

An Introduction to Robot Programming

SECOND EDITION

Programming Sumo Robots with the MRK-2

Eric Ryan Harrison

William J Ashby, PhD

Table of Contents

Foreword 7
by Eric Parker..... 7

Author’s Note 8

Using this textbook 9
Style Conventions 10

Chapter 1 - An Introduction to Sumo Robotics 11
Lesson 1.1 - A Brief Introduction to Sumo (Robots) 11
The Microprocessor..... 12
Sumo Robot Competition Structure 13
Lesson 1.2 - A Tour of Your MRK-2..... 14
Mechanical Parts 14
The Electrical Parts 15
MRK-2 Pin Configuration 16
Lesson 1.3 - Learning the Arduino IDE..... 17
The Toolbar 18
The Editor Window..... 19
The Console 19
Understanding Output 20
Exercises 21

Chapter 2 - Programming Basics 23
Lesson 2.1 - Program Structure..... 23
Code Comments 24
Application Flow 25
Exercises 25
Lesson 2.2 - Variables..... 26
Data Types 28
Integers - <http://bit.ly/ArduinoInt>..... 28
Floating Point Numbers - <http://bit.ly/ArduinoFloat> 30
Booleans - <http://bit.ly/ArduinoBoolean> 30
Arrays - <http://bit.ly/ArduinoArrays>..... 31
Strings - <http://bit.ly/ArduinoStrings> 32
Exercises 32
Lesson 2.3 - Functions 33
A Real World Example 35
Exercises 37

Chapter 3 - Loops and Control Structures	39
Lesson 3.1 - For Loops	39
Exercise.....	40
Lesson 3.2 - If and Else	40
Exercises	43
Lesson 3.3 - Switch.....	44
Exercise.....	47
Chapter 4 - Interacting with Components.....	49
Lesson 4.1 - Digital and Analog Sensors.....	49
Lesson 4.2 - Investigating Inputs	51
The Ultrasonic Sensor.....	52
Infrared Line Detection Sensors	55
The Push Button	57
Exercises	59
Lesson 4.3 - Motors!.....	60
Exercise.....	62
Chapter 5 - Fun with Functions.....	63
Lesson 5.1 - Don't Repeat Yourself.....	63
Lesson 5.2 - Musical Programming with Functions	64
Functional Requirements	67
Task 1 - The Play Function	68
Task 2 - The Array Format	68
Task 3 - Loop Through the Array to Play the Song	69
Exercises	70
Lesson 5.3 - Your First Custom Robotics Library.....	70
Exercises	78
Chapter 6 - Bot: Line Following	79
Breaking Down the Requirements	79
Definitions	82
The Setup.....	83
The Loop.....	84
Exercises	87
Chapter 7 - Bot: Robotics for Fun	89
Breaking Down the Requirements	89
Managing State	90
Developing the Function for State 1	95
Exercises	98

Chapter 8 - Competitive Sumo Robots	99
Starting Your Robot	99
Push Button Pseudo-code	100
Strategies for Autonomy	106
Algorithm #1: Random Rotation Search.....	107
Algorithm #2 - Random Drive Pattern.....	108
Algorithm #3 - The Wobbler.....	108
Creativity and Clever Tricks	109
Implementing the Random Rotation Search Algorithm.....	109
The Compete Function	113
Abort!	114
Attack!	115
Searching	115
Exercises	117
Appendix A - Troubleshooting Common Bugs	119
Common Programming Problems	119
Missing Semicolon.....	119
Scoping Problems	120
Missing Curly Brace	121
Common Hardware Problems	122
Power LED Not Lighting	122
Problem Uploading or Disconnected MRK-2.....	123
Motor Spinning the Wrong Direction.....	124
Rebooting During Competition or Dropping Bluetooth	125
Appendix B – Pin Names & Uses	127
Table of Pin Names, Capabilities, & Uses.....	128
Pin Access	129
Appendix C - Official Sumo Robot Rules	131
Section 1. Definition of the Sumo Match	131
Section 2. Requirements for Ring Area	131
Section 3. Requirements for Robots	132
Section 4. How to Carry Sumo Matches.....	133
Section 5. Start, Stop, Resume, End a Match	133
Section 6. Time of Match.....	134
Section 7. Yuhkoh	134
Section 8. Violations	135
Section 9. Penalties	136
Section 10. Injuries and Accidents during the Match.....	136
Section 11. Declaring Objections.....	137
Section 12. Requirements for Identifications for Robots.....	137

Foreword

by Eric Parker

The journey of Sumo Robot League would be more accurately described as a quest. Quests often occur in fantasy worlds filled with warriors and magic. I like to think that what you hold in your hands now is a great book of magic. Before you learn the magic of logic processes, automation, and embedded systems to take your first steps to becoming a code warrior, indulge me by learning just a bit about our quest...

I first met Eric Harrison at a Hackathon that I had organized in Augusta, GA called “Hack for Education”. The goal of our hackathon was to create better access to educational resources in our community. Over 48 hours we witnessed three projects get built: the construction of a Wi-Fi antenna strong enough to give an entire neighborhood free internet, an educational resources website, and a number of computers made of raspberry pi’s and donated keyboards, mice, and monitors that only cost \$35. Perhaps more importantly, the weekend marked the gathering where our band of coders, designers, engineers, game developers, and cybersecurity professionals decided to open theClubhou.se.

theClubhou.se became our guild, dojo, cultural center, or in Sumo terms ‘*heya*’. In short, it was the place to show off your skills and learn new ones. It was there that Harrison and I (we’ll let history decide which of us is the better Eric) were first introduced to sumo robots. For each of us, it was our first time building a robot. My robot, Hilda, used a discreet brain, old school, 100 solder points and fully controlled by analogue dials to adjust sensor responses and speed, but luckily for me no actual computer programming. Sadly, Hilda did not win the day. The honor went to ‘derp-derp’, Harrison’s bot with lightning quick algorithms and a minimal amount of soldering. We had found our wizard!

As we celebrated our battles, we all wanted to know one thing, “WHEN’S THE NEXT TOURNAMENT?” Thus was born, Sumo Robot League, and the seeds of our quest: to teach every person how to build and program their own robot.

Author's Note

Thank you for buying a Sumo Robot League Mini Robot Kit (MRK) and this associated programming book. The MRK-2 package and this book are the end result of thousands of hours of passionate work from people who share a common belief. The belief that building and programming autonomous competitive robots is the most awesome, empowering way to give teenagers and young adults meaningful, engaging STEM (Science, Technology, Engineering, and Math) experience.

What you hold in your hands is the result of hundreds of hours of writing, coding, revising, testing, and crying. The MRK-2 hardware you own and will learn to program is the result of dozens of hardware revisions, thousands of hours of 3D-printing, and many late nights fighting to overcome unexpected hurdles and bring you something that never existed before -- a 3D-printable, Arduino-based, mini-class Sumo Robot.

We would not be here today without the effort of a lot of individuals, and we'd like to take a few seconds to thank them for their contributions.

- Chris and Amanda Williamson for teaching us how to build robots.
- Eric Parker for daring to dream this up and putting in actual money to make sure we could be successful.
- Grace Belangia for running the show and making sure that we were working in the right direction together.
- Chad DeMeyers and Kevin Huffman from E3 Embedded Systems (<http://www.e3embedded.com/>) for helping us with board revision after board revision. You guys have a lot of patience and we appreciate you.
- To all our families, friends, and supporters: Thank you for giving us the support we needed to help the world learn to build robots.

Eric & Will

Using this textbook

This book is designed, primarily, to serve as a programming textbook. It's arranged to take someone with no prior programming experience through the fundamentals of programming and have them work towards a level necessary to develop new and interesting competitive Sumo Robots. We have made every effort to break this book up into eight concrete Chapters, each with explicit goals intended to mirror the way we've been teaching kids Sumo Robotics.

The first five chapters of this book are dedicated to teaching the basics of ATmega328 programming on the MRK-2 circuit board using the Arduino IDE. We'll cover everything from how variables and functions work along with interacting with the physical hardware and sensors provided in this kit. Each of these chapters is divided into two to three lessons explaining one major subtopic in each lesson, and we've tried to include practical tips and exercises in each lesson to allow you to play around with these concepts in several different ways. These exercises are intended to be a hands-on activity to encourage tinkering and experimentation.

The last three chapters are slightly different and each focuses on a specific individual robot design. Chapter 6 walks you through the design and development of a robot that will compete in a line-following competition. Chapter 7 mixes things up by creating a robot that will randomly navigate around a room and avoid objects as well as showing you ways to build a robot with multiple operational states. Chapter 8 concludes with a bang and walks through the logical design and implementation of a competitive Sumo Robot.

In this text, all code examples and exercise solutions are located on GitHub at <https://github.com/SumoRobotLeague/MRK-2>. These solutions were developed to ensure that all students working their way through this book would have the resources necessary to make it all the way to the end. Although these code examples are available, we encourage you to try and work out the solutions to these exercises on your own. The best way to learn is by doing, and we've tried to give you plenty of opportunities to practice the techniques that are covered in this book.

Style Conventions

All paragraphs and descriptions are written using an Arial font.

All code examples will be listed in Consolas with a 9-point font and will have syntax highlighting that looks like the example below:

```
void setup() {  
    // initialize digital pin 7 as an output.  
    Serial.println("Hello world!");  
}
```

Whenever `functions()`, `variables`, or `SPECIFIC_VALUES` are listed inside of a descriptive paragraph, this book will highlight the special nature of these words by changing the font to an 11-point Courier New, as seen in this paragraph.

Special Notes, Practical Exercises, and Warnings will be sprinkled throughout this book and will be underlined and shown in bold.

If you discover any technical errors in this printing of your book, please email eharrison@p4r75.com or washby@sumorobotleague.com.

Happy botting!

Chapter 1 - An Introduction to Sumo Robotics

Welcome to the exciting world of Sumo Robotics! You're about to enter a world filled with autonomous fighting robots battling it out in a small circle. Defeat your foes with the strength of your algorithms and the torque of your tiny little motors.

Lesson 1.1 - A Brief Introduction to Sumo (Robots)

Inspired by the Japanese sport of Sumo Wrestling, Sumo Robot competitions involve building, programming, and competing with small autonomous robots inside a circular ring with a black bottom and a white strip around the edge. The objective of each match is to have your robot detect your competitor and push that robot out of the ring before it can do the same to yours.

Sumo Robot competitions are generally over very quickly. Most matches last less than a minute, but programming a competitive robot that can win repeatedly is an interesting engineering challenge.



Figure 1.1.1 - Mini-sumo robots engaged in battle.

Sumo Robots compete in certain classes ordered by size and weight ranging from the ultra-tiny Femto-Sumo class that must fit inside a 1cm cube all the way up to the Heavyweight-Sumo that can have a mass up to 57kg. The MRK-2 is an entry-level kit built to compete in the Mini-Sumo class in which every robot must be less than 10cm wide by 10cm long and have a mass of less than 500 grams.

In some competitions, there are Sumo Robot events in which the robots are controlled by a human operator using a radio-powered remote control or by apps built that allow the robot to be driven with a Bluetooth connection. Sumo Robot League (<http://sumorobotleague.com>) focuses primarily on fully-autonomous robot events in which every robot must compete without human intervention.

The Microprocessor

The brain of most autonomous Sumo Robots is a small microcomputer like the Arduino (<http://arduino.cc>) or sometimes something more powerful like a Raspberry Pi (<https://www.raspberrypi.org/>). The MRK-2 uses a custom-built robotics circuit board that contains both the processor as well as all the necessary sensors and motor controllers. The best part about this robotics platform is that it was designed to be fully compatible with Arduino and is programmable using the same programming language and environment. While this book was written to teach the basics of Sumo Robot programming for the MRK-2, the lessons contained within provide an excellent foundation for Arduino-based project development.

This book covers both the foundations of Arduino-based microcomputer programming as well as the specific design and implementation of robot programming. In this book, we take a “crawl, walk, run” approach to instruction by first teaching the general structure of programs, then focus on the specific keywords and concepts used in building programs, before concluding with the design and development of complicated robots using a combination of the techniques learned in previous chapters.

By the end of this book, you will have written several full programs to take advantage of every component and sensor provided by the MRK-2, but the real hope is that we will be able to teach you to think like an engineer. In later chapters of this book, we focus more on the thought processes behind the designs we've chosen to develop in order to teach you how to logically structure your ideas and use programming to take those concepts from vision to reality.

Sumo Robot Competition Structure

This book was written as a companion text to the MRK-2 Sumo Robot Kit, designed and manufactured by Sumo Robot League (<http://sumorobotleague.com>). Sumo Robot League runs classes and holds official tournaments in several locations around the country and this book focuses on developing robots to meet the Sumo Robot League Official Rules, included in this book in [Appendix C](#). However, Sumo Robot League is not the only organization that holds Sumo Robot competitions, and each group may have slightly different rules and procedures for each event.

In general, Sumo Robot matches begin with both players placing their robots somewhere inside the ring. When given instructions by the judge, the players activate their robots and quickly step a few feet back from the ring so that none of the robots will mistakenly detect a leg as an opponent robot. Once the robots have been activated at the start of a match, they are required to wait at least five seconds before they're allowed to begin searching for the opponent.

The robots then have three full minutes to attempt to push the opponent completely out of the ring. At the end of the three minutes, if both robots are still inside the ring, the round is considered a draw and the players are both awarded a point.

Note: Sumo Robot Competitions are completely non-violent. It is a major rules violation to develop a robot that has any mechanism designed to injure your opponent's robot. Pushing is the only acceptable form of attack.

There are a lot of other rules concerning stalemates and other weird edge cases that pop up from time to time, but those really only apply to people that haven't read this book and build robots that malfunction. You're going to do great and your robot is going to kick some serious butt.

Lesson 1.2 - A Tour of Your MRK-2

When you purchased your MRK-2 Sumo Robot Kit, it arrived either fully assembled or as a collection of parts for you to assemble. The components of this robot kit are separated into two primary logical components: the mechanical parts and the electrical parts.

Mechanical Parts

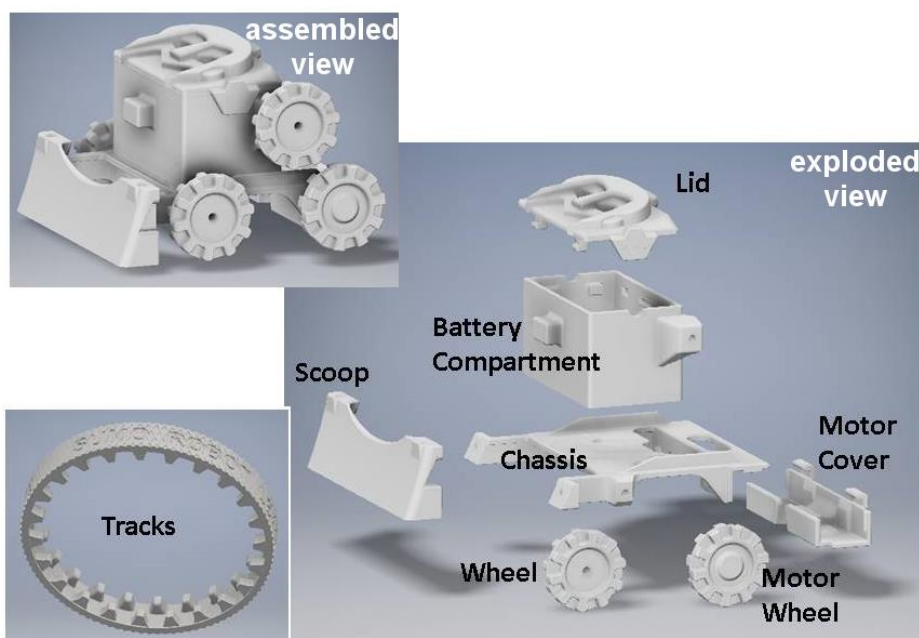


Figure 1.2.1 - MRK-2 mechanical components.

The mechanical parts, as the name implies, are all the components that provide the robot with structure and hold the electrical parts in place.

The Electrical Parts

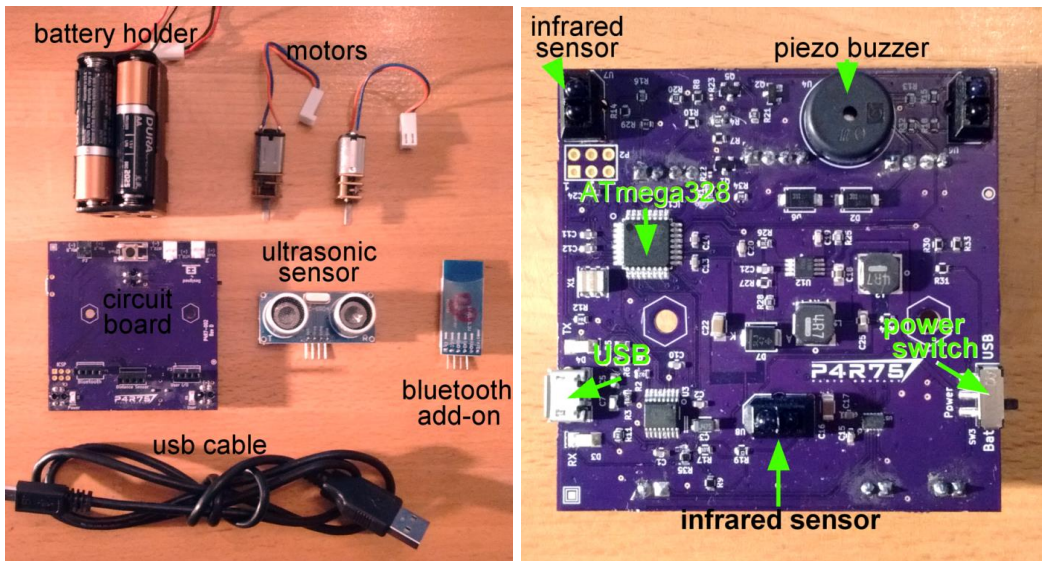


Figure 1.2.2 - MRK-2 electrical components.

The electrical parts are really what makes the MRK-2 a robot. Most of the necessary components and sensors are built into the MRK-2 circuit board that serves as the brain of your robot. The core of the circuit board is the small, square 16-bit Atmel ATmega328 microprocessor. This is the same low-cost processor used by many Arduino boards. The circuit board has connectors for battery power, motors, ultrasound sensor, and other accessories such as a Bluetooth chip or servo motor. The following components are built-in: a power indicator LED, an On/Off Switch for battery power, a user controlled LED, three separate infrared line sensors, two LED indicators for USB communication, a user-controlled LED, a piezo buzzer for music, and a pushbutton switch.

MRK-2 Pin Configuration

In order for a robot to use its electrical components, a programmer has to know what electronics are connected to which pins on the microprocessor

The table below lists all of the MRK-2 pins used in this textbook.

Pin	Component	Description
2	pushbutton, input_pullup	returns a 1 if open and a 0 if pressed.
3	buzzer, output PWM (pulse-width modulation)	plays music! Use the tone() function.
4	IR emitter, output	turns on the infrared emitters attached to the left, right , and rear IR sensors
5	leftSpeed, output PWM (pulse-width modulation)	controls speed on the left motor. Use values from 0 to 255.
6	rightSpeed, output PWM (pulse-width modulation)	controls speed on the right motor. Use values from 0 to 255.
7	leftDirection, output	controls direction on the left motor. HIGH is forward, LOW is reverse.
8	rightDirection, output	controls direction on the right motor LOW is forward, HIGH is reverse.
10	ultrasonic sensor output	triggers a pulse, aka ping, of ultrasound when turned on
13	user LED, output	lights up the front left LED when turned on.
A0	ultrasonic sensor input	listens for the ultrasound to come back
A1	leftIR analog input	senses how much infrared light is absorbed, maximum is 1023, min 0
A2	rightIR analog input	senses how much infrared light is absorbed, maximum is 1023, min 0

See [Appendix B](#) for the complete list of pins and their uses.

Lesson 1.3 - Learning the Arduino IDE

The MRK-2 Sumo Robot Kit was designed to be as easy to program as possible. In competitive sumo robot circles, Arduino boards are commonplace. In order to focus on learning programming and customizing the robot via 3D printing, the MRK-2 circuit board integrates what other robots would have distributed across several different circuit boards (or shields). This means that you don't have to worry as much about wiring electronics together and can still adapt code used by other Arduino sumo robots to your MRK-2.

To make programming the MRK-2 as easy as possible, this board was developed to identify as an Arduino Uno. Throughout this entire book, we'll use the Arduino IDE to write, compile, and upload sketches into your MRK-2.

INFO: If you don't already have a copy of the Arduino IDE, download the latest version at <http://bit.ly/ArduinoIDE>.

When you first launch the Arduino IDE, you should see something that looks like the figure below.



Figure 1.3.1 - Arduino IDE







The IDE is separated into three major zones: the toolbar, the editor window, and the console.

The Toolbar



Figure 1.3.2 - Arduino IDE Toolbar

The Toolbar is the small line of buttons located just below the window's title bar. The following table provides a brief overview of each button.

