EXPERIMENT2

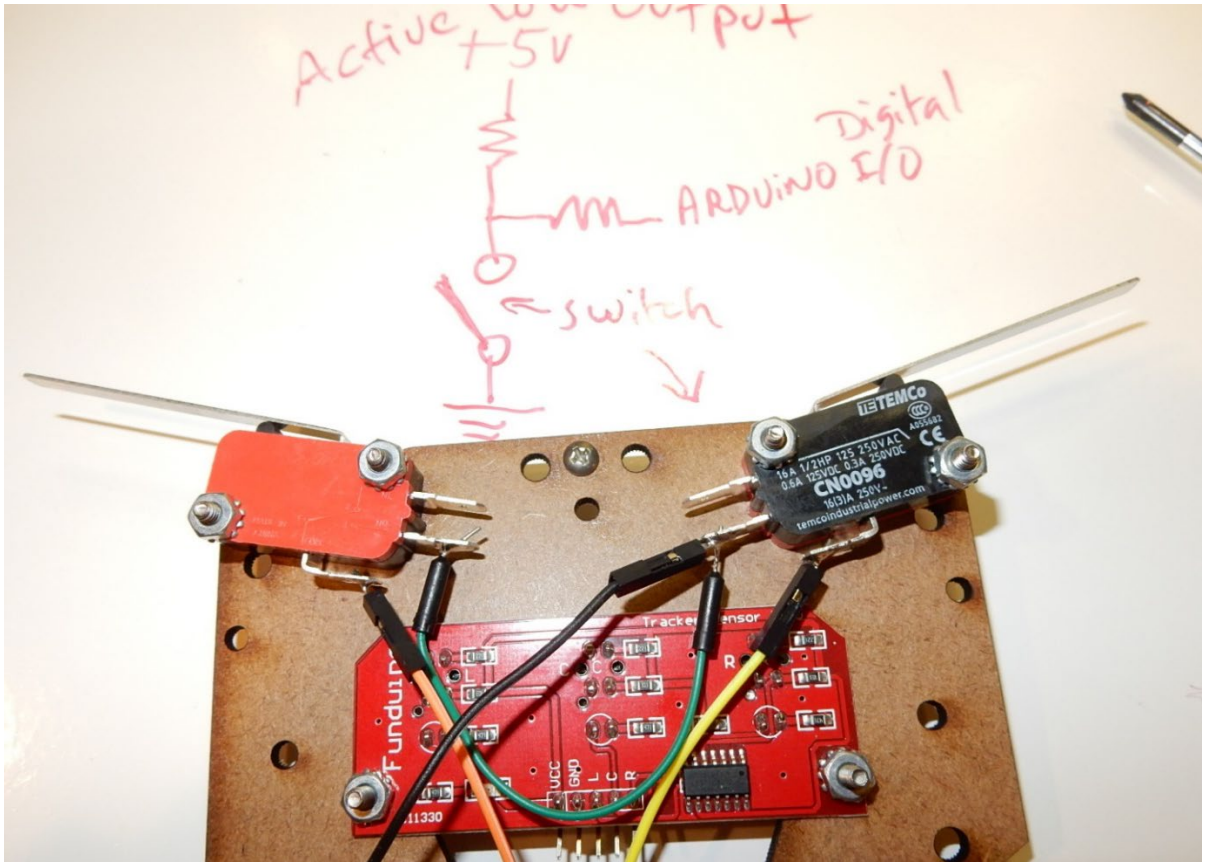
V3 Robot Navigates Autonomously with Feelers and Infrared Sensors

Purpose: Install and program the feeler sensors (switches) plus the adjustable range infrared sensors to navigate autonomously without bumping into objects. Both left and right feelers/IR sensors must work. There is a small introduction to AI program designed to escape corners and must also work. The feelers can act as secondary sensors in case the other sensors fail or encounter blind spots. Leave the feelers connected for the next Experiment 3 as you will use them again.

For the final lab demonstration make sure the robot roams around autonomously using the Feelers and Infrared.

What you will learn:

- Autonomous robot navigation by reading digital values from input sensors
- Intro to Artificial Intelligence algorithm used for autonomous navigation



‘Feelers’ or switches installed on the lower robot chassis

A view from the bottom so you can see where the screws go

If you have not assembled the feelers please do so now. Please refer to the How to build robot guide – Assembling the Feelers or look at the photos above and use the ¾” screws or longer to secure the feelers to the robot’s lower chassis.

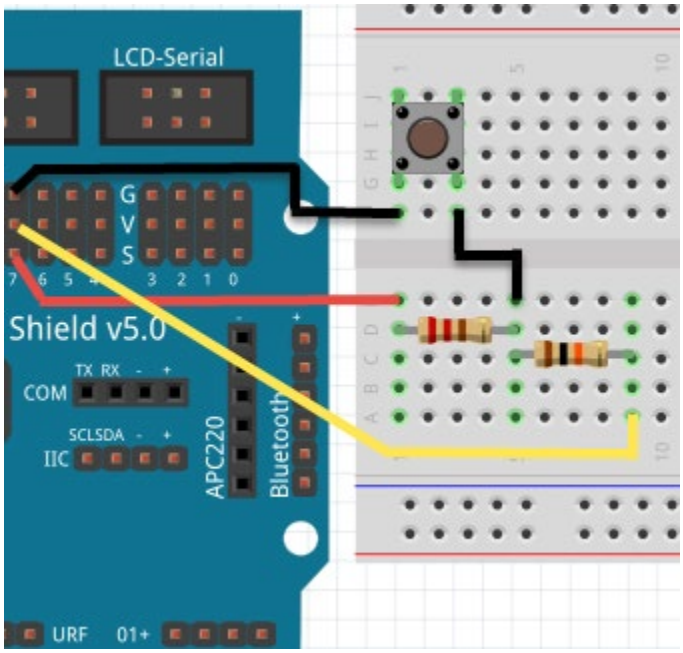
Digital Sensor Input – using ON-OFF switches

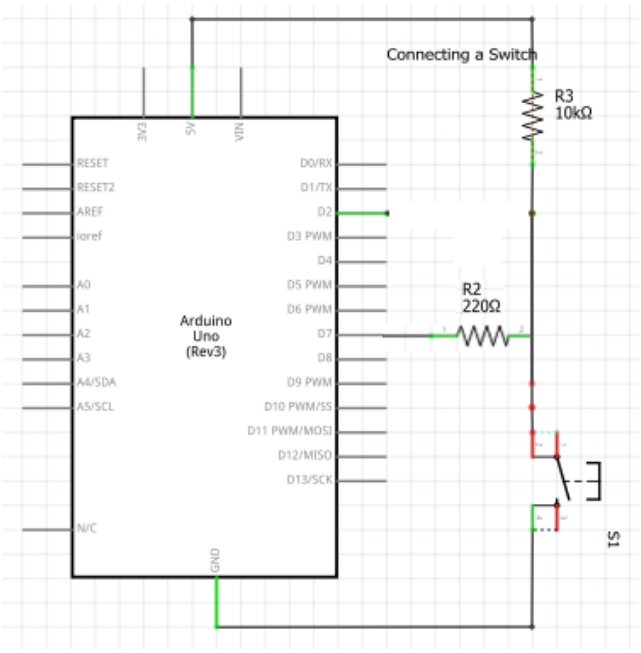
First remember that digital means 1 or 0 in computer terms. The idea here is that digital sensor will return a value of 1 or 0 depending on how you wire them. Microcontrollers and programming languages need to know the real status of a pin input so we must either put a 1 by applying a voltage to the pin or a 0 by applying a sinking value to ground. You will learn more about this during the sensors lecture. This sample code will turn on an LED when you press a simple temporary on button switch.

When the pushbutton is open (un-pressed) there is no connection between the two legs of the pushbutton, so the pin is connected to ground (through the pull-down resistor) and we read a LOW. When the button is closed (pressed), it makes a connection between its two legs, connecting the pin to 5 volts, so that we read a HIGH.

You can also wire this circuit the opposite way, with a pull-up resistor keeping the input HIGH, and going LOW when the button is pressed. If so, the behavior of the sketch will be reversed, with the LED normally on and turning off when you press the button.

If you disconnect the digital I/O pin from everything, the LED may blink erratically. This is because the input is "floating" - that is, it will randomly return either HIGH or LOW. That's why you need a pull-up or pull-down resistor in the circuit.





Once you wire the circuit with the feelers mounted, upload this code to your robot.

```
// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 7;                // the number of the pushbutton pin
const int ledPin =  13;                 // the number of the LED pin – built-in LED

// variables will change:
int buttonState = 0;                            // variable for reading the pushbutton status

void setup() {
                                        // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
                                        // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop(){
                                                // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);
                                                // check if the pushbutton is pressed.
                                                // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
                                                // turn LED on:
    digitalWrite(ledPin, HIGH);
  }
```

