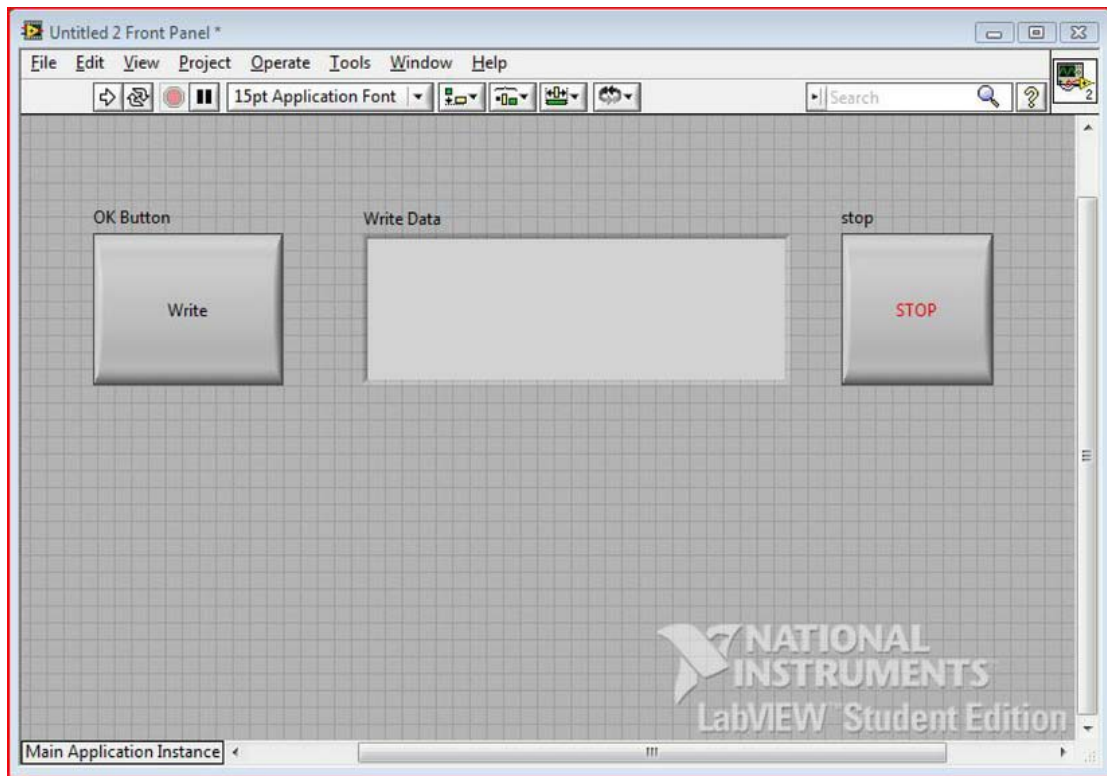


Figure 10-23. Partially completed Front Panel

1. Go to your Block Diagram. You should have three controls on it because we added them on the Front Panel.
2. First, go to the Functions Palette ► Programming ► Structures ► While Loop, and drag the While Loop onto the Block Diagram. Figure 10-24 shows this process.

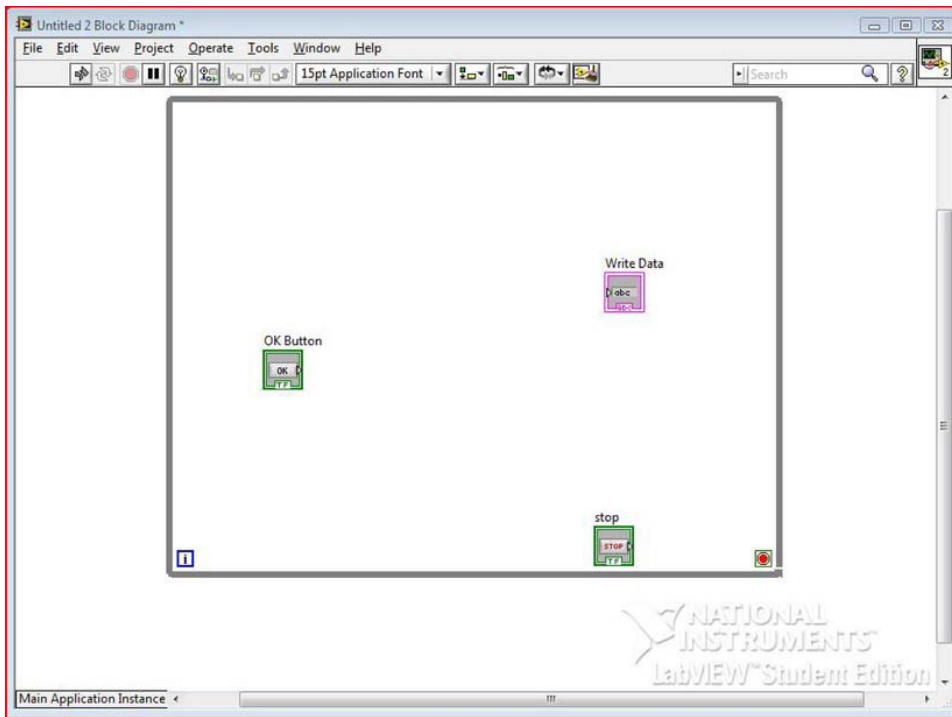


Figure 10-24. Add a While Loop to the Block Diagram.

3. Next, go to the Functions Palette ► Connectivity ► Input Device Control ► Initialize Joystick. Drag this function to the Block Diagram on the outside of the While Loop. Then, click on the device ID terminal on the Initialize Joystick Function and drag the wire to the While Loop.
4. Next, click the error out terminal and drag the wire to the While Loop.
5. After that, go to the Functions Palette ► Connectivity ► Input Device Control ► Acquire Input Data, and drag this function to the Block Diagram, inside the While Loop.
6. Then, attach the device ID from the Initialize Joystick Function that we wired to the While Loop to the device ID terminal on the Acquire Input Data Function.
7. Then, attach the error wire from the Initialize Joystick Function to the error in (no error) terminal on the Acquire Input Data Function.
8. Finally, go to the Functions Palette ► Connectivity ► Input Device Control ► Close Input Device, and drag this function to the Block Diagram to the outside of the While Loop on the right.

9. Then, connect the device ID from the Acquire Input Data Function to the device ID terminal on the Close Input Device Function. These terminals are on the edges of the functions. This particular terminal controls which device will be used.
10. After that, connect the error out terminal on the Acquire Input Data Function to the error in (no error) terminal on the Close Input Device Function.
11. After that, right-click the device ID on the left side of the Initialize Joystick Function, and a pop-up menu should appear. Go to Create ► Control on the pop-up menu; this will add a control to your Block Diagram and Front Panel. Figure 10-25 shows adding the Input Device Control Functions to the Block Diagram. (Make sure you leave plenty of space on the Block Diagram, as we still have some functions to add to it.)

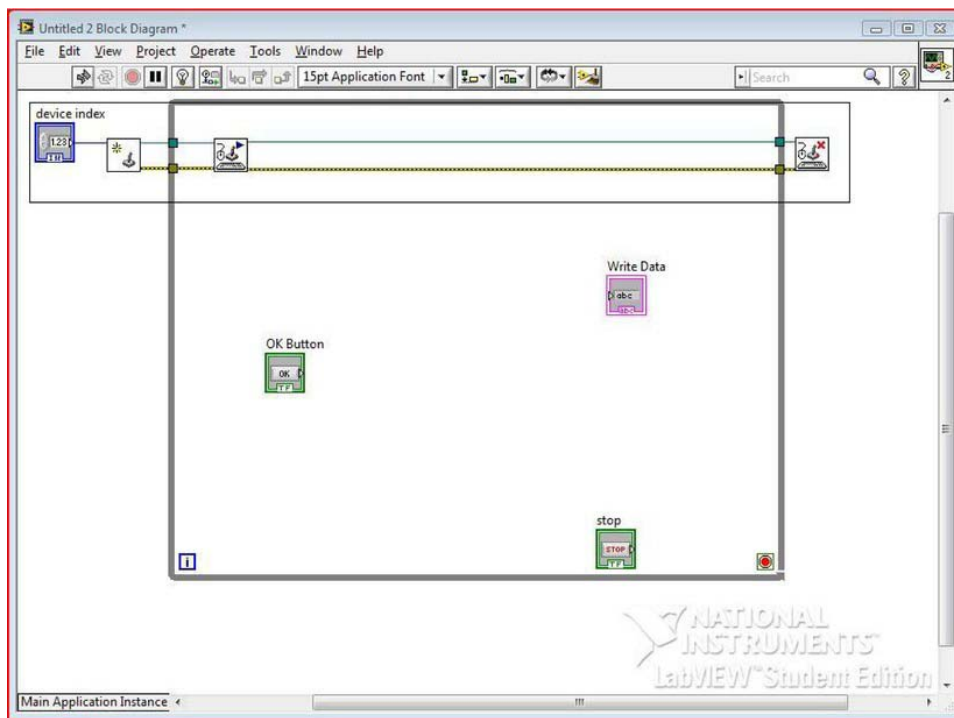


Figure 10-25. Add Input Device Functions to the Block Diagram.

Adding Serial Functions

Now that we have our Input Device Control Functions added and connected on the Block Diagram, we can move on to adding the Serial Functions to the Block Diagram.

1. First, go to the Functions Palette ► Instrument I/O ► Serial ► VISA Configure Serial Port, and drag this function to the outside of the While Loop on the left.

2. Then, go to the Functions Palette ► Instrument I/O ► Serial ► VISA Flush I/O Buffer, and drag this function to the right side of the While Loop, after the VISA Configure Serial Port Function.
3. Connect the VISA resource name out terminal from the VISA Configure Serial Port Function to the VISA resource name terminal on the VISA Flush I/O Buffer Function.
4. Connect the error out (no error) terminal from the VISA Configure Serial Port Function to the error in terminal on the VISA Flush I/O Buffer Function.
5. Add a Case Structure to the Block Diagram inside the While Loop. Go to the Functions Palette ► Programming ► Structures ► Case Structure, and drag it out a little on the Block Diagram.
6. Connect the Write Button to the conditional terminal on the Case Structure (it is the small square with a question mark in it). In the true case, we will need to add a VISA Write Function. Go to the Functions Palette ► Instrument I/O ► Serial ► VISA Write, and drag this function to the inside of the Case Structure to the True Condition.
7. Now we need to connect the VISA resource name out terminal on the VISA Flush I/O Buffer Function to the VISA resource name terminal on the VISA Write Function. Connect the error out terminal on the VISA Flush I/O Buffer Function to the error in (no error) terminal on the VISA Write Function.
8. Wire the VISA resource name out terminal on the VISA Write Function to the right wall of the Case Structure, and do the same for the error out terminal on the VISA Write Function. You will notice that there are two white squares on the right wall of the Case Structure; this is because your false statement has not been wired yet. Go to the false case of the Case Structure and wire from the VISA resource name out terminal on the VISA Flush I/O Buffer Function to the right wall of the Case Structure where the white square is located (these squares are called tunnels). Do the same thing for the error out terminal in the false case of the Case Structure.
9. Add a Sequence Structure to the Block Diagram. To do this, go to the Functions Palette ► Programming ► Structures ► Sequence Structure, and drag this function out a little, inside the While Loop next to the Case Structure.
10. Add a Wait(ms) Function to the inside of the Sequence Structure. To do this, go to the Functions Palette ► Programming ► Timing ► Wait(ms), and drag it to the inside of the Sequence Structure.
11. Right-click the milliseconds to wait terminal on the Wait(ms) Function; a pop-up menu will appear. Go to Create ► Constant; a small constant box will appear next to the Wait(ms) Function. Double-click this box and type in “100.”
12. Now wire the data that is coming from the Case Structure through the Sequence Structure (both error out and VISA reference name out).
13. Then, add another VISA Flush I/O Buffer Function after the Sequence Structure. To do this, go to the Functions Palette ► Instrument I/O ► Serial ►

- VISA Flush I/O Buffer, and drag it to the Block Diagram, after the Sequence Structure.
14. Connect the wires from the Sequence Structure to the VISA Flush I/O Buffer Function.
 15. Add a VISA Flush I/O Buffer Function to the outside of the While Loop, on the right side of the While Loop.
 16. Add a VISA Close Function after the VISA Flush I/O Buffer Function. You can find this function by going to the Functions Palette ► Instrument I/O ► Serial ► VISA Close.
 17. Right-click the VISA resource name terminal on the left side of the VISA Configure Serial Port Function; a pop-up menu will appear. Go to Create ► Control on that pop-up menu, and a control will be created on the Block Diagram and the Front Panel. Figures 10-26 and 10-27 illustrate adding the Serial Functions to the Block Diagram.
 18. Then, right-click the error on the left side of the VISA Configure Serial Port Function; a pop-up menu will appear. Go to Create ► Control on the pop-up menu, and a control will be created on the Block Diagram and Front Panel. Figures 10-26 and 10-27 show the process of adding the error cluster to the Block Diagram.

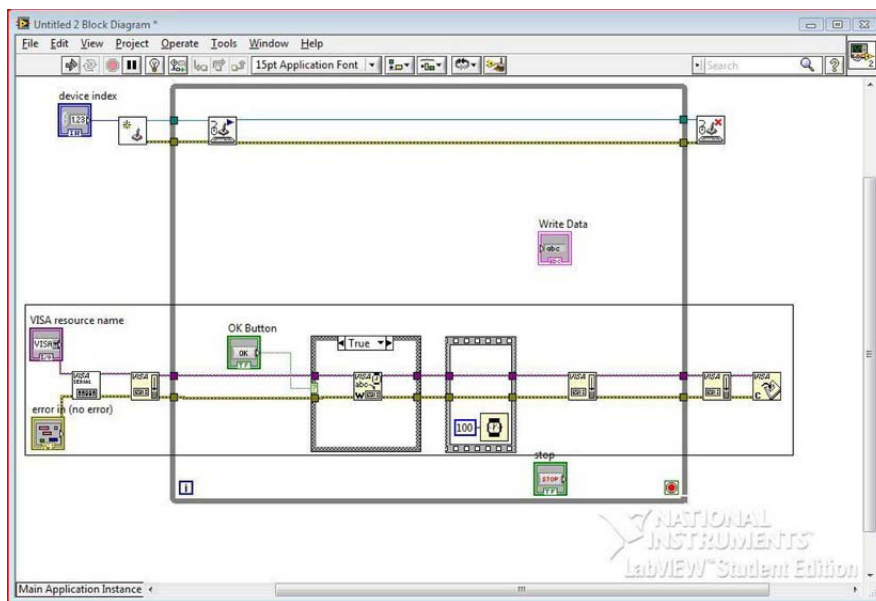


Figure 10-26. Add Serial Communication to the Block Diagram (part 1 of 2).