Button	Name	Description
	Verify	This button will compile your code and make sure it's free of any syntax errors.
	Upload	This button will first compile your code and then attempt to upload your program to the robot you have attached to your computer's USB port.
	New	This button will open a new, blank Sketch for you to begin programming. Whatever Sketch you were working on will remain open in a separate window.
	Open	This button will provide you with a menu to select existing Sketches to open.
	Save	This button will open a "Save As..." dialog box and allow you to save your Sketch. Save often!
	Serial Monitor	This button will open the Serial Monitor and allow you to view messages from your connected device. In later lessons we will explore ways to use the Serial Monitor to debug your programs.

The Editor Window



Figure 1.3.3 - Arduino IDE Editor Window

The Editor Window is the area in which you'll spend most of your time writing and editing code. While the Arduino IDE is not the nicest code editor you'll ever use, it does provide some lightweight syntax highlighting and line numbers (which you can enable in Preferences). The things it lacks, like code completion, a debugger, and advanced configuration options, make the Arduino IDE unsuitable for large-scale engineering projects, but it serves the hobbyist audience well and most of your programs are not going to need such advanced features.

The Console

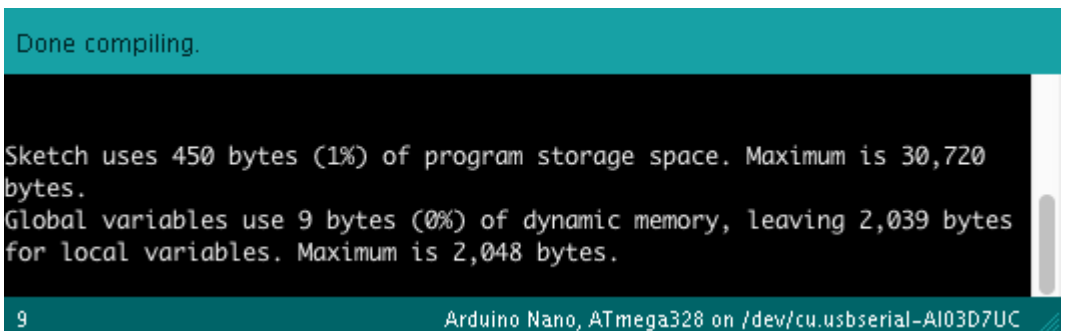


Figure 1.3.4 - Arduino IDE Console

The large black area at the bottom of the window is the Console. This is where the Arduino IDE will warn you about syntax errors and reports back about the status of your compilations and uploads.

Of special note is the status bar at the bottom of the console. In Figure 1.3.4 above, you will see the current board configuration the Arduino IDE has selected. The line “Arduino Uno on COM4” or “Arduino Uno on /dev/cu.usbserial-A103D7UC” lets us know the IDE will be compiling for the Arduino Uno and will be sending compiled sketches to the device attached to the USB port.

Sometimes the Arduino IDE doesn't correctly detect what type of board is connected and you'll need to manually select the correct board from the Tools menu. See [Appendix A - Troubleshooting Common Bugs](#) for solutions to problems like this.

Understanding Output

One of the biggest challenges with robot programming is trying to figure out what your robot is doing after you've uploaded a new sketch. With other types of software development, a huge part of an iterative development process is looking at the output of your program to verify functionality. Because we're trying to program autonomous robots, our eventual goal is to have a robot driving around without any input or output from our computer.

This makes iterative development an interesting challenge. Throughout development, you'll find that the easiest thing to do is to rely on the Serial Monitor and the functions `Serial.print()` and `Serial.println()`.

Exercises

Copy the following Sketch into your Editor Window:

```
void setup() {  
    // initialize digital pin 13 as an output.  
    pinMode(13, OUTPUT);  
}  
void loop() {  
    digitalWrite(13, HIGH); // turn the LED on  
    delay(1000);           // wait for a second  
    digitalWrite(13, LOW);  // turn the LED off  
    delay(1000);           // wait for a second  
}
```

1. What happens when you click the Verify button?
2. What happens when you click the Upload button? Does the LED on your MRK-2 blink?
3. Change the code so that the LED is on for only half a second.
4. Change the code so that the LED is only off for half a second.
5. Save your Sketch in a location you'll remember and name it "BlinkingLED.ino".

Chapter 2 - Programming Basics

Lesson 2.1 - Program Structure

In this lesson, we're going to cover the high-level program structure of the Arduino platform. While the Sumo Robot League MRK-2 robot you have is not an Arduino, the primary processor is the same and the programming characteristics of the MRK-2 and an Arduino are nearly identical.

Software written for an Arduino-based board is called a Sketch, but this is just a fancy term that is a synonym for the word "program" and we'll often use these two terms interchangeably throughout this book. A sketch is a program and a program for the Arduino is a sketch. Open up your Arduino IDE now and we'll take a look at the empty sketch the Arduino IDE generates.

This sketch contains two empty functions, `setup()` and `loop()`. We'll cover functions in greater detail in Lesson 2.3, but a function is basically a logical collection of one or more lines of code. Whenever your code "calls" a function, the processor knows to jump to that set of computer instructions and begin executing those lines of code.

```
void setup() {  
    // put your setup code here, to run once:  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```

The `setup()` function runs exactly one time whenever your robot is first powered on. As you can probably guess, we use the `setup()` function to "setup" the rest of our program. This is where we'll configure pins for input/output, initialize external libraries, and prepare our robot for world domination.

Info: In programming, a Library is a pre-built set of functions you can import into your software to make complicated programming tasks easier. Libraries provide reusable blocks of functionality you can easily embed into your programs to perform complex operations. We will cover libraries in more detail in [Chapter 5](#).

The `loop()` function is where the real magic starts to happen. This function executes repeatedly for however long your MRK-2 board receives power. Every action you program your robot to perform will most likely be started from somewhere in the `loop()` function.

The other thing that you will notice in this default sketch are the two comment lines included inside the functions. These comments are added automatically to help you remember the purpose of those special functions.

Code Comments

A comment is a line included in your code that is not used for program execution. Comments are intended as a mechanism to convey additional meaning and understanding to developers that may be reading your code in the future.

The programming language that we're using to write our programs with has two different formats for comments.

A single-line comment begins with two forward slashes `/**` and will treat everything after the forward slashes as a comment.

```
// This is a single line comment
```

A multi-line comment begins with a forward slash, followed immediately by an asterisk `/*`. Once the compiler sees the `/*`, it will treat everything as a comment until it sees the close comment indicator, which is an asterisk followed by a forward slash `*/`

```
/* This is a really long comment that goes over multiple  
   lines. This comment style is extremely useful. */
```


You will see comments used in many of the programming examples provided in this book as a way to explain the purpose behind some of the concepts that haven't been covered yet. Whenever you see a comment in a code example, it should help you make sense of what the code is supposed to do.

Application Flow

All Arduino-based applications follow the same basic application flow. Everything begins when the board receives power from either the batteries or by being connected to a power supply from the USB connection. After the board has finished its internal startup sequence, the processor begins to execute the program that has been uploaded to the Flash Memory and starts by executing the required function `setup()`. After the `setup()` function has finished execution, the system will begin an infinite loop by repeatedly calling the `loop()` function for as long as the board receives power.

Info: An “Infinite Loop” is a set of computer instructions that will repeat forever.

The functions `setup()` and `loop()` are functions that are *required* in every single program you write for the MRK-2. If you don't include them, you will get an error when you try to compile or upload your sketch.

Exercises

1. Remove the `setup()` function from your sketch and click the Verify button. What happens in the console?
2. Remove the `loop()` function from your sketch and click the Verify button. What happens in the console?

Lesson 2.2 - Variables

“I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.” -- [Linus Torvalds](#)

In Computer Science, almost everything can be simplified down to a few generic data structures and programming your Sumo Robot is no different. The AVR-C language you'll be using to program your robot has a lot of data types, but we'll be able to accomplish most of what we want to do by using just a handful of the most important.

Every program you write will make use of variables, so it's important to spend time to really understand what a variable is and what operations you can perform on them. A variable is basically a fancy term for a bucket capable of holding one or more nuggets of information. If we take a closer look at the code we used in the exercise of [Lesson 1.3](#), we can quickly discover the usefulness of using variables in our code.

```
void setup() {  
    // initialize digital pin 13 as an output.  
    pinMode(13, OUTPUT);  
}  
void loop() {  
    digitalWrite(13, HIGH); // turn the LED on  
    delay(1000);           // wait for a second  
    digitalWrite(13, LOW); // turn the LED off  
    delay(1000);           // wait for a second  
}
```

In this code example, we're passing the number 1000 to the `delay()` function to tell the processor we want it to wait for one second. The number 1000 represents the number of milliseconds the microprocessor will wait, but if you remember changes we had you write in the previous exercises, we had to modify this number to change the speed at which the LED would blink.

In larger programs, there will be many places throughout the code that will share the same variable. Variables enable you to store a value in one location and use (or even modify) that value throughout your code. If you need to make a change, you only have to change it in the place where you've assigned the value to the variable.

A simple change that will improve the readability and utility of your program is to write your code using variables in place of literal values. Here's what that code would look like with that change:

```
int delayMs = 1000; // delay time in milliseconds
void setup() {
    // initialize digital pin 13 as an output.
    pinMode(13, OUTPUT);
}
void loop() {
    digitalWrite(13, HIGH); // turn the LED on
    delay(delayMs);
    digitalWrite(13, LOW); // turn the LED off
    delay(delayMs);
}
```

We've just replaced the value of 1000 we were using in two separate lines of code with a reference to the integer variable "delayMs" which we've initialized with a starting value of 1000. Now if we wanted to change the blink speed, we only have to change the value stored in that variable once and every call to the function `delay(delayMs)` will execute with the new value.

This example is overly simplistic, but nevertheless highlights the usefulness of variables in writing expressive, maintainable code.

Variables also allow you to clearly communicate the intent behind your code. By choosing an expressive name for your variable, you can craft code that reads almost like a book. You can name your variables whatever you want, but you should make an effort to name them clearly and use them to communicate your design intentions to anyone might modify your code in the future.

Good variable naming is a skill that will come with time as you read and write more code.

Practical Exercise: Take the above code example and instead of using one delay variable, use two separate variables. Name one of the variables “durationOn” and the other “durationOff”.

Data Types

Now that we’ve learned a little bit about what a variable is, it’s time to learn about what types of variables we can create. Each variable in a typed programming language, like AVR-C, comes with an attached data type. In the previous example, the word “int” at the beginning of the line defining our variables tells the processor that the type of data we’re going to store in the delayMs variable is an Integer. An Integer is just a fancy computer science word that means “a whole number.”

While AVR-C contains a plethora of complicated data types, robot programming is easy enough that you’ll only need to use a handful of the types. We’ll cover each type in the sections below along with some examples.

For a complete reference of all the supported data types, along with examples of how to use them, please visit the Arduino Language Reference located at <http://bit.ly/ArduinoReference>.

Integers - <http://bit.ly/ArduinoInt>

An Integer, as we previously covered, is any whole number between -32,768 and 32,767. In programming, Integers are one of the most basic and fundamental data types and you’ll use them all the time.

In the blinking LED example we’ve been playing with, we used an Integer variable to store our delay in milliseconds. In the future, you’ll use Integers for setting up timers, approximating attack distances, and for controlling loops.

The cool thing about Integer variables is that we can perform arithmetic operations on them. You can add, subtract, multiply, and divide integers.

The following code example shows how we can use Integers to calculate the number of seconds in a six-hour period.

```
void setup() {  
    Serial.begin(9600);  
}  
void loop() {  
    int seconds = 60; // seconds per minute  
    int minutes = 60; // minutes per hour  
    int hours = 6;    // total number of hours  
    int secondsPerDay = seconds * minutes * hours;  
    Serial.println(secondsPerDay);  
}
```

If we compile this script and send it to our MRK-2 board and then open our Serial Monitor, we should see the number 21,600 displayed over and over again. The variable `secondsPerDay` is created and will be assigned the result of multiplying the variables `seconds` against `minutes` and `hours`, which gives `secondsPerDay` the value of 21,600. The final line inside the `loop()` function is a pre-built Arduino function that will cause the value stored in `secondsPerDay` to be printed to the Serial Monitor on a separate line.

Pro-Tip: You will use the functions `Serial.println()` and `Serial.print()` all the time to help you peek inside your running programs to debug your code.

Integers are the workhorse of programming and should be used whenever possible. Math operations performed on Integers outperform math operations on all other data types and you should always attempt to use the smallest size of integer that will meet your needs.

If you need to work with numbers larger than 32,767, you'll need to use a different data type like an [unsigned int](#) or a [long](#).

Floating Point Numbers - <http://bit.ly/ArduinoFloat>

A floating point number is a number that contains a decimal point. In Sumo Robot programming, we'll often use a floating point number to represent a value provided by one of our sensors. Floating point numbers allow us to get incredibly accurate with our calculations at the expense of speed. Whereas it's fast and efficient to work with integers, floating point calculations often take a lot longer.

In competitive robot programming where performance speed can be the difference between winning and losing, having a lot of complex floating point numbers being calculated in your main loop() function can cause your robot to react sluggishly compared to other robots. Whenever possible, floating point numbers should be used with deliberate intention.

```
float pi = 3.14;
```

In this example, the variable `pi` is declared as a float and has an initial value set to be 3.14.

Booleans - <http://bit.ly/ArduinoBoolean>

A Boolean is a simple true or false value. We use Booleans in conjunction with if statements, which you'll learn about in lesson 3.2, to control the operational flow of your programs.

```
boolean isRunning = true;
if ( isRunning ) {
    isRunning = false;
} else {
    isRunning = true;
}
```

In this simplistic code example, we've declared a boolean variable called `isRunning` and set its initial state to true. Anything inside the parenthesis following an if statement is evaluated to determine if the condition is true or false. In this situation, `isRunning` was set to true, so that if block will evaluate to true and immediately jump to the next line where we set `isRunning` to false.

You will write code similar to this in almost every program you develop. In later chapters, we'll build a simple line following robot that will coordinate the program's current state with several of these types of conditional checks.

Arrays - <http://bit.ly/ArduinoArrays>

Arrays are special types of variables that hold multiple data values. Before we'd learned an integer variable could be assigned like this:

```
int myVariable = 12;
```

If we ever find ourselves in a situation where we want to work with a large list of values, an Array is the type of data structure you'll use to store multiple values in one variable.

Declaring an array is fairly simple. Just like with the other variables we've discussed, you always start by specifying the type data you plan to store in this array.

```
char daysOfTheWeek[] = { 'S', 'M', 'T', 'W', 'T', 'F', 'S' };
```

The one part of this assignment that is slightly different from our other variables are the square brackets immediately following the name of the variable. These square brackets let the compiler know that this variable is an array and has multiple elements. If you leave the brackets empty, like in the example above, the compiler will count the number of elements in the array and create an array of the appropriate size.

If you want to be more explicit in your array sizes, you can specify the number of elements your array will contain in the declaration like this:

```
char daysOfTheWeek[7] = { 'S', 'M', 'T', 'W', 'T', 'F', 'S' };
```

Pro-Tip: There are performance implications behind your decision to declare your array length when you create the array, but this is out of the scope of this course. Most of the time, especially during the early prototyping stages, you're better off just letting the array size get handled by the compiler.

To access a value from that array, we can reference it by it's index number, which starts at 0 and goes up to the highest number of elements.

`daysOfTheWeek[0]` would contain "S"
`daysOfTheWeek[3]` would contain "W"

Arrays are extremely powerful complex data structures we can use to store almost anything. Often, we'll couple arrays with loops (discussed in the next chapter) to rapidly process through a large list of items.

Strings - <http://bit.ly/ArduinoStrings>

Strings are similar to but also unlike arrays of characters. Both can be used to display text. However, the `String` class provides a lot of tools for manipulating and comparing text, such as `toLowerCase()` and `substring()`. For more details please refer to the Arduino reference by following the above hyperlink.

Exercises

```
void setup() {  
    int a = 5;  
    int b = a + 12;  
    a = 10;  
    float c = 7.2;  
    int d = c * a;  
    int e = 9 / 4;  
    int irSensors[] = { A1, A2, A3 };  
}  
void loop() {  
}
```

1. In the given code sample, what is the value of the integer "b"?
2. What is the final value of the variable "a"?
3. Does the line "`int d = c * a;`" cause a compilation error? If not, what is the result when you pass it to `Serial.println()`? Why does it behave this way?
4. What IR sensor pin exists in the `irSensors` array at index 2?
5. What IR sensor pin exists in the `irSensors` array at index 0?
6. What is the value of "e"?

Lesson 2.3 - Functions

Functions are the reusable building blocks of modern software. Any time you find yourself writing several lines of code to do similar things, you should stop and ask yourself “Can I put this code in a function and make the code easier to read?”

Let’s examine our blinking light code from the previous Lesson and see how we can use functions to make our programs more readable and extendable.

```
int delayMs = 1000; // delay time in milliseconds
void setup() {
    // initialize digital pin 13 as an output.
    Serial.begin(9600);
    pinMode(13 OUTPUT);
}
void loop() {
    digitalWrite(13, HIGH); // turn the LED on
    delay(delayMs);
    digitalWrite(13, LOW); // turn the LED off
    delay(delayMs);
}
```

In this code sample, we have the code that actually turns the LED on and off happening inside the loop function. But what if our blinking light code was just one small component of our larger project and we didn’t want to have a bunch of calls to `digitalWrite()` and `delay()` sprinkled around and making a mess of things?

The simple solution would be to extract those lines of code and put them in a separate function like this:

```
void blink(int delayTime) {
    digitalWrite(13, HIGH);
    delay(delayTime);
    digitalWrite(13, LOW);
    delay(delayTime);
}
```

The first line defines the name and arguments of the function. The word “void” lets the program know what type of value this function is going to return. In this case, since we’re not returning a value at all, we’ll use the return type of void. If we were calculating some value and returning that, we would pick a return type that matches the native data types we talked about in Lesson 2.3 like int or float.

The next word is the function name. We’ve given this function the name “blink” because that’s a good name for a function that controls blinking. In software engineering, it’s important to always pick expressive names for functions that properly describe what that function or variable is going to be used for. If we’d chosen a name like the letter “a” for our function name, we’d certainly save time typing that function all the time, but it wouldn’t do a good job of explaining to someone what the function is going to be used for. You would have calls to a () randomly placed throughout your code and it would be difficult to maintain and modify in the future.

Pro-Tip: Always err on the side of caution when picking function and variable names. A good function name provides clarity and allows your programs to be easily understood and modified.

After the function name, we have the argument list enclosed by parentheses (“(” and “)”). Any variables you define inside these parentheses will be available inside the function. When you’re writing functions, you’ll use this list of arguments to define and specify what values your function will need to perform its specific task.

After the argument list, the function’s “block” begins with an opening curly brace “{” and continues until its matching closing curly brace “}”. Any lines of code between those two curly braces are considered to be a part of that function.

In our blink() function example, we’ve specified that we want the function to be called with an integer which we’ll use inside the function with the name of “delayMs”. Now every time we use the blink() function in our program, we’ll have to pass in an integer or we’ll generate a compiler error and our code will not execute.

```
void setup() {  
    // initialize digital pin 13 as an output.  
    Serial.begin(9600);  
}
```

```

        pinMode(13, OUTPUT);
    }
    void loop() {
        blink(1000);
        blink(500);
        blink(1500);
        blink(250);
    }
    void blink(int delayMs) {
        digitalWrite(13, HIGH); // turn the LED on
        delay(delayMs);
        digitalWrite(13, LOW);  // turn the LED off
        delay(delayMs);
    }
}

```

Our newly revised code now has a function called `blink()` that accepts an integer as a required primary argument. This process has cleaned up our code and made it possible to write code that is more expressive than it was before. Now our code can have a call to `blink()` on one line in place of:

```

digitalWrite(13, HIGH); // turn the LED on
delay(delayMs);
digitalWrite(13, LOW);  // turn the LED off
delay(delayMs);

```

This single, simple change has shown how we use functions to clean up large lines of reusable code and place them in named functions that provide insight into what complicated blocks of code are doing.

A Real World Example

In our Sumo Robot, we use an ultrasonic sensor to determine the enemy's range and decide if it's time to attack. This sensor, which we'll cover in greater detail in Chapter 4, basically works by sending out an ultra-high frequency audio signal from one of the speakers and then listening for it to return to the microphone. The value that the sensor will report back to your program will be the amount of time that the signal traveled before it bumped up against something.

This number will be ridiculously high and won't be useful for our purposes of writing clean, elegant code that can be understood by anyone that comes along to look at your program. What you'll want to do is convert that duration, in microseconds, into a usable distance in centimeters. Why centimeters? Because you're an engineer now. Embrace the Metric system!

The first thing we need to do is find the formula to convert the speed of sound into centimeters. Through the "Book-Magic" of me Googling things for you to save time, we come across this formula: $\text{microseconds} / 29 / 2$.

This formula basically divides the input from the ultrasonic sensor (microseconds since response), by 29 (speed of sound in microseconds per centimeter) and then divides that result by 2 to account for the fact that the sound waves had to travel out from your robot, bounce off the enemy, and then return back to your robot.

Let's write a function together that will take the microseconds that our sensor returns and converts it into a distance in centimeters.

The first thing we'll write is our function declaration line which specifies the return type, the function name, and the arguments that we're going to pass into this function.

