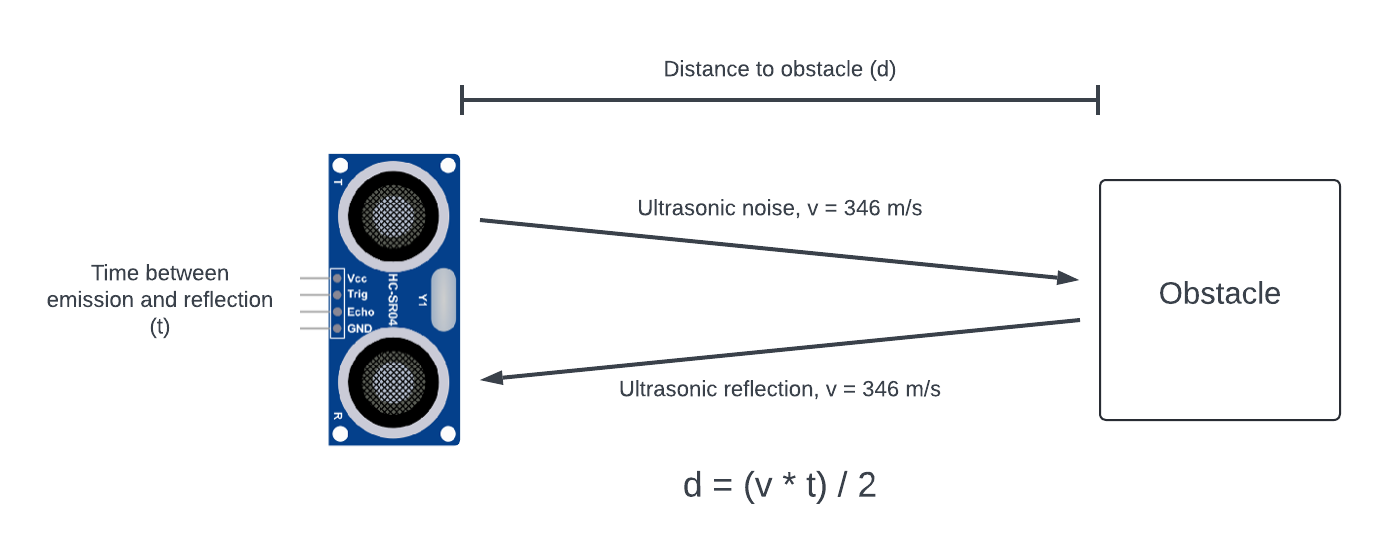
**Lesson 4 – Basic Sensing**

In this lesson we’ll discuss a few useful sensors that return simple digital or analog signals rather than sending data over a communication protocol (we’ll learn about those later). First, we’ll learn the basics of a few common sensors, then discuss how robots use sensor data to respond to their environment. Finally, we’ll install some sensors on our rover and use them to follow a line and avoid an obstacle.

**Time-Of-Flight Sensors**

Robots don’t operate in a limitless void, and so often it's useful to give your robots some way to understand the space around them. Advanced robots will often use radars, lidars, and other complex sensors which cost hundreds or thousands of dollars to scan their local area. These expensive sensors, though, share a lot in common with affordable time-of-flight sensors we can buy from hobby electronics suppliers.

“Time-of-flight" refers to the way these sensors detect distance. These sensors have an emitter which emits some sort of signal, be it radio waves, light, or ultrasonic noise, and a receiver which listens for the reflections of that signal bouncing off an object. A time-of-flight sensor emits a signal when a microcontroller tells it to, then tells the microcontroller when it hears the reflection. The microcontroller can count the time between the emission and reflection- the *time-of-flight*- and then calculate the distance to the object that caused the reflection based on that time and the speed of the signal.



**Figure 4-1:** Basic operation of an ultrasonic time-of-flight sensor

The HC-SR04 is a common time-of-flight sensor that uses pulses of ultrasonic noise to detect and range obstacles. Because it uses ultrasonic noise, the pulses it generates travel at the speed of sound (approximately 346 m/s through room temperature air), which we’ll use to calculate distances. We’ll revisit the HC-SR04 later in this lesson and install one on our rover.