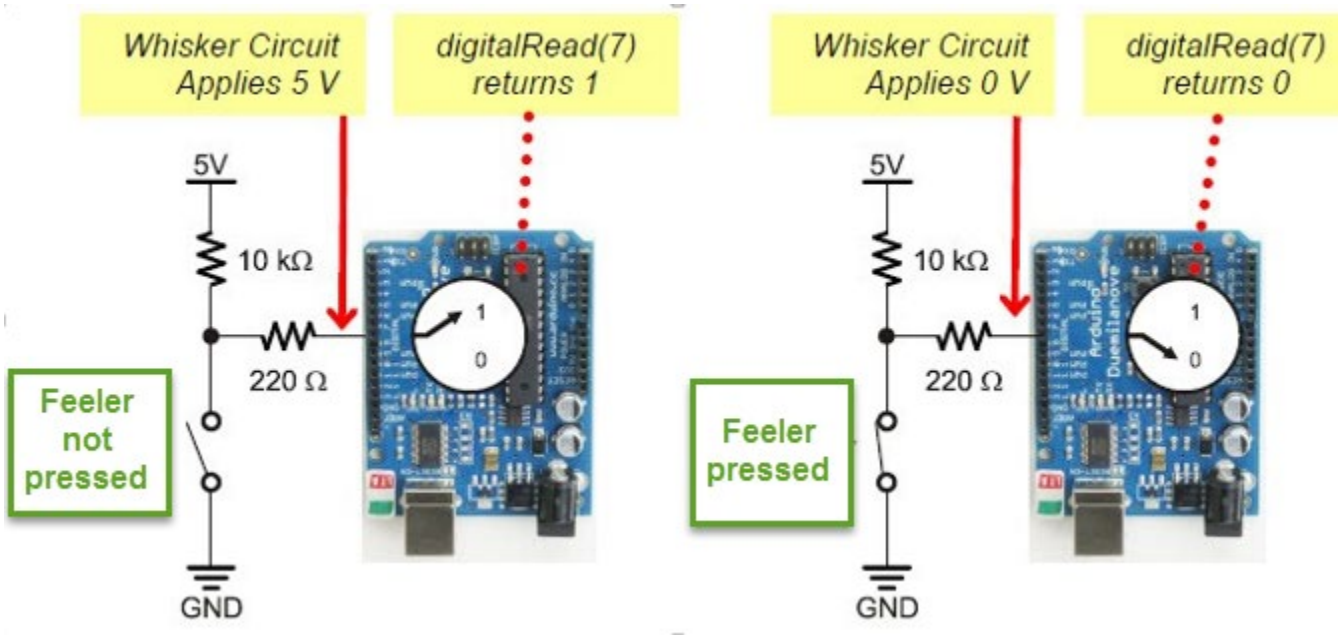
```
  else {
                                        // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

Digital Input – using on-off switches or feelers

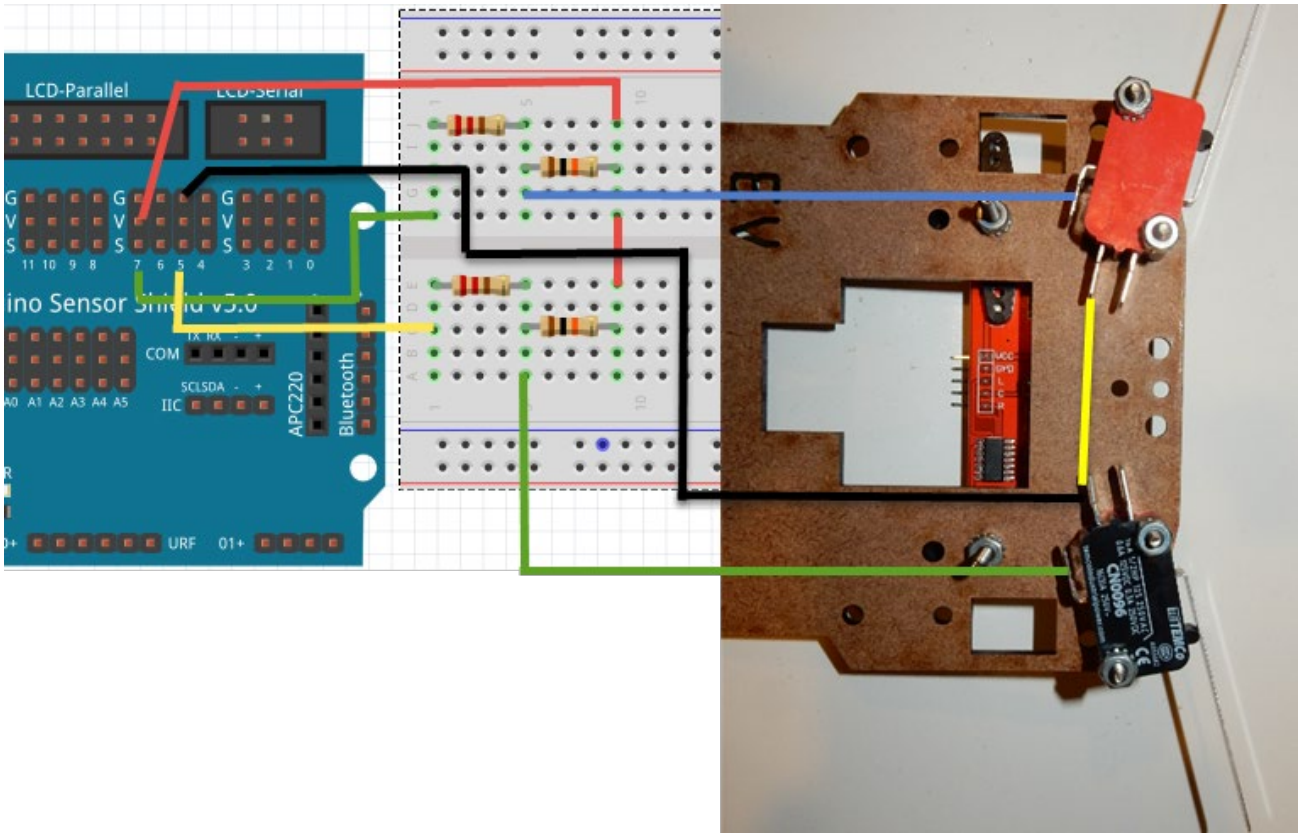
Since each feeler is connected to an I/O pin plus voltage and ground, the Arduino can be programmed to detect which voltage is applied to each circuit, 5 V or 0 V. First, set each pin to input mode with **pinMode(*pin*, *mode*)**, and then detect the pin's state, HIGH or LOW, with **digitalRead(*pin*)** function.

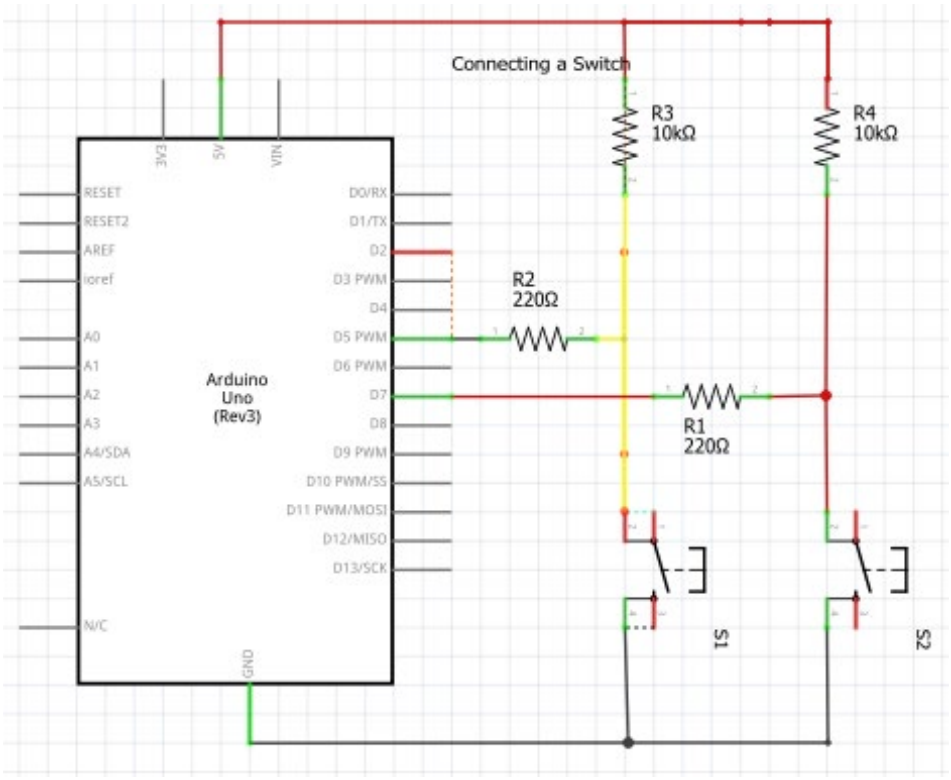
Take a look at the figure below. On the left, the circuit applies 5 V when the feeler is not pressed, so **digitalRead (7)**returns 1 (**HIGH or 5volts on pin 7**). On the right, the circuit applies 0 V when the feeler is pressed, so **digitalRead(7)** returns 0 (**LOW or 0 volts on pin 7**).

Most importantly, your sketch can store the return values in variables, such as **wLeft** and **wRight**, and then use them to trigger actions or make decisions. The next example sketch will demonstrate how.

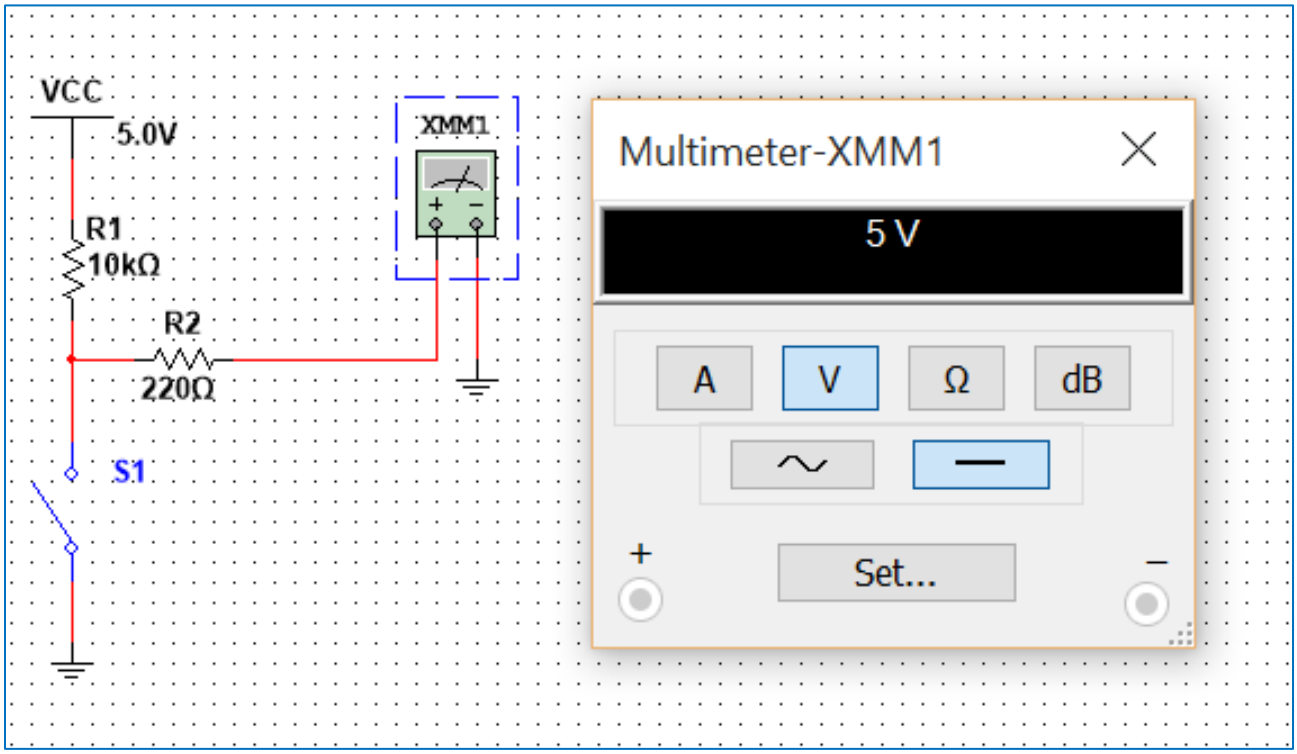


Connecting the feelers to from the robot to your Arduino sensor shield – pins 5 and 7 using 220 ohm and 10K ohm resistors to protect the Arduino board. Wiring color does not matter.

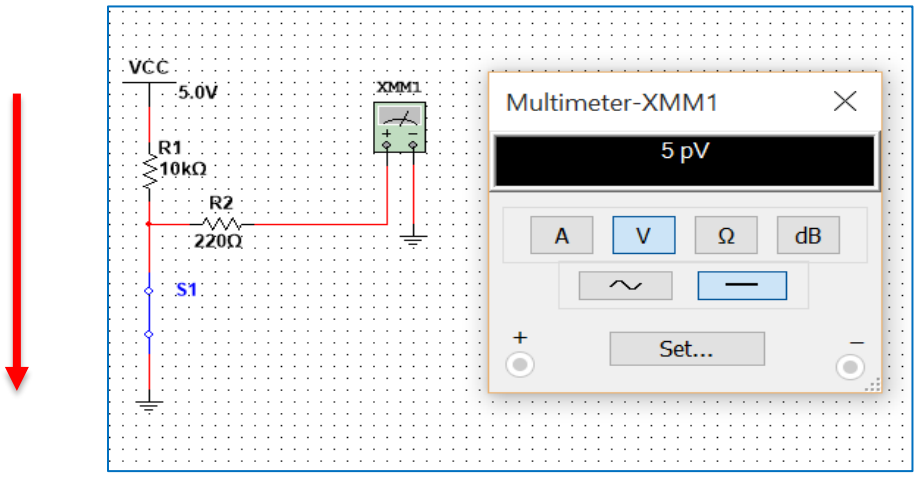




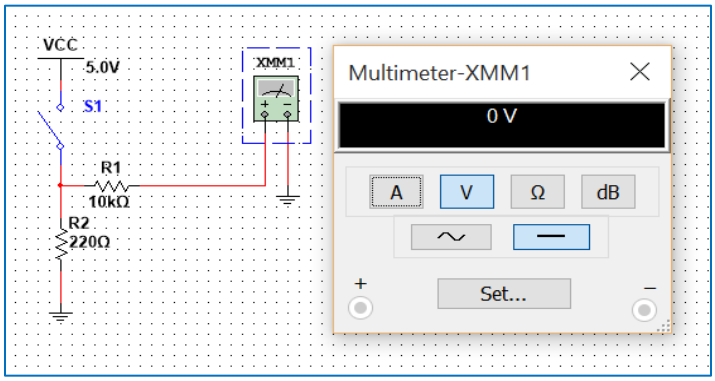
The feeler is basically a normally-open, SPST (Single-Pole, Single-Throw) switch. There are 2 possible choices when wiring a switch; you can either do a resistor pull-up (1) or a resistor pull-down (0) configuration. The pull-up configuration is the most common and it might look reverse of what you are thinking so is on until you push the button then is off.



A **resistor pull-up** is when a "HIGH" or "1" is active by default at the output. The resistor, *R1*, pulls straight from the *VCC* to the *Output* (which is the DMM in these demonstrations).
We can see, on the diagram above, the voltage and current have a clear path to flow (in red) from the *5V* to the *Output*, because the switch is open. *The feelers are setup this way.*

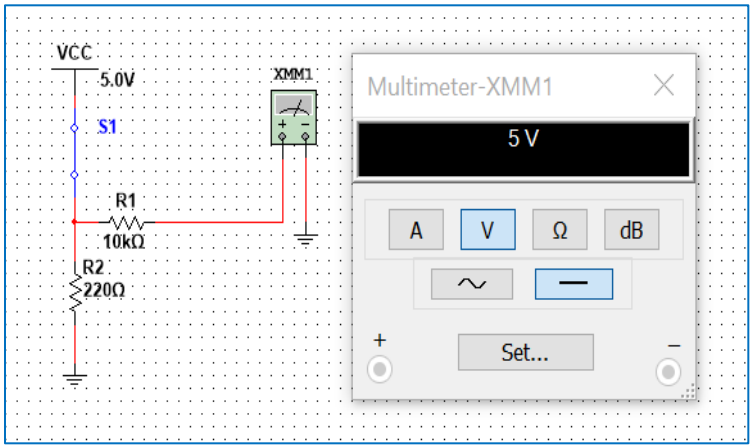


Once the switch is pressed down, the voltage and current will have a clear path to flow (in red) from the *5V* to the *Ground* instead. Note that *5pV* is too low, and is considered *0V*.



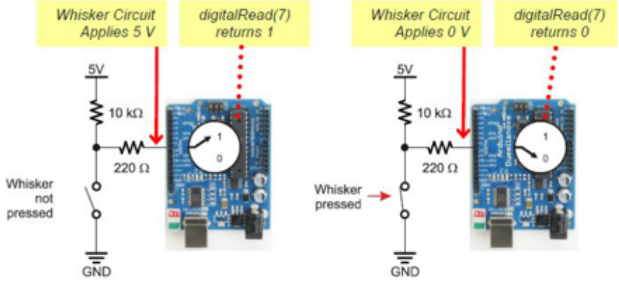
A **resistor pull-down** is when a "LOW" or "0" is active by default at the output instead. Since the switch is open, there are no voltage or current going to through.