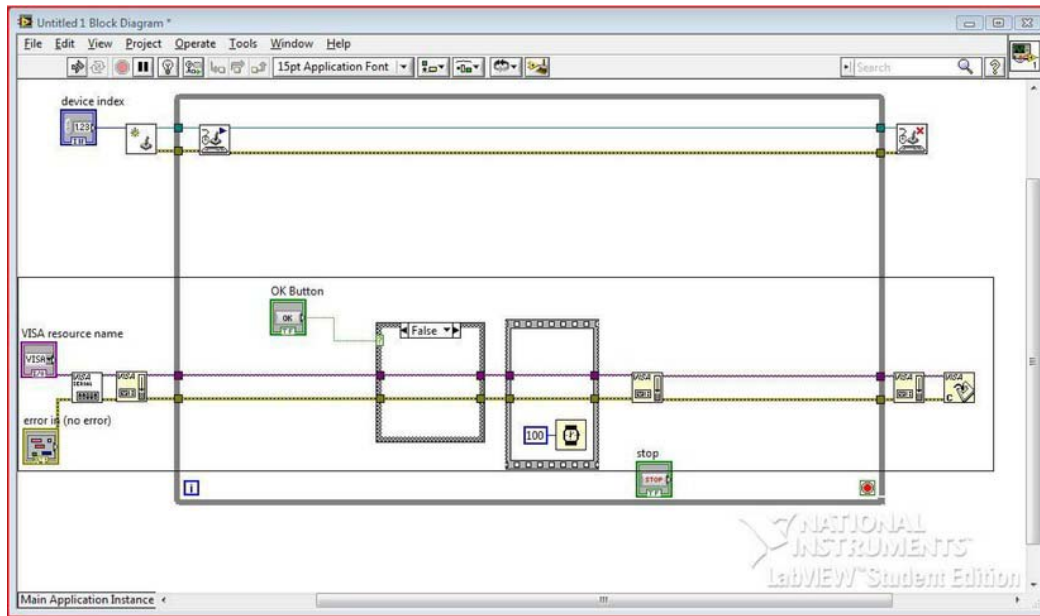


Figure 10-27. Make sure you wire the false condition of the Case Structure (part 2 of 2).

Completing the While Loops Condition

Now, we need to complete the While Loops condition:

1. First, add an Or Function to the Block Diagram. To do this, go to the Functions Palette ► Programming ► Boolean ► Or, and drag it next to the conditional terminal of the While Loop.
2. Then, connect the x.or.y? terminal to the conditional terminal on the While Loop.
3. After that, connect the Stop Button to the bottom terminal on the Or Function.
4. Next, add an Unbundle by Name Function to the Block Diagram. To do this, go to the Functions Palette ► Programming ► Cluster, Class, & Variant ► Unbundle by Name, and drag it to the Block Diagram, next to the top terminal on the Or Function.

Adding a Merge Errors Function

Now, we need to add a Merge Errors Function to the Block Diagram. To do this, follow these instructions:

1. Go to the Functions Palette ► Programming ► Dialog & User Interface ► Merge Errors, and drag it to the Block Diagram and connect the error out terminal to the Unbundle by Name Function.

2. Then, connect the status terminal of the Unbundle by Name Function to the top terminal on the Or Function.
3. Connect the error out terminal from the Acquire Input Data Function to the first terminal on the Merge Errors Function.
4. Then, attach the error out terminal from the VISA Flush I/O Buffer Function (the one inside the While Loop) to the second terminal on the Merge Errors Function. Figure 10-28 shows the completed While Loop condition.

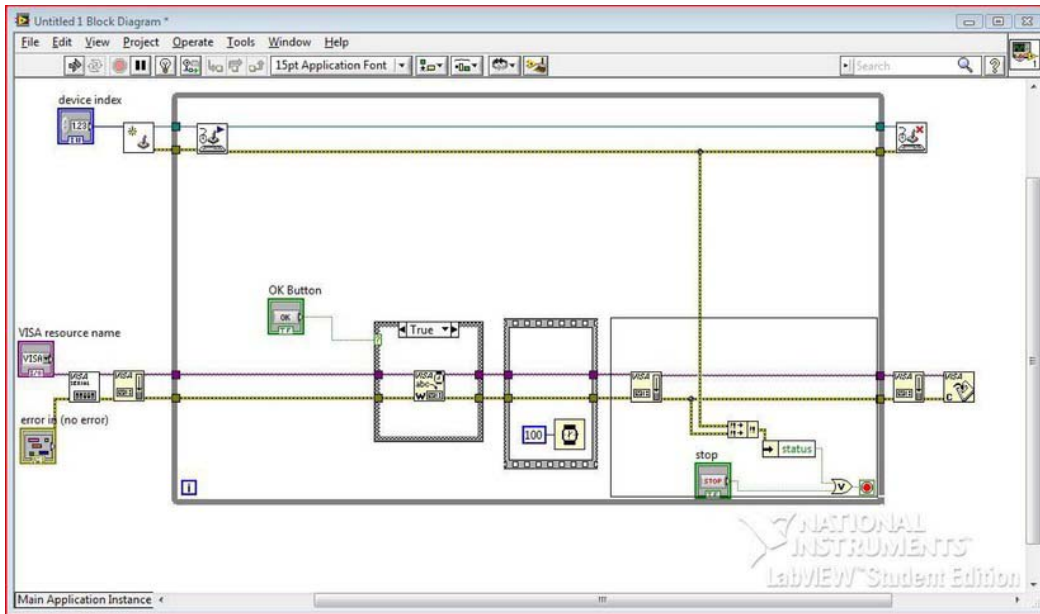


Figure 10-28. Complete the While Loop by adding in stop conditions.

Next, we need to add a SubVI to our program.

Adding a SubVI

A SubVI is much like the subroutines we create when we program with the Arduino IDE. Every function that we have put on the Block Diagram has been a SubVI, but we are about to add a SubVI that LabVIEW does not come with. You can find this SubVI with the source code from Chapter 10 at www.apress.com.

1. Once you have downloaded the SubVI (the SubVI's name is `ScaleandControl.vi`) to your Desktop, you can drag and drop it onto the Block Diagram.
2. Then, connect the axis info terminal on the Acquire Input Data Function to the axis info terminal on the Scale and Control Function.
3. Next, connect the button info terminal on the Acquire Input Data Function to the button info terminal on the Scale and Control Function.

4. After that, connect the string terminal on the Scale and Control Function to the Write Data String Indicator.
5. Finally, connect the string terminal on the Scale and Control Function to the write buffer terminal on the VISA Write Function. Figure 10-29 shows the completed SubVI.

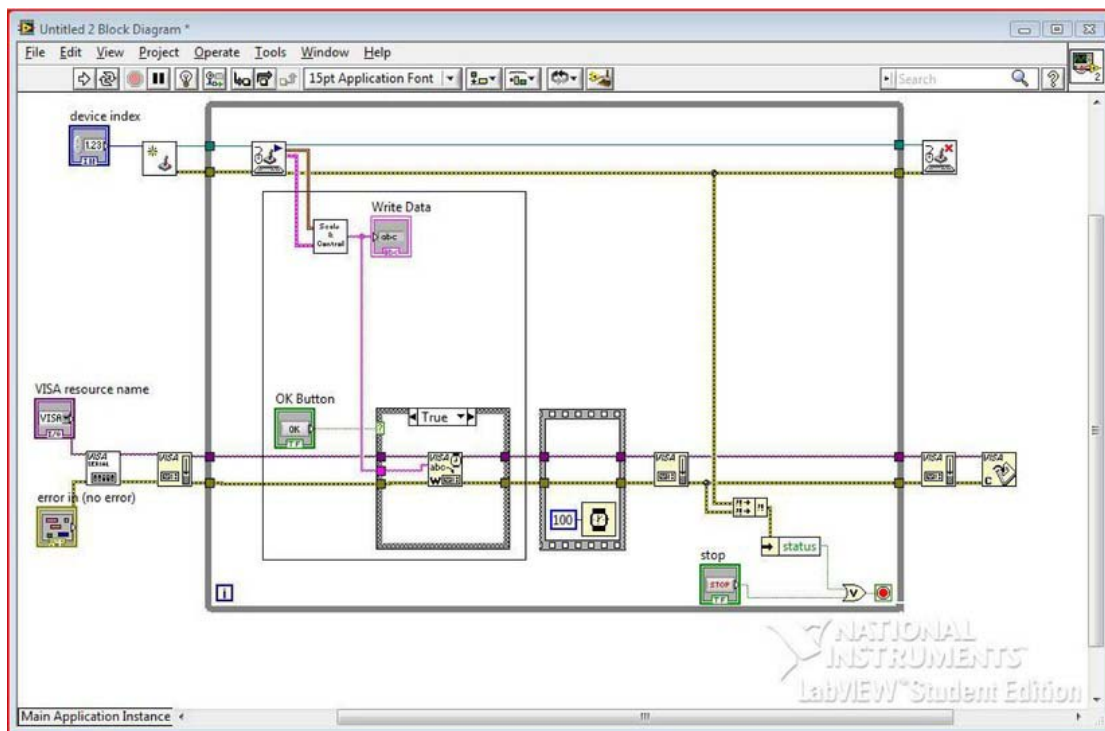


Figure 10-29. Add a SubVI to the Block Diagram that will scale the Xbox controller's values and dictate which direction to move the robot.

Error Handling

Now, we need to complete the error handling for this project.

1. First, add a Merge Errors Function to the outside of the While Loop, after the VISA Close Function.
2. Next, attach the error out terminal on the VISA Close Function to the second terminal on the Merge Errors Function.
3. Then, attach the error out terminal on the Close Input Device Function to the first terminal on the Merge Errors Function.

4. Finally, right-click the error out terminal on the Merge Errors Function, and a pop-up menu should appear. Go to Create ► Indicator, and an error indicator should be added to your Block Diagram and Front Panel.

Now, you can modify the Front Panel as you see fit because we have no more controls or indicators to add to it. That should do it for the LabVIEW software. Figures 10-30 and 10-31 show the completed software for this project.

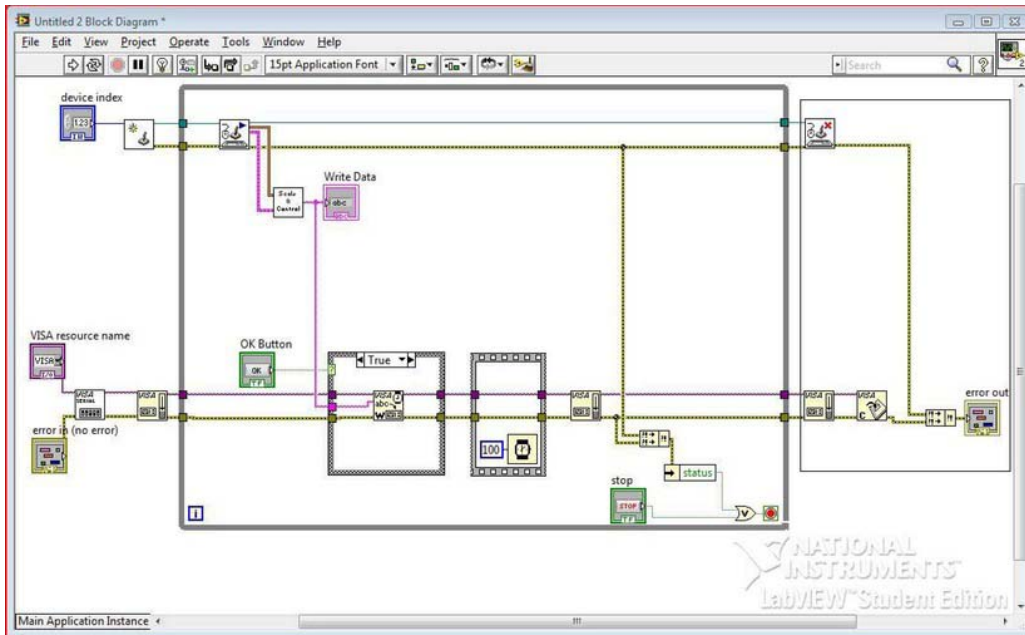


Figure 10-30. Finish Error Handling of the LabVIEW software to finish the program.

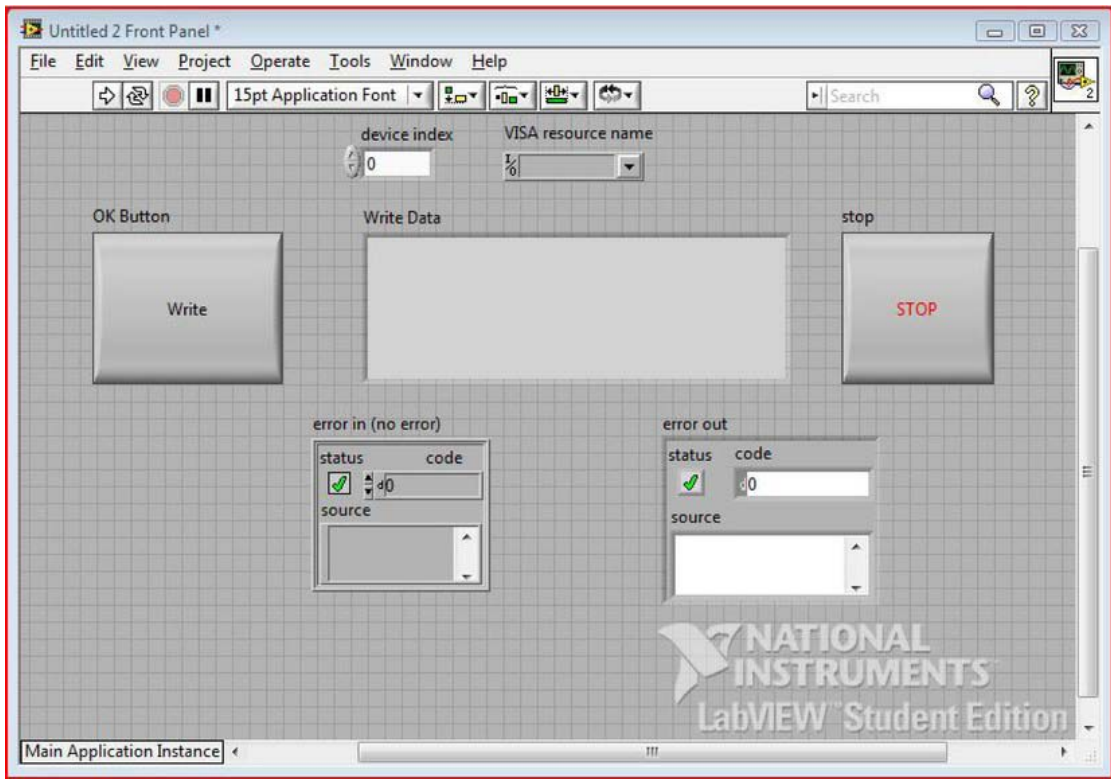


Figure 10-31. The finished Front Panel

Uploading the Code to the Arduino

Now that we have written our LabVIEW software, we need to make sure the correct code is uploaded to the Arduino. Listing 10-1 shows the Arduino code for this project.