**Infrared Light Sensors**

Sometimes it can be useful for your robots to have some awareness of color or reflectance. In a warehouse, for instance, painted lines could be used to guide robots or mark boundaries they shouldn’t cross. In the field, detecting a reflection could tell a robot if the ground in front of it is dropping away. For applications like these, infrared light (IR) sensors can be useful. IR sensors typically consist of an IR LED and an [IR sensitive phototransistor](https://www.sciencedirect.com/science/article/pii/B9780124201651000214). A phototransistor behaves a lot like a normal transistor, except rather than opening when voltage is applied to the base, it opens when it senses light. When the IR sensor is near a reflective surface, the phototransistor detects the reflected light and opens. Then we can use a circuit called a *comparator* to let our microcontroller know whenever the phototransistor detects light.

A [comparator](https://www.electronics-tutorials.ws/opamp/op-amp-comparator.html) is a circuit which compares a varying input voltage (Vin) to a steady reference voltage (V\_ref). It needs to be connected to a supply voltage (V\_cc), ground, and some input and reference voltages. When Vin is *lower* than V\_ref, the output of the comparator will be at ground. When the Vin is *higher* than V\_ref, the output voltage goes high to V\_cc. For instance, if the circuit is connected to a 5V power supply, and the input voltage is above the reference voltage, then the comparator will output 5V.

A diagram of a circuit

Description automatically generated

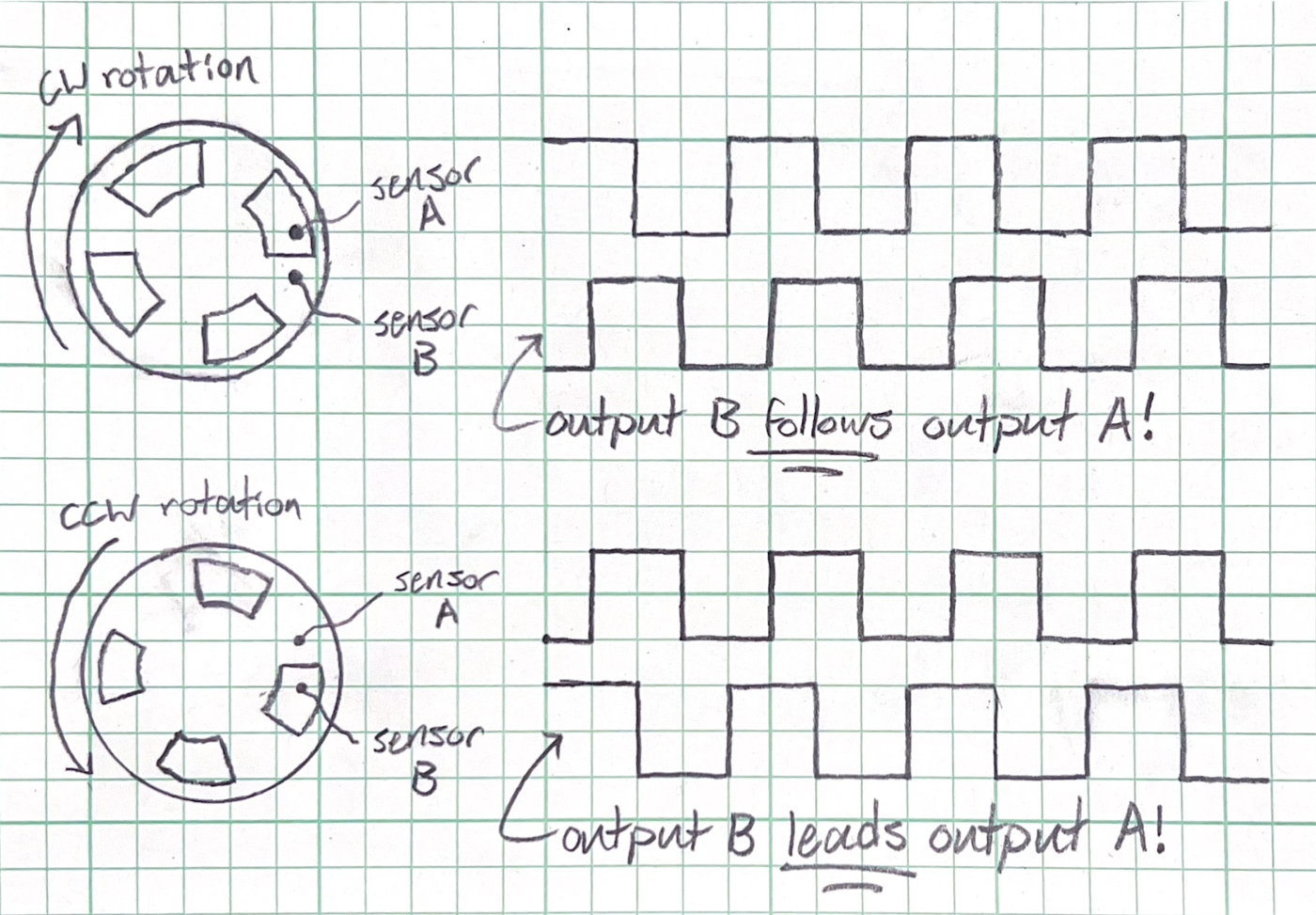
**Figure 4-2:** Behavior of the LM393 comparator, assuming [use of a pull-up resistor](https://www.youtube.com/watch?v=ib-JzYZzV0o)