```
long msToCm(long microseconds) {
```

The first word in this line of code specifies that we're going to be returning a value that is of the type "long". We have to use a long data type because the ultrasonic sensor could potentially return values much higher than 32,768. To be on the safe side, the sensor manufacturer recommends using a long.

The next word is our function name, which we've chosen to call `msToCm()`. This function name abbreviates both microseconds and centimeters so that you can save a bit of time typing, but we could have just as easily named the function `microsecondsToCentimeters()`. At this point, it mostly comes down to stylistic decisions and you're free to name your functions however you want. Just remember: **err on the side of clarity**.

The arguments list then specifies that we'll be accepting one argument to this function, which is a `long` called `microseconds`. That is a direct value we'll be receiving from our sensor.

We then end the line with an opening curly brace to let the compiler know that everything that follows is going to be inside the `msToCm()` function.

```
long msToCm(long microseconds) {  
    return microseconds / 29 / 2;  
}
```

The bolded line in the middle that says "`return microseconds / 29 / 2;`" is simply the most efficient way to return a value from this function. The `return` keyword specifies that whatever follows will be returned, or sent back, out of the function. In this case, we're going to "return" the calculated value of `microseconds` divided by 29, then divided by 2.

The last line is the closing curly brace. If you don't include this closing curly brace, the compiler will get confused, angry, and spew hateful error messages at you in the console. **Always include your curly braces!**

Whatever line of code that called that function probably called it like this:

```
long distance = msToCm(pingDistance);
```

Our new function will return our calculated number of centimeters and store that value in the newly created `distance` variable.

Exercises

1. Write a function that will accept two integers named "A" and "B" as arguments and will return their values multiplied together.
2. Your computer science teacher is very pro-America and doesn't like seeing nasty words like "Centimeters" in your code. Write a new function called `msToInches()` that takes an argument of: `long microseconds` and will return the distance in inches.
[Hint: The formula to calculate the speed of sound in inches is: `(microseconds / 74 / 2)`]

Chapter 3 - Loops and Control Structures

Lesson 3.1 - For Loops

Looping is a crucial part of software engineering. A loop allows you to write a few lines of code and then rapidly repeat each line of code multiple times. There are several different types of loops, but in this book we're going to cover the `For` Loop.

A `For` Loop has a simple structure. Let's look at the code snippet below and examine what it is doing in detail:

```
void setup() {  
    Serial.begin(9600);  
    int magicCounter = 0;  
    for ( int i = 0; i < 10; i++ ) {  
        magicCounter = magicCounter + i;  
        Serial.println(magicCounter);  
    }  
}
```

The first line defines how the for loop is going to behave. Inside the curly braces, we have defined our “index” integer with a name of “`i`” and set its initial value to 0. We then add a semicolon and move to the control portion of the parameters.

The line `i < 10` means that this loop will happen as long as the number `i` is less than the number 10.

The next section inside the control block after that last semicolon is the incrementation instruction. We're telling the for loop to increment the value of `i` by 1 each time this loop passes around to the top. We used the shorthand version of variable incrementation `i++`, but we could have just as easily used `i = i + 1` as those two versions essentially do exactly the same thing.

As with functions, we always end our loop declaration line with an open curly brace. Everything between the open curly brace and the close curly brace will be treated as a part of the loop and will be evaluated on each cycle of the loop.

Inside the loop, we have two statements. The first takes the `magicCounter` variable and adds the current value of `i` to it.

`Serial.println(magicCounter);` just shows us what is happening in our loop inside the Serial Debug Window.

Run this full program now and see if you are seeing the result you expect:

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

    int magicCounter = 0;
    for ( int i = 0; i < 10; i++ ) {
        magicCounter = magicCounter + i;
        Serial.println(magicCounter);
    }
}

void loop() {
    // do nothing here
}
```

What was the last number that printed in your Serial Monitor? Was it the number you expected? Why or why not?

Exercise

1. Write a for loop that prints out the numbers from 10 to zero.

Lesson 3.2 - If and Else

An autonomous robot is nothing if it's incapable of making decisions. An If block is one of the most effective ways to quickly and easily force your program to make a decision and act upon it.

You make decisions like this every day: “If it’s raining, take an umbrella.”, “If I’m cold, turn up the thermostat.”

These simple decisions manifest themselves in almost every aspect of software engineering and they’re a feature that you’ll use time and time again.

In engineering, we’ll often define our program’s structure using pseudo-code examples that describe (in mostly plain English) what it is we want our programs to do. There is no right or wrong way to write pseudo-code. pseudo-code is just one technique to use to logically think through the operation of a complex program.

Let’s take a look at some pseudo-code for how we want our eventual Sumo Robot to operate.

```
SETUP FUNCTION:
    SET SERIAL COMMUNICATIONS RATE
    SET UP PINS FOR INPUT AND OUTPUT
LOOP FUNCTION:
    GET ULTRASONIC SENSOR VALUES
    GET INFRARED SENSOR VALUES

    IF ULTRASONIC SENSOR DISTANCE IS LESS THAN 6:
        ATTACK!
    ELSE:
        IF INFRARED SENSOR DETECTS EDGE OF RING:
            ABORT!
        ELSE:
            SEARCH!
```

This pseudo-code basically covers all of the operational requirements for a fairly competitive Sumo Robot.

Starting from the top, we’ve specified that we have a **SETUP FUNCTION** and that this function will **SET UP PINS FOR INPUT AND OUTPUT** and configures the Serial line for communication.

Inside the primary **LOOP FUNCTION**, we loosely define our program’s logical flow.

The first thing we do is get the values from the ultrasonic sensor. We will store that value in a variable so that we can use it later in the function. We then get the values from the infrared sensors that are located at the bottom of the board. We'll store these values in a variable too so that we'll know when we've reached the edge of the ring.

Now that we've reached the section with our major IF blocks, the fun of programming is really going to show itself to us. When we use IF/ELSE blocks, what we're really doing is writing code that is capable of handling multiple forms of external information and then **acting on it!** For the first time, you'll be able to have your program make a decision. These are the building blocks of true Artificial Intelligence.

Let's dive into the pseudo-code we've written and see if we can logically walk through what it is we're trying to accomplish.

Pseudo-Code	Meaning
IF ULTRASONIC DISTANCE LESS THAN 6	If the sensor returns a distance less than 6, execute the next line of code, otherwise skip it.
ATTACK!	Perform some sort of attack routine, probably stored in a function somewhere.
ELSE	The sensor returned a distance greater than 6, so we're going to execute a different block of code.
IF INFRARED SENSOR DETECTS EDGE OF RING	If our IR sensors find the white line in the bottom, we need to back our robot up, immediately.
ABORT!	Perform some sort of abort motion routine, probably stored in a function somewhere.
ELSE	None of the other conditions matched, do whatever is in this block by default.
SEARCH!	Perform your searching routine, probably stored in a function somewhere.

Our first IF block checks to see if the **ULTRASONIC DISTANCE IS LESS THAN 6** (centimeters). If we find an object that is closer to us than 6 centimeters, we'll know that we've found a bad guy and need to charge forward at full speed and **ATTACK!**

If the sensor distance is greater than 6, the **ATTACK!** code will be skipped and we'll move into the **ELSE** block to continue the programs operation. The first thing we check in the **ELSE** block is another IF statement asking if our infrared sensors have detected the edge of the ring. If they have, we want to immediately **ABORT!** whatever we are doing.

If our infrared sensors AND our ultrasonic sensors haven't found anything, the final **ELSE** block will be activated and we'll want our robot to go into **SEARCH!** mode.

When designing complicated programs, it's often helpful to first sit down and think through what it is that you want your program to do. Writing pseudo-code helps you quickly get those concepts out onto paper so that you can walk through the logic and make sure everything makes sense.

Exercises

```
int height = 72;
if ( height < 42 ) {
    Serial.println("Not tall enough. Access denied! ");
} else {
    Serial.println("Climb aboard!");
}
```

1. In this code sample above, we have an if block that checks a person's height before allowing them to ride an amusement park ride. If the height they provide is less than 42, they'll get a warning message that they're too short to ride. Can you change this if block so that only people OVER 42 are unable to ride?
2. Write pseudo-code that describes a robot that will sit motionless and will look forward using the ultrasonic sensor. Every time the sensor responds with an object coming within 10 centimeters of the front of your robot, turn on the LED and leave it on until the object is removed.

3. **BONUS:** Write an `if` block that accepts a person's name. If the name provided is YOUR name, have it tell you how awesome you are. If it is someone else's name, tell them to mind their own business.

Lesson 3.3 - Switch

In Lesson 3.2, we covered the control capabilities provided by the `If` and `Else` blocks. `If` blocks work great in complex situations when you need to check multiple conditions with specific outcomes. The other type of control method we'll use in robot development is called a `Switch`. Switches are an interesting control structure because they operate on a single variable and perform specific actions when that variable has specific values.

The easiest way to wrap your head around the value of using switches is to take a simplistic program design and then implement your code using an `If` block and a `Switch`.

The easiest way to wrap your head around using switches is to work with a simple example of a small application and then build it using both `if` statements and with a `switch`.

For this example, let's assume we're building a simple program that takes an individual's age and then prints out a list of things that person can do at specific ages. In this example, we'll have our program print out what the individual is allowed to do at the following ages:

- Age 15 - Get a learner's driving permit.
- Age 16 - Get a driver's license.
- Age 18 - Buy lottery tickets.
- Age 25 - Get a major discount on car insurance.

If we were going to develop this using a bunch of `if` statements, our code would look something like this:

```
if ( age == 15 ) {  
    Serial.println("learner's permit");  
}  
if ( age == 16 ) {  
    Serial.println("driver's license");  
}
```

```

}
if ( age == 18 ) {
    Serial.println("lottery tickets");
}
if ( age == 25 ) {
    Serial.println("insurance discount");
}

```

This code works perfectly, but it doesn't read very well and **good programming is about writing code that is easy to read**. Computers don't need us to write beautiful code that's easy to read. Every single line of code you write is compiled into cryptic machine instructions. **Code is meant for humans** and it should always be your goal to write code that is easy to understand at a glance.

Rewriting this simple program using a `Switch` would condense these lines of code into something easier to read and easier to maintain.

The syntax for switch statements is very simple. It begins with the keyword "switch" and is followed by the variable you'll be switching on surrounded by parenthesis. As with our other blocks, it then begins with an open curly brace and ends with a closed curly brace.

```

switch (age) {
}

```

Inside the curly braces, you'll put each condition on a separate line with the keyword "case" followed by the value you expect and then a colon. After the colon, you will write the lines of code that you want to execute whenever that condition matches the value currently stored in that variable. After you've finished each line of code, you'll end that case with the keyword "break", which lets the processor know that you've finished that condition. If you do not put in the `break` instruction, the `switch` will move to the next condition and automatically execute that code as well. This will allow you to build systems that can flow down and perform multiple tasks at once.

The following lines of code replicate our age and permission system that we developed above.

```
switch (age) {
    case 15:
        Serial.println("learner's permit");
        break;
    case 16:
        Serial.println("driver's license");
        break;
    case 18:
        Serial.println("lottery tickets");
        break;
    case 25:
        Serial.println("insurance discount");
        break;
}
```

This use of `switch` highlights the strengths of using `switch` over `if` in certain situations. Bear in mind that this choice is almost entirely stylistic and it's up to you to decide which type of control structure you want to use in any given situation.

Switches do offer a few other stylistic niceties that come in handy from time to time. The coolest feature is a special case called “default” that you can use to catch values that didn't match one of your defined cases. We could add one to our age example to print a message that says “no permission change” for ages that don't offer new permissions. It would look something like this:

```
switch (age) {
    case 15:
        Serial.println("learner's permit");
        break;
    case 16:
        Serial.println("driver's license");
        break;
    case 18:
        Serial.println("lottery tickets");
        break;
    case 25:
        Serial.println("insurance discount");
        break;
    default:
        Serial.println("no permission change");
        break;
}
```

Practical Exercise: Put this code inside the `setup()` function and define an integer variable named "age". What is printed to the Serial Monitor if you set age to 16? What is printed if you set age to 21?

Switches are an important control structure because they narrow the focus down to one specific variable and make the possible options clear to future developers. There is very little ambiguity about your intent when you control your program using a `switch` statement. In later chapters when we dive into specific robot designs, we'll use switches to control the operational state of our robot and decide what logic to perform. Switches provide a very clean and logical way to branch your code execution paths from a single variable.

Exercise

1. Develop a new switch statement that will monitor an integer named "state" and will print "Attack!" when the value is 1, "Abort!" when the value is 2, and "Searching..." when the value is anything else.

Chapter 4 - Interacting with Components

At its core, robot development is fundamentally about receiving input from sensors and providing output to motors, buzzers, and lights.

In this chapter, we're going to explore the various types of components that come pre-installed on your MRK-2 and teach you how to write code to interact with them.

Lesson 4.1 - Digital and Analog Sensors

There are two primary types of components that we'll be using in our robot: digital, and analog. Digital components are the easiest to understand. They're either ON or OFF. There is no in between with digital components.

If we hook a digital signal up to an oscilloscope, it will look something like what you see in Figure 4.1.1 below.

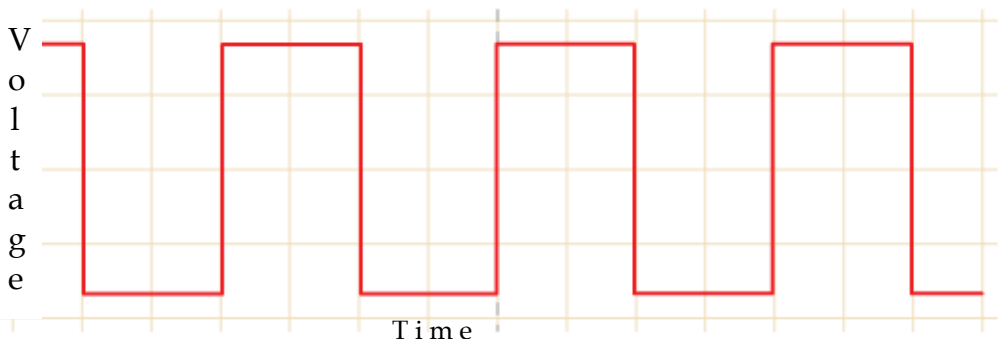


Figure 4.1.1 - Example of a digital signal read by an oscilloscope.

When working with digital components, we'll often refer to their current state as either HIGH or LOW. This corresponds with a setting of on or off. High merely refers to the digital device getting the maximum amount of electrical power it is able to safely accept.

A LOW signal to a digital device means it's getting either no power, or such little power the device itself registers the miniscule amount of voltage as effectively non-existent.

The “Blinking Light” code we used before was based around sending a digital signal to an LED. We used the Arduino native `digitalWrite()` function to turn the LED on and off by setting the value to either HIGH or LOW.

The code we used before is listed below.

```
void blink(int delayMs) {  
    digitalWrite(13, HIGH); // turn the LED on  
    delay(delayMs);  
    digitalWrite(13, LOW);  // turn the LED off  
    delay(delayMs);  
}
```

The LEDs that we're using require a certain voltage level before they'll turn on, though some LEDs do accept variable levels of power and can be dimmed. In the MRK-2, all of our LEDs need to be controlled using the `digitalWrite()` function.

An analog signal is a signal that can have a varied range of values, often seen between -255 and +255.

An analog signal looks somewhat like the image shown in Figure 4.1.2 below.

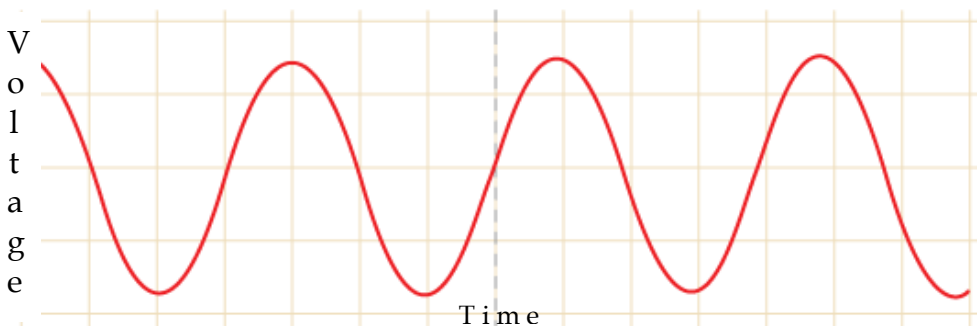


Figure 4.1.2 - Analog signal read by an oscilloscope.

Analog signals will have peaks and valleys that could correspond to a HIGH/LOW voltage value, but we're often most interested in what happens in between. The infrared sensors installed at the bottom-front of your MRK-2 are one of the most important analog sensors on the robot.

Once during every cycle of the main `loop()` function, these sensors sample the light reflecting from the ground below them and return a value somewhere between 0 and 1023, depending on the lighting conditions in the room. A low value indicates that your robot has discovered the white edge of the ring and should immediately take action to back up. A high value means that you're in the black part of the ring and should be on the hunt.

And with every rule, there's always one smart-alec that thinks they're going to be funny and have a component that requires both analog and digital input and output.

Both the motors and the IR line sensors on your MRK-2 require digital and analog input/output and we're going to cover both of these components in greater detail in later chapters.

Lesson 4.2 - Investigating Inputs

"I can show you the world. Shining, shimmering, splendid. Tell me, princess, now when did you last let your heart decide? I can open your eyes. Take you wonder by wonder. Over, sideways and under on a magic carpet ride. A whole new world. A new fantastic point of view. No one to tell us no or where to go or say we're only dreaming." -- ["A Whole New World" from Aladdin. Lyrics: Tim Rice.](#)

Without input, your robot is no different than a glorified toaster oven. Sure, you can have it sit there and blink angrily, but until we start using our sensors to reach out and explore the world around us, we're left with a battery powered brick with a dim red flashlight. If this was Blinking Robot League, you'd be sure to win first place.

The Ultrasonic Sensor

The most important sensor on your MRK-2 is the ultrasonic sensor. This is the component that lets you find and detect your opponents by sending out ultrasonic “pings” and counting the time it takes to hear the echo.

The ultrasonic sensor is one of the few components on our board that you’ll actively control as both an input and an output.

In our `setup()` function, we’ll need to specify the pins we’ll be using for input and output.

The “echoPin” on the MRK-2 is located at pin A0. This is the pin that will be constantly listening for a response from our “pingPin” which is located on pin 10.

Our setup function to prepare these two pins should look something like this:

```
#define echoPin A0
#define pingPin 10
void setup() {
    Serial.begin(9600);
    pinMode(echoPin, INPUT);
    pinMode(pingPin, OUTPUT);
}
```

The first thing that you should notice is the two lines at top that begin with the word `#define`. `#define` is a trick the compiler uses to swap out whatever value that you declare in the third position any time the word is detected. In programming, we’ll often place several `#define` constructs up at the top of a program to indicate that these are configuration values that can be changed.

Using `#define echoPin A0` will have the compiler replace all instances of the word “echoPin” with the value “A0” when the code is being compiled. This will transform the line of code that says `pinMode(echoPin, INPUT);` into `pinMode(A0, INPUT);` automatically.

Pro-Tip: Do not include a semicolon at the end of a #define declaration, as this will cause a compiler error.

In our `setup()` function, we have defined both of the pins we'll need for our ultrasonic sensor to properly operate. The `pingPin` will happily chirp out an ultrasonic pulse and the `echoPin` will sit there and wait until it hears the echo of ultrasound come back.

Let's write a quick function to capture that response and print it to our Serial Monitor.

```
#define echoPin A0
#define pingPin 10
void setup() {
    Serial.begin(9600);
    pinMode(echoPin, INPUT);
    pinMode(pingPin, OUTPUT);
}
void loop() {
    long duration = ping();
    delay(500); // We don't want this to run too fast.
    Serial.println(duration);
}
long ping() {
    long duration;
    digitalWrite(pingPin, LOW);
    delayMicroseconds(2);
    digitalWrite(pingPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(pingPin, LOW);
    duration = pulseIn(echoPin, HIGH);
    return duration;
}
```

Before we upload this sketch onto your MRK-2, let's briefly touch on some of the important parts. As we described earlier, the `setup()` function turns on Serial communication with a rate of 9600, then it sets the `echoPin` for INPUT and the `pingPin` for OUTPUT.

Inside our main `loop()`, we call the `ping()` function and store the result in a `long` variable named `duration`. We then manually introduce a delay of half a second to slow things down enough for you to see something more than just a giant whiz of numbers flying by on the Serial Monitor. After that, we print the number to the Serial Monitor.

In the `ping()` function, we create a different variable named `duration` to store our result and then turn off the `pingPin` by calling `digitalWrite(pingPin, LOW)`. This is not always necessary, but it's helpful to always clear your inputs before you try and use them.

The call to `delayMicroseconds(2)` slows things down just enough to make sure that everything has had enough time to reset before we actually blast out a signal using the ultrasonic sensor. Calling `digitalWrite(pingPin, HIGH)` slams a few volts into the ultrasonic sensor which sends out an inaudible chirp, flying away from the robot at the speed of sound. We then wait ten more microseconds before shutting down the `pingPin` by setting its voltage back to `LOW`.

The last step in this process is to read the time that it took to receive the echo by using the `pulseIn()` function on the `echoPin`. The value that `pulseIn()` returns is the amount of time in microseconds that it took to receive the response.