Listing 10-1. Controls a Robot with a Simple Sentence or Sends GPS Location to a microSD Card

```
#include <SdFat.h>
#include <SdFatUtil.h>
#include <ctype.h>

#include <TinyGPS.h>

#include <NewSoftSerial.h>

TinyGPS gps;

NewSoftSerial nss(2,3);
```

```

void printFloat(double number, int digits); // function prototype for printFloat() function

int numCount = 0;

const int fields = 4; // how many fields are there? right now 4
int motorPins[] = {4,5,7,6}; // Motor Pins
int index = 0;           // the current field being received
int values[fields];      // array holding values for all the fields

Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;

char name[] = "GPSData.txt";    // holds the name of the new file
char LatData[50];              // data buffer for Latitude
char LongData[50];

void setup()
{
    Serial.begin(9600); // Initialize serial port to send and receive at 9600 baud
    nss.begin(4800);    // Begins software serial communication

    pinMode(10, OUTPUT); // Pin 10 must be set as an output for the SD communication to
                        // work.
    card.init();         // Initialize the SD card and configure the I/O pins.
    volume.init(card);   // Initialize a volume on the SD card.
    root.openRoot(volume);

    for (int i; i <= 3; i++) // set LED pinMode to output
    {
        pinMode(motorPins[i], OUTPUT);
        digitalWrite(motorPins[i], LOW);
    }

    Serial.println("The Format is: MotoADir,MotoASpe,MotorBDir,MotoBSpe\n");
}

void loop()
{
    if( Serial.available())
    {
        char ch = Serial.read();

        if (ch == 'G') // if Serial reads G
        {
            digitalWrite(motorPins[0], LOW);
            digitalWrite(motorPins[2], LOW);
            analogWrite(motorPins[1], 0);
            analogWrite(motorPins[3], 0);
        }
    }
}

```

```

while (numCount == 0)
{
    if (nss.available() > 0) // now gps device is active
    {
        int c = nss.read();
        if(gps.encode(c))    // New valid sentence?
        {

            // Initialize Longitude and Latitude to floating point numbers
            float latitude, longitude;

            // Get longitude and latitude
            gps.f_get_position(&latitude,&longitude);

            Serial.print("Lat: ");
            // Prints latitude with 5 decimal places to the Serial Monitor
            Serial.println(latitude,7);

            Serial.print("long: ");
            // Prints longitude with 5 decimal places to the Serial Monitor
            Serial.println(longitude,7);

            file.open(root, name, O_CREAT | O_APPEND | O_WRITE);    // Open or create the file
                                                                    // 'name'
                                                                    // in 'root' for writing
                                                                    // to the
                                                                    // end of the file.

            file.print("Latitude: ");
            printFloat(latitude, 6);
            file.println("");
            file.print("Longitude: ");
            printFloat(longitude, 6);
            file.println("");
            file.close();    // Close the file.
            delay(1000);    // Wait 1 second

            numCount++;
        }
    }
}
}
else if(ch >= '0' && ch <= '9') // If the value is a number 0 to 9
{
    // add to the value array
    values[index] = (values[index] * 10) + (ch - '0');
}
else if (ch == ',') // if it is a comma
{
    if(index < fields -1) // If index is less than 4 - 1...
        index++; // increment index
}
else

```

```

{
    for(int i=0; i <= index; i++)
    {
        if (i == 0 && numCount == 0)
        {
            Serial.println("Motor A");
            Serial.println(values[i]);
        }
        else if (i == 1)
        {
            Serial.println(values[i]);
        }
        if (i == 2)
        {
            Serial.println("Motor B");
            Serial.println(values[i]);
        }
        else if (i == 3)
        {
            Serial.println(values[i]);
        }

        if (i == 0 || i == 2) // If the index is equal to 0 or 2
        {
            digitalWrite(motorPins[i], values[i]); // Write to the digital pin 1 or 0
            // depending on what is sent to the arduino.
        }

        if (i == 1 || i == 3) // If the index is equal to 1 or 3
        {
            analogWrite(motorPins[i], values[i]); // Write to the PWM pins a number between
            // 0 and 255 or what the person has entered
            // in the serial monitor.
        }

        values[i] = 0; // set values equal to 0
    }

    index = 0;
    numCount = 0;
}

}

void printFloat(double number, int digits)
{

```

```

// Handle negative numbers
if (number < 0.0)
{
    file.print('-');
    number = -number;
}

// Round correctly so that print(1.999, 2) prints as "2.00"
double rounding = 0.5;
for (uint8_t i=0; i<digits; ++i)
    rounding /= 10.0;

number += rounding;

// Extract the integer part of the number and print it
unsigned long int_part = (unsigned long)number;
double remainder = number - (double)int_part;
file.print(int_part);

// Print the decimal point, but only if there are digits beyond
if (digits > 0)
    file.print(".");

// Extract digits from the remainder one at a time
while (digits-- > 0)
{
    remainder *= 10.0;
    int toPrint = int(remainder);
    file.print(toPrint);
    remainder -= toPrint;
}
}

```

If you would like to read the discussion of this program, please read the “Writing the Software” section in Chapter 5’s final project.

■ **Note** Make sure the Arduino and the Xbox controller are connected to your computer.

Operation

The following steps will guide you through the operation of this project:

1. To operate the LabVIEW software, you will need to first know the serial com port to which the Arduino is connected. Go to the Front Panel of the LabVIEW software and click the arrow on the VISA resource name control; a drop-down menu will appear, and you can select the correct serial port to which the Arduino is connected.

2. After that, you need to select the correct number ID with which the Xbox controller is associated. To do this, click inside the white area of the device index control and type in the number (the best thing to do is type in 1; if that doesn't work, type in 2; if that doesn't work, type in 3; repeat these steps with a larger number each time).
3. Then, click the white arrow at the top-left of the screen. After that, your program should run. Click the Write Button, and the robot should move when you move the joystick up, down, left, and right.

Now that we have written the LabVIEW software and Arduino software, we can move on to fixing any bugs or issues we might have had with both the hardware and software. In the next section, we will discuss how to debug the software.

Debugging the LabVIEW Software

LabVIEW has several built-in debugging tools. I will explain only two of them: Highlight Execution and probes.

Highlight Execution is used to see the flow of your program and to find where errors are occurring. It slows the program down considerably and runs through the code function by function. To use this debugging method, click the Highlight Execution button at the top of the Block Diagram (the button has a light bulb on it). If you click this function while the program is running, you will see bright lines showing you where you are in the code.

Probes are used to see specific values that controls or indicators are giving or receiving. For instance, if you wanted to see at what point a logical error is occurring in your code, you could use a probe to view data on the wires to figure out where the mistake is. To use probes, right-click any wire on the Block Diagram. A pop-up menu will appear. Click probe, and a display will pop up with an indicator showing you the value on that wire.

We can use these tools to figure out issues that the LabVIEW software might be having. If you have a broken arrow, you might have connected something incorrectly, or you forgot to connect a terminal on one of the functions. Make sure your code is exactly like that shown in Figure 10-30. Now, if you have a white arrow, and you are still having issues with your LabVIEW software, you might have encountered a logical error. Use probes and Highlight Execution to find these issues. Because every Xbox controller will have different calibration, you might need to adjust the limits on the Xbox controller. To do this, double-click the Scale and Control SubVI, and the Block Diagram for this SubVI should appear. Then adjust the constants, shown in Figure 10-42, to a higher or lower value, depending on what your Xbox controller is doing.

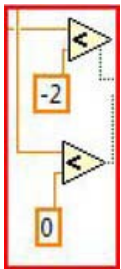


Figure 10-32. Change the constants on these functions to set the limits for vertical and horizontal movements of the joystick on the Xbox controller.

If the software is working, and you are still having issues, refer to the next section on troubleshooting the hardware.

Troubleshooting the Hardware

If you are having problems with your hardware, you should first make sure that everything is on, and the right selections are made on your GPS shield. Also make sure your H-bridge's components are all attached correctly. If your robot is not moving, make sure you have a 9V battery connected to your H-bridge, and your motors are connected. If you are still having issues, please read the “Configuring the Hardware” section from Chapter 5.

■ **Note** If you are not receiving any longitude or latitude data, this could mean that your GPS has not found a lock on its location. Your GPS module's LED will blink when it has found a lock.

Now, if your robot is working, you can move on to the final step of the engineering process.

Finished Prototype

Well, if everything is working, you have a finished prototype ready to deliver to the customer. Figure 10-33 shows the finished prototype.

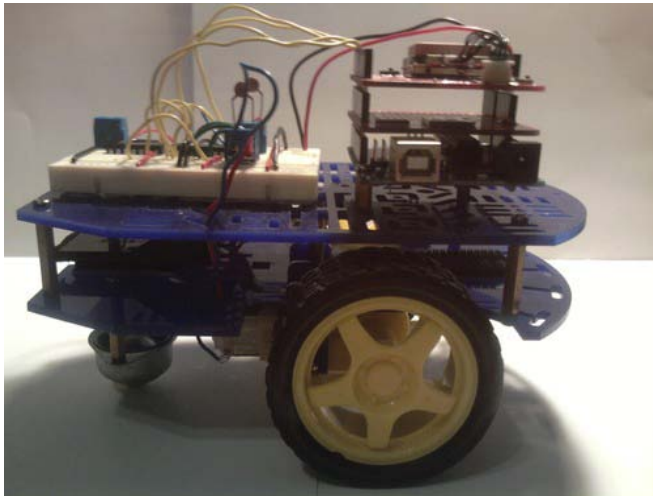


Figure 10-33. The finished prototype for this project

Summary

We first learned about the LabVIEW environment, including the Front Panel and Block Diagram. Then, we went through an introduction of the various functions that LabVIEW has to offer, such as Serial Functions, Numeric Functions, and Structures. Then, we went through a project that used LabVIEW to interface with the Xbox controller. We used the engineering process to help us configure the hardware and write the software for this project. You can make this project more robust by sending the latitude and longitude data to the LabVIEW application, instead of storing it on a microSD card, or by adding a string indicator that will tell you which direction your robot is moving. It is now up to you have fun tinkering with this robot.

Controlling Your Project: Bluetooth Arduino

In this chapter, we will build a robot that we can control wirelessly, using a Bluetooth shield that we'll also create in this chapter. We will continue to use the LabVIEW software that we created in the previous chapter, with a few minor tweaks. However, the Arduino software will change substantially, as we will not be using the GPS shield or microSD shield. We will not be doing any preliminary projects for this chapter, as we have worked with the Bluetooth Mate Silver from SparkFun before. Instead, we get right to the customer's requested project. Of course, after all that's done, we will troubleshoot any issues that the hardware configuration might have encountered. Then, we will debug any issues that occurred with that the LabVIEW and Arduino software. We should then have a finished prototype, ready for the customer to use.

Gathering Requirements and Creating the Requirements Document

The customer would like a robot that they can control wirelessly with an Xbox controller, using the LabVIEW software, created in the previous chapter, and the Arduino. The only new requirements are that the robot needs to use the Bluetooth Mate Silver, and the Bluetooth Mate Silver needs to be mounted on an Arduino shield for easy connection and disconnection. Again, we will not use the microSD shield or the GPS shield for this project, as the customer wants a wireless robot (using Bluetooth) that they can control using the Xbox controller.

The LabVIEW software will need to use a baud rate of 115200 because the Bluetooth Mate Silver is configured at this rate. The LabVIEW software will also use a different SubVI that no longer has the functionality to send "G" to the serial port. The customer wants a control added to the LabVIEW software to control the baud rate. The customer also wants any code associated with the GPS and microSD shield removed to free up memory resources on the Arduino. The robot should still use the same chassis from the previous project. The customer needs the robot to be powered by two 9V batteries. The robot needs to operate for 15 to 30 minutes (this will depend on the terrain).

Now that we have the notes for this project, we can compile them into a requirements document that will help us stay focused on the tasks for this project. The first section will discuss the hardware requirements for this project. Figure 11-1 shows the hardware used in this project, excluding the 9V batteries and the 9V battery connectors.

Gathering Hardware

Figure 11-1 shows some of the hardware being used in this project. (Not pictured: chassis from Chapter 4, two 9V batteries, two 9V battery connectors.)

- Arduino Deumilanove (or UNO)
- Bluetooth Mate Silver
- Chassis and motor driver from previous project
- DIY shield for Arduino from Adafruit
- 6-pin female stackable header
- Two 9V batteries
- Two 9V connectors (see the final project in Chapter 7 final project for more details)
- Extra wire

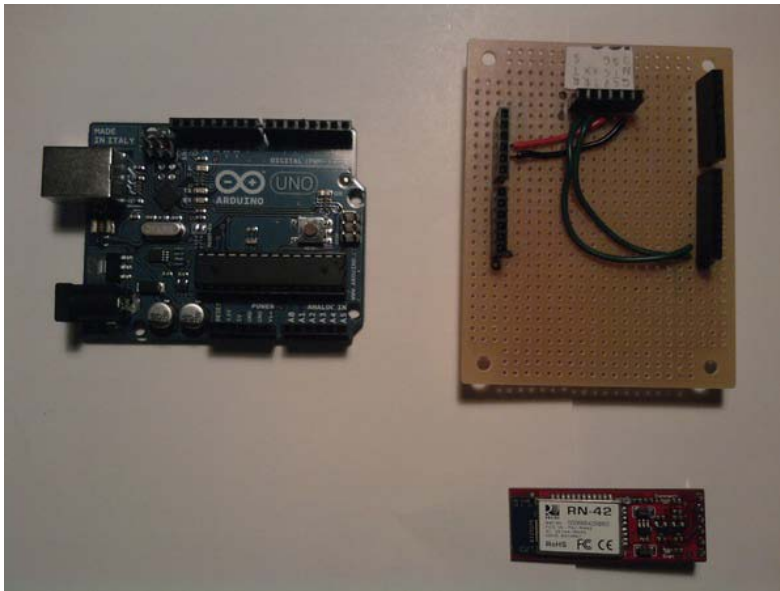


Figure 11-1. Hardware for this project

■ **Note** The DIY shield (Bluetooth shield) will be created in the “Configuration the Hardware” section of this chapter.