Once the switch is pressed, *5V* will flow to both the Arduino board *Input* and *Ground*, as seen below. Unlike the resistor pull-up configuration, we can see a slight disadvantage, with the setup below; the current becomes divided with the parallel resistors. Of course, a much higher *R2* can force more current to flow to *R1* instead, and the Adurino does not require much current to even recognize a voltage value of "HIGH" or "1".

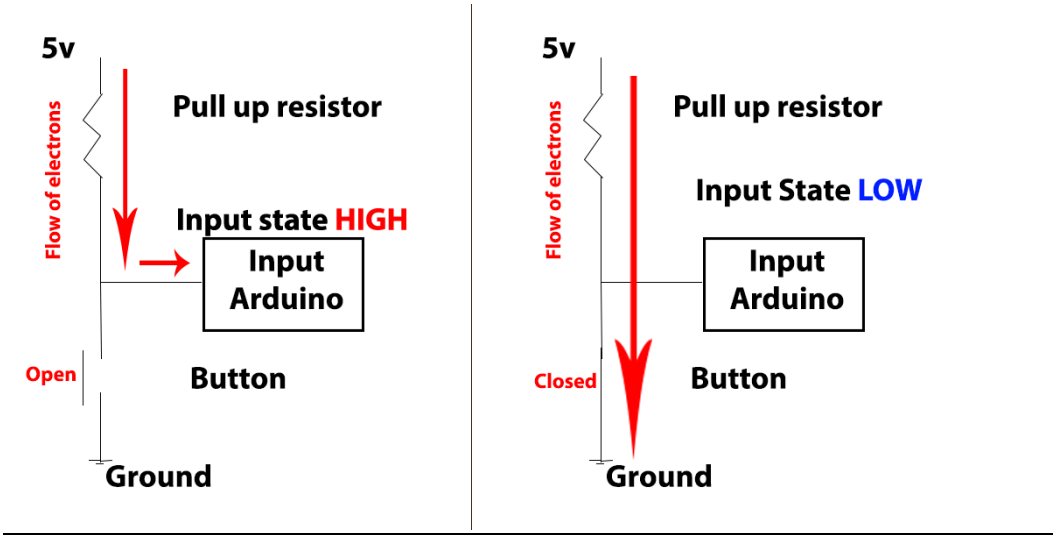


Most sensors work in this manner of sourcing or sinking. These sensors below are industrial sensors used for programmable logic controllers and robots.

The NPN names is referred to the transistor logic that the controller has



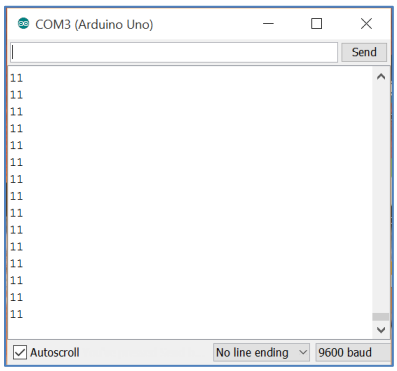
Sinking and Sourcing the current. Sinking Example->



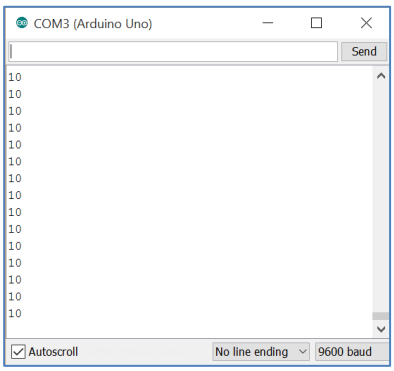
Feeler's Reaction

Because the feelers are setup with the resistor pull-up configuration, the Arduino board will be reading a "HIGH" or "1" from the start. It is considered an *Active-Low Output*, meaning when the feelers (acting as switches) are pressed, an output of "LOW" or "0" will be activated.

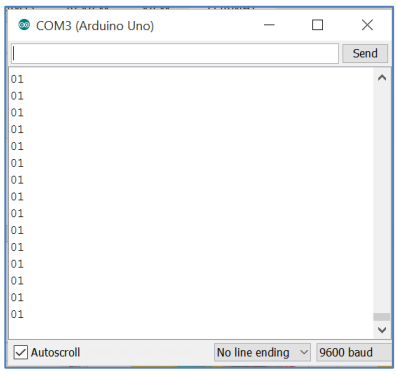
This become useful when having 2 feelers, because we can now know when both, or either sides of the feeler making contact with the wall or object. We can use the "*Serial Monitor*" to see the results like below.



No feeler contact



left feeler pressed

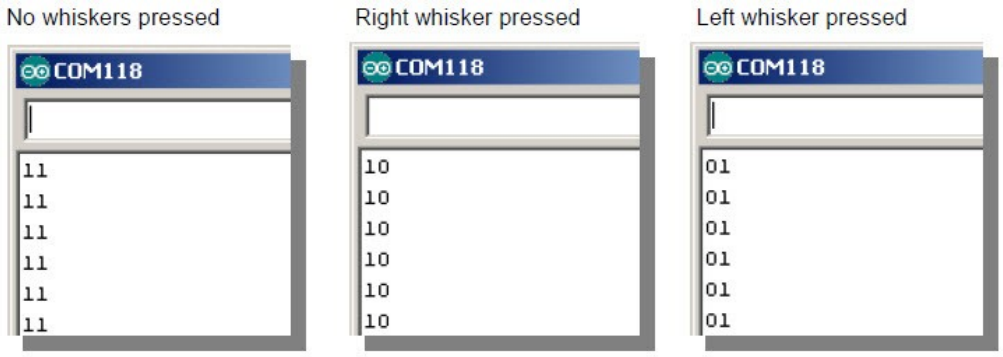


right feeler pressed

Testing the Feelers

The next sketch tests the feelers to make sure they are functioning properly, by displaying the binary values returned by digitalRead(7) and digitalRead(5). This way, you can press each feeler against its 3-pin header on the breadboard, and see if the Arduino's digital pin is sensing the electrical contact.

When neither feeler is pressed up against its 3-pin header, you can expect your Serial Monitor to display two columns of 1's, one for each feeler. If you press just the right feeler, the right column should report 0, and the display should read 10. If you press just the left feeler, the left column should report 1 and the display should read 01. Of course, if you press both feelers, it should display 00.



Active-low Output

The feeler circuits are wired for *active-low output*, which means that they each send a low signal when they are pressed (active) and a high signal when they are not pressed. Since digitalRead returns 0 for a low signal and 1 for a high signal, 0 is what tells your sketch that a feeler is pressed, and 1 tells it that a feeler is not pressed.

Enter, save, and upload TestFeelers to your Arduino.

As soon as the sketch is finished uploading, open the Serial Monitor.

Leave the USB cable connected so that the Arduino can send serial messages to the Serial Monitor.