Upload your sketch to your MRK-2 and open up your Serial Monitor.

You should start seeing numbers scroll past on your Serial Monitor similar to what we see in Figure 4.2.1.

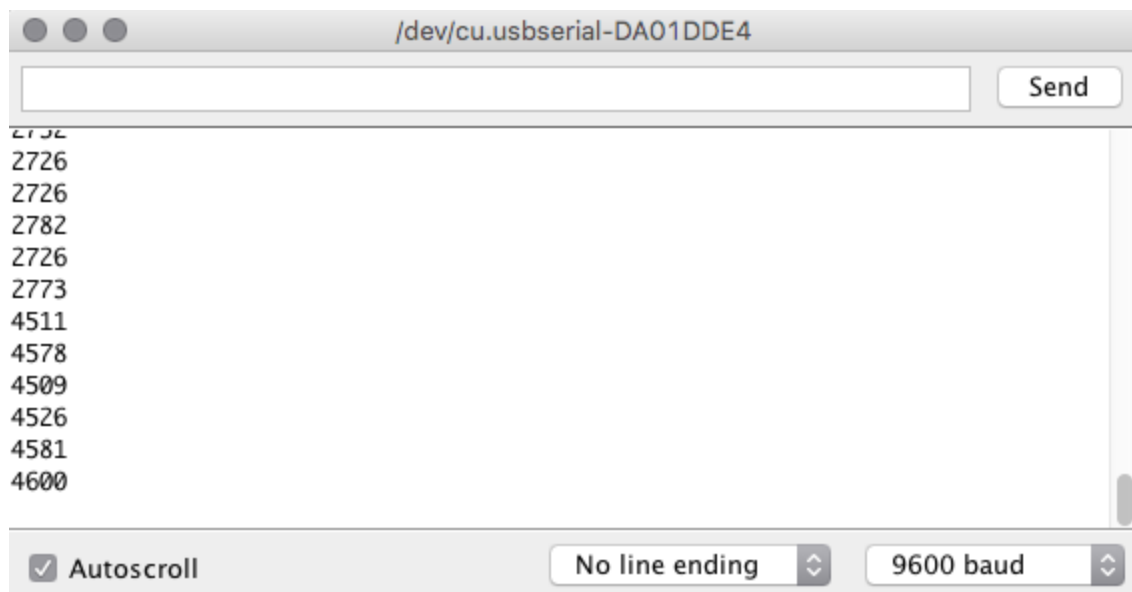


Figure 4.2.1 - Serial Monitor output of ultrasonic sensor output.

Practical Exercise - In [Lesson 2.3](#), we worked through an example function to convert microseconds to centimeters to solve this exact scenario. You now have an ultrasonic sensor that is happy to spit out microseconds, so modify your new function to return centimeters instead of microseconds.

Infrared Line Detection Sensors

There are two other sensors located at the bottom of your MRK-2 near the front edge of your blade. These simple sensors emit an infrared signal and read the reflection coming back from whatever surface your robot is currently sitting on.

The competitive Sumo Ring is designed specifically for this purpose. The middle part of the ring is painted a matte black that poorly reflects infrared. The outer edge of the ring is painted with a bright, glossy, white paint that does a much better job of reflecting infrared and should return much lower values to your line detection sensors.

The left sensor is assigned to pin A1 and the right sensor is attached to pin A2. Of special note is that the infrared emitter operates on a separate pin. This pin, located at 4, is a digital output that sends voltage to the IR emitter attached to each of the IR sensors.

As with the ultrasonic sensor we worked with in the previous section, we'll need to set up these sensors to accept input inside our `setup()` function. We'll also need to specify that we plan to use pin 4 for output.

```
#define leftSensor A1
#define rightSensor A2
#define IREmitter 4
void setup() {
    Serial.begin(9600);
    pinMode(leftSensor, INPUT);
    pinMode(rightSensor, INPUT);
    pinMode(IREmitter, OUTPUT);
}
```

Reading these sensors will give you your first opportunity to use the `analogRead()` function. If you remember from our earlier lessons, the difference between digital outputs and analog outputs is the type of value that the sensor is going to be providing. A digital sensor will tell you if it's "on" or "off" by returning `HIGH` or `LOW`.

These infrared detection sensors will give back a possible range of values between 0 and 1023. The high value of 1023 means that it received none of the infrared beam back. In practice, you'll only ever really see values in the high 1020s most of the time due to ambient infrared light scattered throughout the world.

When the sensor reaches a more reflective surface, like the white line that borders the Mini Sumo ring, it will respond with a value much lower than 1020. In optimum conditions, this number can sometimes be as low as 800.

To gain additional accuracy for these sensors, we must pass a signal of `HIGH` to the IR Emitter provided on pin 4. We'll want to always pass this `HIGH` value to the emitter at the start of every call to the `loop()` function so that our IR sensors will have a bright, stable source of infrared to capture.

Many Android-based smartphones and tablets will display an infrared light source when viewed through the built-in camera. Apple iPhones and iPads have an infrared filter in their camera that will hide the infrared light from your emitter.

Pro-Tip - Whenever you're about to compete at a new location, it's always a good idea to place your robot on the black and record the baseline values. Then place it on the white and record the values of the ring edge.

The code needed to read the values from these sensors is simple. Here's an example script you can place inside your `loop()` function to read the values and print them to the console.

```
void loop() {
  digitalWrite(IREmitter, HIGH);
  int rightInput = analogRead(rightSensor);
  int leftInput = analogRead(leftSensor);
  Serial.print("Left/");
  Serial.print(leftInput);
  Serial.print(" - Right/");
  Serial.println(rightInput);
  delay(500);
}
```



```
}
```

In this example, we're reading the values coming back from the `rightSensor` and `leftSensor` and storing them in the integer variables `rightInput` and `leftInput`, respectively. By using the `analogRead()` function, we're able to get whatever value the sensor provides back to us.

Compile and upload your sketch and examine the values being returned from a number of different surfaces.

The Push Button

The last major input that we're going to discuss in this book is the small push button mounted on the top of your MRK-2 board in the back. In competitive Sumo Tournaments, we use this button to start our game countdown, as the rules require a five second delay before a match can begin to give each player time to step away from the board.

Using the push button in our code is extremely easy, but it does require a special `pinMode()` setting that we haven't mentioned previously. All of the components we've discussed previously have been configured as either an `INPUT` or an `OUTPUT`. The push button is a switch that outputs a value of `HIGH` when it's not pressed and a `LOW` value when it is pressed. This is the reverse of the way our other components operate, and to get this to work we'll configure this pin inside our `setup()` function like this:

```
#define buttonPin 2
void setup() {
    Serial.begin(9600);
    pinMode(buttonPin, INPUT_PULLUP);
}
```

As with our other components, we `#define` the pin address of 2 to the word `buttonPin`. Inside `setup()`, we prepare the pin for use by calling the `pinMode()` function and passing in `INPUT_PULLUP` as the second argument. Using `INPUT_PULLUP` for this pin will reverse the normal way this component operates. With `INPUT_PULLUP`, the component will return a value of `HIGH` whenever the button is ***not*** pressed and will return a value of `LOW` when it is. This is the exact opposite of how most of our other components operate, and it's important to keep this in mind when you're writing code to work with buttons and switches.

Warning: If you try to use pin 2 with any value other than `INPUT_PULLUP`, it will not work. The author of this book spent two days pulling out his hair trying to figure out why the danged button wasn't working. Do not be like this author. Use `INPUT_PULLUP`.

One of the easiest ways to demonstrate the functionality of the push button is to write a simple script to turn on the LED whenever we press the button.

In [Lesson 2.2](#) we covered the LED with a simple blinking light demonstration. If you recall, our LED is a simple digital output connected to pin 13. Let's walk through the logic of what it is we want to accomplish before diving in and writing code.

For our setup, we'll want to configure pin 2 as `INPUT_PULLUP` and pin 13 as simple `OUTPUT`. In our main loop function we'll check the result of `digitalRead()` on pin 2. If the result of that call to `digitalRead()` is a 0, we'll call `digitalWrite()` with a value of `HIGH` to pin 13. It's important to note that once set, the LED will continue to receive power until a second call is made to `digitalWrite()` that returns its state to `LOW`. We'll need to use an `else` with our `if` block to reset it whenever the result of the `digitalRead()` is 1 and we know our button has been released.

With such simple requirements, we can quickly develop a simple program to perform this exact function. Copy the code below into your editor and send it to your MRK-2.

```

#define buttonPin 2
#define led 13

void setup() {
    Serial.begin(9600);
    pinMode(buttonPin, INPUT_PULLUP);
    pinMode(led, OUTPUT);
}

void loop() {
    if ( digitalRead(buttonPin) == 0 ) {
        // button is pressed, turn on LED
        digitalWrite(led, HIGH);
    } else {
        // button is not pressed, turn it off
        digitalWrite(led, LOW);
    }
}

```

That is all it takes to make use of your push button switch. In [Chapter 8](#) when we build our competitive robot, we'll need to use the push button to start the timer that will set your robot on a path to victory.

Exercises

1. What would happen if you changed `analogRead()` to `digitalRead()` for your IR sensors?
2. What would happen if you removed the `digitalWrite()` call that turns on the IR Emitter? What values do your infrared sensors report with the IR Emitter turned off?
3. **Bonus:** Rewrite the push button code example in a way that causes the LED to turn on when you push the button once and remain on until you've pushed the button a second time. Turn your LED switch into a toggled option rather than a simple press-and-release system.

Lesson 4.3 - Motors!

"The true measure of a man is not what he dreams, but what he aspires to be; a dream is nothing without action. Whether one fails or succeeds is irrelevant; all that matters is that there was motion in his life. That alone affects the world." -- Mike Norton, The White Mountain

It's time to make your robot do more than just sit there blinking. Your MRK-2 is designed to be a dual-tracked vehicle, similar to a modern battle tank. Each track can operate independently allowing for incredible control and precision movement.

Moving the left track in full speed reverse while having the right track moving a full speed forward will result in your robot spinning in a perfect circle. By controlling the speed and direction of each track, you gain incredible precision in driving the vehicle.

Do not let this power lull you into a false sense of security. These motors are one of the most challenging components in the MRK-2 and to write code to properly control them is one of the more interesting aspects of robot development.

Let's first take a look at how we set up the dual-pin configuration for one motor. The right motor has two pins that we're concerned with: pin 6 and pin 8. Pin 6 is an pulse-width modulation (PWM) capable pin. By fluctuating on and off very rapidly it imitates an analog voltage and controls the speed that the motor will attempt to turn. Pin 8 is a digital pin that will tell the motor which direction to turn (forwards or backwards).

```
#define rightMotorSpeed 6
#define rightMotorDirection 8
void setup() {
    Serial.begin(9600);
    pinMode(rightMotorSpeed, OUTPUT);
    pinMode(rightMotorDirection, OUTPUT);
    delay(1000);
}
```

Once you've configured your pins for OUTPUT, your right motor is ready to accept your control. If we want our right motor to begin spinning forward at maximum speed, we'll insert the following lines of code:

```
digitalWrite(rightMotorDirection, LOW);  
analogWrite(rightMotorSpeed, 255);
```

If we place these commands inside our `loop()` function, the right motor will spin along at it's highest speed until you disconnect the power.

If we want our robot's wheel to turn in reverse, we have to do something somewhat counterintuitive. We want to keep the speed set to 255, but we have to change the voltage to the `rightMotorDirection` pin to `HIGH`.

The only weird catch here is that the left motor is installed on your robot in the opposite direction. For the left motor, `HIGH` is forward and `LOW` is reverse, the exact opposite of your right motor.

Practical Exercise - Using the information you've just learned, implement the left motor in your sketch and have it set to full speed in reverse. Your robot should remain mostly in place as it spins around its center axis. The pin for `leftMotorDirection` is 4 and the pin for `leftMotorSpeed` is 9.

The next major task you'll want to accomplish when working with motors is to write a helper function that will make it easy to set the speed and direction for each track independently.

This will be your first major exercise, but we're going to walk through the steps required in pseudo-code before letting you loose on the Arduino IDE.

Exercise

Transform this pseudo-code into a working implementation of a function to handle setting the speed and direction for each motor.

```
FUNCTION setSpeed TAKES PARAMETERS, int motor AND int speed:
  IF motor IS 0, WE'LL SET THE SPEED OF THE RIGHT MOTOR
  IF motor IS 1, WE'LL SET THE SPEED OF THE LEFT MOTOR

  IF speed IS LESS THAN 0, WE WANT TO GO IN REVERSE
    IF motor is 0:
      SET motorDirection TO HIGH
    IF motor is 1:
      SET motorDirection to LOW
    SET motorSpeed to abs(speed)

  IF speed is GREATER THAN 0, WE WANT TO GO FORWARD
    IF motor IS 0:
      SET motorDirection to HIGH
    IF motor IS 1:
      SET motorDirection to LOW
    SET motorSpeed to speed
```

1. Write a function called `setSpeed(int motor, int speed)` and have it function according to the specification roughly defined in that pseudo-code. **Hint:** The `abs()` function will take whatever negative number you provide and turn it into a positive number. This will allow you to call `setSpeed(1, -255)` and have it know to go full speed in reverse. Make sure all “reverse” calls have the speed wrapped in the `abs()` function in your call to `analogWrite()`.
2. Test your newly written function inside your `setup()` function by having your robot do the following three tasks:
 - a. Drive full speed ahead.
 - b. Drive full speed in reverse.
 - c. Drive in a complete loop in place, with one motor going forward and the other going backward.

Chapter 5 - Fun with Functions

Lesson 5.1 - Don't Repeat Yourself

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. -- [The Pragmatic Programmer](#)

One important concept that you'll hear in software engineering is "Don't Repeat Yourself", sometimes just referred to as DRY. The basic concept behind this is that you should only ever write a specific line of code once. If you find yourself writing the same lines of code over and over again, it's absolutely time to refactor and place those lines of code inside of a reusable function.

We've already seen the value of functions in [Chapter 2](#) when we briefly discussed moving your "blinking light" code out of the main loop and placing it in its own function. We then tackled a more realistic robotics challenge by creating a function to convert the amount of time it took our ultrasonic sensor to "ping" back to us and converted that time in microseconds to centimeters.

Your most recent exercise at the end of [Lesson 4.3](#) is a perfect example of how we can use functions to simplify common operations and clean up what would be an otherwise messy codebase. Had we not created a simple function to set the speed and direction for your motors, any place in your code where you wanted to make a change to the motors would have involved manually writing in code like this:

```
// ATTACK!
digitalWrite(rightMotorDirection, LOW);
analogWrite(rightMotorSpeed, 255);
digitalWrite(leftMotorDirection, HIGH);
analogWrite(leftMotorSpeed, 255);

// Resume searching
digitalWrite(rightMotorDirection, LOW);
analogWrite(rightMotorSpeed, 255);
digitalWrite(leftMotorDirection, LOW);
analogWrite(leftMotorSpeed, 255);
```

With our function, our code can be simplified to be cleaner and easier to maintain and quickly understand.

```
if ( foundOpponent ) {  
    // ATTACK!  
    setSpeed(0, 255);  
    setSpeed(1, 255);  
} else {  
    // Resume searching  
    setSpeed(0, 255);  
    setSpeed(1, -255);  
}
```

Those are perfect examples of the DRY philosophy you should always strive for. Remember, code is meant to be read. The computer goes out of its way to translate the beautiful words you write into boring Machine Language instructions for the compiler to understand. **Be expressive!**

In this lesson, we're going to briefly touch on some of the more advanced concepts that you'll see later on and the ways we can choose to simplify our programs using functions.

Lesson 5.2 - Musical Programming with Functions

You've officially survived the first half of your Sumo Robot Programming course. It's time we celebrated!

In this lesson, we're going to use the on-board buzzer to make sweet, sweet music. Well, very tinny sounding, sweet, sweet music.

Generating specific tones for your MRK-2 is a simple matter of sending the correct frequency to that pin as output. In this lesson, we'll also learn how to import an external library containing most of the commonly used pitches and the corresponding note.

As with all components, you first need to configure the buzzer pin for output. On the MRK-2 board, the buzzer is located at pin 3, so we'll need to configure it like this:


```

#define buzzer 3
void setup() {
    Serial.begin(9600);
    pinMode(buzzer, OUTPUT);
}

```

With our buzzer configured to receive output, let's try and send it a few solid tones.

```

void loop() {
    tone(buzzer, 1397, 250); // Freq. 1397 = F6
    delay(1000);
}

```

Running the above code will generate a delightfully shrill beeping sound, perfect to turn your robot into the world's most annoying alarm clock. The first line of code: `tone(buzzer, 1397, 250)` tells the processor to send a Pulse-Width Modulation (PWM) signal to the buzzer component at a frequency of 1397Hz for 250 milliseconds. We then add a `delay(1000)` line to turn the constant stream of noise into a steady beep every second.

In the interest of inspiring our future 8-bit musical prodigies, let's first create a simple library that equates frequencies into musical notes. After that, we can write a simple function to play some music in a format that will be easier to work with.

First, open up a new tab in your Arduino IDE by clicking the down arrow to the far left of the top of your editor window:



Figure 5.2.1 - Creating a new tab in the Arduino IDE

Select “New Tab” and you should see an empty tab open up right next to your current sketch. The nice folks at Arduino were kind enough to provide us with a full list of common musical notes and their associated frequency, so let’s open up a browser and grab their list. Go to <http://bit.ly/ArduinoPitches> and select all of the text on that page and copy it into your new, empty tab.

Once all of those constants have been pasted, save that sketch in your current directory under the name “Pitches.h”. Go back to your main tab and replace your setup code with this:

```
#include "Pitches.h"
#define buzzer 3
void setup() {
    Serial.begin(9600);
    pinMode(buzzer, OUTPUT);
}
```

The `#include` directive tells the compiler to grab the file specified and compile that before it begins to compile your code. Anything you’ve added with an `#include` will be available to your program. From now on, any time you want to play a specific note, you can reference that note by the name it was defined as inside `Pitches.h` and won’t have to refer to it by frequency.

Let’s test this out with the first little bit of Twinkle Twinkle Little Star:

```
void loop() {
    tone(buzzer, NOTE_C5, 250);
    delay(500);
    tone(buzzer, NOTE_C5, 250);
    delay(500);
    tone(buzzer, NOTE_G5, 250);
    delay(500);
    tone(buzzer, NOTE_G5, 250);
    delay(500);
    tone(buzzer, NOTE_A5, 250);
    delay(500);
    tone(buzzer, NOTE_A5, 250);
    delay(500);
    tone(buzzer, NOTE_G5, 500);
    delay(1000);
}
```

Uploading this sketch should cause your MRK-2 to start humming the opening bars of the familiar song. Each call to `tone()` tells the buzzer which note to play and for how long. The `delay()` after each call to `tone()` separates the notes and helps make them sound distinct.

With our powerful understanding of arrays and functions, we can make writing songs a much less tedious process. Let's lay out some requirements and begin developing our musical helper functions.

Functional Requirements

In engineering, whenever you're given a programming task, the first place you should start is with a list of functional requirements. These requirements lay out exactly what your program is supposed to do and help you know when your code has met all of your required objectives.

For this exercise, our functional requirements are very simple.

1. Develop a function that will take 3 arguments and will play the note provided for the duration specified and then pause for the time specified by the rest duration.
 - a. a note
 - b. a duration to play the note
 - c. a duration to rest between notes
2. Define an array format that will contain all of these values.
3. Loop through the array to play the song.

If this was a professional programming assignment and these were the requirements you were given, your next step would probably be to take these requirements and break them down into programming tasks. As this is a simple exercise, the requirements basically already are arranged in a way that lends itself to being developed.

Task 1 - The Play Function

Our first requirement specifies that we must write a function that accepts three arguments and plays the note provided for the proper length of time and then pauses for the time specified.

The requirements do not specify any type of data being returned, so we can safely assume that this function will be defined with a void return type. Let's write the declaration together, and then you'll write the rest.

```
void playNote(int note, int duration, int rest) {  
    // Write in your code here.  
}
```

Task 2 - The Array Format

As Linus Torvalds said, *“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.”*

The easiest approach that lends itself to this type of problem is a simple, one-dimensional flat array of integers. If we write our code like the array listed below, it will be easy for our composer to easily write music in a style that isn't overly cumbersome.

```
int notes[] = {  
    // NOTE, DUR, REST  
    NOTE_C5, 250, 500,  
    NOTE_C5, 250, 500,  
    NOTE_G5, 250, 500,  
    NOTE_G5, 250, 500,  
    NOTE_A5, 250, 500,  
    NOTE_A5, 250, 500,  
    NOTE_G5, 500, 1000  
};
```

Task 3 - Loop Through the Array to Play the Song

Looping through a one-dimensional array of integers like this is a trivial task that we can handle in just a few lines of code. The secret to this is that we really want our loop to skip to every fourth element. Inside the loop we'll capture the three elements we care about and pass those to our function `playNote()`.

The tricky part about working with arrays of numbers in the MRK-2 is that the `sizeof()` function returns the size of the array in bytes.