The [LM393 is a comparator](https://www.ti.com/product/LM393?bm-verify=AAQAAAAJ_____yB_dQaNquRoBRmt9teVrTe6WzCz43MBT9mCkkbS3oVpF1PZlmyxRQt0IgtyfuUURUhaE37xRm5ud5xFdoCNrPtH0kvCMEOlfOzrxmeJ523iVkxl41QL-EG5oGtFyaBCjTW6LYC6rStQrWICUgrXzA4kk-ukWCgaN6Jq4L3tgs8vIEstaYQOMR32qiAASAJVzWNwySNhGp_Y1OJzEe1PAzjghOwS1p3aydtplQanbEN8IFrcCXSHqiZU9iGy0glpcQ8N7h7jICf4zlkJxDN1TuDrCVQodaI3BTwVQVMvSdpx_A) packaged up in a small [integrated circuit chip](https://www.ansys.com/blog/what-is-an-integrated-circuit#:~:text=An%20integrated%20circuit%20(IC)%20—,top%20of%20the%20silicon%20surface.). The IR sensors we’ll use on our robot include the LM393 comparator chip to determine when they detect a strong reflection. The output (emitter) of the phototransistor is connected to ground. This means when the phototransistor sees a strong reflection, it opens and the input of the LM393 is pulled down to ground. Because of that, the comparator outputs 0V when the sensor sees a strong reflection. On the other hand, when the phototransistor sees no reflection, it stays closed and the input of the LM393 is pulled up to V\_cc. The comparator will output V\_cc when the sensor sees no (or a weak) reflection. These signals can easily be detected with a digital pin on the Arduino, and we’ll use this sensor to detect stark changes in color later in this lesson.

**Rotary Encoders**

The human body is an incredible sensory system. We all learn about the five senses (sight, sound, smell, taste, and touch), but humans have another sense you may not have heard about: proprioception. Proprioception is the ability of your body to sense the movements it makes and where it is in space. Your body is full of nerves that provide detailed sensory feedback to your brain which makes complicated motions like running, climbing, or gymnastics possible. Robots are very similar to humans; they need proprioception too to perform complex tasks.

One type of sensor we can use to sense our robot’s joint positions is a [quadrature rotary encoder](https://howtomechatronics.com/tutorials/arduino/rotary-encoder-works-use-arduino/). Quadrature encoders use two sensors to detect increments on a rotor. When an increment passes the two sensors, it causes one sensor to go high before the other. Based on which sensor goes high first, we can determine the direction of rotation.