```
/* DisplayFeelerStates
* Display left and right feeler states in Serial Monitor.
* 1 indicates no contact; 0 indicates contact */

void setup()                  // Built-in initialization block
{
tone(4, 3000, 1000);          // Play tone for 1 second
delay(1000);                  // Delay to finish tone
pinMode(7, INPUT);            // Set right feeler pin to input
pinMode(5, INPUT);            // Set left feeler pin to input

//prepare to display values on the serial monitor
Serial.begin(9600);           // Set data rate to 9600 bps
}
void loop()                   // Main loop auto-repeats
{
byte wLeft = digitalRead(5);  // Copy left result to wLeft
byte wRight = digitalRead(7);// Copy right result to wRight
Serial.print(wLeft);          // Display left feeler state
Serial.println(wRight);       // Display right feeler state
delay(50);                    // Pause for 50 ms to prevent buffer overflow and refresh data
}
```

Look at the values displayed in the Serial Monitor. With no feelers pressed, it should display 11, indicating 5 V is applied to both digital inputs (5 and 7).

Press the right feeler into its three-pin header, and note the values displayed in the Serial Monitor. It should now read 10.

Release the right feeler and press the left feeler into its three-pin header, and note the value displayed in the Serial Monitor again. This time it should read 01.

Press both feelers against both three-pin headers. Now it should read 00.

If the feelers passed all these tests, you're ready to move on. If not, check your sketch and circuits for errors.

Nesting Function Calls to save lines of code

Your sketch doesn't actually need to use variables to store the values from **digitalRead**. Instead, the (1 or 0) value that **digitalRead** returns can be used directly by nesting the function call inside **Serial.print** and sending its return value straight to the Serial Monitor. In that case, your **loop** function would look like this:

```
void loop()                          // Main loop auto-repeats
{
Serial.print(digitalRead(5));        // Display wLeft
Serial.println(digitalRead(7));      // Display wRight

delay(50);                           // Pause for 50 ms
}
```

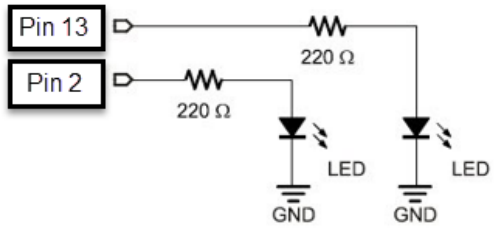
Replace the **loop** function with the one above, upload the sketch, and test the feelers to verify that it functions the same.

Test the Feelers using LEDs (Optional circuit and test program)

What if you have to test the feelers at some later time away from a computer? In that case, the Serial Monitor won't be available, so what can you do? One solution would be to use LED circuits to display the feeler states. All it takes is a simple sketch that turns an LED on when a feeler is pressed or off when it's not pressed.

Parts: (2) resistors, 220 Ω (red-red-brown) (2) LEDs, any color

Build the LED Feeler Testing Circuits (be sure to unplug the robot from USB port)



Programming the LED Feeler Testing Circuits (Optional test program)

Try the following programs to test the feeler sensors.

```
/* TestFeelersWithLeds
* Display left and right feeler states in Serial Monitor.
* 1 indicates no contact; 0 indicates contact.
* Display feeler states with LEDs. LED on indicates contact;
* off indicates none*/

void setup()              // Built-in initialization block
{
pinMode(7, INPUT);  // Set right feeler pin to input
pinMode(5, INPUT);  // Set left feeler pin to input
pinMode(13, OUTPUT);        // Left LED indicator -> output
pinMode(2, OUTPUT);         // Right LED indicator -> output
tone(4, 3000, 1000); // Play tone for 1 second
delay(1000);              // Delay to finish tone

Serial.begin(9600);    // Set serial data rate to 9600
}
void loop()               // Main loop auto-repeats
{
byte wLeft = digitalRead(5);  // Copy left result to wLeft
byte wRight = digitalRead(7);// Copy right result to wRight
if(wLeft == 0)                    // If left feeler contact
{
digitalWrite(13, HIGH);         // Left LED on
}
else  // if no left feeler contact
{
digitalWrite(13, LOW);          // Left LED off
}
if(wRight == 0)                   // If right feeler contact
{
digitalWrite(2, HIGH);          // Right LED on
}
else                        // If no right feeler contact
{
digitalWrite(2, LOW);           // Right LED off
}
Serial.print(wLeft);    // Display wLeft
Serial.println(wRight);         // Display wRight
delay(50);                // Pause for 50 ms
}
```

Recall that **if...else**statements execute blocks of code based on conditions. Here, if **wLeft** stores a zero, it executes the **digitalWrite(13, HIGH)**call. If **wLeft** instead stores a 1, it executes the **digitalWrite(13, LOW)** call. The result? The left LED turns on when the left feeler is pressed or off when it's not pressed. The second **if…else**statement does the same job with **wRight** and the right LED circuit.

Save and upload to your Arduino.

Test the sketch by gently pressing each feeler against its 3-pin header post in the breadboard. The red LEDs on the side of the breadboard where you pressed the feeler should emit light to indicate that the feeler has made contact.

Navigation with feelers (digital sensors) your first autonomous robot!

Previously, our sketches only made the Robot execute a list of movements predefined by you, the programmer. Now that you can write a sketch to make the Arduino monitor feeler switches and trigger action in response, you can also write a sketch that lets the Robot drive and select its own maneuver if it bumps into something. This is an example of *autonomous robot navigation*. Congratulations this will be your first autonomous robot!

Feeler Navigation Overview

The Navigating With Feelers sketch makes the Robot go forward while monitoring its feeler inputs, until it encounters an obstacle with one or both of them. As soon as the Arduino senses feeler electrical contact, it uses an if…else if…else statement to decide what to do. The decision code checks for various feeler pressed/not pressed combinations, and calls navigation functions from the previous experiments to execute back-up-and-turn maneuvers and the robot resumes forward motion until it bumps into another obstacle.

Navigating With Feelers (don't forgot your servo values will vary, used servo values from lab1)

Load the sketch and put the robot on the floor, and try letting it roam. When it contacts obstacles in its path with its feeler switches, it should back up, turn, and then roam in a new direction.