Gathering Software

The following list explains the software requirements for this project:

- Modify the LabVIEW software to use 115200 baud rate instead of 9600. The customer wants this to be a control rather than a constant.
- The SubVI will no longer have the functionality to send a “G” character. A new SubVI will take its place with the modifications.
- Modify the Arduino software so that it no longer uses any code associated with the GPS shield or the microSD shield.

Now that we have the hardware and software requirements, we can create the flowchart, shown in Figure 11-2, which will help guide us through the software portion of this project.

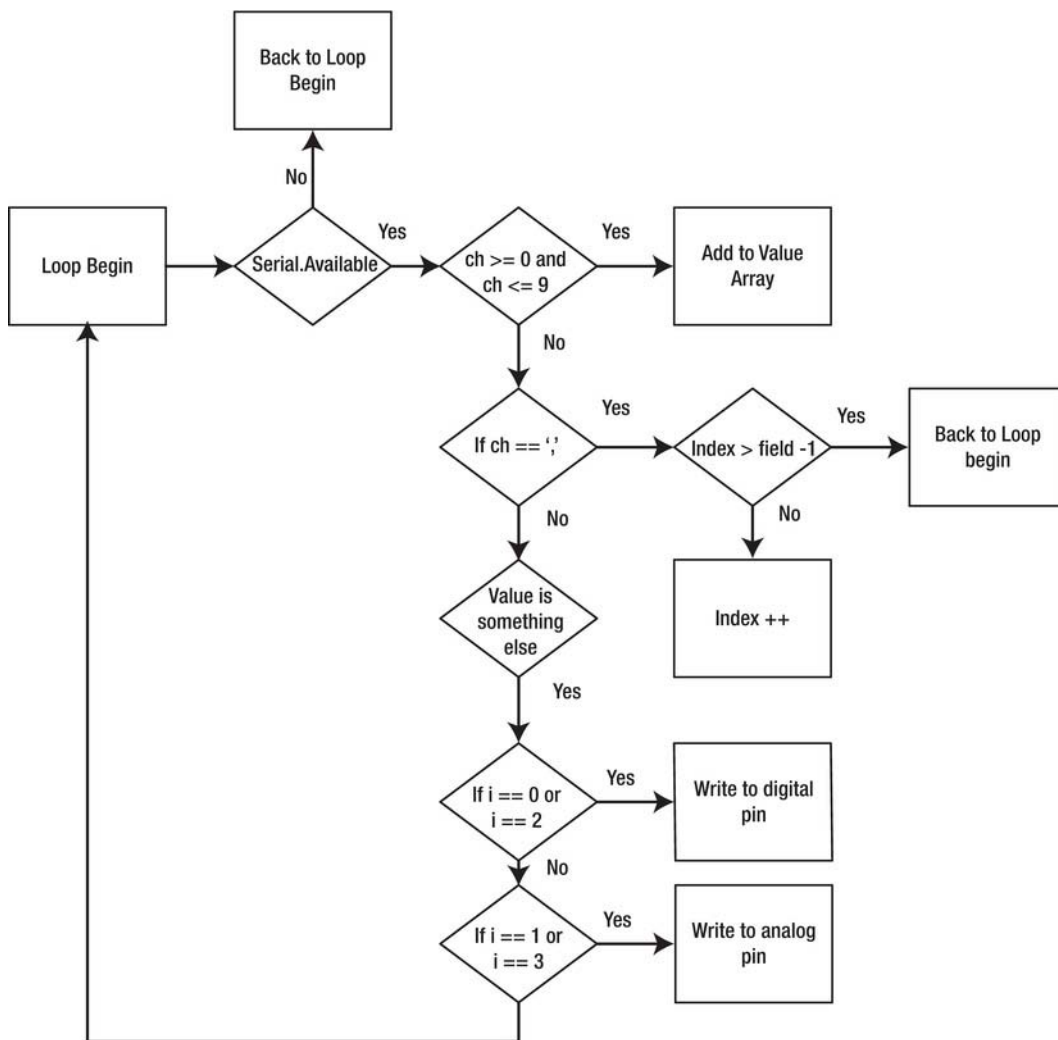


Figure 11-2. Flowchart for this project

Next, let's move on to configuring the hardware for this project.

Configuring the Hardware

The first thing we will do is create the Bluetooth shield using the DIY shield from Adafruit. This kit includes one special 8-pin female stackable header; it is special because all of its pins are slanted so that you can use it with the Arduino (you could also use the Proto shield from SparkFun for this project).

Begin by putting all of the stackable headers onto the DIY shield so that you can determine where the headers should be. To do this, put the special 8-pin female stackable header onto the DIY shield and

attach it to the Arduino. Now you should be able to put the rest of the headers on. Then, use a marker to mark where your headers are located. Figure 11-3 shows this process.

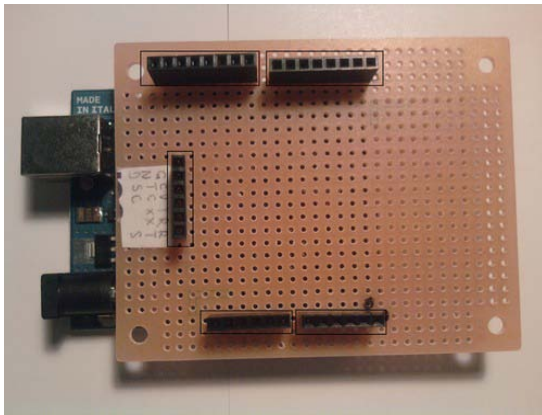


Figure 11-3. You've added the headers, marked them, and are now ready to solder.

■ **Note** Make sure that the special 8-pin female stackable header has its pins facing toward the other 8-pin stackable header; otherwise, the shield will not connect correctly to the Arduino. Figure 11-4 shows the directions the pins should face on the special 8-pin stackable header.

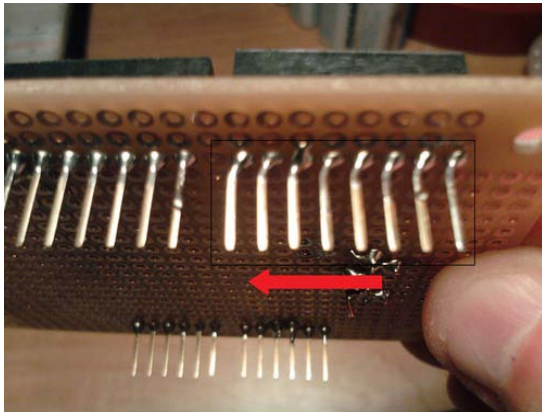


Figure 11-4. Make sure that the special 8-pin female stackable header is pointing toward the other 8-pin header.

Soldering the Headers

Now that you've finalized the location for the headers, you can solder them to the DIY shield. Figure 11-5 shows this process.



Figure 11-5. Solder on the headers that will connect to the Arduino.

Next, you need to solder on the 6-pin stackable header that will allow you to connect the Bluetooth Mate Silver to the Bluetooth shield. After you have the 6-pin header soldered on, you will need to cut the extra pin length off; otherwise, the header will touch the Arduino, and that can have severe consequences. Figures 11-6 and 11-7 show this process.

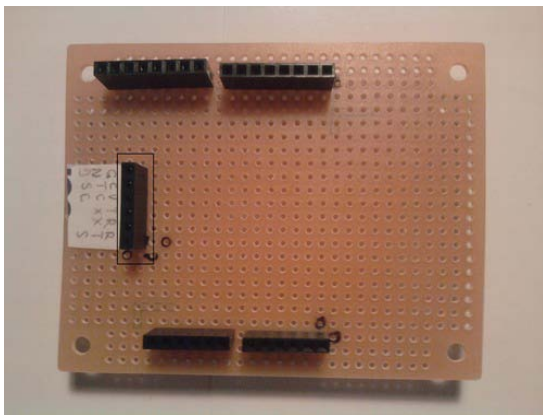


Figure 11-6. Solder on the 6-pin header that will allow the Bluetooth Mate Silver to attach to the Bluetooth shield.

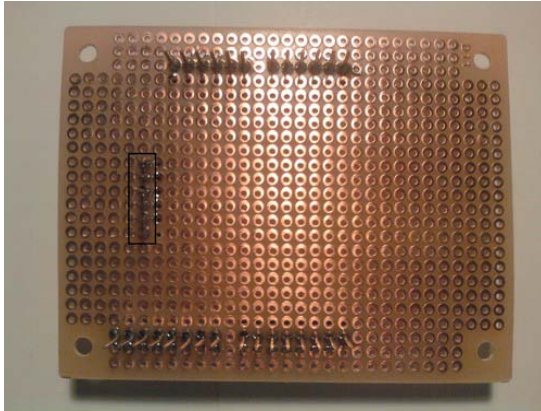


Figure 11-7. Solder the 6-pin header.

Now that we have connected all of the headers, let's move ahead and work on the pins and more.

Pins and Beyond

It might be a good idea to make sure you can attach the Bluetooth shield to the Arduino, so gently place the Bluetooth shield onto the Arduino. If it fits, that's great. If it doesn't, you might need to make sure that the pins are not slanting. After you have made sure that the Bluetooth shield can attach to the Arduino, take it off the Arduino. Then, solder on the ground pin of the Bluetooth Mate Silver header to the ground pin on the Bluetooth shield, and solder the CTS and RTS pins to one another. Figures 11-8 through 11-10 show this process.

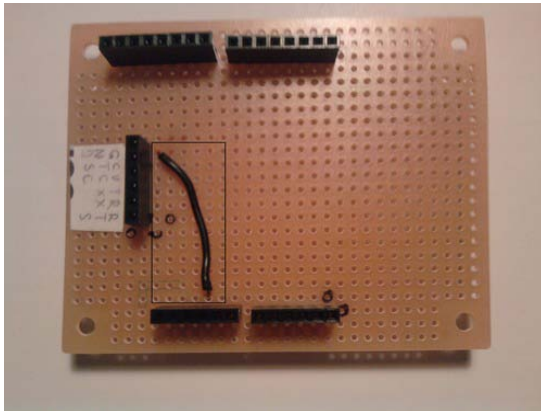


Figure 11-8. Solder the ground wire from the 6-pin header to the ground on the Bluetooth shield.



Figure 11-9. Solder the CTS and RTS pins together from the 6-pin header.

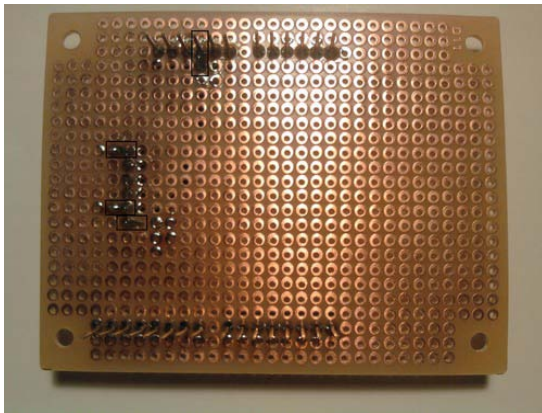


Figure 11-10. Solder the ground from the 6-pin header to the ground on the Bluetooth shield, and then solder the CTS and RTS together from the 6-pin header.

Next, you need to solder the VCC pin of the Bluetooth Mate Silver header to the power (+5V) on the Bluetooth shield. Figures 11-11 and 11-12 show this process.

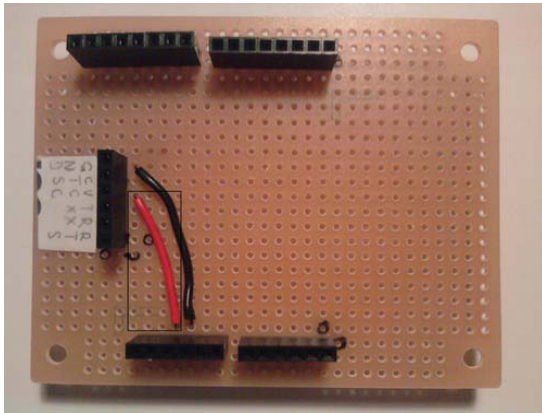


Figure 11-11. Solder the VCC pin on the 6-pin header to the +5V pin on the Bluetooth shield.

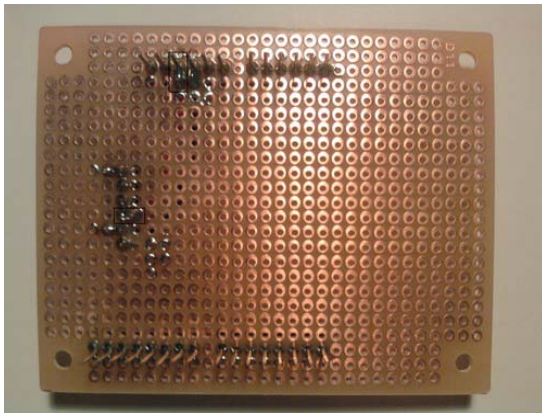


Figure 11-12. Solder the VCC wire.

Now you need to solder the TX pin on the Bluetooth Mate Silver header to the RX pin on the Bluetooth shield. Then, solder the RX pin on the Bluetooth Mate Silver header to the TX pin on the Bluetooth shield. Figures 11-13 and 11-14 show this process.