**Figure 4-3:** Based on which output leads, we can find the direction of rotation

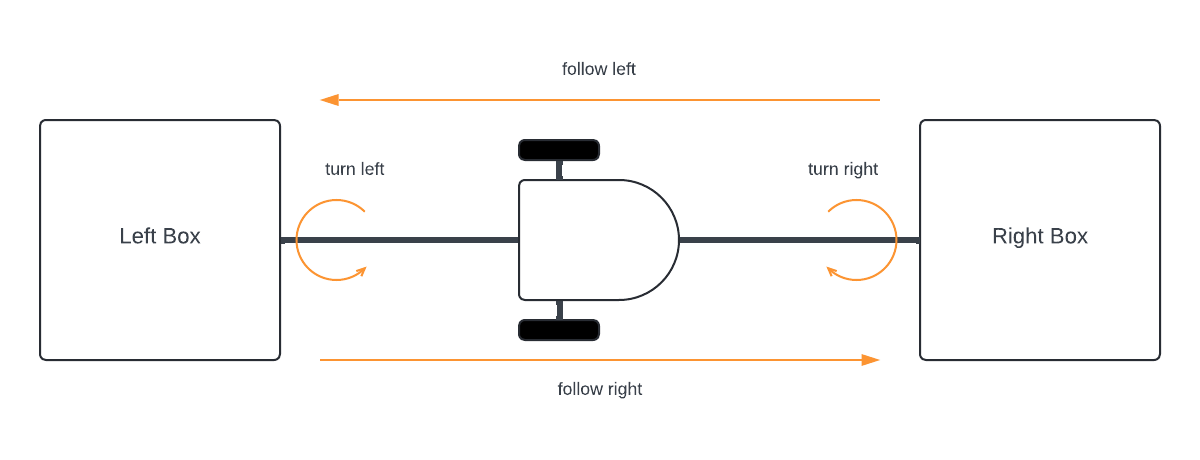
We can detect the pulses generated by each sensor in the encoder with the pins of a microcontroller and use the timing and frequency of these pulses to decode the position of the rotor. If the rotor is attached to a motor shaft ([like it is on our motors](https://www.pololu.com/product-info-merged/5225)), then we can measure the position of the motor, which we often refer to as its *state*.

Measuring the state of our robot’s joints allows us to employ some advanced robotics techniques. We can compare our measured wheel speed to a target value we want to hit and adjust our motor supply voltage accordingly, so we move at the exact speed we intend ([this is called feedback control](https://medium.com/@yakupcengiz/exploring-control-systems-part-2-feedback-control-systems-cb3e1a30c274)). We could also measure the position of our robot’s wheels and feed that info into some [kinematics equations](https://medium.com/@nahmed3536/wheel-odometry-model-for-differential-drive-robotics-91b85a012299), so we can estimate where our robot is in the world. This process is called [*odometry*](https://en.wikipedia.org/wiki/Odometry), and the use of odometry is fundamental to more complicated robot algorithms such as Simultaneous Localization and Mapping (SLAM).

**Event Driven Programming and the Finite State Machine**

Alright, time to figure out how we’ll use all this sensory information to program a robot that makes decisions and operates autonomously. The beauty of our sensors is that they can detect *events* our robot should respond to. That could be anything from driving within a certain distance to an obstacle, seeing a line on the floor, or driving a certain distance. But the instant the robot detects that something interesting to our application has occurred, it should perform a set of actions (called a *service*) to respond to that event. This structure is called [*event driven programming*](https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_eventdriven_programming.htm).

Event driven programming works well for robotics particularly when combined with another concept called the [*finite state machine*](https://en.wikipedia.org/wiki/Finite-state_machine) (FSM). A finite state machine consists of a set of states and a set of specific transitions between those states. An event-driven FSM represents the combination of these two concepts, where events trigger the transition between states, and the states themselves are services- the set of actions a robot takes to respond to events. To practice creating an FSM, let’s consider a robot that follows a line between two boxes, and when it gets close to each box it turns around and goes the other way:



**Figure 4-4:** A robot following a line between two boxes

There are four different states our robot will be in as it drives its circuit between the boxes: follow right, turn right, follow left, and turn left. The event that triggers the turns will be coming within 10cm of the boxes, while the event that triggers the line following will be detecting the line. For the sake of learning, we’ll throw in another wrinkle. We want our robot to stop, wait, and turn around each time it has travelled 10m+. This adds yet another state and four transitions, which gives us this [*state transition diagram*](https://www.geeksforgeeks.org/state-transition-diagram-for-an-atm-system/) (STD):