```
//   NavigatingWithFeelers
// Go forward.  Back up and turn if feelers indicate Robot bot bumped into
something.
#include <Servo.h>        // Include servo library
Servo servoLeft;          // Declare left and right servos
Servo servoRight;
void setup()              // Built-in initialization block
{
pinMode(7, INPUT);        // Set right feeler pin to input
pinMode(5, INPUT);        // Set left feeler pin to input
tone(4, 3000, 1000);           // Play tone for 1 second
delay(1000);
                          // Delay to finish tone
servoLeft.attach(11);     // Attach left signal to pin 11
servoRight.attach(10);    // Attach right signal to pin 10
}
void loop()               // Main loop auto-repeats
{
byte wLeft = digitalRead(5);         // Copy left result to wLeft
byte wRight = digitalRead(7);        // Copy right result to wRight
if((wLeft == 0) && (wRight == 0))    // If both feelers contact
{
backward(1000);      // Back up 1 second
turnLeft(800);       // Turn left about 120 degrees
}
else if(wLeft == 0) // If only left feeler contact
```

```
{
backward(1000);          // Back up 1 second
turnRight(400);          // Turn right about 60 degrees
}
else if(wRight == 0)     // If only right feeler contact
{
backward(1000);     // Back up 1 second
turnLeft(400);      // Turn left about 60 degrees
}
else                // Otherwise, no feeler contact
{
forward(20);        // Forward 1/50 of a second
}
}
void forward(int time)   // Forward function
{
servoLeft.writeMicroseconds(1700); // Left wheel counterclockwise
servoRight.writeMicroseconds(1300); // Right wheel clockwise
delay(time);             // Maneuver for time ms
}
void turnLeft(int time)  // Left turn function
{
servoLeft.writeMicroseconds(1300);  // Left wheel clockwise
servoRight.writeMicroseconds(1300); // Right wheel clockwise
delay(time);        // Maneuver for time ms
}
void turnRight(int time) // Right turn function
{
servoLeft.writeMicroseconds(1700); // Left wheel counterclockwise
servoRight.writeMicroseconds(1700); // Right wheel counterclockwise
delay(time);             // Maneuver for time ms
}
void backward(int time)  // Backward function
{
servoLeft.writeMicroseconds(1300); // Left wheel clockwise
servoRight.writeMicroseconds(1700); // Right wheel counterclockwise
delay(time);  // Maneuver for time ms
}
```

How Navigating With Feelers Work

The if...else if...elsestatement in the loop function checks the feelers for any states that require attention. The statement starts with if((wLeft == 0) && (wRight == 0)). Translated to English, it reads "if the wLeft variable AND the wRight variable both equal zero." If both variables are zero, the two calls in the if statement's code block get executed: backward(1000) and turnLeft(800).

```
if((wLeft == 0) && (wRight == 0)) // If both feelers contact
{
backward(1000); // Back up 1 second
turnLeft(800); // Turn left about 120 degrees
}
```

In the **if…else if…else** statement, the sketch skips code blocks with conditions that are not true, and keeps checking until it either finds a condition that's true or runs out of conditions. When the sketch finds a true statement, it executes whatever is in its code block, then it skips to the end of the **if…else if…else** statement without checking any more conditions, and moves on to whatever else comes next in the sketch.

So, if both feelers are not pressed, that first **if** statement is not true and its code block is skipped. The sketch will check the first **else if** statement. So, maybe the left feeler is pressed and the calls in this statement's code block will run. After backing up for one second and turning left for 0.4 seconds, the sketch skips the rest of the conditions and moves on to whatever comes after that last **else** statement.

```
else if(wLeft == 0) // If only left feeler contact
{
backward(1000);  // Back up 1 second
turnRight(400);    // Turn right about 60 degrees
}
```

If it's the right feeler that detects an obstacle, the first two code blocks will be skipped, and the **if(wRight ==0)** block will run.

```
else if(wRight == 0)        // If only right feeler contact
{
backward(1000);             // Back up 1 second
turnLeft(400);              // Turn left about 60 degrees
}
```

An **else** condition functions as a catch-all for when none of the statements preceding it were true. It's not required, but in this case, it's useful for when no feelers are pressed. If that's the case, it allows the Robot to roll forward for 20 ms. Why so little time before the loop repeats? The small forward time before re-checking allows the Robot to respond quickly to changes in the feeler sensors as it rolls forward.

```
else                    // Otherwise, no feeler contact
{
forward(20);            // Forward 1/50 of a second
}
```

The forward, backward, turnLeft and turnRight functions were introduced earlier in lab1 1.

Try this: You can also modify the sketch's if...else if...else statements to make the LED indicators broadcast which maneuver the Robot is running. Just add digitalWrite calls that send HIGH and LOW signals to the indicator LED circuits. Here is an example:

```
if((wLeft == 0) && (wRight == 0)) // If both feelers contact
{
digitalWrite(13, HIGH);  // Left LED on
digitalWrite(2, HIGH);   // Right LED on
backward(1000);          // Back up 1 second
```

```
turnLeft(800);           // Turn left about 120 degrees
}
else if(wLeft == 0)      // If only left feeler contact
{
digitalWrite(13 HIGH);   // Left LED on
digitalWrite(2, LOW);    // Right LED off
 backward(1000);         // Back up 1 second
turnRight(400);          // Turn right about 60 degrees
}
else if(wRight == 0)          // If only right feeler contact
{
digitalWrite(13, LOW);   // Left LED off
digitalWrite(2, HIGH);   // Right LED on
backward(1000);          // Back up 1 second
turnLeft(400);           // Turn left about 60 degrees
}
else                     // Otherwise, no feeler contact
{
digitalWrite(13, LOW);   // Left LED off
digitalWrite(2, LOW);    // Right LED off
forward(20);             // Forward 1/50 of a second
}
```

Modify the **if...else if...else** statement in NavigatingWithFeelers to make the Robot broadcast its maneuver using the LED indicators.

Remember to set the digital pins to outputs in the setup function so they can actually supply current to the LEDs.

```
pinMode(13, OUTPUT);      // Left LED indicator -> output
pinMode(2, OUTPUT);       // Right LED indicator -> output
```

Intro to Artificial Intelligence for Escaping Corners – Source: http://parallax.com

You may have noticed that with the last sketch, the Robot tends to get stuck in corners. As it enters a corner, its left feeler contacts the wall on the left, so it backs up and turns right. When the robot moves forward again, its right feeler contacts the wall on the right, so it backs up and turns left. Then it contacts the left wall again, and then the right wall again, and so on forever.

Artificial intelligence is a branch of computer science that aims to create intelligent machines. It has become an essential part of the technology industry. (source : techopedia.com)

Research associated with artificial intelligence is highly technical and specialized. The core problems of artificial intelligence include programming computers for certain traits such as:

- Knowledge
- Reasoning

- Problem solving
- Perception
- Learning
- Planning
- Ability to manipulate and move objects

Knowledge engineering is a core part of AI research. Machines can often act and react like humans o**nly if they have abundant information relating to the world (in this case if sensors are activated 4 times do something else).** Artificial intelligence must have access to objects, categories, properties and relations between all of them to implement knowledge engineering. Initiating common sense, reasoning and problem-solving power in machines is a difficult and tedious approach.

On the other hand Machine learning is another core part of AI. Learning without any kind of supervision requires an ability to identify patterns in streams of inputs, whereas learning with adequate supervision involves classification and numerical regressions. Classification determines the category an object belongs to and regression deals with obtaining a set of numerical input or output examples, thereby discovering functions enabling the generation of suitable outputs from respective inputs. Mathematical analysis of machine learning algorithms and their performance is a well-defined branch of theoretical computer science often referred to as computational learning theory.

Programming to Escape Corners

The trick to solving this problem is to count the number of times that alternate feelers make contact with objects. To do this, the sketch has to remember what state each feeler was in during the previous contact. Then, it has to compare those states to the current feeler contact states. If they are opposite, then add 1 to a counter. If the counter goes over a threshold that you (the programmer) have determined, then it's time to do a U-turn and escape the corner, and also reset the counter.

This next sketch relies on the fact that you can nest if statements, one inside another.
The sketch checks for one condition, and if that condition is true, it checks for another condition within the first if statement's code block. We'll use this technique to detect consecutive alternate feeler contacts in the next sketch.

Example Sketch: EscapingCorners (source parallax.com)

This sketch will cause your Robot to execute a reverse and U-turn to escape a corner at either the fourth or fifth alternate feeler press, depending on which one was pressed first.

Test this sketch pressing alternate feelers as the Robot roams. It should execute its reverse and U-turn maneuver after either the fourth or fifth consecutive, alternate feeler contact.