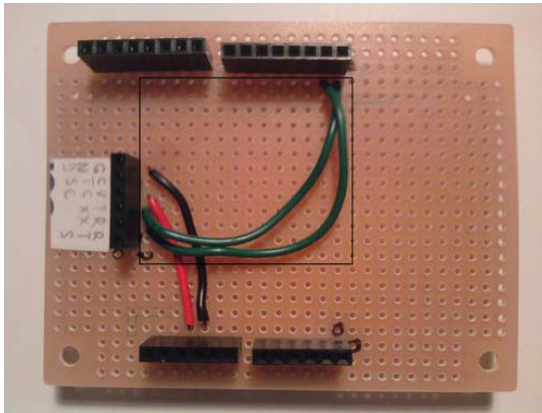


Figure 11-13. Solder the RX pin from the 6-pin header to the TX pin on the Bluetooth shield. Then, solder the TX pin from the 6-pin header to the RX pin on the Bluetooth shield.

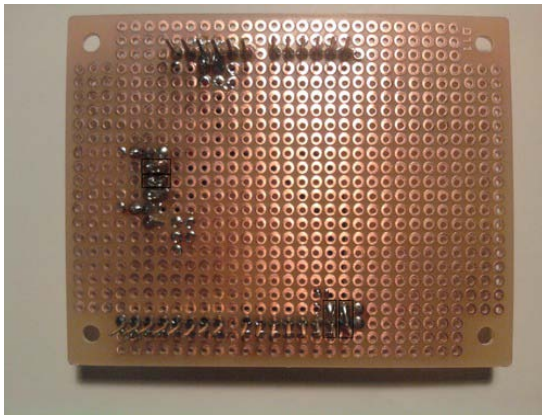


Figure 11-14. Solder the RX and TX wires.

Next, you need to add the Bluetooth Mate Silver to the 6-pin header. That should do it; you've now finished the Bluetooth shield. We will not connect the Bluetooth shield to the Arduino until we upload the software. Figure 11-15 shows the Bluetooth Mate Silver attached to the Bluetooth shield.

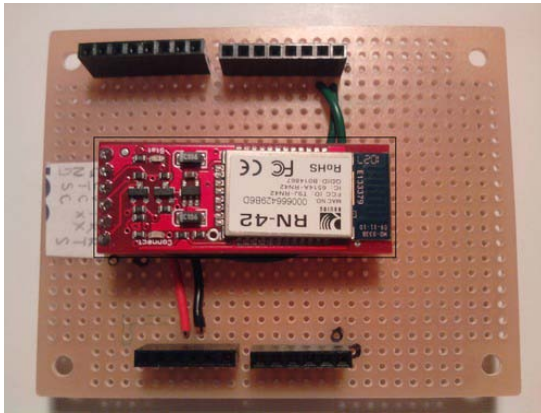


Figure 11-15. Attach the Bluetooth Mate Silver to the Bluetooth shield.

Configuring the Chassis and Arduino

Now that we have finished creating the Bluetooth shield, we can complete the configuration of the chassis and Arduino. To complete the partial hardware configuration, attach the Arduino to the chassis. We only need to do a partial hardware configuration because we cannot add the Bluetooth shield that we created in the previous section, as it will conflict with the serial port that uploads the code to the Arduino. Figure 11-16 shows the partial hardware configuration.

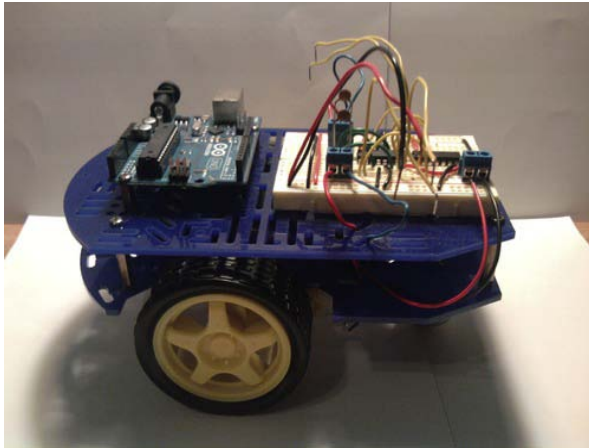


Figure 11-16. Partial hardware configuration for this project

Now that we have completed the Bluetooth shield, and partially configured the hardware, we can create the software that will run on the Arduino and the software that will allow us to integrate the Bluetooth Mate Silver and the Xbox controller. In the next section, we will discuss writing the software.

Writing the Software

We will first work on the LabVIEW software for this project. Open `ChapterTenFinalProject.vi`, and the Front Panel should open.

Reviewing the LabVIEW Software

Open the Block Diagram by going to `Window ► Show Block Diagram`. The first modification that you need to complete is to replace `ScaleandControl.vi` with `ScaleandControlChapter11.vi`; this SubVI no longer uses “button A” to send the “G” command to the serial port. To do this, first left-click `ScaleandControl.vi` in the Block Diagram, then press the Delete key. The LabVIEW software should have a broken arrow and some broken wires. Figure 11-17 shows the Block Diagram with no `ScaleandControl.vi`.

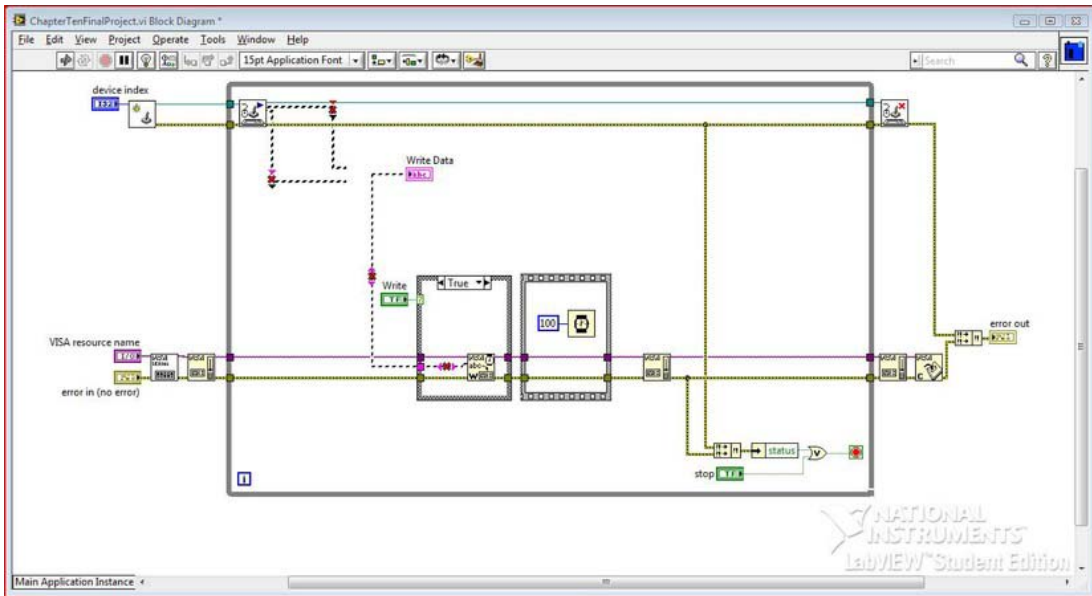


Figure 11-17. Delete the `ScaleandControl.vi` from the Block Diagram.

Now you need to get rid of the broken wires in the Block Diagram. To do this, press `Ctrl+B`. Figure 11-18 shows this process.

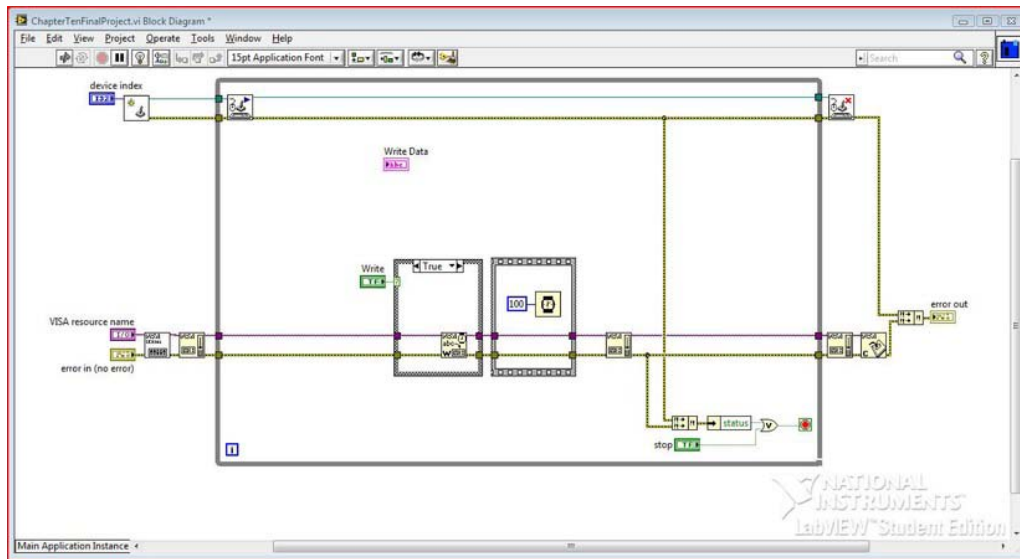


Figure 11-18. Press Ctrl+B to delete the broken wires.

Now we need to add `ScaleandControlChapter11.vi` to the Block Diagram and connect it to the appropriate controls and indicators. First, connect axis info terminal from the Acquire Input Data function to the axis info terminal on `ScaleandControlChapter11.vi`. Next, connect the String terminal on `ScaleandControlChapter11.vi` to the Write Buffer terminal on the VISA Write function. Figure 11-19 shows this process.

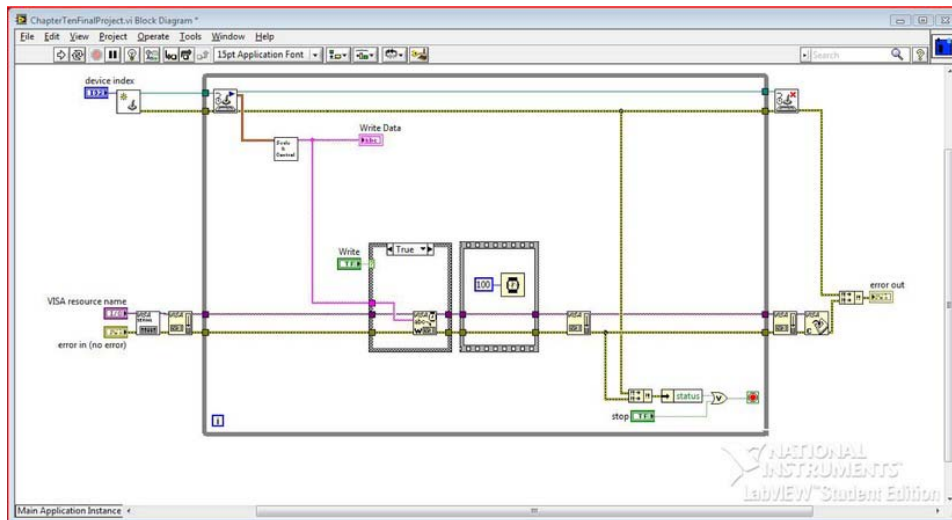


Figure 11-19. Connect the terminals of `ScaleandControlChapter11.vi` to the appropriate areas.

Finally, we need to add the baud rate control to the Front Panel. To do this, go to the Block Diagram and right-click the VISA Configure Serial Port's baud rate terminal; a pop-up menu should appear. Go to Create ► Control to add a control to the Block Diagram and Front Panel. Go to the Front Panel and move the baud rate control to an appropriate position. Figures 11-20 and 11-21 show the completed LabVIEW software for this project.

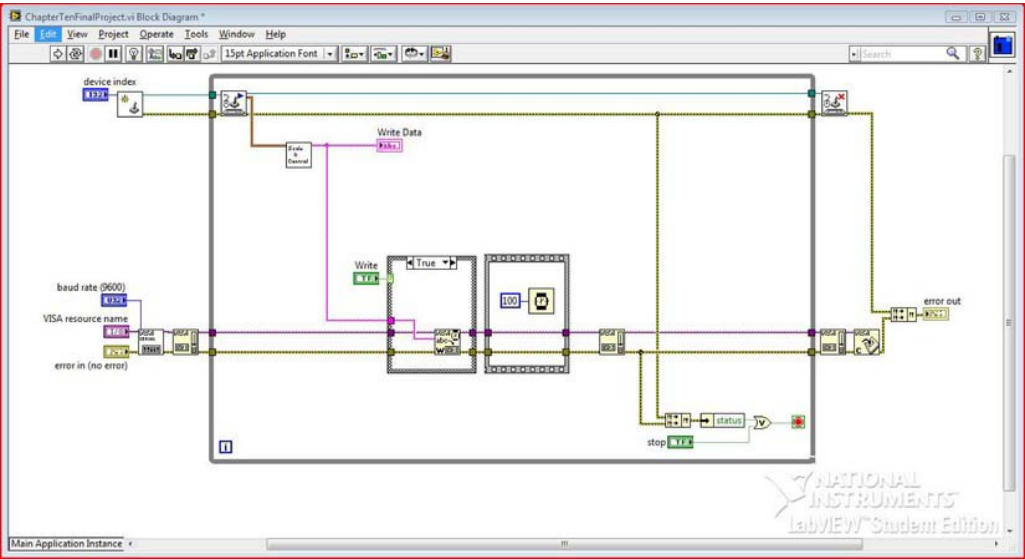


Figure 11-20. Add the baud rate control to the Block Diagram and the Front Panel.

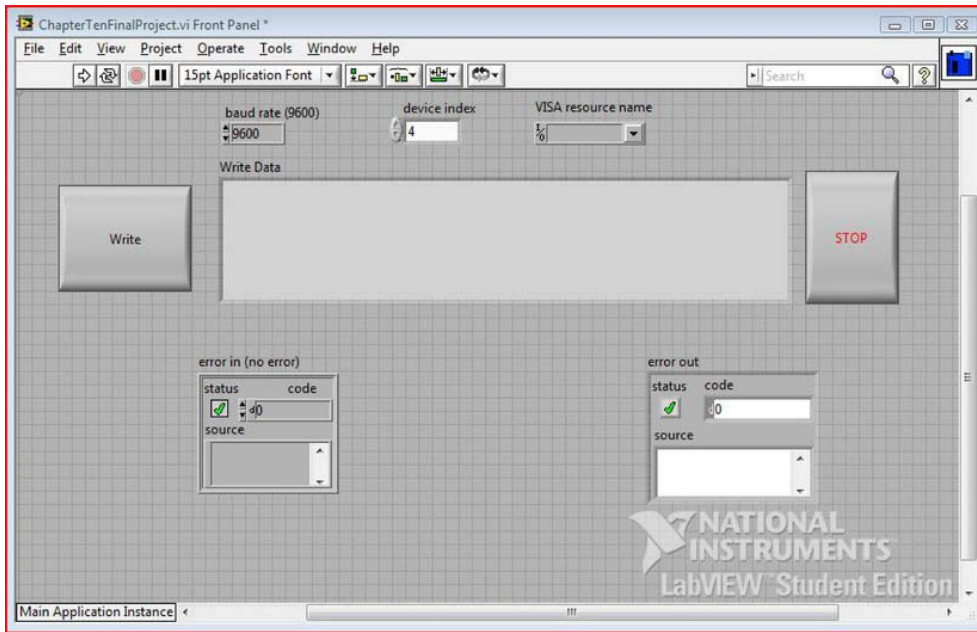


Figure 11-21. Completed LabVIEW application

Reviewing the Arduino Software

Now that we have completed the LabVIEW software, we need to take a look at the Arduino software for this project. Listing 11-1 shows the code that we will upload to the Arduino.