If you run add this line of code inside your `setup()` function:

```
Serial.println(sizeof(notes));
```

It would spit out the number 42, which is much larger than our array of 21 elements. The reason should be obvious. Each integer in 16-bit AVR-C is 2 bytes long. The safe way to calculate the total size of the array is to use this formula:

```
int arrayLength = (sizeof(notes)/sizeof(int));
```

Dividing the `sizeof()` value of our array of integers by what the processor knows is the size of a standard integer will always return back the correct size of the overall number of elements in this array.

In our data structure, we defined each musical note with a duration to play the note along with a rest. We know that every fourth element in the array is going to be a new musical note, which allows us to cleverly skip over three elements in every pass through our loop and efficiently play each note.

```
int arrayLength = (sizeof(notes)/sizeof(int));
for ( int i = 0; i < arrayLength; i = i + 3 ) {
    playNote(notes[i], notes[i+1], notes[i+2]);
}
```

Even though we're skipping three array elements with every pass of the loop, the integer counter `i` still knows where it is when it begins to execute the code inside curly braces. By referencing `notes[i]`, `notes[i+1]`, and `notes[i+2]`, we've effectively grabbed all three values that our `playNote()` function requires in one pass of the loop.

Exercises

1. Put together all you've learned in the previous 3 tasks to implement a functional music player.
2. **Bonus** - Write a new little melody and enjoy your own private concert.

Lesson 5.3 - Your First Custom Robotics Library

At some point in your software engineering career, you're going to want to take the DRY philosophy to another level. You'll have written such beautiful code that you have no choice but to take out all the good bits and put it into a library so the whole world can use your wonderful software.

In this lesson, we're going to take the work that you did creating a set of functions to control your robot's motors and put all that code inside a simple, reusable library.

If you don't remember, back in Chapter 1 we defined a library as *a pre-built set of functions that you can import into your software to make **complicated** programming tasks **easier***. Libraries provide reusable blocks of functionality that you can easily embed into your programs to perform complex operations.

While that may sound complicated, in reality, most libraries are simple programs just like the ones you've been writing in previous chapters.

How simple should they be? A good rule of thumb is that a library should only do **one** thing and should try to do that **one** thing as well as it can.

You may be tempted with your soon-to-be-developed library-writing powers to make one giant library that controls your motors, buzzers, LEDs, and ultrasonic sensor. This would be a big mistake and would make the Library useless to anyone that doesn't have your exact same product configuration. In Computer Science terms, your libraries should almost always be the smallest possible unit of sharable work.

In the previous lesson, we created a simple library that defined a list of frequencies that your code was able to use to make music creation an easier process. We did this by tossing a long list of preprocessor definitions inside a file called Pitches.h and added it to our code using the `#include` declaration.

To write a functional library, we're actually going to need to write two separate files. The header file, usually included with a `.h` file extension, is a file that sets up the functional capabilities of the library. A second file, written with a `.cpp` extension, actually contains the meat of the operations.

Inside the header file, we're going to set up the structure of how your library will interact with your program. We'll do this by first creating a class and then defining the public and private members of that class. Let's start by jumping in and examining the required structure of your class.

Open a new tab in your Arduino IDE and name this file `Motor.h`.

The first thing you should always add to your header file is a comment describing what the Library is for, who wrote it, and when. It's also good form to include a license if you intend for other people to use your code.

```
/*  
*****  
Motor.h - A Library for Controlling the Motors of the MRK-2  
Author - Eric Ryan Harrison <me@ericharrison.info>  
This library is released into the public domain.  
*****  
*/
```

Immediately after your comment block, you must include the following three lines:

```
#ifndef Motor_h  
#define Motor_h  
#include "Arduino.h"
```

The first two lines keep our library safe from being included twice by accident and the third line imports all of the standard Arduino constants and data types. `Arduino.h` is automatically included behind the scenes in every sketch that you upload to your robot, but we have to manually include it whenever we write a library.

At the very bottom of your library on the last line of code, you'll have to add this:

```
#endif
```

The `#ifndef` declaration at the top basically says "if not already defined:", and we have to end this at the end of the file with `#endif` to close off that condition.

Your entire header should look like this before any code is added:

```
/*
Motor.h - A Library for Controlling the Motors of the MRK-2
Author - Eric Ryan Harrison <me@ericharrison.info>
This library is released into the public domain.
*/
#ifndef Motor_h
#define Motor_h
#include "Arduino.h"

// YOUR CODE WILL GO HERE

#endif
```

It's time to jump into Object Oriented Programming for the first time. In simplistic terms, an Object is simply a special type of variable that can encapsulate variables and functions. Inside this encapsulation, we can choose to make the variables and functions of the object either public or private.

If a function or variable is listed as public, that means any part of your code can touch that function or change the value of that variable. If it is private, your code will not be able to see it.

Inside libraries like the one we're about to write, we create the potential for Objects by writing Classes. A Class is simply a definition for how an object is expected to behave. Inside the header file, we'll only be writing the blueprint of that class. Inside the .cpp file we're going to write next, we'll actually implement the functions we've defined here.

Let's start by thinking about the requirements that our `Motor` class will need to meet. Previously, we wrote a simple function called `setSpeed()` that took an argument for which motor to control and a speed from -255 to 255.

While that design worked well for early prototyping, now that we're getting ready to start developing a world-class competitive robot, we need to think about what type of code we'll want to write to drive our robot.

When we've finally implemented our library, we'll have an Object at our disposal called `motor`. It would make sense to expose two public functions called `motor.left()` and `motor.right()` and have those functions just take a speed argument. We are going to have to remember that our motors each have two pins, an analog and a digital, and we'll need to remember to have some way of setting those up in our `setup()` function.

To define a class inside a header file, we need to write a class declaration like this:

```
class Motor {
public:
    Motor();
    void setupRight(int rightSpeed_pin, int rightDirection_pin);
    void setupLeft(int leftSpeed_pin, int leftDirection_pin);
    void right(int speed);
    void left(int speed);
private:
    int _rightMotorSpeedPin;
    int _rightMotorDirectionPin;
    int _leftMotorSpeedPin;
    int _leftMotorDirectionPin;
}
```

This class declaration may seem strange, but it has everything we need to implement a really powerful Motor control library for our MRK-2 robot.

Inside the `class Motor { }` block, you should see the `public:` and `private:` keys we've defined that let our program know what is available to be accessed from outside programs. We've made all of the pin assignments private because we don't really need our program to muck around with those once they've been configured.

At the very top of the `public:` list of function signatures, you'll see one special signature called `Motor()` that doesn't have a return type specified. This is the class constructor and must have the same name as your class. If your class was called `LaserGun`, your class constructor would have to be called `LaserGun()`. Many libraries will have the class constructor do the primary configuration for the pin assignments, but since our motor class is somewhat complicated and has four separate pins, we're going to leave that constructor without any arguments and manually assign the pins in separate functions.

In the `private:` list, you will see we've left room for four variables called `_rightMotorSpeedPin`, `_rightMotorDirectionPin`, `_leftMotorSpeedPin`, and `_leftMotorDirectionPin`. By convention, many developers put an underscore before private variables so that any future programmer will know that the variable is intended to be private, but this is optional.

Pro-Tip: The underscore is never optional. Always include it in your private variables. ;)

Your entire header file should now look like this:

```
/*
*****
Motor.h - A Library for Controlling the Motors of the MRK-2
Author - Eric Ryan Harrison <me@ericharrison.info>
This library is released into the public domain.
*****
*/

#ifndef Motor_h
#define Motor_h
#include "Arduino.h"

class Motor {
public:
    Motor();
    void setupRight(int rightSpeed_pin, int rightDirection_pin);
    void setupLeft(int leftSpeed_pin, int leftDirection_pin);
};
```

```

        void right(int speed);
        void left(int speed);
    private:
        int _rightMotorSpeedPin;
        int _rightMotorDirectionPin;
        int _leftMotorSpeedPin;
        int _leftMotorDirectionPin;
    }

#endif

```

Now that we've defined what our Motor class is going to look like to the outside world, it's time to open up a new tab in your Arduino IDE and create a new file named "Motor.cpp". Now the fun begins!

As with your header file, it's always a good idea to include a large comment at the top letting future developers know what this code is for.

```

/*****
Motor.cpp - A Library for Controlling the Motors of the MRK-2
Author - Eric Ryan Harrison <me@ericharrison.info>
This library is released into the public domain.
*****/

```

Next we'll need to manually include both Arduino.h and our newly created Motor.h.

```

#include "Arduino.h"
#include "Motor.h"

```

Now all that's left to do is implement the functions that we've defined previously in our header file.

We start with our class constructor, `Motor()`. As mentioned previously, in our Motor class, our constructor doesn't do anything because we're going to be handling the pin assignments manually.

All we need to write to implement a minimalistic constructor is to write the following lines of code.

```

Motor::Motor() {
    // Intentionally do nothing
}

```

Every function that you write inside your class is going to follow the same unusual format:

```
ReturnType Classname::FunctionName() {  
    // code goes here  
}
```

The exception to this rule is the class constructor, which has no return type.

It's generally best to start at the top of your list of public method definitions and work your way down, so now that we've coded our class constructor method, lets move on to our first public function: `setupRight()`.

This function's purpose is to take the two pin assignments your code will pass in as arguments and then store those values in the private variables we've created for them. After the values are stored in our private variables, we'll set the `pinMode()` to `OUTPUT` for each and activate them so that our other functions can use them.

```
void Motor::setupRight(int rightSpeed_pin, int rightDirection_pin) {  
    // Store the pin in a private variable for later use  
    _rightMotorSpeedPin    = rightSpeed_pin;  
    _rightMotorDirectionPin = rightDirection_pin;  
  
    // Set up the pins for output  
    pinMode(_rightMotorSpeedPin,    OUTPUT);  
    pinMode(_rightMotorDirectionPin, OUTPUT);  
}
```

That's all it takes. Now your right motor is ready to force your opponents out of the ring.

Practical Exercise - Write the code to finish the `setupLeft()` function.

Now that we have both of our motors ready to accept input, we need to write the `right()` and `left()` functions.

In Lesson 4.3, we learned that our motors are controlled via two pins. The digital direction pin controls the motors spin direction, with `HIGH` being reverse on the right motor and `LOW` being forward on the right motor. For the left motor, everything is reversed with `HIGH` being forward and `LOW` being reverse. The other pin we use is the analog speed pin that controls the speed from 0 to 255. In our `setSpeed()` function that we originally developed, we abstracted away the need to know which direction we were going by having the digital pin controlled entirely by the value provided for the speed. For numbers below 0, we set the direction pin to either `LOW` or `HIGH` (depending on the motor we're working with) and set the analog speed with a positive number returned from the native `abs()` function.

We are going to do the exact same thing here, so the code should look extremely familiar. The only thing that is different is how we define our function so that it exists as a member of the `Motor` class.

```
void Motor::right(int speed) {
    if ( speed < 0 ) {
        digitalWrite(_rightMotorDirectionPin, HIGH);
        analogWrite(_rightMotorSpeedPin, abs(speed));
    } else {
        digitalWrite(_rightMotorDirectionPin, LOW);
        analogWrite(_rightMotorSpeedPin, speed);
    }
}
```

That's all there is to it. In a perfect world, you would want to add some error handling for cases in which the speed was either above 255 or below -255, but we needed to leave something for you to do in the Lesson Exercises.

Practical Exercise - Implement the `left()` function of the `Motor` class.
Remember that the left motor uses the opposite values for forward and backward in the direction pin.

Your class is now completely finished! You've written your first reusable library and you're ready to take on the world, one tiny robot at a time.

To implement your new library and class in your main program, you'll need to include your header file at the top of your program and then create an instance of the object.

The following simple code sketch below will show you the basics.

```
#include "Motor.h"

Motor motor;

void setup() {
    motor.setupLeft(5, 7);
    motor.setupRight(6, 8);
    motor.left(255);
    motor.right(-255);
}
void loop() {}
```

WARNING - Since our constructor does not take any arguments, do not initialize your new motor object with `Motor motor()` ; or that will throw tons of random compiler errors. If you write a constructor that does take an argument, then you would include parenthesis in your variable declaration like `Motor motor(MAX_SPEED)` ; or something along those lines.

Exercises

1. Write in some error checking code to `Motor::left()` and `Motor::right()` to make sure that `int speed` is never greater than 255 or less than -255.
2. Write a new function called `Motor::attack()` that instantly drives the robot ahead at full speed.
3. Write a new function called `Motor::abort()` that instantly drives the robot in reverse at full speed for 3 seconds. (You don't want to drive off the board, do you?)

Chapter 6 - Bot: Line Following

Now that we've covered all of the basics, let's dive in and start building some functional robots that actually perform a useful action. One of the competition types that are popular in Sumo Robot League events is the Line Following Competition and we'll try to build a competitive line following robot in this chapter.

In the Line Following Competition, your robot is placed at the start of a white line on the floor. The line has several turns and the objective is to have your robot reach the end as quickly as possible while keeping the white line centered underneath.

Breaking Down the Requirements

In the first five chapters of this book, you've learned all of the programming techniques that you'll need to build this autonomous robot, so in this lesson we're going to focus on defining the requirements and planning the implementation details that we're going to use to build this robot.

"Everyone has a plan 'till they get punched in the mouth." -- Mike Tyson

Most projects of any size start with a simple set of requirements that we'll use to determine the scope of work. Looking over the description for the Line Following Contest shows two major requirements:

1. Robot must straddle and follow the line of tape on the ground.
2. Robot must be able to adjust direction when tape direction changes.

The sub-requirement is that we want our robot to move as fast as possible while still maintaining accuracy.

Let's take a quick pass at writing some pseudo-code to think through the logic for how our robot will operate.

```
LOOP FUNCTION:
  SET LEFT MOTOR SPEED
  SET RIGHT MOTOR SPEED
  GET LEFT IR SENSOR VALUE
  GET RIGHT IR SENSOR VALUE
  IF LEFT IR SENSOR VALUE SEES TAPE LINE
    DECREASE LEFT SPEED VARIABLE BY 10%
  IF RIGHT IR SENSOR VALUE SEES TAPE LINE
    DECREASE RIGHT SPEED VARIABLE BY 10%
```

Let's walk through our first pass at the pseudo-code and think about how this program will function.

Our first assumption is that we're operating inside a continuous loop. We want to perform all of these actions every time the loop runs.

Our next assumption is that we should be moving forward at around half our maximum speed. We don't want to start out moving so fast that our robot doesn't have enough time to react to input from the sensors, so our first version will work from the assumption that half speed is probably a good starting point.

After we've actually implemented the code and seen the robot in action, we can tweak these values and improve our overall performance. In engineering, this type of iterative development is common when you're working with unknown performance characteristics.

Once we've started our robot on its forward path, we'll want to read the current sensor values from both the left and right IR sensors, just like we covered in [Lesson 4.2](#). As soon as we've read those values, we need to check to see if our sensor thinks we're touching a line. If we are, the smart thing to do is to decrease the motor speed on the side of the robot that had a sensor detect a white line and slightly increase motor speed on the opposite side. That should be enough to straighten your robot out and get it back on track.

Let's implement this code now and test out our theory with a straight line of white masking tape on a darker surface. (A light surface and dark tape also works, but you'll have to reverse the logic shown below.) For now, we'll test this on a simple straight line to make sure our assumptions for robot course correction are accurate. We'll also be able to use our newly developed Motor library to easily control our motors and speeds.


```

#include "Motor.h"

#define leftSensor A1
#define rightSensor A2
#define IREmitter 4
#define MOVE_SPEED 150
#define LINE_VALUE 100
#define DELAY_TIME 10

int leftSpeed = MOVE_SPEED;
int rightSpeed = MOVE_SPEED;

Motor motor;

void setup() {
    Serial.begin(9600);
    pinMode(IREmitter, OUTPUT);
    pinMode(leftSensor, INPUT);
    pinMode(rightSensor, INPUT);
    motor.setupLeft(5, 7);
    motor.setupRight(6, 8);
}

void loop() {
    digitalWrite(IREmitter, HIGH);
    motor.left(leftSpeed);
    motor.right(rightSpeed);
    delayMicroseconds(1);
    int leftReading = analogRead(leftSensor);
    int rightReading = analogRead(rightSensor);
    if ( leftReading < LINE_VALUE ) {
        leftSpeed = leftSpeed*9/10;
    } else {
        leftSpeed = MOVE_SPEED;
    }
    if ( rightReading < LINE_VALUE ) {
        rightSpeed = rightSpeed*9/10;
    } else {
        rightSpeed = MOVE_SPEED;
    }
    delay(DELAY_TIME); // give our motors time to turn
}

```

How did your robot perform? Did it successfully straddle and follow the straight line? Try it now on a line with some curves and see how it handles bigger changes in direction.

Let's examine this code sample in detail to make sure we fully understand what is going on in this complicated example before we move on to making improvements.

Definitions

This program begins in the same way that we've been writing our programs previously, with a large section of definitions that give us some flexibility in configuration.

```
#include "Motor.h"

#define leftSensor A1
#define rightSensor A2
#define IREmitter 4
#define MOVE_SPEED 150
#define LINE_VALUE 100
#define DELAY_TIME 10

int leftSpeed = MOVE_SPEED;
int rightSpeed = MOVE_SPEED;
```

We begin on the first line by importing our Motor control library, `Motor.h`. This will give us the ability to easily control our motors. By convention, it's encouraged to always put library `#include` declarations at the very top of your programs. This will make it easy for future programmers to immediately know what capabilities you've included that were brought in from external sources.

Immediately following your library inclusion, we define a list of constants that we want to make configurable. The first three definitions specify which pins the IR sensors use, while the other three are our custom configuration options for this program.

We start by defining our assumption for how fast we want our robot to move by setting our default movement speed to 150, which is roughly half of the maximum motor speed. We also define the value we expect to receive whenever our IR sensor drives over the white line of the tape, aka `LINE_VALUE`. In this example, we set it to 100, but you should test the output of your IR sensors in your specific environment. If you used a black electrical tape on a white tile floor, you would need to completely reverse this logic (the two lines with `< LINE_VALUE` would become `> LINE_VALUE`) and probably to change `LINE_VALUE` to 500.

The last value that we define is a delay time which should slow down the robot operations enough to give the motors enough time to alter their course.

After making these definitions we declare two integer variables one for each motors speed (left and right) and initialize them to the MOVE_SPEED value we defined a few lines earlier.

Practical Exercise - Modify some of these values and watch how the robot performs. Change the DELAY_TIME to be higher and lower and tweak the MOVE_SPEED. Can you find a sweet spot between the movement speed you've specified and the amount of time you delay between each loop cycle?

The Setup

The next major section of code is the setup() function and our global motor object creation.

```
Motor motor;
void setup() {
    Serial.begin(9600);
    pinMode(IREmitter, OUTPUT);
    pinMode(leftSensor, INPUT);
    pinMode(rightSensor, INPUT);
    motor.setupLeft(5, 7);
    motor.setupRight(6, 8);
}
```

You may be wondering why the line reading “Motor motor;” is being included here even though it’s clearly outside the void setup() function. By instantiating this object outside the function, we’ve made it globally available to the entire program. If we’d defined our new motor inside the setup() function, only the setup() function would be able to use the program and our attempts to control the motors inside the loop() function would generate errors.

In Figure 6.1 below, you can see an example error message that is caused by trying to use a variable outside of the scope in which it was defined.

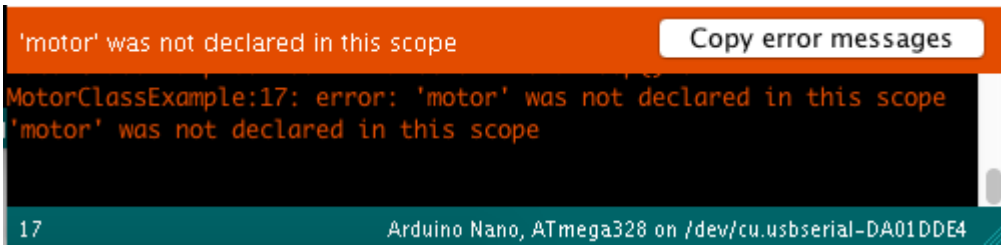


Figure 6.1 - Example of a variable scope error.

Scoping rules can be rather complex, but the simplest way to think of this is to remember that any variable defined within a block of curly braces, { and }, will only be available inside those curly braces. If you want your variables to be globally available, you must define them outside of your functions. Variables defined inside of a function will only be available to that specific function.

After our `motor` object creation, we enter our main `setup()` function. All of this should be second nature by now, beginning with setting the communication rate to our Serial Monitor and moving down through pin assignments.

The last thing we do in our `setup()` function is configure our motor object calling the `motor.setupRight()` and `motor.setupLeft()` functions. After this is complete, our robot is fully configured and ready to enter its main operational loop.

The Loop

As we've seen, the main `loop()` function is where the meat of our programs reside. This robot is no different, with the main loop containing the entirety of the logic we want our robot to follow.

```
void loop() {
    digitalWrite(IREmitter, HIGH);
    motor.left(leftSpeed);
    motor.right(rightSpeed);
    delayMicroseconds(1);
    int leftReading = analogRead(leftSensor);
    int rightReading = analogRead(rightSensor);
    if ( leftReading < LINE_VALUE ) {
        leftSpeed = leftSpeed*9/10;
    } else {
```

```

        leftSpeed = MOVE_SPEED;
    }
    if ( rightReading < LINE_VALUE ) {
        rightSpeed = rightSpeed*9/10;
    } else {
        rightSpeed = MOVE_SPEED;
    }
    delay(DELAY_TIME); // give our motors time to turn
}

```

We begin by turning on the `IREmitter` so that our infrared sensors will have a good strong source of infrared to read. Leaving this out will have a negative impact on the values that the analog inputs will receive as we discovered in the exercises in [Lesson 4.2](#).