```
/*   EscapingCorners with AI like algorithm
Count number of alternate feeler contacts, and if it exceeds 4, get out of the
corner */

#include <Servo.h>
Servo servoLeft;
Servo servoRight;

byte wLeftOld;      // Previous loop feeler values
byte wRightOld;
byte counter;       // For counting alternate corners
void setup()        // Built-in initialization block
{
pinMode(7, INPUT);  // Set right feeler pin to input
pinMode(5, INPUT);  // Set left feeler pin to input
tone(4, 3000, 1000);      // Play tone for 1 second
delay(1000);              // Delay to finish tone
servoLeft.attach(11);     // Attach left signal to pin
servoRight.attach(10);    // Attach right signal to pin
wLeftOld = 0;       // Init. previous feeler states
wRightOld = 1;
counter = 0;        // Initialize counter to 0
}
void loop()
{
// Corner Escape
byte wLeft = digitalRead(5);   // Copy right result to wLeft
byte wRight = digitalRead(7);  // Copy left result to wRight
if(wLeft != wRight)            // One feeler pressed?
{
                               // Alternate from last time?
if ((wLeft != wLeftOld) && (wRight != wRightOld))
{
counter++;                // Increase count by one
wLeftOld = wLeft;         // Record current for next rep
wRightOld = wRight;
if(counter == 4)          // Reached 4 - Stuck in a corner?
{
wLeft = 0;          // Set up for U-turn
wRight = 0;
counter = 0;        // Clear alternate corner count
}
}
else                // Not alternate from last time
{
counter = 0;        // Clear alternate corner count
}
}
// Feeler Navigation
if((wLeft == 0) && (wRight == 0))   // If both feelers contact
```

```
{
backward(1000);     // Back up 1 second
turnLeft(800);      // Turn left about 120 degrees
}
else if(wLeft == 0) // If only left feeler contact
{
backward(1000);     // Back up 1 second
turnRight(400);     // Turn right about 60 degrees
}
else if(wRight == 0) // If only right feeler contact
{
backward(1000);     // Back up 1 second
turnLeft(400);      // Turn left about 60 degrees
}
else                // Otherwise, no feeler contact
{
forward(20);        // Forward 1/50 of a second
}
}

void forward(int time)   // Forward function
{
servoLeft.writeMicroseconds(1700);  // Left wheel counterclockwise
servoRight.writeMicroseconds(1300); // Right wheel clockwise
delay(time);                        // Maneuver for time ms
}

void turnLeft(int time)             // Left turn function
{
servoLeft.writeMicroseconds(1300);       // Left wheel clockwise
servoRight.writeMicroseconds(1300); // Right wheel clockwise
delay(time);                        // Maneuver for time ms
}

void turnRight(int time)            // Right turn function
{
servoLeft.writeMicroseconds(1700);       // Left wheel counterclockwise
servoRight.writeMicroseconds(1700);      // Right wheel counterclockwise
delay(time);                        // Maneuver for time ms
}

void backward(int time)             // Backward function
{
servoLeft.writeMicroseconds(1300);  // Left wheel clockwise
servoRight.writeMicroseconds(1700); //Right wheel counterclockwise
delay(time);                        // Maneuver for time ms
}
```

Let's look at this sketch

First, three global byte variables are added: wLeftOld, wRightOld, and counter. The wLeftOld and wRightOld variables store the feeler states from a previous feeler contact so that they can be compared with the states of the current contact. Then counter is used to track the number of consecutive, alternate contacts.

```
byte wLeftOld;        // Previous loop feeler values
byte wRightOld;
byte counter;         // For counting alternate corners
```

These variables are initialized in the **setup** function. The **counter** variable can start with zero, but one of the "old" variables has to be set to 1. Since the routine for detecting corners always looks for an alternating pattern, and compares it to the previous alternating pattern, there has to be an initial alternate pattern to start with. So, **wLeftOld** and **wRightOld** are assigned initial values in the **setup** function before the **loop** function starts checking and modifying their values.

```
wLeftOld = 0;          //        Initialize previous feeler
wRightOld = 1;         //        states
counter = 0;           //        Initialize counter to 0
```

The first thing the code below // **Corner Escape** has to do is check if one or the other feeler is pressed. A simple way to do this is to use not-equal operator (**!=**) in an **if** statement. In English, **if(wLeft != wRight)** means "if the **wLeft** variable is not equal to the **wRight** variable…"

```
// Corner Escape
if(wLeft != wRight)          // One feeler pressed?
```

If they are not equal it means one feeler is pressed, and the sketch has to check whether it's the opposite pattern as the previous feeler contact. To do that, a nested **if** statement checks if the current **wLeft** value is different from the previous one and if the current **wRight** value is different from the previous one. That's **if ((wLeft != wLeftOld) && (wRight != wRightOld))**. If both conditions are true, it's time to add 1 to the **counter** variable that tracks alternate feeler contacts. It's also time to remember the current feeler pattern by setting **wLeftOld** equal to the current **wLeft** and **wRightOld** equal to the current **wRight**.

```
if((wLeft != wLeftOld) && (wRight != wRightOld))
{
counter++;           // Increase count by one
wLeftOld = wLeft;    // Record current for next rep
wRightOld = wRight;
```

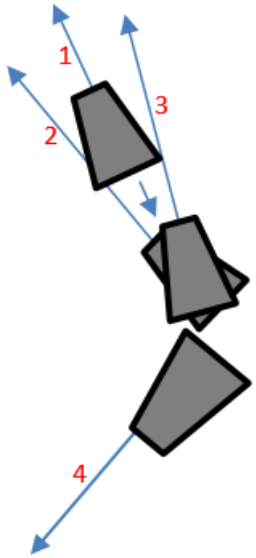
If this is the fourth consecutive alternate feeler contact, then it's time to reset the **counter** variable to 0 and execute a U-turn. When the **if(counter == 4)**statement is true, its code block tricks the feeler navigation routine into thinking both feelers are pressed. How does it do that? It sets both **wLeft** and **wRight** to zero. This makes the feeler navigation routine think both feelers are pressed, so it makes a U-turn.

```
if(counter == 4)     // Stuck in a corner?
{
wLeft = 0;           // Set up feeler states for U-turn
wRight = 0;
```

```
counter = 0;          // Clear alternate corner count
}
```

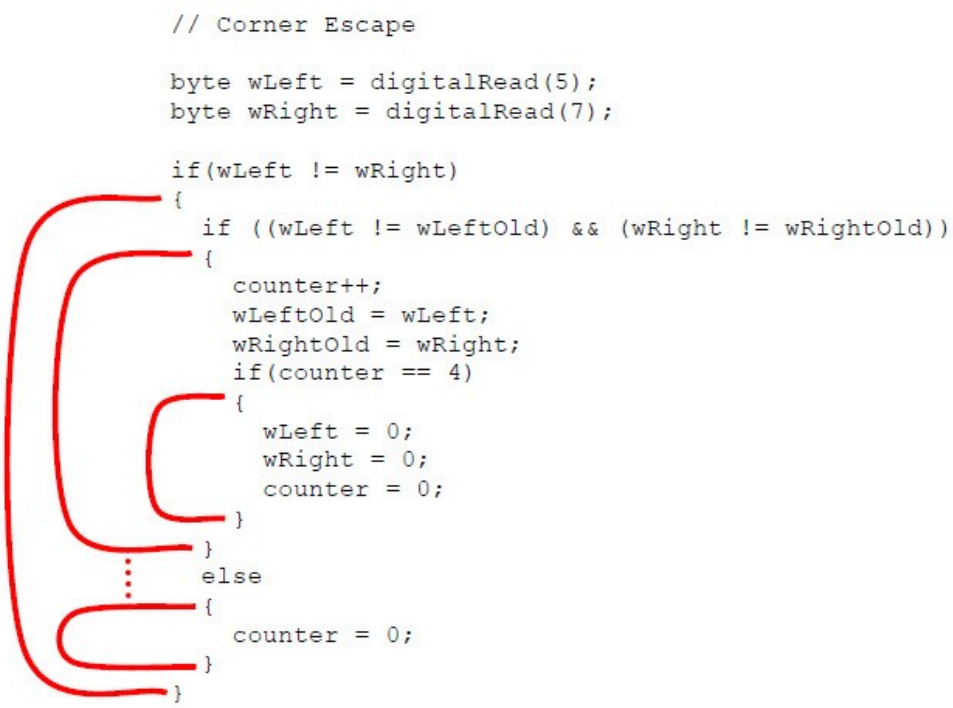
But, if the conditions in **if((wLeft != wLeftOld) && (wRight != wRightOld))** are not all true, it means that this is not a sequence of alternating feeler contacts anymore, so the Robot must not be stuck in a corner. In that case, the **counter** variable is set to zero so that it can start counting again when it really does find a corner.

```
else // Not alternate from last time
{
counter = 0;          // Clear alternate corner count
}
}
```



Path traveled based on algorithm

One thing that can be tricky about nested **if** statements is keeping track of opening and closing braces for each statement's code block. The picture below shows some nested **if** statements from the last sketch. In the Arduino editor, you can double-click on a brace to highlight its code block.



```
// Corner Escape

byte wLeft = digitalRead(5);
byte wRight = digitalRead(7);