Listing 11-1. Code for This Project

```
const int fields = 4; // how many fields are there? right now 4
int motorPins[] = {4,5,7,6}; // Motor Pins
int index = 0; // the current field being received
int values[fields]; // array holding values for all the fields

void setup()
{
  Serial.begin(115200); // Initialize serial port to send and receive at 115200 baud

  for (int i; i <= 3; i++)          // set motors pinMode to output
  {
    pinMode(motorPins[i], OUTPUT);
    digitalWrite(motorPins[i], LOW);
  }
}

void loop()
```

```

{

  if( Serial.available())
  {
    char ch = Serial.read();

    if(ch >= '0' && ch <= '9') // If the value is a number 0 to 9
    {
      // add to the value array
      values[index] = (values[index] * 10) + (ch - '0');
    }
    else if (ch == ',') // if it is a comma
    {
      if(index < fields -1) // If index is less than 4 - 1...
        index++; // increment index
    }
    else
    {
      for(int i=0; i <= index; i++)
      {
        if (i == 0 || i == 2) // If the index is equal to 0 or 2
        {
          digitalWrite(motorPins[i], values[i]); // Write to the digital pin 1 or 0
          // depending what is sent to the arduino.
        }

        if (i == 1 || i == 3) // If the index is equal to 1 or 3
        {
          analogWrite(motorPins[i], values[i]); // Write to the PWM pins a number between
          // 0 and 255 or what the person has enter
          // in the serial monitor.
        }

        values[i] = 0; // set values equal to 0
      }

      index = 0;
      Serial.flush();
    }
  }
}

```

Now, let's examine the code in more detail.

First, we get rid of everything to do with the GPS shield and the microSD shield. Then, we add on a new piece of code right after the `index = 0;` (the one at the end of the code) statement. The

`Serial.Flush()` function flushes the buffer on the serial port, allowing for more data to be processed. That's it for the Arduino software.

Uploading the Software and Attaching the Bluetooth Shield

Now we need to upload this program to the Arduino. After you have uploaded the code, you can attach the Bluetooth shield to the Arduino (which should be attached to the chassis). Then, connect the power (+5V) and ground from the motor driver to the power (+5V) and ground on the Bluetooth shield. Next, connect pin 7 from the H-bridge to digital pin 4 on the Bluetooth shield. Then, connect pin 1 on the H-bridge to digital pin 5 on the Bluetooth shield. After that, connect pin 9 on the H-bridge to digital pin 6 on the Bluetooth shield. Finally, connect pin 10 on the H-bridge to digital pin 7 on the Bluetooth shield. The motor driver was created in Chapter 4. Figure 11-22 shows the completed hardware configuration for this project.

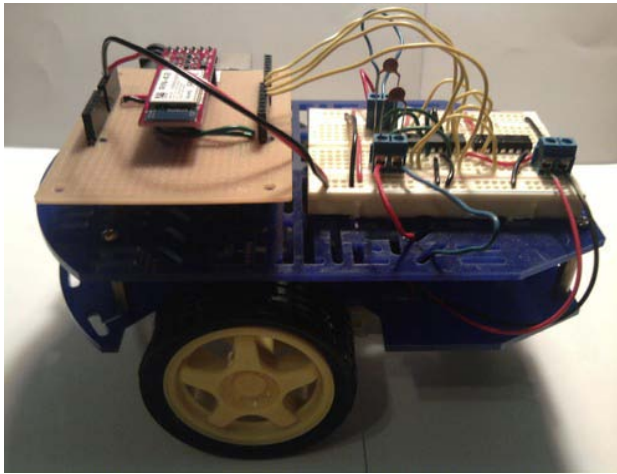


Figure 11-22. Completed hardware configuration for this chapter

■ **Note** Make sure you use new 9V batteries.

Operating the Robot

Now you should be ready to operate the Bluetooth robot. There is a particular sequence that you should follow; otherwise, you could run into issues with the robot. Here is the sequence:

1. Plug in the Xbox controller to one of your USB ports.
2. Attach one 9V battery to the motor driver and one 9V battery to the Arduino.
3. Open `ChapterTenFinalProject.vi` that we modified in this chapter, and the Front Panel should appear.

4. Make sure that the baud rate control is set to 115200, the correct COM port is selected, and the correct device index is selected for the Xbox controller (mine is set to 4).
5. Start the LabVIEW application by clicking the white arrow.
6. Wait until the Bluetooth Mate Silver's LED is solid (not blinking) before you press the Write button on the Front Panel.
7. You should now be able to control the Bluetooth robot with the Xbox controller.

There are also shutdown procedures that you should use. They are as follows:

1. Press the Write button on the Front Panel (you should no longer be able to control the Bluetooth robot).
2. Press the Stop button on the Front Panel, and the program should stop.
3. Remove the batteries from the Bluetooth robot.
4. Disconnect the Xbox controller from the computer.

Now that we have completed the hardware configuration, and we have completed writing the software, we can move on to debugging the software for this project.

Debugging the Software

As long as you followed the configuration of the LabVIEW software correctly, you shouldn't have any issues with it. If you do have problems, it is more than likely that one of the configuration controls (baud rate, COM port, and device index) is set to the wrong setting (you might have to mess around with the device index in order to get it working correctly). The Arduino software should also be working, but if it is not, make sure that you have removed all of the code associated with the GPS shield and microSD shield. You should now have working software.

Now that we have debugged the software, we can move on to troubleshooting the hardware for this project.

■ **Note** It might also be a good idea to make sure you are following the correct startup procedures as explained in the “Writing the Software” section of this chapter.

Troubleshooting the Hardware

Because we made our own shield, we might have had a few issues with the Bluetooth shield. The first thing you should do is check for continuity of your circuit. For example, if you are having continuity issues with your RX pin (that is, you do not think it is soldered properly and is not connecting from the RX pin on the Bluetooth Mate Silver header to the TX pin on the Arduino), you should put a wire in the RX pin of the Bluetooth Mate Silver header and put another wire in the TX pin of the Arduino. Then, you

can use a multimeter to tell whether the two connections have continuity. Figure 11-23 shows this process.

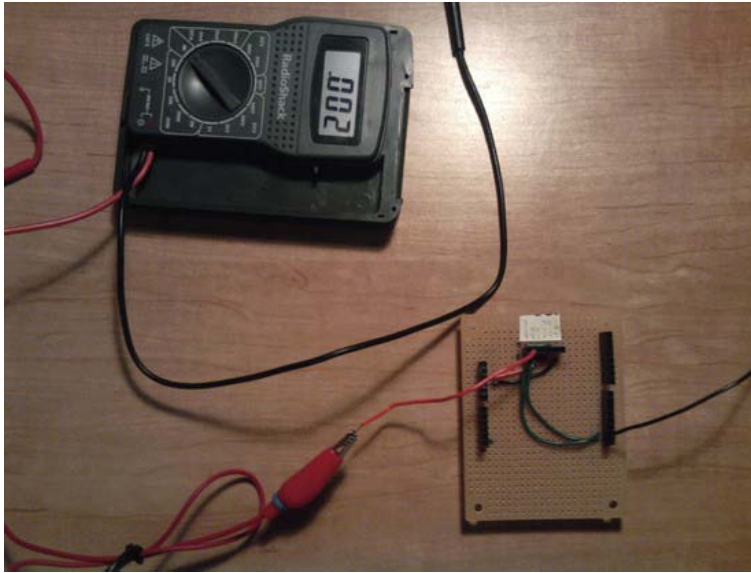


Figure 11-23. Checking whether two terminals are connected; if the multimeter displays “OL,” then the terminals are not connected, which could be an issue.

If you are still having problems, it might be a good idea to make sure that some of your pins are not connecting to the wrong pins; you can do this with the same continuity test, but instead of seeing some resistance (ohm), you should see “OL” (Open Loop, or Over Limit). (For more information on your multimeter, please consult your multimeter user manual.) This means that your terminals are not connected, which is a good thing if you are making sure pins are not touching. Figure 11-24 shows this process.

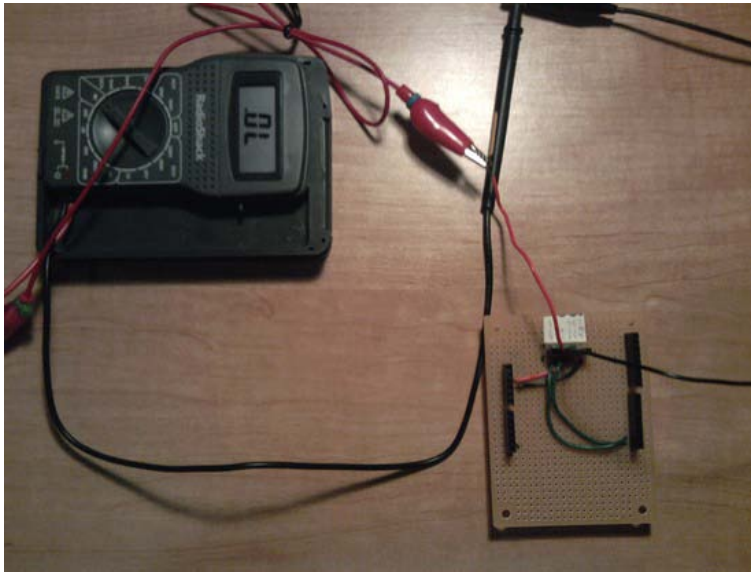


Figure 11-24. These two terminals are supposed to be disconnected, which is why the multimeter reads “OL”; if it were to read a number, then those two terminals are connecting, which can be an issue.

If everything is working, and you can drive your robot wirelessly, then you are ready to move on to the finished prototype.

Finished Prototype

Now that we have completed the debugging of this project, we can give the customer the finished prototype. Figure 11-25 shows the finished prototype.

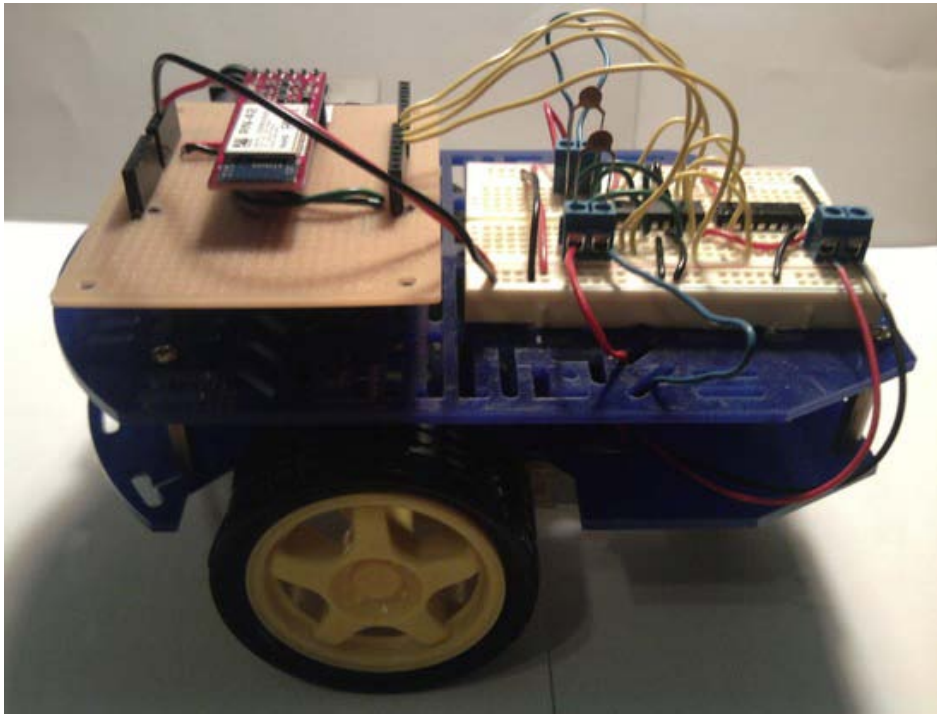


Figure 11-25. Finished prototype

Summary

You should now be familiar with the engineering process, as it relates to the Arduino. You can use this process in any project and can also modify it to make it even more robust. Throughout this chapter and this book, we have been using various pieces of hardware to create many different gadgets and robots. For this chapter, we made a robot that we could control wirelessly from a computer with the Xbox controller.

I hope that this book illustrated the engineering process well enough for you to understand how you can implement it in all of your projects. If you want to expand on this project, here are a few things you could do:

- Add GPS location to the robot.
- Add a battery-checking device.
- Make the operating area a bit larger with a ZigBee (there is a kit that you can purchase from SparkFun).
- Make this robot both controlled by the user and autonomous (see the final project in Chapter 7 for information on autonomous vehicles).

I hope you enjoyed reading this book as much as I did writing it, and that the knowledge you gained from the projects we developed will help you in your own future plans.