Once the `IREmitter` is on, we move on to the next lines that turn on the motors and get them moving forward at half speed.

```

motor.left(MOVE_SPEED);
motor.right(MOVE_SPEED);

```

This code should be familiar to you and shouldn't contain any real surprises. We're setting our movement speed to equal the value that we defined at the top of the file.

To allow our robot to begin to move, the next line has a very briefly delay, one microsecond. This also allows any voltage irregularities or noise from firing up the motors to settle out so that we get a good reading from the infrared sensors. The next two lines of code capture the values from the front two IR sensors. As we've previously learned, these sensors respond with a number between 0 and 1023 depending on the amount of infrared that the sensor is able to detect. A low number represents a larger source of infrared light. A high value such as 500 or 600 is generally a safe bet that your robot is driving over a dark surface that is fairly infrared absorptive. Lower numbers indicate that the material directly beneath the robot is infrared reflective.

Pro-Tip: Not everything that looks black to the eye is black to infrared. We've found that many inkjet printers create a visible black that actually reflects almost as much infrared as blank white paper.

After we've captured the values, the next thing we want to do is examine them and adjust our heading to try and keep the white line directly beneath us. In our pseudo-code, we made the assumption that a simple decrease of 10% to the motor on the same side as the triggered sensor will correct the direction, especially when repeated over and over until the sensor is clear of the line. The reason for this logic is simple: we want to rotate our robot in a direction that will take it away from the line that it discovered.

```
if ( leftReading < LINE_VALUE ) {  
    leftSpeed = leftSpeed*9/10;  
} else {  
    leftSpeed = MOVE_SPEED;  
}
```

If the left IR sensor does not see the line, then the left motor speed is set to the default speed, aka MOVE_SPEED.

Programming is about working with the inputs and data that you've been given. This robot was designed on the assumption that your robot would be driving along a dark-colored floor with a line of more reflective white masking tape. In that scenario, your IR sensors would return a value close to 500 when driving over the floor and a lower value when such as 70 when encountering the white masking tape. With this case, our logic is sound. We "detect" a line when the sensor reading is less than the LINE_VALUE that we defined.

```
if ( leftReading < LINE_VALUE ) {
```

If our environment was configured differently, such as a light tile or linoleum floor with a line of black electrical tape, this logic would need to be completely reversed. Doing so would be as simple as changing your two `if` blocks to read:

```
if ( leftReading > LINE_VALUE ) {  
}  
if ( rightReading > LINE_VALUE ) {  
}
```

In competitive programming tournaments, it's always wise to take your pre-built code and configure your values and thresholds to ensure that your robot's operational performance matches the real-world conditions.

The last line of code we include in our loop is a call to the `delay()` function. This call stops all other robot operations until it completes and gives our robot enough time to move away from the line and not overreact. Changing the number of milliseconds we have the robot delay will change the total distance that our robot turns before it resumes primary forward motion.

In this chapter, we've thoroughly covered the primary design and implementation of a simple line-following robot. All of the techniques covered should have prepared you for building a robot to follow a line and correct course if necessary.

Exercises

1. Set up a more complicated course for your robot to navigate with several turns at different angles. How did your robot perform? Can you improve on your algorithm to make your robot better cope with certain types of paths?
2. How fast can your robot navigate? Ramp up the speed until you're navigating these paths as quickly as possible.

Chapter 7 - Bot: Robotics for Fun

The next robot that we're going to build is a fun little robot that will perform several different functions depending on how many times you press the push button. Our goal for this robot is to explore interesting ways to use your MRK-2 to create non-competitive robots. We're also going to use this chapter to explore randomness and some strategies for movement and searching. These skills will carry over into the next chapter when we focus entirely on building a robot to compete in Sumo competitions.

Breaking Down the Requirements

Because this robot is designed for fun, and not to meet some specific set of goals, we have a lot of flexibility in what we want to build. This chapter will walk you through the design and development of a few specific use cases for a fun robot design, but you are strongly encouraged to experiment and play with new techniques and algorithms.

What do we want this robot to do? When you're not competing in tournaments, it would be fun to have a robot that you can use to demonstrate your programming prowess and experiment with multiple methods of operation. Let's list some of the cool things we could have this robot do.

1. Have the robot pick a random direction and drive in that direction until it discovers an object blocking its path at a distance of 10 centimeters. Once an object is detected, it will change direction randomly until it discovers a direction that it can move.
2. Have the robot perform a 360-degree search looking for the closest object. When that object is detected, it will drive in the direction of that object in an attempt to bump against it. When the object is less than 1 centimeter away, it will stop. If the object moves, the robot

will attempt to reacquire the target by performing a search for the closest object and repeat the attack.

3. Have the robot follow the procedures of 1, but play the buzzer as we're driving based on the current value passed back by the IR sensors on the bottom. Higher values return lower sounds and lower IR values play higher sounds.

Practical Exercise: Can you think of other fun implementations that your robot can perform using the sensors and motors we've learned about?

Managing State

Knowing that we intend to develop at least 3 separate robot functions, we're going to want to track and maintain the current operational state of our robot. There are a lot of ways to do this, but the simplest way is to use a simple integer variable to maintain the current operational state of our robot. This will also allow us to continue adding features if we want to add more programs to our demo robot.

To make this work, we'll assume that the robot begins in state 1. Every time we press the button, we'll switch to the next state and blink the LED the number of times matching the current state. On state 1, we will blink the LED once. On state 3, we'll blink the LED three times. We'll also need to specify the total number of states that we have developed and if we press the button when we're already at the last state, the state variable will reset to 1.

This functionality can be easily accomplished by mixing some of the things we've done before in new ways. In the interest of writing good maintainable code, we will want to develop a blink function that accepts the number of blinks as an argument. Let's dive in with the simplest piece of functionality and focus on building a robot that will sit and cycle through states with each button press, flashing the LED as an indication of which state its currently in.

We'll begin by defining our current pin assignments and creating a configuration parameter to specify the number of seconds we want to wait after a new mode is selected before we begin robot operations.

```
#define button 2
#define led 13
#define stateSwitchDelay 3000
```

Our push button is an `INPUT_PULLUP` available on pin 2 and our LED is a digital `OUTPUT` assigned to pin 13. We also create a definition of a constant value `stateSwitchDelay` and set that to 3000 milliseconds. We assign this here so that in the future we can change this value easily to change how long our robot will wait between button presses.

The next thing that we're going to create are three global variables that will store our current state, inform our program of the total number of states, and a boolean variable to determine if our button is currently pressed.

```
const int states = 3;
int state = 1;
bool buttonPressed = false;
```

Of special note is the word `const` included before the line `int states = 3;`. The `const` keyword informs the Arduino compiler that this variable should be treated like a constant value that can never be changed. While this isn't a hard requirement, it's always a good programming practice to make your intentions clear. Specifying states to be a constant instantly informs any future developer that you don't intend for this value to change during the operation of the program.

We also initialize state to begin with a value of 1. When your robot is turned on, we want it to immediately start up in state 1 and begin driving around avoiding obstacles. We also create a boolean `buttonPressed` variable that will let us track the current state of the button. When we introduced the button in an earlier chapter, we learned that the default state of the button is to return a `HIGH` value. When the button is pressed, it switches to `LOW` and remains low until the button is released. In order to capture a single button press, we need to watch for the change from `LOW` to `HIGH`, which would imply that you have pressed the button and fully released it. For a refresher on this topic, please revisit [Lesson 4.2](#) and take a look at your solution for the third exercise in that lesson.

```
void setup() {
    Serial.begin(9600);
    pinMode(led, OUTPUT);
    pinMode(button, INPUT_PULLUP);
    // TODO: call blink() function, delay w/ stateSwitchDelay
}
```

Our first pass at a `setup()` function shouldn't contain any surprises. We begin by initializing the Serial port for communication at a 9600 baud rate and then configure out two pins for use. You'll notice the placeholder comment we've left to remind ourselves to start the initial operation inside the `setup()` function after we've finished our pin configuration.

Next is our main `loop()` function code, which will serve as the primary interface for this robot's program. Take a look at the code listing below and see if you can understand what is happening before we examine it closer.

```
void loop() {
    if ( digitalRead(button) == 0 ) {
        // the button is being held down
        buttonPressed = true;
    }
    if ( buttonPressed && digitalRead(button) == 1 ) {
        // the button was held down and was just released
        buttonPressed = false;

        if ( state == states ) {
            // we've reached our maximum state, reset to 1
            state = 1;
        } else {
            state = state + 1;
        }
        // TODO: call blink() function, delay w/ stateSwitchDelay
        //TODO: call delay() w/ stateSwitchDelay
    }
}
```

This example has a lot of complicated logic, but it should be fairly easy to walk through. At the beginning, we have a simple `if` block that checks to see if the button is pressed down. If it is, we'll set the boolean variable `buttonPressed` to `true`. From then on, this robot will not do anything until the button is released. When the button is released, the second `if` block will evaluate to `true` because `buttonPressed` has previously been set to `true` and the return from `digitalRead(button)` will equal 1. The double ampersand (`&&`) means that **both** conditions must evaluate to `true` before the `if` block will evaluate as `true`.

As soon as those conditions are both met, the next line of code resets `buttonPressed` to `false` so that it will be available for future button presses. If we don't set this back to `false`, then we'll never be able to press the button again because every time the `loop()` function executes, it will continue to think that we've just released the button and our second `if` block will continue to execute.

Inside this `if` block, we check to see if the `state` is already at our maximum value stored inside the `states` constant. If it is, we reset the `state` to 1. If it isn't, we'll simply increment the value of the `state` variable by 1.

The last thing we have inside this main `if` block is a comment to remind us to add our `blink()` and `delay()` functions once the blink function has been developed.

The next major thing to tackle before we move on to programming the logic behind our three robot operational modes is to develop a simple blink function that will blink the same number of times as we currently have defined as the active state.

This function can be as simple as this:

```
void blink(int blinks) {
    for ( int i = 1; i <= blinks; i++ ) {
        digitalWrite(led, HIGH);
        delay(500);
        digitalWrite(led, LOW);
        delay(500);
    }
}
```

This function simply takes a single argument for the number of blinks we want to perform and then loops from 1 to that number, turning on and off the LED with a small delay added at each step.

Practical Exercise: Implement this fully functioning state changing mechanism and add the calls to `blink()` and `delay()` in the two places where we've left placeholder comments.

Now that we have a robot that can switch between three separate operational states, let's walk through the implementation of the first operational state we defined above.

Our first operational requirement was to "have the robot pick a random direction and drive in that direction until it discovers an object blocking its path at a distance of 10 centimeters. Once an object is detected, it will change direction randomly until it discovers a direction that it can move."

We could translate this into pseudo-code that would look something like this:

```
LOOP FUNCTION IN STATE 1:  
  CALL STATE_1 FUNCTION:  
    GET DISTANCE IN FRONT OF ROBOT  
    IF DISTANCE IS LESS THAN 10 CENTIMETERS:  
      PICK A RANDOM DIRECTION  
      TURN RANDOM DIRECTION  
    IF DISTANCE IS MORE THAN 10 CENTIMETERS:  
      DRIVE FORWARD
```

The logic behind this is fairly simple when it's written out as pseudo-code. Inside our main `loop()` function, if the value of the `state` variable is set to 1, we can have our robot continuously perform these actions. We'll always want to start out with the distance in front of us, so we know we'll need to bring in the ultrasonic control code that we worked on in [Lesson 4.2](#). We also covered how to convert the return from the ultrasonic sensor from milliseconds to centimeters in [Lesson 2.3](#) when we introduced functions. We can use our Motor control library to handle the turning and driving, but how do we pick a random direction when we reach an obstacle?

The Arduino standard library provides us with a function called `random()` that accepts two arguments, a min and a max. To pick between two choices, left and right, we can simply call `random()` and pass in the values 1 and 3. You can decide which value corresponds to which direction, but the following code solution will use 1 for left and 2 for right. You'll notice that we passed in 1 and 3 even though we want the `random()` function to give us either a 1 or a 2. This is because the value passed in for the max argument is excluded from the possible list of numbers to return. If we wanted to get a random number between 1 and 100, we would have to call `random(1, 101)`.

Inside our `loop()` function, add the following lines of code near the bottom.

```
switch(state) {  
    case 1:  
        search_and_avoid();  
        break;  
}
```

This `switch` statement will evaluate the current value of the state variable and, in the case of state 1, call the function `search_and_avoid()`, which we're about to create. The `switch` statement is the perfect way to programmatically express that we're changing our logical operation based on the current value of one variable. We're going to develop the first operation together, but this will lay the groundwork for you to add as many different operational programs as you can dream up in a concise way that will be easily understood.

Developing the Function for State 1

In the interest of clear naming, we've chosen to name our function `search_and_avoid()`, so let's begin by adding that function to the bottom of our code editor. This function won't be returning a value, so its return type will be `void`.

```
void search_and_avoid() {  
    // get distance in centimeters  
    // if distance is less than 10, pick a random direction  
    // if distance is greater than 10, drive forward  
}
```

We know that the first thing we're going to want to do is get the distance in front of our robot in centimeters. The easiest thing to do in this scenario is to reuse the code we've written previously.

At the very top of your program, define your ultrasonic sensors pins using the `#define` preprocessor keyword.

```
#define led 13  
#define button 2  
#define echoPin A0  
#define pingPin 10  
#define stateSwitchDelay 3000
```

And inside your `setup()` function, set the `pinMode` to both of those pins appropriately.

```
void setup() {
  Serial.begin(9600);
  pinMode(led, OUTPUT);
  pinMode(button, INPUT_PULLUP);
  pinMode(echoPin, INPUT);
  pinMode(pingPin, OUTPUT);

  blink(state);
  delay(stateSwitchDelay);
}
```

We can also add the `ping()` function that we developed in [Lesson 4.2](#). This will give us the time in microseconds, which we'll have to convert using the `msToCm()` function, so bring that in as well.

```
long ping() {
  long duration;
  digitalWrite(pingPin, LOW);
  delayMicroseconds(2);
  digitalWrite(pingPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(pingPin, LOW);
  duration = pulseIn(echoPin, HIGH);
  return duration;
}

long msToCm(long microseconds) {
  return microseconds / 29 / 2;
}
```

With those helper functions in place, we can now look in front of our robot and see if we're about to bump into anything. We need to declare our variables globally at the top of our program so that we'll have a place to store these results, so at the top of your file, add a `long` variable called `cm` and initialize it to 0. We should also take this opportunity to import our motor library and set up our motors. Don't forget to `#include Motor.h`.

```
#include "Motor.h"

const int states = 3;
int state = 1;
bool buttonPressed = false;
```



```

// global operational variables
long cm = 0;
const int turnDuration = 600; // time to turn in ms

// Define motor
Motor motor;
void setup() {
    Serial.begin(9600);
    pinMode(led, OUTPUT);
    pinMode(button, INPUT_PULLUP);
    pinMode(echoPin, INPUT);
    pinMode(pingPin, OUTPUT);

    // set up our motors
    motor.setupLeft(5, 7);
    motor.setupRight(6, 8);

    blink(state);
    delay(stateSwitchDelay);
}

```

With that in place, we're ready to start coding our logic.

```

void search_and_avoid() {
    // random direction - 1 = left, 2 = right
    int turnDirection = 1;

    // get distance in centimeters
    cm = msToCm( ping() );

    // if distance is less than 10, pick a random direction
    if ( cm < 10 ) {
        turnDirection = random(1, 3);

        if ( turnDirection == 1 ) {
            motor.left(255);
            motor.right(-255);
            delay( turnDuration ); // turn for turnDuration
        } else {
            motor.left(-255);
            motor.right(255);
            delay( turnDuration ); // turn for turnDuration
        }
    } else {
        // if distance is greater than 10, drive forward
        motor.left(255);
        motor.right(255);
    }
}

```

Upload this sketch to your MRK-2 and turn on your robot. It should start up in `state 1` and start driving around the room after 3 seconds.

You should have everything you need to continue developing additional operational states. Adding more now is a simple matter of creating new functions to control the robot's behavior and calling those functions from inside your switch statement in your `main loop()` function.

Exercises

1. Develop an operational function called `search_and_destroy()` that will have the robot perform a full 360-degree search looking for the closest object. When that object is detected, your robot will drive in the direction of that object and stop when it reaches a distance of 1 centimeter. If that object moves, the robot will attempt to reacquire the target by performing another 360-degree search.
2. Develop an operational function called `search_and_avoid_musically()` that will follow the same operational parameters we developed in the first function in this chapter, but have the robot constantly read from the IR sensors on the bottom of the MRK-2 board. Have the robot play a note with the buzzer as it drives along, playing high notes for highly reflective surfaces and low notes for less reflective surfaces.
3. **Bonus -** Design and develop a fourth operational function that does whatever you want. Be creative!

Chapter 8 - Competitive Sumo Robots

In the last few chapters, we've explored robot design and development from several different perspectives, but now it's time to focus on the implementation necessary to program a robot to compete in Sumo competitions. In this chapter, we're going to cover the requirements needed to meet competition guidelines and discuss the implementation of a basic Sumo Robot searching algorithm.

Starting Your Robot

[Section 5, Article 8](#) of the official Sumo Robot League rules state that the player must begin the match by pressing a start switch, at which time the robot will begin a 5 second countdown before starting operation.

"Press the switch when the chief judge announces the start of the round. After 5 seconds, the robot is allowed to start operating, before which players must clear out of the Ring Area." -- Sumo Robot League Official Rules, Section 5, Article 8

We've written code to control the operational state of our robot in Chapter 7, but let's improve this functionality to meet the official competition requirements and provide better feedback about what is going on. One popular mechanism is to have the robot play an audible note for the last three seconds, then play one higher note before beginning.

Push Button Pseudo-code

Our robot should have two primary states: idle and competition. When the robot first powers on, it should go into the idle state and wait for the first press of the button. Since our button reports back with a HIGH value whenever it's not pressed, we will have our robot change state every time the button goes from LOW (0) back to HIGH (1). Assuming we were in the idle state, this would begin a 5 second countdown, playing a note at the start of the second second and playing a note on each consecutive second until it reaches the fifth second and is allowed to begin searching.

Our pseudo-code for this logic would look something like this:

```
INCLUDE PITCHES.H HEADER
INCLUDE MOTOR.H HEADER
```

```
DEFINE IR SENSOR PORTS
DEFINE BUTTON PIN PORT
DEFINE BUZZER PIN PORT
DEFINE ULTRASONIC SENSOR PORTS
```

```
CREATE MOTOR OBJECT
```

```
INITIALIZE STATE VARIABLE TO 1
INITIALIZE BUTTON_STATE VARIABLE TO 1
SETUP FUNCTION:
```

```
    INITIALIZE BUTTON PIN
    INITIALIZE BUZZER PIN
    INITIALIZE ULTRASONIC SENSOR
    INITIALIZE IR SENSORS
    SETUP MOTORS
```

```
LOOP FUNCTION:
```

```
    IF BUTTON_STATE IS 1 AND BUTTON IS PRESSED:
        SET BUTTON_STATE TO 0
```

```
    IF BUTTON_STATE IS 0 AND BUTTON IS NOT PRESSED:
        SET BUTTON_STATE TO 1
```

```
    IF STATE IS 1:
        CALL BEGIN_COUNTDOWN FUNCTION
```

**IF STATE IS 2:
 SET STATE TO 1**

BEGIN_COUNTDOWN FUNCTION:

**LOOP 3 TIMES:
 DELAY 1000 MS
 PLAY BUZZER WITH NOTE_G3 FOR 200 MS**

**DELAY 1000 MS
 PLAY BUZZER WITH NOTE_G4 FOR 500 MS
 DELAY 1000 MS
 SET STATE TO 2**

By this point, initialization and setup should be second nature to you. Even the push button and countdown functionality are just modifications of what we've done in previous chapters, but let's walk through the flow with code examples to make sure this structure makes sense.

As always, we'll start by importing the scripts we know we'll need and defining some constant values at the top. We know we're going to need the Motor library we've developed, and we're also going to want to import the Pitches header file from [Chapter 5.2](#). The code sample provided below is a good functional foundation for the rest of the robot we're going to program. Copy this code into a new Arduino project and import Pitches.h, Motor.h, and Motor.cpp into your editor in new tabs.

```
#include "Motor.h"
#include "Pitches.h"

// pin configurations
#define leftSensor A1
#define rightSensor A2
#define IREmitter 4
#define button 2
#define buzzer 3
#define echoPin A0
#define pingPin 10

// Motor object
Motor motor;

// global variables
int buttonState = 1; // 1 = up, 0 = pressed
int state      = 1; // 1 = idle, 2 = competing
```

```

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

    // pin setup
    pinMode(button, INPUT_PULLUP);
    pinMode(buzzer, OUTPUT);
    pinMode(leftSensor, INPUT);
    pinMode(rightSensor, INPUT);
    pinMode(IREmitter, OUTPUT);
    pinMode(pingPin, OUTPUT);
    pinMode(echoPin, INPUT);

    // Motor setup
    motor.setupLeft(5, 7);
    motor.setupRight(6, 8);

    Serial.println("Robot powered up. Waiting for button
press.");
}

```

Our setup code configures our entire MRK-2 board and all associated sensors so that we can use everything at our disposal in the creation of our autonomous Sumo Robot. Everything is enabled and we end our `setup()` function with a call to `Serial.println()` with a message letting us know that the robot is ready to begin. You'll only see this message if you have the robot connected to your computer with your USB cable and have the Serial Monitor pulled up, but it will be a useful debugging method that we'll use throughout the entire software development lifecycle for this project.