if(wLeft != wRight)
{
  if ((wLeft != wLeftOld) && (wRight != wRightOld))
  {
    counter++;
    wLeftOld = wLeft;
    wRightOld = wRight;
    if(counter == 4)
    {
      wLeft = 0;
      wRight = 0;
      counter = 0;
    }
  }
  else
  {
    counter = 0;
  }
}
```

In this picture, the **if(wLeft != wRight)**statement's code block contains all the rest of the decision-making code. If it turns out that **wLeft** is equal to **wRight**, the Arduino skips to whatever code follows that last closing brace }. The second level **if**statement compares the old and new **wLeft** and **wRight** values with **if**

((wLeft != wLeftOld) && (wRight != wRightOld)). Notice that its code block ending brace is just below the one for the **if(counter==4)**block. The **if ((wLeft != wLeftOld) && (wRight != wRightOld))**statement also has an **else** condition with a block that sets **counter** to zero if the feeler values are not opposite from those of the previous contact.

Study the code in the picture carefully.

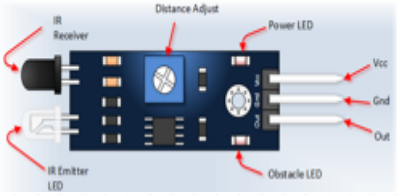
Imagine that **wLeft** = 0 , **wRight** = 0 and **counter** == 3, and think about what this statement would do.

Imagine that **wLeft** = 1 , **wRight** = 0 , **wLeftOld** = 0 , **wRight** = 1 and **counter** == 3. Try walking through the code again line by line and explain what happens to each variable at each step.

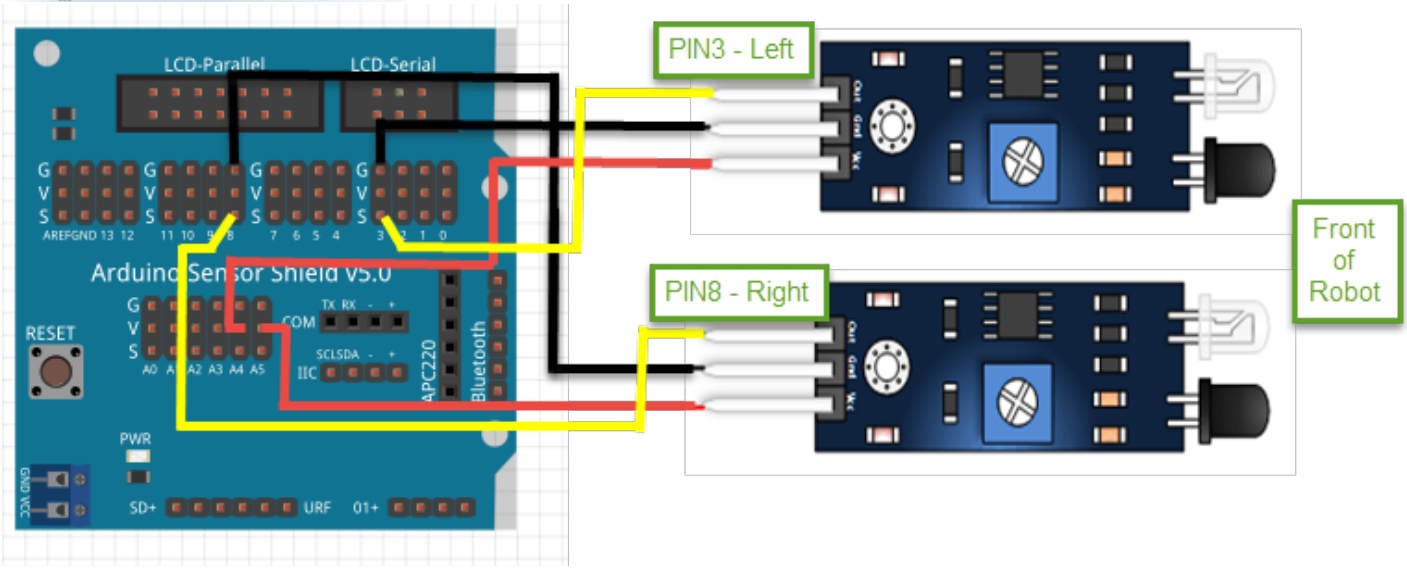
Adding Infrared Sensors for autonomous navigation – You will figure how to add these extra sensors on your own and include the demo and code in your report

As part of this experiment we are going to add infrared sensors for autonomous navigation.
These particular sensors already have an analog to digital converter built-in so they are connected to digital ports and their output is 1 or 0 just like the feelers. Connect the IR collision avoidance sensors as shown below. We use pins 3 and 8 as the signal pins. These are primary sensors and feelers are secondary sensor.

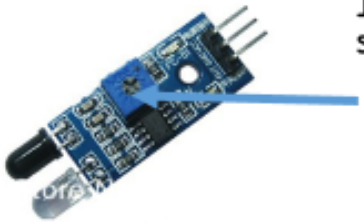
Installing top infrared collision avoidance sensors



Paying attention to the Ground (G), signal (S) and VCC (V) pins. Use V pins as shown below



Installing top infrared collision avoidance sensors

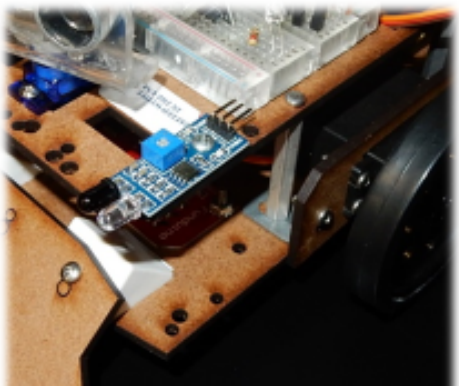


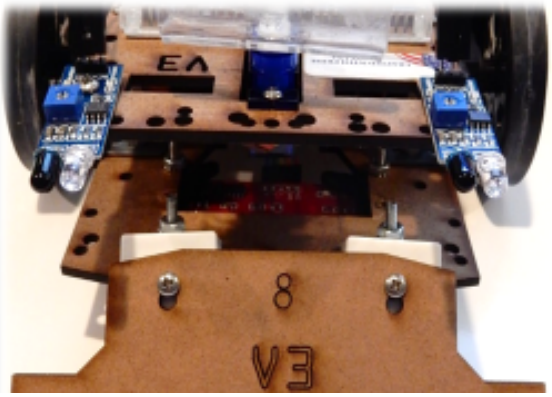
Install the sensors on the top chassis as shown using the ½" crews and nylon nuts

You can adjust the sensitivity of the distance manually using your star screwdriver. Green LED turns ON when it detects an object. The output of the signal on this sensor is 0 or 1

Parts:

(2) nylon nuts
(2) ½" screws
(2) IR adjustable sensors
Lower laser cut chassis
star screwdriver
(3) F-F DuPont wires





Module parameters:

1. when the module detects obstacles in front of the signal, the green indicator on the circuit board light level and at the same time the OUT port output low level signal, the detection module from 2 ~ 30cm, 35°detection Angle, test distance can be adjusted through the potentiometer, adjustable potentiometer clockwise, the detection distance increases; Counter-clockwise tuning potentiometer, detection distance is reduced.
2. sensors, active infrared reflection detection, therefore the reflectivity and shape of the target is the key of the detection range. White one black detection range, minimum maximum; Small area of the object distance, large distance.
3. the sensor module output port OUT can be directly connected to the microcontroller IO port can, also can drive a 5v relay directly; Connection mode: the VCC - VCC; GND - GND; The OUT - the IO
4. the comparator USES the LM393, working stability;
5. can be used for 3 to 5V DC power to power supply module. When power on, the red power indicator light lit;
6. 3 mm screw holes, easy fixed, installation;
7. circuit board size: 3.2cm*1.4cm
8. each module is shipped already compare threshold voltage through the potentiometer to adjust good, not special circumstances, please do not arbitrarily adjust the potentiometer.

Module interface specification:

1. VCC voltage is 3.3V to 5V converter (which can be directly connected to 5V single-chip microcontroller and 3.3V)
2. GND external GND
3. OUT of small plate digital output interface (0 and 1)

10

The Robot can already use the feelers (mechanical switches) to get around, but it only detects obstacles when it bumps into them.

The infrared receiver that sends the Arduino high/low signals to indicate whether it detects the infrared LED's light reflected off an object.

Infrared Light Signals

Infrared is abbreviated IR, and it is light the human eye cannot detect. The IR LEDs introduced in this chapter emit infrared light, just like the red LEDs we've been using to emit visible light.

The infrared receivers in this chapter detect infrared light, these infrared receivers are not just detecting ambient light, but they are designed to detect infrared light flashing on and off very quickly from pulses that the Arduino will output.

The infrared LED that the Robot will use as a tiny headlight is actually the same one you can find in just about any TV remote. The TV remote flashes the IR LED to send messages to your TV. The microcontroller in your TV picks up those messages with an infrared receiver like the one your Robot will use.
The TV remote sends messages by flashing the IR LED very fast, at a rate of about 38 kHz (about 38,000 times per second). The IR receiver only responds to infrared if it's flashing at this rate. This prevents infrared from sources like the sun and incandescent lights from being misinterpreted as messages from the remote. So, to send signals that the IR receiver can detect, your Arduino will have to flash the IR LED on/off at 38 kHz.

Some fluorescent lights do generate signals that can be detected by the

IR receivers.
These lights can cause problems for your Robot's infrared headlights. One of the things you will do in this chapter is develop an infrared interference "sniffer" that you can use to test the fluorescent lights near your Robot courses.

The light sensors inside most digital cameras, including cell phones and webcams, can all detect infrared light. By looking through a digital camera, we can "see" if an infrared LED is on or off. These photos show an example with a digital camera and a TV remote. When you press and hold a button on the remote and point the IR LED into the digital camera's lens, it displays the infrared LED as a flashing, bright white light.

With a button pressed and held, the IR LED doesn't look any different.

Through a digital camera display, the IR LED appears as a flashing, bright white light.

The pixel sensors inside the digital camera detect red, green, and blue light levels, and the processor adds up those levels to determine each pixel's color and brightness. Regardless of whether a pixel sensor detects red, green, or blue, it detects infrared. Since all three pixel color sensors also detect infrared, the digital camera display mixes all the colors together, which results in white.

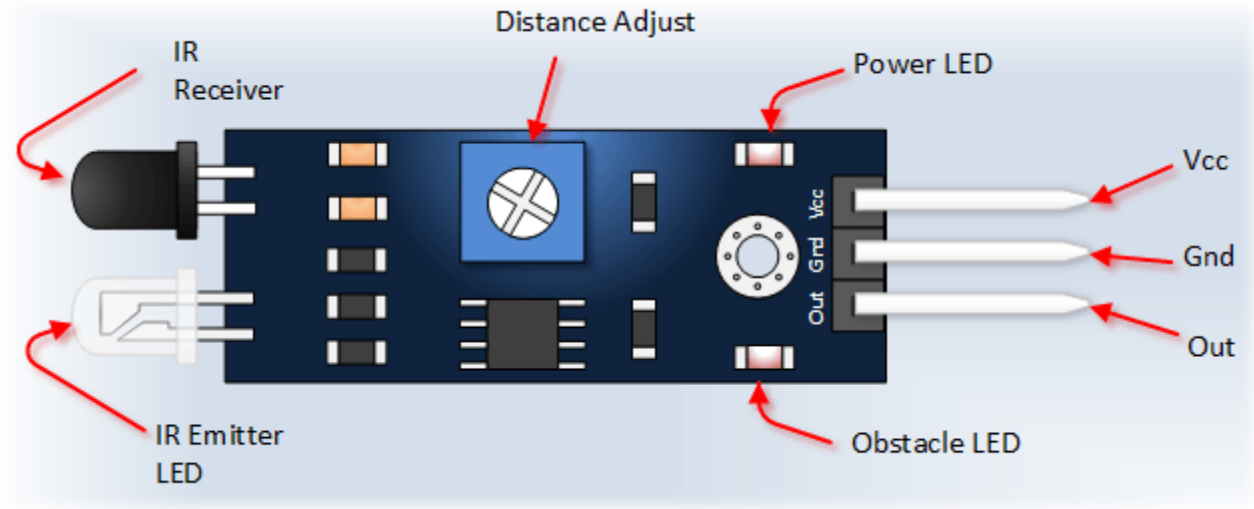
***Infra* means *below*, so *infrared* means *below red*.**
The name refers to the fact that the frequency of infrared light waves is less than the frequency of red light waves. The wavelength our IR LED transmits is 980 nanometers (abbreviated nm), and that's the same wavelength our IR receiver detects. This wavelength is in the near-infrared range. The far infrared range is 2000 to 10,000 nm, and certain wavelengths in this range are used for night-vision goggles and IR temperature

Use the following code to test each IR sensor.