APPENDIX A

Hardware and Tools

This Appendix shows you what hardware and tools you will need for each chapter of this book. The Miscellaneous section has hardware that you will need in most chapters of this book. This Appendix will tell you the product number, company that makes the product, website of the company, the cost of the product (subject to change), and a brief description of the product. Here is the hardware you will need for this book:

Part Number	Company	Website	Cost (\$)	Description
Miscellaneous Hardware				
				Pushbutton Normally- Closed Momentary Switch (4-pack)
275-1548	Radio Shack	www.radioshack.com	3.69	
278-1224	Radio Shack	www.radioshack.com	7.39	22AWG Hookup wire
271-312	Radio Shack	www.radioshack.com	9.99	500 resistors
275-406	Radio Shack	www.radioshack.com	2 X 3.19	SPST slide switch
PRT-09518	Sparkfun	www.sparkfun.com	2.95	9v to Barrel jack adapter
PRT-10265	Sparkfun	www.sparkfun.com	9.95	9v LI-Ion battery charger
PRT-10053	Sparkfun	www.sparkfun.com	2 X 4.95	9v Li-ion battery
270-324	Radio Shack	www.radioshack.com	2.99	9v snap connectors
Chapter 1 Hardware				
DEV-10812	Sparkfun	www.sparkfun.com	49.99	Arduino and LabVIEW bundle
COM-09592	Sparkfun	www.sparkfun.com	0.35	Green LED
Chapter 3 Hardware				

DEV-09815	Sparkfun	www.sparkfun.com	24.95	Motor driver shield
PRT-10007	Sparkfun	www.sparkfun.com	8 X 1.50	Stackable Header kits
RB-Dag-47	Robot shop	http://www.robotshop.com/store	27 C	hassis for robot
Chapter 4 Hardware				
LCD-00709	Sparkfun	www.sparkfun.com	15.95	White on Black LCD
LCD-09363	Sparkfun	www.sparkfun.com	39.95	Color LCD shield
497-2936-5-ND	DigiKey	www.digiKey.com	4.7	H-bridge
296-3542-5-ND	DigiKey	www.digiKey.com	0.60	Hex Inverter
COM-08653	Sparkfun	www.sparkfun.com	3.95	Keypad
276-003	Radio Shack	www.radioshack.com	8.99	medium solderless breadboard
276-002	Radio Shack	www.radioshack.com	14.99	large solderless breadboard
276-1388	Radio Shack	www.radioshack.com	3 X 2.39	Terminal blocks
N/A	N/A	N/A	N/A	.01microF ceramic capacitors
Chapter 5 Hardware				
GPS-00465	Sparkfun	www.sparkfun.com	59.95	GPS Module
GPS-10710	Sparkfun	www.sparkfun.com	16.95	GPS Shield
DEV-09802	Sparkfun	www.sparkfun.com	14.95	microSD Shield
N/A	N/A	N/A	9.99	512mb microSD card
Chapter 6 Hardware				
WRL-10393	Sparkfun	www.sparkfun.com	39.95	Bluetooth Mate Silver
PRT-00553	Sparkfun	www.sparkfun.com	4 X 1.95	Male Headers
BOB-08745	Sparkfun	www.sparkfun.com	1.95	Logic Level Converter
SEN-10167	Sparkfun	www.sparkfun.com	9.95	Humidity and temp sensor
SEN-09375	Sparkfun	www.sparkfun.com	6.95	Force Sensitive Resistor
SEN-09418	Sparkfun	www.sparkfun.com	5.95	Digital Temp Sensor

SEN-10289	Sparkfun	www.sparkfun.com	1.95	Tilt sensor
SEN-10264	Sparkfun	www.sparkfun.com	7.95	Flex Sensor
PRT-07916	Sparkfun	www.sparkfun.com	3.95	Mini proto board
SEN-09088	Sparkfun	www.sparkfun.com	2 X 1.50	Photoresistor
270-1805	Radio Shack	www.radioshack.com	3.99	Medium Enclosure
Chapter 7 Hardware				
910-28015A	parallax	www.parallax.com	39.99	Ping))) with bracket kit
COM-07950	Sparkfun	www.sparkfun.com	1.95	Buzzer
276-003	Radio Shack	www.radioshack.com	8.99	Medium size breadboard
270-326	Radio Shack	www.radioshack.com	2 X 1.19	9v battery holder
Chapter 8 Hardware				
276-033	Radio Shack	www.radioshack.com	10.19	Passive infrared sensor
270-1807	Radio Shack	www.radioshack.com	7.39	Large enclosure
270-1805	Radio Shack	www.radioshack.com	3.99	Medium enclosure
Chapter 9 Hardware				
CEL-09607	Sparkfun	www.sparkfun.com	99.95	GSM Shield
CEL-00675	Sparkfun	www.sparkfun.com	7.95	Antenna
TOL-00298	Sparkfun	www.sparkfun.com	5.95	Wall adapter for Arduino
N/A	CVS	www.cvs.com	19.99	Cheap T-Mobile prepaid phone
Chapter 10 Hardware				
N/A	Gamestop	www.gamestop.com	29.99-39.99	Wired Xbox controller
Chapter 11 Hardware				
187	Adafruit	www.adafruit.com	6.00	DIY Shield

Tools for this
Book

64-083	Radio Shack	www.radioshack.com	17.99	Wire Strippers
TOL-10029	Sparkfun	www.sparkfun.com	3.95	Wire cutters
TOL-09146	Sparkfun	www.sparkfun.com	0.95	Flat-head screw driver
TOL-00080	Sparkfun	www.sparkfun.com	3.95	Phillips screw driver set
TOL-00082	Sparkfun	www.sparkfun.com	3.95	Soldering Vacuum
63-194	Radio Shack	www.radioshack.com	9.99	Lighted magnifier
64-2071	Radio Shack	www.radioshack.com	10.99	Soldering iron
64-002	Radio Shack	www.radioshack.com	4 X 3.89	Solder
N/A	Dremel	www.dremel.com	N/A	

Index

■ A

Alarm system

- Arduino software, 212
- debugging, 214
- finished prototype, 214–215
- passive infrared (PIR) sensor, 197–198
- requirements document gathering and creation
 - hardware, 207–208
 - hardware configuration, 209–212
 - software writing, 208–209, 212–214
- security (door alarm) (*see* Door alarm system)
- troubleshooting, 214

Arduino engineering process

- configuring hardware, 10–12
- creating requirements document, 9–10
- debugging software, 13–14
- finished prototype, 14
- gathering hardware, 10
- hardware components
 - Arduino Duemilanove or UNO, 2
 - Arduino shields, 4–5
 - ArduinoBT or Bluetooth Mate Silver, 2–3
 - miscellaneous components, 7
 - sensors, 5–6
 - servos and motors, 6–7
 - solderless breadboard, 3–4
 - wire, 4
- tools, 8, 9
- troubleshooting hardware, 14
- writing software, 12–13

Arduino shields, 4

Arduino software

- analog communication commands, 21
- creating basic Arduino program, 25–26
- digital communication commands, 20–21
- libraries
 - ColorLCDShield Library, 24
 - NewSoftwareSerial, 24

- TinyGPS, 24
- programming components
 - conditional statements, 18–19
 - loops, 19–20
 - setup() and loop(), 15–16
 - variables, 16–17
- serial communication commands, 21–23

Arrays, 17

AT commands. *See* Attention (AT) commands

attach(), 168

Attention (AT) commands, 218–219

Automated robot software, 186

Automation. *See* Robot perception

■ B

Block Diagram, 241–242

Bluetooth Mate Silver, 199, 202–204, 206, 208–211, 213, 214

Bluetooth shield

- attaching to chassis, 293
- attachment with Bluetooth Mate Silver, 287–288

chassis and Arduino configuration, 287

debugging, 294

finished prototype, 296–297

hardware configuration

headers soldering, 282–283

pins, 283–287

requirements document gathering and creating

hardware, 278

software requirements, 279–280

robot operation, 293–294

software

Arduino software, 291–293

LabVIEW Software, 288–291

uploading, 293

troubleshooting, 294–297

Buzzer, 167–168

■ C

Capacitors, [7](#)
 Case Structure, [244](#)
 Cellular Shield
 door alarm with SMS messaging, [225–230](#)
 hardware, [217–218](#)
 text message sending, [219–225](#)
 Color LCD shield, [61–62](#)
 slot machine creation, [76–80](#)
 ColorLCDShield library, [24](#), [66–67](#)
 Comparison Functions, [249–250](#)
 Concatenate string, [248](#)
 Conditional statements, [18](#)
 Control and instrumentation. *See* LabVIEW
 software
 Controls Palette, [241](#)

■ D

Dagu 2WD Beginner Robot Chassis V2, [28–29](#)
 Data logger
 hardware, [115](#)
 hardware configuration, [115–116](#)
 software writing, [116–119](#)
 DHT22 library, [138](#)
 DHT22 sensor
 definition, [137](#)
 hardware configuration, [153–154](#)
 hardware gathering, [152–153](#)
 software writing, [154](#)
 Digital level
 hardware configuration, [150–151](#)
 hardware gathering, [149–150](#)
 software writing, [151–152](#)
 Digital ruler
 hardware, [169](#)
 hardware configuration, [170](#)
 software writing, [170–172](#)
 Digital temperature and humidity sensor. *See*
 DHT22 sensor
 Digital temperature sensor (I2C), [137](#)
 Dikes, [8](#)
 Diodes, [7](#)
 DIY shield, [280](#)
 do . . . while loop, [20](#)
 Door alarm system
 hardware, [198–199](#)
 hardware configuration, [200–204](#)
 software writing, [204–207](#)

Door alarm with SMS messaging, [225–230](#)

■ E

Error messages and commands
 AT commands, [218–219](#)
 Cellular Shield, [217–218](#)
 debugging, [236](#)
 requirements document gathering and
 creating
 hardware, [230–231](#)
 hardware configuration, [232–233](#)
 software requirements, [231–232](#)
 software writing, [233–235](#)
 troubleshooting, [236](#)

■ F

Finished prototype, [237](#)
 Flex sensor
 definition, [136](#)
 hardware, [147](#)
 hardware configuration, [148](#)
 software writing, [148–149](#)
 Flower pot analyzer
 hardware configuration, [140–142](#)
 hardware gathering, [139](#)
 software writing, [143–144](#)
 for loop, [19](#)
 Force sensitive resistor (FSR) sensor
 definition, [136](#)
 hardware configuration, [145–146](#)
 hardware gathering, [144–145](#)
 software writing, [146–147](#)
 Front Panel, [240](#)
 Function prototype, [15](#)
 Functions Palette, [242–243](#)

■ G

GPS communication
 creating car finder
 hardware, [108–109](#)
 hardware configuration, [109–110](#)
 software writing, [110–114](#)
 data logger
 hardware, [115](#)
 hardware configuration, [115–116](#)
 software writing, [116–119](#)
 debugging, [129–130](#)

- finished prototype, 130
- hardware troubleshooting, 130
- libraries
 - SdFat library, 100–101
 - TinyGPS library, 99–100
- microSD Shield, 97–98
- NMEA Protocol, 98–99
- requirements document gathering and creating
 - hardware, 119–120
 - hardware configuration, 122–123
 - software writing, 120–121, 123–129
- writing GPS data to monochrome LCD
 - hardware, 104
 - hardware configuration, 105–106
 - software writing, 106–108
- writing raw GPS data to serial monitor
 - hardware, 101–102
 - hardware configuration, 102–103
 - software writing, 103–104
- GPS shields, 5
- GSM communication
 - door alarm with SMS messaging
 - hardware, 225–226
 - hardware configuration, 226–228
 - software writing, 229–230
 - text message sending
 - hardware, 219–220
 - hardware configuration, 220–224
 - software writing, 224–225
- GSM shields, 5
- GUI design, 258–259

■ H

- Hardware and tools, 299–302
- H-bridge, 27–28

■ I, J

- if statements, 18
- Input Device Control Functions, 252–253
- Inter-integrated circuits (I2C) digital
 - temperature sensor, 137

■ K

- Keypad usage
 - hardware configuration, 82
 - hardware gathering, 81

- software writing, 82–84

■ L

- LabVIEW software, 239, 288–291
 - debugging, 274
 - finished prototype, 275
 - functions
 - Case Structure, 244
 - Comparison Functions, 249–250
 - Input Device Control Functions, 252–253
 - Numerical Functions, 246–247
 - Sequence Structure, 245
 - Serial Functions, 250–252
 - String Functions, 247–249
 - While Loop, 244
 - hardware, 253–254
 - hardware configuration, 254–256
- LabVIEW environment
 - Block Diagram, 241–242
 - controls palette, 241
 - front panel, 240
 - Functions Palette, 242–243
 - Tools Palette, 243
- software requirements, 254
- troubleshooting, 275
- writing software
 - adding Merge Errors Function, 265–266
 - adding serial functions, 262–265
 - adding SubVI, 266–267
 - Arduino code uploading, 269–273
 - error handling, 267–269
 - GUI designing, 258–259
 - operation, 273–274
 - programming application, 259
 - steps, 256–258
 - While Loops condition, 265
- LCD shields, 5
- LCDs
 - color LCD shield configuration, 61–62
 - control
 - displaying multiple sensor values, 67–71
 - keypad usage to communicate, 81–84
 - menu creation, 71–76
 - slot machine creation, 76–80
 - customer's robot creation
 - hardware configuration, 87–91
 - hardware gathering, 85–86

LCDs, customer's robot creation (*cont.*)
 requirements document gathering and
 creation, 84–85
 software writing, 91–94
 debugging Arduino software, 95
 finished prototype, 96
 libraries
 ColorLCDShield library, 66–67
 LiquidCrystal library, 64–66
 monochrome, 62–64
 troubleshooting, 95
 LED, 7
 Libraries
 ColorLCDShield Library, 24
 NewSoftwareSerial, 24
 TinyGPS, 24
 LiquidCrystal library, 64–66
 loop(), 15
 Loops, 19

■ M

Magnifying glass, 8
 Merge Errors Function, 265
 microSD Shield, 97–98
 Monochrome LCD, 62–64
 hardware configuration, 72–74
 hardware gathering, 71–72
 software writing, 74–76
 Motor control
 multiple motor with Arduino
 hardware configuration, 39–41
 hardware gathering, 38–39
 software writing, 41–43
 speed and direction
 hardware configuration, 44–46
 hardware gathering, 43–44
 software writing, 46–47
 speed with potentiometer
 hardware configuration, 35
 hardware gathering, 35–37
 software writing, 37–38
 turning motor with switch
 hardware configuration, 30–33
 hardware gathering, 29–30
 software writing, 33–34
 with serial commands
 debugging Arduino Software, 55–58
 hardware configuration, 52–53
 hardware gathering, 49

software requirements, 50–51
 software writing, 53–55
 troubleshooting, 58

Motor shields, 5

Multimeter, 8

Multiple sensor values display
 hardware configuration, 68–70
 hardware gathering, 67–68
 software writing, 70–71

■ N

Needle-nose pliers, 8
 NewSoftwareSerial, 24
 NMEA Protocol, 98–99
 Numeric Palette, 246–247

■ O

Object alarm
 hardware, 172
 hardware configuration, 173
 software writing, 173–175
 Object detection. *See* Object alarm