The main `loop()` function is where our autonomous robot control code begins. The pseudo-code we're basing this initial programming effort on is concerned with the primary robot operational state and a musical countdown that plays when we push the button and switch from idle mode to competition mode.

Our pseudo-code for the first section of the loop reads:

```

LOOP FUNCTION:
    IF BUTTON_STATE IS 1 AND BUTTON IS PRESSED:
        SET BUTTON_STATE TO 0

```

This logic can easily be translated into actual code.

```
void loop() {  
    if ( buttonState == 1 && digitalRead(button) == 0 ) {  
        buttonState = 0;  
    }  
}
```

This code checks to see if our `buttonState` is currently set to 1, and if so, also checks to see if the physical button is currently being pressed down. If both of those conditions are true, we set `buttonState` to 0 so that it will be ready for the robot state change.

The next thing we need to add is the code to handle the state change when the button is released. The logic we designed before was written like this:

**IF BUTTON_STATE IS 0 AND BUTTON IS NOT PRESSED:
SET BUTTON_STATE TO 1**

**IF STATE IS 1:
CALL BEGIN_COUNTDOWN() FUNCTION
IF STATE IS 2:
SET STATE TO 1**

The code to make this functional will be the opposite of the previous if block. When the `buttonState` variable is set to 0, we know that the button had been pressed down. If the button is no longer pressed down, we know that the button has just been released and its time to switch the value stored in the `state` variable. The first thing we'll need to do in this case is set the `buttonState` variable back to 1 so that the next time the `loop()` function is called, the button will be ready for the next time it's pressed.

Once we've reset the `buttonState` variable, we will check the value of our `state` variable. If it's currently set to 1, we know we're idling and that it's time to begin our countdown. If it's set to 2, we know that our robot has been in competition mode and that we need to go back to idling. The code listed in bold on the next page contains our new additions to the `loop()` function.

```

void loop() {
  if ( buttonState == 1 && digitalRead(button) == 0 ) {
    buttonState = 0;
  }
  if ( buttonState == 0 && digitalRead(button) == 1 ) {
    buttonState = 1;

    if ( state == 1 ) {
      Serial.println("Competition countdown.");
      Serial.println("Good luck!");
      begin_countdown();
    } else {
      Serial.println("Robot entering idle mode.");
    }
  }
}

```

The logic defined in this code matches what we had drawn up in our pseudo-code. If we tried to compile this code now, we'd see a console message informing us that the `begin_countdown` function was not declared in this scope. We get this error because we haven't written it yet. Let's code that now and walk through each line of code with the explanation listed on each line.

Code	Explanation
<code>void begin_countdown() {</code>	Define our function with a return type of “void”.
<code>for (int i = 0; i < 3; i++) {</code>	Start a loop to repeat the next lines of code exactly three times.
<code>delay(1000);</code>	Wait 1 full second.
<code>playNote(NOTE_G3, 200, 15);</code>	Play the note G3 for 200 milliseconds with a 15 millisecond rest.
<code>}</code>	End our loop.
<code>delay(1000);</code>	Wait 1 more second before playing the next note.
<code>playNote(NOTE_G4, 500, 15);</code>	Play our final note, G4 for half of a second.
<code>delay(1000);</code>	Wait 1 more second before continuing.
<code>state = 2;</code>	Now that the countdown has finished, set the <code>state</code> variable to 2 so the robot will enter competition mode.
<code>}</code>	End our function and return to the main <code>loop()</code> .

This code requires the `playNote()` function that we developed in [Chapter 5.2](#).

The `playNote` function is a simple function that takes the arguments `note`, `duration`, and `rest` and then sends those values to be played by the buzzer.

```
void playNote(int note, int duration, int rest) {
    tone(buzzer, note, duration);
    delay(duration + rest);
}
```

Try compiling this program now and make sure it works. Once you've uploaded your sketch to the MRK-2 board, leave it plugged in to your computer with your USB cable and open up the Serial Monitor.

You should see the message "Robot powered up. Waiting for button press." pop up. When you press the button, does it play your startup countdown and print that it's entering competition mode?

Practical Exercise: Write the code needed to implement the LED and have it flash every time you play a note during your countdown. Intimidate your enemies with your powerful red lantern!

Strategies for Autonomy

There are many different strategies for approaching the development of an autonomous robot to compete in Sumo Robot Competitions. Some strategies involve complicated maneuvers designed to trap an opponent's robot in a compromised position while others take an approach more comparable to a direct frontal assault. Sumo Robot Competitions in this weight class are often won or lost in the first 30 seconds and sometimes come down to who can program the most efficient search algorithm.

We're going to explore several possible strategies and consider the strengths and weaknesses of each strategy. In the next section of this chapter, we'll implement one of these strategies and leave room for configuration options that let you tweak and modify the functionality of this algorithm to compete in Sumo Robot competitions. Please bear in mind that while the algorithm we develop together is strong enough to put up a great fight and has the potential to win with regularity, in competitions you'll be competing against people with years of experience in building and programming competitive robots. Creativity, experimentation, and constant refinement are the long-term keys to continued victory. There are no shortcuts to an undefeated robot and you'll have to take the lessons you've learned in this book and apply them in new and interesting ways in order to win the majority of your battles.

Algorithm #1: Random Rotation Search

The Random Rotation Search algorithm is the simplest algorithm to develop and is the one we'll be building together in this chapter. The core premise is that the best defense is a good offense and this algorithm works by efficiently finding the enemy robot and attacking immediately.

In this algorithm, we have our robot randomly pick a direction and spin in place for a pre-configured number of seconds. If an object passes in front of the robot at a range of less than the pre-configured number of centimeters, our robot immediately switches to attack mode and drives forward in an attempt to push the enemy robot outside of the ring. If our ultrasonic sensor suddenly returns a value higher than our pre-configured distance threshold, we know that our enemy has moved past the front of our robot and we abort our attack and go back to searching.

If at any time our infrared sensors pick up the edge of the ring, we immediately abort any action and move backwards in full reverse in an attempt to remain inside the ring.

The biggest strength to this algorithm is the speed at which we're able to acquire targets. Two robots using this algorithm will be able to find each other within seconds and will begin their attack sequence, at which point the competition comes down to brute force. Traction and torque will become the deciding factor in these competitions. The main goal of using this algorithm is the desire to detect the enemy robot and push him out of the ring while it's still searching for your robot. Ramming into the side of most robots should give you the upper hand and let you push your opponent outside of the ring before it can react.

The biggest flaw in this algorithm is that it can sometimes take too long to find other robots, and it's search pattern of spinning in place can be defeated by some of the more challenging algorithms and solutions.

Algorithm #2 - Random Drive Pattern

The Random Drive Pattern algorithm is another fun, simple algorithm that is ideal for the beginner. In this algorithm, your robot randomly picks a direction of left or right and then drives gradually in that direction. When this robot detects the white border it will aggressively turn away from that border before resuming the gradual turn, only this time towards the opposite direction.

If, at any time, it detects another robot in front of it at a distance of less than the pre-configured distance, it switches to attack mode and attempts to ram the robot out of the ring at full speed.

This algorithm's biggest strength is that the constant side-to-side motion makes this robot a difficult target for other robots to detect and defeat. It performs particularly well against robots developed using the Random Rotation Search algorithm by being extremely difficult to pin down.

The biggest flaw to this algorithm is that the search pattern is highly inefficient and this robot often loses due to searching near the ring border and being pushed out by another robot.

Algorithm #3 - The Wobbler

This algorithm is an interesting search concept that involves searching in an ever-widening pattern from an initial point of origin. At the start of the search, this robot will first turn in either direction for a small amount of time looking for any object within the preconfigured distance. If it does not detect the enemy, at the end of the short time interval, it will reverse the search direction and look the opposite way for two times the original time interval. If the enemy is still not detected, it will again reverse direction and search for three times the original interval. This process will repeat until the enemy is detected and the robot goes into attack mode.

If the distance to the enemy suddenly increases, we know that it's moved past our forward field of view and we restart the search using the original time interval.

This algorithm, while slow to start, is extremely effective at re-engaging fast moving enemies. An enemy moving fast horizontal to the robot will frequently be immediately reacquired within the next few seconds. This search algorithm often produces the hilarious result of chasing the enemy robot around the ring and pushing it out from behind.

The biggest flaw with this strategy is that if your opponent is able to get behind your robot, you'll often get pushed out of the ring before you even detect that the enemy is there.

Creativity and Clever Tricks

There is no silver bullet that will let you build a robot to beat every opponent. Even the best robots can be beaten by a poorly designed robot that gets lucky at the right time. The best strategies are often simple and easy to implement, though it's always fun to explore new ideas and implement clever tricks.

One interesting idea would be to implement a quick flanking maneuver. If your robot is in attack mode and has been in attack mode for more than 10 seconds, we can assume that your robot is pushing directly against the enemy robot. Rather than keep pushing until one of the robots slips off the board, an interesting idea would be to switch into flank mode and quickly reverse at a sharp angle. The enemy robot will continue to push forward while your robot slides neatly to the side and then starts pushing against the side of the enemy robot.

That is just one example of a clever algorithm you could use to enhance your robot's performance.

Can you think of any other cool ideas?

Implementing the Random Rotation Search Algorithm

Let's build one version of the Random Rotation Search algorithm and put it to the test against other robots. We'll start, like we always do, by designing our logic in pseudo-code to make sure that what we're designing makes sense from a logical standpoint.

Most of what we're going to be building is a modification of the code we developed for our fun demo robot in Chapter 7. The biggest change here is that we're going to aggressively attack when we find an enemy robot close to us rather than changing direction to avoid running into obstacles.

Along with our operational states of idle and competition mode, we're also going to have three separate states that the robot will use inside the competition mode: searching mode, attack mode, and abort mode.

In searching mode, our robot will pick a random direction and rotate for a configured number of seconds. As it spins, it will use the ultrasonic sensor to look ahead and see if an object is detected within the distance threshold that we'll `#define` at the top of our program. If an object is detected at a range below the distance threshold, we'll move to mode 2 which is our attack mode.

Attack mode is the simplest mode and involves driving straight ahead at maximum speed in the hope of quickly ramming the enemy robot out of the ring. If the distance sensor suddenly begins reporting values above the distance threshold, we know that the enemy has moved out of our forward field of view and we'll go back to search mode.

Abort mode is a special mode we'll enter if our front facing IR sensors detect the edge of the ring. As we've seen in earlier chapters, the center area of the black ring returns high values while the white border returns low values. If our robot suddenly reports that it is over the border of the ring, we'll immediately halt and proceed in reverse at full speed for a preconfigured number of seconds.

With that logical description of what we'll need, we can modify the code we developed earlier in this chapter to include some of the variables and definitions we'll need. In your Arduino IDE, add the following definitions to the top of the file.

```
#define abortThreshold 900
#define searchTime 2000
#define attackDistance 12
```

These values are merely intelligent guesses as to what we will need for an effective robot. As you test your robot on the board, you'll want to tweak these values to match better real world conditions. If your IR sensors report 950 as the value of the white line, you'll need to increase your `abortThreshold` value to be above 950 in order to successfully go into abort mode. If your motor speed causes your robot to turn too far while searching, you'll want to decrease the amount of time your robot turns by lowering the `searchTime` definition.

We're also going to need to create some global variables. We will need to track the elapsed number of milliseconds in a turn and the turn direction we've randomly selected. These variables need to be global so that we can access this from several different functions without worrying about passing around arguments.

```
int turnDirection = 1;           // 1:left, 2:right
unsigned long turnTime = 0;      // holds the return from millis()
```

In previous exercises, we were able to call the `delay()` function directly to perform an action for a certain number of milliseconds. Unfortunately, calls to `delay()` halt all future operations and would prevent our robot from searching during a turn. In order to have our robot be capable of turning and searching at the same time, we're going to track the amount of time we spend in a turn by calling the `millis()` function. This function will return the number of milliseconds that have elapsed since the program was started.

When we first enter the search state, we can call `millis()` and store that result in `turnTime`. In every subsequent call to our search function from our primary `loop()`, we'll simply compare the result of `millis()` subtracted from `turnTime` to see if that time has exceeded our defined `searchTime` value. If it has, we reset `turnTime` to 0 and restart the search. This is an efficient way to time an event and have your robot continue to perform other tasks that need to occur during your main `loop()`.

To read more about the `millis()` function and see additional examples of its use, please refer to the Arduino Reference library and the `millis()` entry located at <http://bit.ly/ArduinoMillis>.

With timing handled, we can move on to developing the meat of our competition algorithm. We'll want to put all this search, attack, and abort code in their own separate functions, but we'll control these operations in a standalone function named `compete()`. Let's add these method stubs to our code now and include the call to `compete()` at the very end of our `loop()` function.

```
void loop() {
    if ( buttonState == 1 && digitalRead(button) == 0 ) {
        buttonState = 0;
    }
    if ( buttonState == 0 && digitalRead(button) == 1 ) {
        buttonState = 1;

        if ( state == 1 ) {
            Serial.println("Competition countdown.");
            Serial.println("Good luck!");
            begin_countdown();
        } else {
            Serial.println("Robot entering idle mode.");
        }
    }

    // if the countdown has ended, we're now in competition mode
    if ( state == 2 ) {
        compete();
    }
}

void compete() {
    // competition control code goes here
}

void search() {
    // algorithm for search mode
}

void attack() {
    // algorithm for attack mode when an opponent is found
}

void abort() {
    // algorithm for handling the ring border.
}
```


Creating empty functions and providing a quick description of the function's purpose is a good way to approach the development of complex systems. We've already given a lot of thought to the structure of our competition system, so creating these empty functions will let us start from the top and work our way down in small, discrete blocks of work.

The Compete Function

This is our primary competition mode function. It is called every clock cycle from the processor at the end of the `loop()` function whenever we're in state 2. From here, this function will monitor the current competition state and pass off control to the appropriate function.

Our logical setup for this algorithm involves a few simple rules:

1. Check the distance in front of our robot using the ultrasonic sensor.
2. Check the current return from the IR sensors below us.
3. If the IR sensors are reporting that we're at the ring border, immediately call the `abort()` function.
4. If the ultrasonic sensors are reporting that there is an enemy robot in front of us, call the `attack()` function.
5. If ultrasonic shows nothing and we're in the black area, call the `search()` function.

These rules are fairly simple and we can start by working our way down the requirements.

```
void compete() {  
    // read sensor values  
    int distance = msToCm( ping() );  
    int leftIR   = digitalRead(leftSensor);  
    int rightIR  = digitalRead(rightSensor);  
}
```

At the start of every call to `compete()`, we want to grab the current sensor values. This will let us see what is happening around us and make the appropriate decisions for requirements 3 through 5. To these requirements, we're going to take these sensor values and quickly cycle through the conditions and call the correct control function.

```

void compete() {
    // read sensor values
    int distance = msToCm( ping() );
    int leftIR    = digitalRead(leftSensor);
    int rightIR   = digitalRead(rightSensor);

    if ( leftIR < abortThreshold || rightIR < abortThreshold ) {
        abort();
    } else if ( distance < attackDistance ) {
        attack();
    } else {
        search();
    }
}

```

With that code in place, we are ready to move on and start developing the individual components.

Abort!

The `abort()` function is one of the most critical and time sensitive functions. One of the worst ways to lose a match is to drive straight out of the ring without any help from your opponent. Do not commit seppuku!

The simplest way to approach this task is to simply immediately throw both motors in full reverse whenever a ring boundary is detected. We want to put as much distance between our robot and the edge of the ring as quickly as we can, and the easiest way to handle this is to simply drive backwards at full speed for 2 to 3 seconds.

```

void abort() {
    motor.left(-255);
    motor.right(-255);
    delay(2000);
}

```

By the time the delay has finished, you should be far enough back that the next cycle of your `compete()` function will have your robot resuming the search implementation.

Practical Exercise: This implementation is extremely naive and has room for improvement. One approach would be to determine which sensor detected the ring border and drive backwards away from that side on a slight curve. Try and implement a better version of this `abort()` function.

Attack!

The `attack()` function is also quite simple and is basically the opposite of our naive approach to ring border detection. Whenever our `attack()` function is called, we know that our `compete()` function has detected an object directly in front of us at a range that is below our `attackDistance`. The only thing we need to do here is charge forward at full speed and hope for the best.

```
void attack() {  
    motor.left(255);  
    motor.right(255);  
}
```

For added style points, one common addition is to play a musical tone with the buzzer whenever your robot is in attack mode. Adding a call to `playNote()` here would spice things up and make your matches a little more melodic.

Searching

The implementation of your `search()` function is bound to be one of the most challenging aspects to building your competitive robot. The logic we'll be using in this function is nothing new, but the introduction of the `millis()` function call for timing could make this seem more difficult than it really is.

To recap our logic, let's quickly walk through the functional steps this function will handle each time it's called.

1. If `turnTime` is not 0, we need to compare the difference between the current return of `millis()` against the initial time we stored in the

turnTime variable. If that number is greater than the value we've set in our searchTime configured definition, we can reset turnTime to 0 so that our robot can pick another random direction in the next functional step.

2. If the current value of turnTime is 0, we know that we either just finished a full search sequence and the value has been reset, or that we've just called our search function for the first time. We'll need to pick a direction and set that in our global variable and save the current time returned by millis() into the turnTime variable.
3. Turn the motors in whichever direction we've specified and end the function.

Coding these rules should be fairly simple, so let's give it a shot and move on to real-world testing.

```
void search() {  
    // our time has exceeded our configured searchTime, reset.  
    if ( turnTime != 0 && (millis() - turnTime > searchTime) ) {  
        turnTime = 0;  
    }  
    // get random direction and store the start time  
    if ( turnTime == 0 ) {  
        direction = random(1, 3);  
        turnTime = millis();  
    }  
  
    // start our turn, 1 = left, 2 = right  
    if ( direction == 1 ) {  
        motor.left(255);  
        motor.right(-255);  
    } else {  
        motor.left(-255);  
        motor.right(255);  
    }  
}
```

This implementation is extremely efficient, wasting very few CPU cycles on unnecessary things. It also leaves you with a lot of room for experimentation and should provide you with a lot of opportunities to try unique things.

The only limit to what you can accomplish is your imagination.

Exercises

1. Compete!
2. Tweak your configuration.
3. Repeat as needed.
4. **Bonus** – Have Fun!

Appendix A - Troubleshooting Common Bugs

A large part of good engineering is knowing how to solve most problems. With Arduino-based development, sometimes the easiest way to find the solution to an error is to directly Google whatever error message your console displays.

For hardware troubleshooting, often the best thing you can do is to write the smallest possible program to test just that one component. If one of your motors isn't working, writing code to operate on just that motor is often the first step in troubleshooting. This method of debugging can be referred to as isolationism in which you isolate the problem and focus on the core of what is going wrong.

Once you've isolated the problem, if a solution isn't quickly found in a Google search, you can often ask about the problem on Stack Overflow and get good suggestions from a community. Stack Overflow is a programming question/answer site that often has excellent answers for most problems. The Arduino-specific section of Stack Overflow can be found at <http://stackoverflow.com/questions/tagged/arduino>.

Common Programming Problems

Missing Semicolon

Semicolons are required at the end of every normal line of code and it's common to leave these out. The console will often tell you that there is an expected semicolon, but will often point you to the line of code directly below where you're missing the semicolon.

```

void setup() {
    // put your setup code here, to run once:
    Serial.begin(9600)
}

void loop() {
    // put your main code here, to run repeatedly:
}

```

In the above example, the line `Serial.begin(9600)` is missing a semicolon to end the line. Trying to compile this code will result in the following error message displayed in the Console.

```

error.ino: In function 'void setup()':
error:4: error: expected ';' before '}' token
expected ';' before '}' token

```

This error message is telling you that in the function `setup()`, line 4 believes that there is an error because it expects a semicolon before the closing curly brace. Since line 4 is the end of our function, looking up to line 3 where we have the call to `Serial.begin()` shows the actual line that is missing the semicolon.

Scoping Problems

As we saw throughout this book, we need to declare variables in the appropriate place for them to be available inside other functions.

If we had the following code sample and tried to compile it, we would get an error message saying that the variable `a` was not declared inside the scope of the function `loop()`.

```

void setup() {
    // put your setup code here, to run once:
    int a = 6;
}

void loop() {
    // put your main code here, to run repeatedly:
    a = 5;
}

```

Compiling this code would result in the error message you see below.


```
error.ino: In function 'void loop()':  
error:7: error: 'a' was not declared in this scope  
'a' was not declared in this scope
```

The problem is that we defined `int a` for the first time inside the `setup()` function. Doing so makes this variable only available inside `setup()`. If we wanted this variable to be available to every function we write, we would need to declare it as a global variable by putting it outside of any functions.

```
int a;  
void setup() {  
    // put your setup code here, to run once:  
    a = 6;  
}  
void loop() {  
    // put your main code here, to run repeatedly:  
    a = 5;  
}
```

Warning: You will get this exact same error message if you have a typo in your variable name.