```
// IR Obstacle Collision Detection Sensors Testing

int LED = 13;                 // Use the onboard Uno LED
int isObstaclePin = 3;        // This is our input pin
int isObstacle = HIGH;        // HIGH MEANS NO OBSTACLE

void setup() {
  pinMode(LED, OUTPUT);
  pinMode(isObstaclePin, INPUT);
  Serial.begin(9600);
```

```
  }


void loop() {
  isObstacle = digitalRead(isObstaclePin);
  if (isObstacle == LOW)
  {
    Serial.println("OBSTACLE!!, OBSTACLE!!");
    digitalWrite(LED, HIGH);
  }
  else
  {
    Serial.println("clear");
    digitalWrite(LED, LOW);
  }
  delay(200);
}
```

Change the pins isObstaclePin = 3 to pin 8 to test the other side.



You can adjust the distance sensor (how far away they are able to detect an obstacle in front of it) by turning the small screws on top of it as shown by the arrow on the graphic below.

To complete the sketch with using both feelers and IR sensors see tips below.
Just like the "feeler" switches in the front detect something when they are triggered the same way the IR sensors sense something when you put an object in front of it and it gives you a value of 0 or 1.
In the instructions you have a sample of how to wire and test one infrared sensors as shown below. One goes to pin 3 and the other to pin 8.
So the challenge is for you to figure it out, but here are some hints.

- Just declare the two IR sensors just like the switches, but with different names so instead of *byte wLeft = digitalRead(5);* it would be for example byte *irLeft = digitalRead(3);*. They act the same way in that they provide an output of 1 or 0.

- Remember that && is AND and || is OR so just add the two new sensors to the same lines your already have for the switches. Think of the switches and IR sensors being in parallel with each other.
Example: if((wLeft == 0) && (wRight == 0)) you would do something like this if((wLeft == 0) && (wRight == 0) || (irLeft==0)&&(irRight==0))

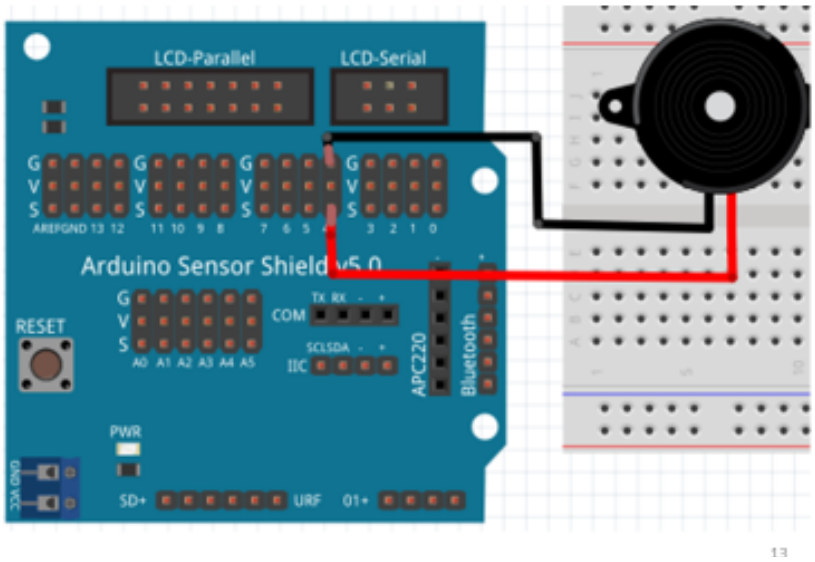
Another example: else if(wLeft == 0) // If only left feeler contact would be: else if((wLeft == 0) || (irLeft==0)) // If only left feeler or IR contact

Go ahead and wire the speaker on pin 4 before doing the final program.

Installing the speaker

Parts:
(1) piezo buzzer
(2) M-F DuPont wires

While you can use any color wires you want, you still need to be careful that the positive (+) side is connected to pin 4 signal (S) pin. Connect the other wire to any available ground (G) pin of the sensor shield



Final Solution to Experiment 2 so you can check your own program. Remember that servo values will vary. Also the IR sensors may need to be adjusted using the small star screwdriver. There are many ways to solve this program including using degrees instead of microseconds. As part of this challenge please make sure to add the AI escaping corners part to this program. You will need to demo this.

```
#include <Servo.h>
// robot navigates autonomously using infrared and feeler (switches)
Servo leftServo, rightServo;     //Declares servos
byte leftIRsensor =  3;   // Left IR sensor input
byte rightIRsensor = 8;   // Right IR sensor input
byte leftIRvalue = HIGH;  // HIGH means no obstacle
byte rightIRvalue = HIGH;   //HIGH means no obstacle
byte leftSwitch =  5;     //Left switch
byte rightSwitch = 7;     //Right switch
byte leftSwitchvalue = HIGH;
byte rightSwitchvalue = HIGH;
byte speaker = 4;     //Assigns speaker to pin 4

void setup() {
```

```
  leftServo.attach(10);   //Attach left servo to pin 10
  rightServo.attach(11);  //Attach right servo to pin 11
  pinMode(leftIRsensor, INPUT);  //Assigns leftIRsensor as input
  pinMode(rightIRsensor, INPUT); //Assigns rightIR sensor as input
  pinMode(leftSwitch, INPUT);    //Assigns leftSwitch as input
  pinMode(rightSwitch, INPUT); //Assigns rightSwitch ass input
  tone(speaker, 1000, 1000);

}

void loop()
{
  leftIRvalue = digitalRead(leftIRsensor);
  rightIRvalue = digitalRead(rightIRsensor);
  leftSwitchvalue = digitalRead(leftSwitch);
  rightSwitchvalue = digitalRead(rightSwitch);

if ((leftIRvalue == LOW || leftSwitchvalue == LOW)&&(rightIRvalue == LOW || rightSwitchvalue == LOW))

  {
    backward(500);
    turnRight(1850);

  }
else if ((leftIRvalue == LOW || leftSwitchvalue == LOW) && (rightIRvalue == HIGH || rightSwitchvalue ==
HIGH))
  {
    backward(300);
    turnRight(1000);

  }
else if ((leftIRvalue == HIGH || leftSwitchvalue == HIGH) && (rightIRvalue == LOW || rightSwitchvalue ==
LOW))

  {
    backward(300);
    turnLeft(1000);

  }
  else
  {
    forward(50);
  }
}
```

```
void forward(int time)

{

  leftServo.writeMicroseconds(1700);

  rightServo.writeMicroseconds(1300);
  delay(time);

}

void backward(int time)
{
  leftServo.writeMicroseconds(1300);
  rightServo.writeMicroseconds(1700);
  delay(time);

}

void turnLeft(int time)
{
  leftServo.writeMicroseconds(1300);
  rightServo.writeMicroseconds(1300);
  delay(time);
}

void turnRight(int time)
{
  leftServo.writeMicroseconds(1700);
  rightServo.writeMicroseconds(1700);
  delay(time);
}
```

End of Experiment 2

```
  This is another way to doing counting using the modulo operator. Try it
  and see if you can redo escaping corners using the modulo operator.

/*  State change detection (edge detection): Often, you don't need to know the
state of a digital input all the time, but you just need to know when the input
changes from one state to another. For example, you want to know when a button
goes from OFF to ON.  This is called state change detection, or edge detection.
This example shows how to detect when a button or button changes from off to on
and on to off.

This example code is in the public domain.
http://www.arduino.cc/en/Tutorial/ButtonStateChange */

  // this constant won't change:
```

```
const int  buttonPin = 7;  // the pin that the pushbutton is attached to
const int ledPin = 13;     // the pin that the LED is attached to

// Variables will change:
int buttonPushCounter = 0;  // counter for the number of button presses
int buttonState = 0;        // current state of the button
int lastButtonState = 0;    // previous state of the button

void setup() {
  pinMode(buttonPin, INPUT);
  pinMode(ledPin, OUTPUT);
  Serial.begin(9600);
}


void loop() {
  // read the pushbutton input pin:
  buttonState = digitalRead(buttonPin);

  // compare the buttonState to its previous state
  if (buttonState != lastButtonState) {
    // if the state has changed, increment the counter
    if (buttonState == HIGH) {
      // if the current state is HIGH then the button
      // wend from off to on:
      buttonPushCounter++;
      Serial.println("on");
      Serial.print("number of button pushes:  ");
      Serial.println(buttonPushCounter);
    } else {
      // if the current state is LOW then the button
      // wend from on to off:
      Serial.println("off");
    }
    // Delay a little bit to avoid bouncing
    delay(50);
  }
  // save the current state as the last state,
  //for next time through the loop
  lastButtonState = buttonState;

  // turns on the LED every four button pushes by
  // checking the modulo of the button push counter.
  // the modulo function gives you the remainder of
  // the division of two numbers 4/4=1 with remainder 0:
  if (buttonPushCounter % 4 == 0) {
    digitalWrite(ledPin, HIGH);
  } else {
    digitalWrite(ledPin, LOW);
  }

}
```