■ P, Q

Parallax Ping))) bracket kit, 175
 Passive infrared (PIR) sensor, 197–198
 Photoresistor, 135
 Ping))) Parallax Ultrasonic Sensor, 165–166
 PIR sensor, 6
 printFloat() function, 236

■ R

Resistors, 7
 Robot engineering requirements
 finished prototype, 59–60
 hardware
 Dagu 2WD Beginner Robot Chassis V2,
 28–29
 H-bridge, 27–28
 motor control
 multiple motors with Arduino, 38–43
 speed and direction control, 43–47
 speed control with potentiometer, 34–38
 turning motor with switch, 29–34
 with serial commands, 47–58

Robot integration. *See* GPS communication

Robot perception

debugging, 190–193

digital ruler

hardware, 169

hardware configuration, 170

software writing, 170–172

finished prototype, 194

hardware

buzzer, 167–168

servos, 167

ultrasonic sensor, 165–166

object alarm

hardware, 172

hardware configuration, 173

software writing, 173–175

requirements document gathering and creating

hardware, 180–181

hardware configuration, 182–186

software writing, 181–182, 186–190

Servo Library, 168

solar controller

hardware, 175–176

hardware configuration, 176–178

software, 178–180

troubleshooting, 194

■ S

Scientific calculator, 8

SdFat library, 100–101

Security system. *See* Door alarm system

Sensors, 5

debugging, 163

DHT22 sensor

hardware configuration, 153–154

hardware gathering, 152–153

software writing, 154–156

digital level

hardware configuration, 150–151

hardware gathering, 149–150

software writing, 151–152

digital temperature and humidity sensor, 137

final prototype, 163

flex sensor

definition, 136

hardware, 147

hardware configuration, 148

software writing, 148–149

flower pot analyzer

hardware configuration, 140–142

hardware gathering, 139

software writing, 143–144

force sensitive resistor (FSR), 136, 144–147

inter-integrated circuits (I2C) digital temperature sensor, 137

libraries

DHT22 library, 138

Wire library, 138

photoresistor, 135

requirements documents gathering and creating

hardware configuration, 159–161

hardware gathering, 156–157

software requirements, 157–158

software writing, 162–163

tilt sensor, 135

troubleshooting, 163

weight detection sensor

hardware configuration, 145–146

hardware gathering, 144–145

software writing, 146–147

wireless temperature monitor, 156

Sequence Structure, 245

Serial Functions, 250–252, 262

Serial.available(), 22

Serial.begin(baud), 21

Serial.end(), 23

Serial.println(), 22

Serial.read(), 22

Serial.write(), 22

Servos, 167

Servos and motors, 6–7

setup(), 15

Slot machine creation

hardware configuration, 77–78

hardware gathering, 76–77

software, 78–80

Solar controller

hardware, 175–176

hardware configuration, 176–179

software writing, 178–181

Solder, 8

Soldering iron, 8

Sonar sensors, 6

Stacked Sequences, 245

String Functions, 247–249

SubVI, 266

Switch statement, 18

■ T

- Temperature sensor, 6
- Tilt sensor, 135
- TinyGPS, 24
- TinyGPS library, 99–100
- tone() function, 173
- Tools Palette, 243
- Transistors, 7

■ U

- Ultrasonic sensor, 165–166

■ V

- Variables, 16
- Voltage divider, 133–134

■ W

- Weight detection sensor
 - hardware configuration, 145–146
 - hardware gathering, 144–145
 - software writing, 146–147
- While Loop, 244
- While Loops Condition, 265
- Whitespacing, 17
- Wire Library, 138
- Wire stripper, 8
- write(), 168

■ X, Y, Z

- Xbox Controller. *See* LabVIEW software

Practical Arduino Engineering



Harold Timmis

Apress®

Practical Arduino Engineering

Copyright © 2011 by Harold Timmis

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-3885-0

ISBN-13 (electronic): 978-1-4302-3886-7

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights. *Practical Arduino Engineering* is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation. LabVIEW™ is a trademark of National Instruments. This publication is independent of National Instruments, which is not affiliated with the publisher or the author, and does not authorize, sponsor, endorse or otherwise approve this publication.

President and Publisher: Paul Manning

Lead Editor: James Markham

Technical Reviewers: Andreas Wischer, Coleman Sellers

Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick,

Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff

Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing,

Matt Wade, Tom Welsh

Coordinating Editor: Corbin Collins

Copy Editors: Heather Lang, Tracy Brown, Vanessa Moore, Tiffany Taylor

Compositor: Bytheway Publishing Services

Indexer: SPI Global

Artist: SPI Global

Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media, LLC., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at www.apress.com. You will need to answer questions pertaining to this book in order to successfully download the code.

To my wife and my family because you are all always there for me.

Contents

■ About the Author	xii
■ About the Technical Reviewer	xiii
■ Acknowledgments	xiv
■ Preface	xv
■ Chapter 1: The Process of Arduino Engineering	1
Gathering Your Hardware	1
Gathering Your Tools	8
Understanding the Engineering Process.....	9
Requirements Gathering.....	9
Creating the Requirements Document	9
Gathering the Hardware	10
Configuring the Hardware	10
Writing the Software.....	12
Debugging the Arduino Software	13
Troubleshooting the Hardware	14
Finished Prototype.....	14
Summary	14
■ Chapter 2: Understanding the Arduino Software	15
Getting Started with setup() and loop().....	15
Initializing Variables.....	16
Writing Conditional Statements	18

Working with Loops	19
Communicating Digitally	20
Communicating with Analog Components	21
Serial Communication	21
Using Arduino Libraries	23
NewSoftwareSerial	24
TinyGPS	24
ColorLCDShield Library	24
Putting Together the Arduino Language Basics	25
Summary	26
■ Chapter 3: Robot Engineering Requirements: Controlling Motion	27
Hardware Explained: The H-bridge	27
Gathering the Hardware for this Chapter	28
Understanding the Basics of Motor Control	29
Project 3-1: Turning on a Motor with a Switch	29
Project 3-2: Controlling the Speed of a Motor with a Potentiometer	34
Project 3-3: Controlling Multiple Motors with the Arduino	38
Project 3-4: Controlling Speed and Direction	43
Project 3-5: Controlling Motors with Serial Commands	47
Summary	60
■ Chapter 4: Adding Complexity to the Robot: Working with LCDs	61
Configuring a Color LCD Shield	61
Introducing the Monochrome and Color LCD Shields	62
Working with the LiquidCrystal and ColorLCDShield (Epson or Phillips) Libraries	64
Using the LiquidCrystal Library	64
ColorLCDShield Library	66

Exploring the Basics of LCD Control	67
Project 4-1: Displaying Multiple Sensor Values.	67
Project 4-2: Creating a Menu on the Monochrome LCD	71
Project 4-3: Creating a Slot Machine with the Color LCD Shield	76
Project 4-4: Using a Keypad to Communicate with the Color LCD	81
Project 4-5: Creating the Customer's Robot	84
Summary	96
■ Chapter 5: Robot Integration Engineering a GPS Module with the Arduino	97
Hardware Explained: microSD Shield	97
Understanding NMEA Protocol	98
Libraries Explained: TinyGPS and SdFat Libraries	99
TinyGPS	99
SdFat Library	100
The Basics of GPS Communication with the Arduino	101
Project 5-1: Writing Raw GPS Data to the Serial Monitor	101
Project 5-2: Writing GPS Data to a Monochrome LCD	104
Project 5-3: Creating a Car Finder	108
Project 5-4: GPS Data Logger	114
Requirements Gathering and Creating the Requirements Document	119
Hardware	119
Software	120
Summary	131
■ Chapter 6: Interlude: Home Engineering from Requirements to Implementation	133
Understanding the Voltage Divider	133
Hardware Explained: Sensors	134
Photoresistor	135

Flex Sensor	136
Force Sensitive Resistor (FSR)	136
Digital Temperature and Humidity Sensor	137
Digital Temperature Sensor (I2C)	137
Libraries Explained: Wire Library and DHT22 Library	138
Wire Library	138
DHT22 Library	138
Understanding the Basics of Sensors	139
Project 6-1: Flower Pot Analyzer	139
Project 6-2: Using a FSR Sensor	144
Project 6-3: Using a Flex Sensor	147
Project 6-4: Digital Level	149
Project 6-5: Using a DHT22 Sensor with a Monochrome LCD	152
Project 6-6: Wireless Temperature Monitor	156
Requirements Gathering and Creating the Requirements Document	156
Writing the Software	162
Troubleshooting the Hardware	163
Final Prototype	163
Summary	164
■ Chapter 7: Robot Perception: Object Detection with the Arduino	165
Hardware Explained: Ultrasonic Sensor, Servo, and Buzzer	165
Ultrasonic Sensor	165
Servo	167
Buzzer	167
Libraries Explained: The Servo Library	168
Basics of the Ultrasonic Sensor and the Servo	169
Project 7-1: Digital Ruler	169

Project 7-2: Object Alarm	172
Project 7-3: Solar Controller	175
Requirements Gathering and Creating the Requirements Document	180
Hardware	180
Software	181
Summary	195
■ Chapter 8: Mature Arduino Engineering: Making an Alarm System Using the Arduino	197
Basic Security System	198
Project 8-1: Door Alarm	198
Requirements Gathering and Creating the Requirements Document	207
■ Chapter 9: Error Messages and Commands: Using GSM Technology with Your Arduino	217
Hardware Explained: Cellular Shield	217
Understanding the AT Command Set	218
The Basics of GSM Communication	219
Project 9-1: Sending a Text Message	219
Project 9-2: Door Alarm with SMS Messaging	225
Requirements Gathering and Creating the Requirements Document	230
Summary	237
■ Chapter 10: Control and Instrumentation: The Xbox Controller and the LabVIEW Process	239
Introduction to the LabVIEW Environment	239
The Front Panel	240
The Controls Palette	241
The Block Diagram	241
The Functions Palette	242

The Tools Palette	243
LabVIEW Functions Explained	244
The While Loop	244
The Case Structure	245
The Sequence Structure	245
Numerical Functions	246
String Functions	247
Comparison Functions	249
Serial Functions	250
Input Device Control Functions	252
Gathering Requirements and Creating the Requirements Document	253
Getting Started	256
Designing the GUI	258
Programming the Application	259
Adding Serial Functions	262
Completing the While Loops Condition	265
Adding a Merge Errors Function	265
Adding a SubVI	266
Error Handling	267
Uploading the Code to the Arduino	269
Operation	273
Summary	276
■ Chapter 11: Controlling Your Project: Bluetooth Arduino	277
Gathering Requirements and Creating the Requirements Document	277
Configuring the Hardware	280
Soldering the Headers	282
Pins and Beyond	283
Configuring the Chassis and Arduino	287

Writing the Software.....	288
Reviewing the LabVIEW Software.....	288
Reviewing the Arduino Software	291
Uploading the Software and Attaching the Bluetooth Shield.....	293
Operating the Robot.....	293
Summary	297
■ Appendix A: Hardware and Tools.....	299

About the Author



■ **Harold Timmis**, since he was a small child, has fostered a curiosity for technology, taking apart everything in his parents' house just to see how it all worked. This fueled his thirst for knowledge of computer science, programming, and its uses. He has worked with LabVIEW and Arduino for the past three years. During that time, he has been involved in several professional projects using LabVIEW, as well as many hobbyist projects utilizing both Arduino and LabVIEW. Harold attended the Florida Institute of Technology, where he studied computer engineering and was introduced to LabVIEW and Arduino. Later, he worked at the Harris Corporation and General Electric, where he created several LabVIEW projects for trains and became very interested in the Arduino, data acquisition, and control theory.

About the Technical Reviewers



■ **Andreas Wischer** holds a German degree in electronics. Since 1997 he has worked as a software consultant across Europe. With a solid background in expert systems and C programming, he now spends most of his working time on Java and Lotus Notes. For the latter he's certified for both development and administration.



■ **Coleman Sellers** studied computer science at the Florida Institute of Technology where he received his Bachelor of Science degree. It was there he was first exposed to PIC embedded programming. His first job out of college was in Software Quality Assurance, on which he worked for a year. He currently works as a Win32 C programmer.

Acknowledgments

I would like to thank my beautiful wife Alexandria for being very patient with me for the past seven months while I wrote this book. I would also like to thank my amazing Aunt Sue for helping me correct the syntax of this book. I want to thank Mom, Dad, Daphne, sister Amanda, and brother George for always being there for me and helping me achieve my goals.

To the many editors at Apress, thank you for making my book clear and concise (Corbin Collins, James Markham, Michelle Lowman, Heather Lang, Tracy Brown, Vanessa Moore, and Tiffany Taylor).

I want to thank my tech reviewers, Coleman Sellers and Andreas Wischer, for reading my book and giving very detailed feedback to fix any glitches.

To the many suppliers: Sparkfun, Radio Shack, Adafruit, Digikey, and Robotshop—without you this book could not have been completed.

Also to the many developers that wrote the libraries in this book: Arduino Team, Mark Sproul, Peter Davenport, William Greiman, John Boxall, Coleman Sellers, Vlado Brajer (terminal program), nethoncho, and Mikal Hart. You guys/gals are the reason open source hardware and software exists.

And of course to the Arduino Team, what can I say? The Arduino is the greatest microcontroller I have ever been privileged enough to use. Thank you for everything you have done and still are doing to this day.

Preface

I wrote this book so that students, hobbyists, and engineers alike can take advantage of the Arduino platform by creating several projects that will teach them about the engineering process. I also wanted to guide the reader through introductory projects so that they could get a firm grasp on the Arduino Language, and how to incorporate several pieces of hardware to make their own projects.

This book offers so much information on the Arduino, not just the basic LED projects but it offers different peripherals such as Ultrasonic sensor, the Xbox® controller, Bluetooth, and much more. This book also teaches the non-engineer to follow a process that will help them in future project (not just Arduino projects).

This book is for anyone wanting to learn about the Arduino and the Engineering process can read this book. That includes but not limited to: students, hobbyists, and engineers. The reader will need some basic skills in breadboarding and soldering.

In this book you learn how to use various pieces of hardware with the Arduino. Examples: Bluetooth, Xbox controller, ultrasonic sensor, PIR sensor, several other sensors, GPS, microSD card reader, GSM module, and a motor driver. You will also learn how to apply an engineering process that will streamline your projects, making them more efficient in time and resources.

See Appendix A for detailed information about the hardware used in this book.