In this example, we've created a boolean variable named `typo` and set it to `false`. But when we used it inside the `setup()` function, we accidentally spelled the variable as `tpyo`, swapping the `p` and the `y` incorrectly. The console will warn you that the variable `tpyo` is not available in this scope because that variable was never actually created.

```
bool typo = false;  
void setup() {  
    // put your setup code here, to run once:  
    tpyo = false;  
}  
void loop() {  
    // put your main code here, to run repeatedly:  
}
```

Missing Curly Brace

If blocks, functions, and switches all require matching opening and closing curly braces. If you forget one of these curly braces, you will get an error message that can sometimes be cryptic.

```
void setup() {
```

```

        // put your setup code here, to run once:
    }
    void loop()
        // put your main code here, to run repeatedly:
    }

```

The code above (missing the opening curly brace after the `loop()` function) would cause the following error.

```

error:6: error: expected initializer before '}' token
error:6: error: expected declaration before '}' token
expected initializer before '}' token

```

This error is letting us know that we've somehow forgotten to include the opening curly brace. If we had forgotten our closing curly brace, the error message would look like this:

```

error.ino: In function 'void loop()':
error:4: error: expected '}' at end of input
expected '}' at end of input

```

Common Hardware Problems

Power LED Not Lighting

When the MRK-2 is connected via USB cable to a computer, the power indicator LED (green) will turn on regardless only when the power switch on the circuit board is set to USB. Once the robot is receiving power either via USB or from the batteries by moving the switch to On, it will execute the code currently on it.

If the LED fails to turn on when switching the robot's battery power to On, check the installation of your batteries. If it still doesn't come on, next check the wires connecting the batteries to the board. If both the battery installation and wires seem correct, try some batteries that you know are good. If the robot still fails to power up, use a Voltmeter to check the voltage on the MRK-2 circuit board where the batteries connect. Note that the copper pads are probably completely covered with solder so just push the probes against the solder. If you don't make a solid connection between the probes and the metal (copper or solder) on the circuit board, then the Voltmeter will read zero or read an unexpectedly low Voltage such as 1.23 V. With the robot turned off the voltage should be in the range of 5 to 6.5 Volts. If it is below this, find new batteries.

If the LED fails to turn on when connected via USB, the following steps will help you troubleshoot and identify the cause. First, check that the USB port on the computer has power. If a laptop has fallen asleep or a computer is turned off, it probably will not provide power. If the computer is on and the USB cable is properly connected to both the robot and computer, try using a different USB cable. You can also test your computer and cable by borrowing a friend's robot that has a lit LED when connected over USB.

Problem Uploading or Disconnected MRK-2

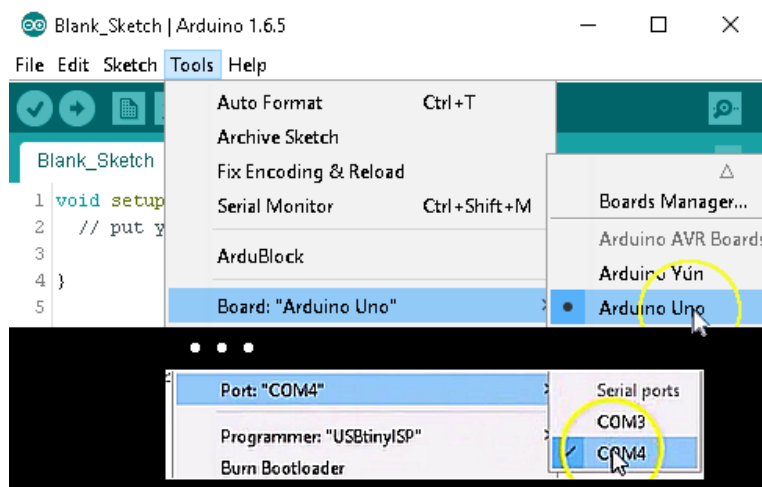


Figure A.1 – Select the correct board and port for your MRK-2 robot.

After connecting your board to your computer using the usb cable and selecting the correct board and port in the Arduino IDE (see Figure C.1), the Arduino IDE is supposed to auto-detect the MRK-2 board whenever it's connected with your USB cable. However, sometimes it becomes unseated or just stops working and you'll get a cryptic error message like this:

```
avrdude: ser_open(): can't open device "/dev/cu.usbserial-  
DA01DDE4": No such file or directory  
ioctl("TIOCMGET"): Inappropriate ioctl for device
```

or simply a message saying:

Problem uploading to board. See
<http://www.arduino.cc/en/Guide/Troubleshooting#upload> for
suggestions

The best way to correct these sorts of errors is to simply unplug the USB cable from both the computer and the MRK-2 and reconnect it. Hopefully the issue will go away after being reconnected. If the problem persists, then you may have selected the wrong port out of multiple available ports. Try selecting one of the other ports and then try to upload again. If you still have trouble, try to isolate the cause of the problem by getting with a friend who has successfully uploaded onto their robot. Test your robot with first their USB cable, then their computer, and then both their USB cable and computer. This should enable you to isolate the problem to either your computer, your usb cable, or your robot. After you isolate the problem, test it to verify that you have correctly identified the problem. If you've successfully isolated the problem, then the Arduino IDE should still fail to connect or upload on the good equipment until you replace the faulty piece with your friend's good one.

Motor Spinning the Wrong Direction

One of your motors will always be opposite the other motor, and you'll need to write your motor control code to correct this deficiency. This code is covered in detail in [Chapter 4](#) when we learn about writing code to control our motors.

Rebooting During Competition or Dropping Bluetooth

In order to keep the microprocessor functioning properly as long as possible the MRK-2 circuit board contains a voltage regulation system that decreases voltage from fresh batteries to a steady 5V and boost voltage from low batteries to 5V. As batteries get too low and motors are suddenly switched on or change direction, the regulated voltage may temporarily drop so low that the microprocessor reboots as if you had turned the power off and then back on. If you have added Bluetooth onto your robot, sometimes the Bluetooth chip may lose power before the microprocessor. This can be seen visually because the Bluetooth chip's red power indicator LED will turn off or will go from being solid (connected) to blinking. You will probably also get an error message like "Broken Pipe" on your app or simply see that your robot is stuck in whatever command it received last.

Appendix B – Pin Names & Uses

This book refers to pins by the Arduino conventions and not the microprocessor manufacturers naming convention. The following table displays the ATmega328 pin names, Arduino pin names, extra capabilities of each pin, and MRK-2 usage.

There are 3 pins labelled N/A in the Arduino column because Arduino does not give them a name. This is because they are connected to the crystal that keeps time and the reset signal necessary for uploading new code over USB. Consequently, they cannot be used as an input or output (i/o) pin. Otherwise all the Arduino pins labelled with only a number can be used for both input and output. The pins labelled with A and then a number, for example A3, can be used as Analog inputs as well. The MRK-2 board does not have any true analog outputs but can mimic analog output via pulse-width modulation (PWM) on pins 5,6,9,10,11.

One of the many improvements of the MRK-2 compared to the MRK-1 is that the pin assignments have been optimized to avoid loss of functionality when using the Servo and Tone libraries provided by Arduino. Because using the Servo library prevents PWM on pin 9 and 10, we have placed pin 9 in the front-left header for use with servo motors, and wired pin 10 to the ultrasonic sensor output (aka trigger) which does not use PWM. Similarly, the buzzer has been attached to pin 3 so that use of the tone() doesn't affect the robots performance even though it does eliminate the ability to do PWM on pin 3 and 11.

Getting all this capability -- bluetooth communication via the SoftwareSerial library, music via tone(), control of a servo motor via the Servo library, ultrasonic and infrared sensors, and variable speed and direction motor control -- out of a mere ATmega328 is a great example of why Arduino programming is so exciting. Being able to simultaneously use these capabilities is what makes the MRK-2 really stand out as one of the most cost-effective and user-friendly entry platforms into Sumo Robot Competition.

Table of Pin Names, Capabilities, & Uses

ATmega Name	Arduino Name	Extra Capabilities	MRK-2 Uses
PD0	0	Atmega RX	USB connection
PD1	1	Atmega TX	USB connection
PD2	2	interrupt	Pushbutton
PD3	3	PWM**, interrupt	buzzer
PD4	4		IR emitter
PD5	5	PWM	leftSpeed
PD6	6	PWM	rightSpeed
PD7	7		leftDirection
PB0	8		rightDirection
PB1	9	PWM*	servoPin on front-left header
PB2	10	PWM*	ultrasound trigger
PB3	11	PWM**	MOSI (ISP header)
PB4	12		MISO (ISP header)
PB5	13		user LED & SCK (ISP header)
PC0	A0		ultrasound input
PC1	A1		leftIR
PC2	A2		rightIR
PC3	A3		rearIR
PC4	A4	SDA, I2C (3.3V logic)	BT_RXPin on front-right header
PC5	A5	SCL,I2C (3.3V logic)	BT_Txpin on front-right header
ADC6	A6		extra analog input, on front-left header
ADC7	A7		battery monitor circuit
PB6	N/A	ext oscillator	
PB7	N/A	ext oscillator	
PC6	N/A	reset	

*The ATmega328 can not do PWM on 9 and 10 and when using the Servo Library

**Using the tone library interferes with PWM capability on 3 and 11

Pin Access

Because the MRK-2 circuit board has a motor controller, a piezo buzzer, user pushbutton, and 3 infrared sensors, some pins are not easily accessible. However, there are 9 accessible I/O pins in the headers at the front of the board and are labelled “ICSP”, “Bluetooth”, “Distance Sensor”, and “User I/O” that can be used for anything you want to try. Please keep in mind that although the Bluetooth header provides 5V on the V+ pin, pins A4 and A5 operate at 3.3V logic. The MRK-2 circuit board has a built-in 3.3V logic converter for these two pins.

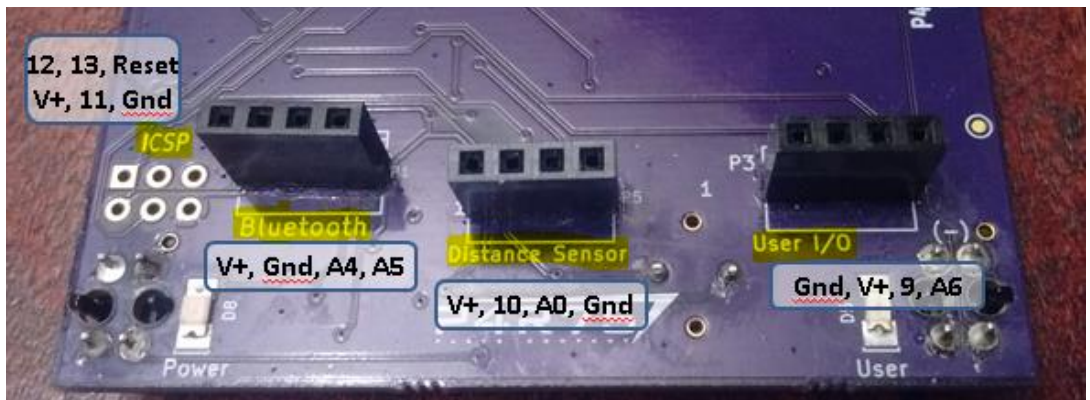


Figure B.1 – Accessible pins in the ICSP, Bluetooth, Distance Sensor, and User I/O connectors

Appendix C - Official Sumo Robot Rules

Section 1. Definition of the Sumo Match

Article 1. (Definition) The match shall be fought by two teams (At the event, one team consists of one robot with two team members, one of which is a leader. Other team members must watch from the audience), according to these Rules for Sumo matches (hereafter called "these Rules"), with each team's robot made by each team competing to get the effective points (hereafter called Yuhkoh), within the perimeter of the defined Sumo Ring. The judges will decide which team wins. A single person can also compete with a Robot Sumo, with the same rules that apply to teams.

Section 2. Requirements for Ring Area

Article 2. (Definition of Ring Area) The Ring Area means the Sumo Ring and the space outside the Ring. Anywhere outside this Ring Area is called Outer Area.

Article 3. (Sumo Ring)

- The Ring shall be in circular shape with its height being 2.5 cm and its diameter 77 cm (including the outside of the line that divides the inside of the Ring from its outside). The ring shall be made of any material fitting this qualification, normally MDF painted black with a white outside line.
- Shikiri lines (where robots stand at the beginning of the match) are the two parallel lines with 10 cm distance between the lines, drawn in the center of the Ring. The Shikiri lines are painted in brown (or equivalent for reflection of IR light), 1 cm wide and 10 cm long.
- The Ring shall be marked by a white circular line of 2.5 cm thickness. The Ring is within the outside of this circular line.

Article 4. (Space) There should be the space of more than 50 cm wide outside the outer side of the Ring. This space can be of any color except white, and can be of any materials or shape, as long as the basic concept of these rules are observed. This area, with the ring in the middle, is to be called the: "Ring Area". If there are markings or part of the ring platform outside these dimensions, this area will also be considered in the Ring Area.

Section 3. Requirements for Robots

Article 5. (Specifications) A robot must be in such a size that it can be put in a square tube of 10 cm wide and 10 cm deep. A robot can be of any height. A robot must not be in such a design that its body will be physically separated into pieces when a match starts. The robot with such a design shall lose the match. The design to stretch a robot's body or its parts shall be allowed, but must remain a single centralized robot. Screws or nuts or other robot parts, with a mass of less than 5 grams total, falling off from a robot's body shall not cause the loss of match.

The mass of a robot must be under 500 grams including the attachments and parts.

For stand-alone robots, any control mechanisms can be employed. Stand-alone models must be so designed that a robot starts operating a minimum of five seconds after a start switch is pressed (or any method that invokes the operation of a robot). Microcomputers in a robot can be of any manufacturers and any memory sizes can be chosen.

Article 6. (Don'ts in manufacturing a robot)

- Jamming devices, such as an IR LED intended to saturate the opponents IR sensor, are not allowed.
- Do not use parts that could break or damage the Ring. Do not use parts that are intended to damage the opponent's robot. Normal pushes and bangs are not considered intent to damage.
- Do not put into a robot's body devices that can store liquid, powder, or air, in which are thrown at the opponent.
- Do not use any inflaming devices.
- Do not use devices that throw things at your opponent.
- Do not stick a robot down onto the Ring, using sucking devices or

glue, or use any type of sticky tires (such as double sticky foam tape) or any device to assist in adding more down force (such as a vacuum device).

Section 4. How to Carry Sumo Matches

Article 7. (How to Carry Sumo Matches) One match shall consist of 3 rounds, within a total time of 3 minutes, unless extended by the Judges.

The team who wins two rounds or receives two "Yuhkoh" points first, within the time limit, shall win the match. A team receives a "Yuhkoh" point when they win a round. If the time limit is reached before one team can get two "Yuhkoh" points, and one of the teams has received one Yuhkoh point, the team with one Yuhkoh point shall win.

When the match is not won by either team within the time limit, the extended match shall be fought during which the team who receives the first Yuhkoh point shall win. However, the winner/loser of the match may be decided by judges or by means of lots, or there can be a rematch.

One Yuhkoh point shall be given to the winner when the judges' decision was called for or lots were employed.

Section 5. Start, Stop, Resume, End a Match

Article 8. (Start) With the chief judge's instructions, the two teams bow in the Outer Ring (For example, stand facing each other, outside the ring platform or "ring area", with ring between), go up to the Ring, and place a robot on or behind the Shikiri line or the imaginary extended Shikiri line. (A robot or a part of a robot may not be placed beyond the front edge of the Shikiri line toward the opponent.). A match starts with the following rules:

For stand-alone robots, be ready to press a start switch. Press the switch when the chief judge announces the start of the round. After 5 seconds, the robot is allowed to start operating, before which players must clear out of the Ring Area.

Article 9. (Stop, Resume) The match stops and resumes when a judge announces so.

Article 10. (End) The match ends when the chief judge announces so. The two teams bring the robots out of the Ring Area, and bow.

Section 6. Time of Match

Article 11 (Time of Match) One Match will be fought for a total of 3 minutes, starting and ending by the chief judge's announcements. For stand-alone robots, the clock shall start ticking 5 seconds after the start is announced.

Article 12. An extended match shall be for 3 minutes, if called by the Judge.

Article 13. The following are not included in the time of the Match:

- The time elapsed after the chief judge announces Yuhkoh and before the match resumes. 30 seconds shall be the standard before the match resumes.
- The time elapsed after a judge announces to stop the match and before the match resumes.

Section 7. Yuhkoh

Article 14. (Yuhkoh) One Yuhkoh point shall be given when:

- You have legally forced the body of your opponent's robot to touch the space outside the Ring, which includes the side of the ring itself.
- A Yuhkoh point is also given in the following cases:
 - Your opponent's robot has touched the space outside the Ring, on its own.
 - Either of the above takes place at the same time that the End of the Match is announced.
 - When a robot has fallen on the Ring or in similar conditions, Yuhkoh will not be counted and the match continues.
- When judges' decision is called for to decide the winner, the following points will be taken into considerations:
 - Technical merits in movement and operation of a robot

- Penalty points during the match
- Attitude of the players during the match
- The match shall be stopped and a rematch shall start when:
 - Both robots are in clinch and stop movements for 5 seconds, or move in the same orbit for 5 seconds, with no progress being made. If it is not clear if progress is being made or not, the Judge can extend the time limit for a clinch or orbiting robots up to 30 seconds.
 - Both robots move, without making progress, or stop (at the exact same time) and stay stopped for 5 seconds without touching each other. However, if one robot stops its movement first, after 5 seconds, he shall be considered not having the will to fight, and the opponent shall receive a Yuhkoh, even if the opponent also stops. If both robots are moving and it isn't clear if progress is being made or not, the Judge can extend the time limit up to 30 seconds.
 - If both robots touch the outside of the ring at about the same time, and it can not be determined which touched first, a rematch is called.

Section 8. Violations

Article 15. (Violations) If the players perform the deeds as described in Articles 6, 16, and 17, the players shall be declared as violating the rules.

Article 16. The player utters insulting words to the opponent or to the judges or puts voice devices in a robot to utter insulting words or writes insulting words on the body of a robot, or any insulting action.

Article 17. A player:

- Enters into the Ring during the match, except when the player does so to bring the robot out of the Ring upon the chief judge's announcement of Yuhkoh or stopping the match. To enter into the Ring means:
 - A part of the player's body is in the Ring, or

- A player puts any mechanical kits into the Ring to support his/her body.
- Performs the following deeds:
- Demands to stop the match without appropriate reasons.
- Takes more than 30 seconds before resuming the match, unless the Judge announces a time extension.
- Remotely operating the robot.
- Starts operating the robot within 5 seconds after the chief judge announces the start of the match (for stand-alone robots).
- Does or says that which should disgrace the fairness of the match.

Section 9. Penalties

Article 18. (Penalties) Those who violate the rules with the deeds described in Articles 6 and 16 shall lose the match. The judge shall give two Yuhkoh points to the opponent and order the violator to clear out. The violator is not honored with any rights.

Article 19. Each occasion of the violations described in Article 17 shall be accumulated. Two of these violations shall give one Yuhkoh to the opponent.

Article 20. The violations described in Article 17 shall be accumulated throughout one match.

Section 10. Injuries and Accidents during the Match

Article 21. (Request to stop the Match) A player can request to stop the game when he/she is injured or his/her robot had an accident and the game cannot continue.

Article 22. (Unable to continue the Match) When the game cannot continue due to player's injury or robot's accident, the player who is the cause of such injury or accident loses the match. When it is not clear which team is such a cause, the player who cannot continue the game, or who requests to stop the game, shall be declared as the loser.

Article 23. (Time required to handle injury/accident) Whether the game should continue in case of injury or accident shall be decided by the judges and the Committee members. The decision process shall take no longer than five minutes.

Article 24. (Yuhkoh given to the player who cannot continue) The winner decided based on Article 22 shall gain two Yuhkoh points. The loser who already gained one Yuhkoh point is recorded as such. When the situation under Article 22 takes place during an extended match, the winner shall gain one Yuhkoh point.

Section 11. Declaring Objections

Article 25. (Declaring Objections) No objections shall be declared against the judges' decisions.

Article 26. The lead person of a team can present objections to the Committee, before the match is over, if there are any doubts in exercising these rules. If there is no Committee member present, the objection can be presented to the Judge, before the match is over.

Section 12. Requirements for Identifications for Robots

Article 27. (Identifications for Robots) Some type of name or number, to identify the robot (as registered in the contest) must be easily readable on the robot's body, while the robot is in competition.

Section 13. Miscellaneous

Article 28. (Flexibility of Rules) As long as the concept and fundamentals of the rules are observed, the rules shall be so flexible that they will be able to encompass the changes in the number of players and of the contents of matches.

Article 29. (Change in Rules) Any changes to or obsolescence of these rules shall be decided by the General Committee Meeting based on the Sumo Match Committee Rules.