A diagram of a network

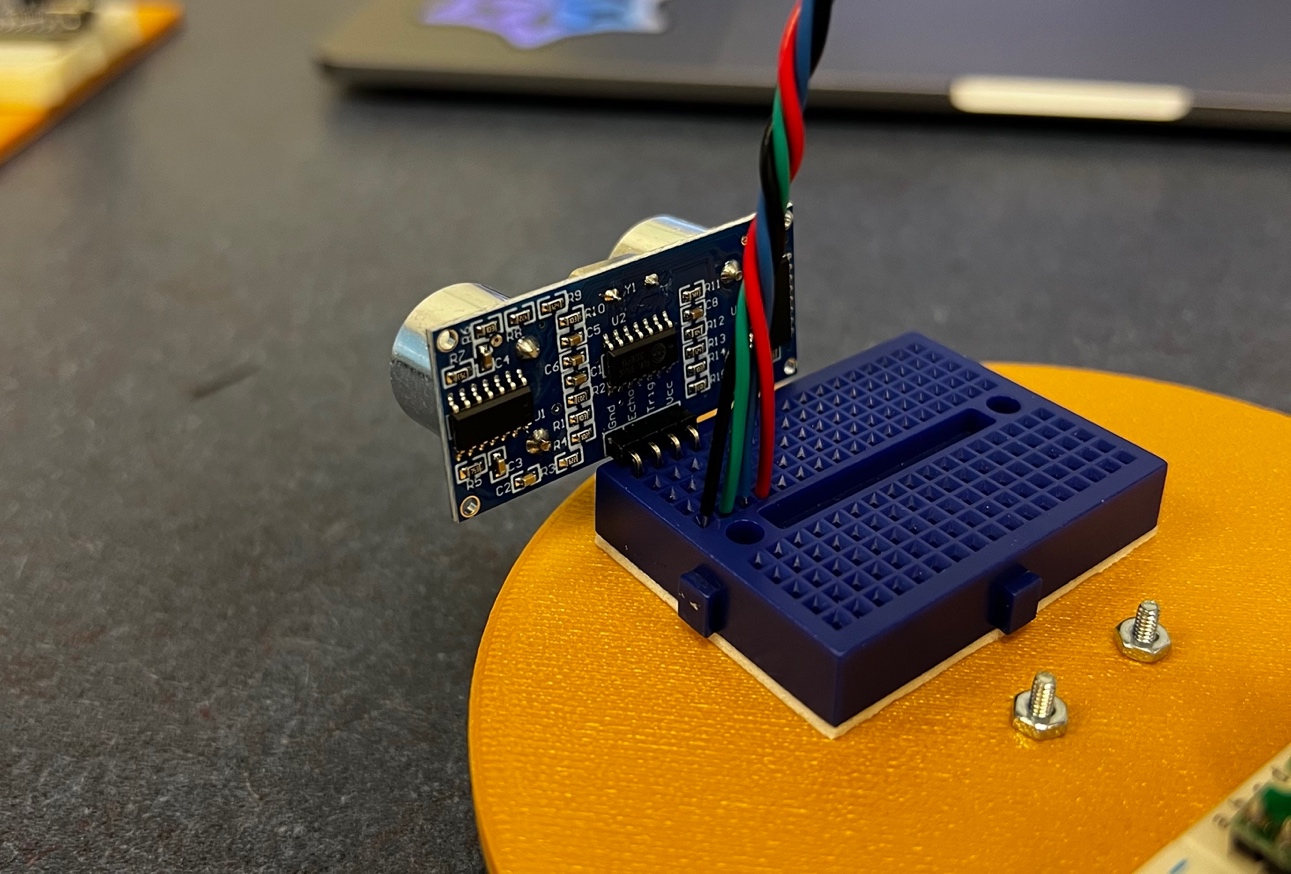
Description automatically generated

**Figure 4-5:** The FSM that follows a line between two boxes

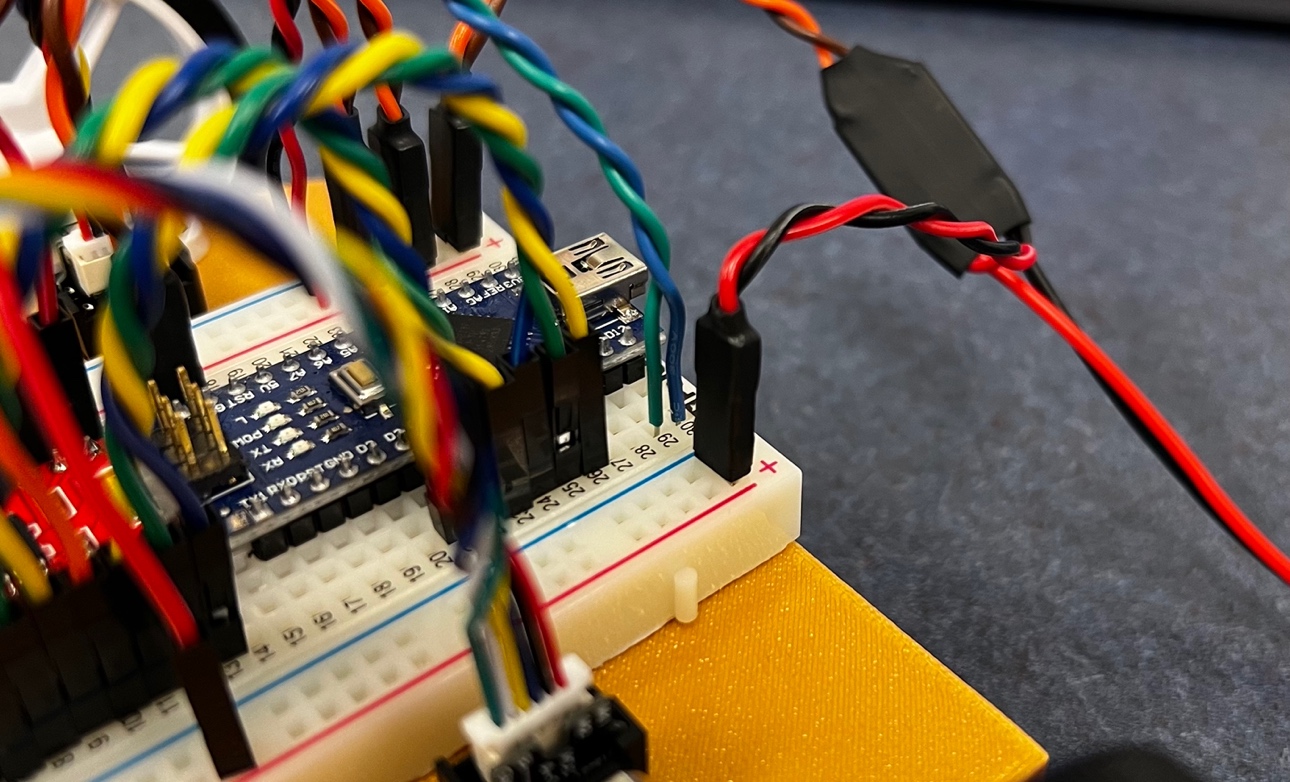
In the Arduino workspace you created for this workshop, open “lesson\_4\_basic\_sensing.ino” in the Arduino IDE. You’ll notice this code is almost identical to the code we wrote in lesson 3. Before we organized our problem with layers of abstraction and used Arduino functions to write modular code we can easily build on top of. Now we cash in on that hard work and build on top of that code to create a robot that senses its environment and makes decisions.

**Integrating the Ultrasonic Time-Of-Flight Sensor**

The time-of-flight sensor we’ll use on our rover is called the HC-SR04. Plug your HC-SR04 sensor into the top left corner of the mini breadboard, with the eye-like emitter and receiver facing forward. The HC-SR04 has just four pins: one for 5V power, one for ground, and two pins called trig and echo which we use to trigger a pulse of noise and then listen for reflections. We’ll use two jumper cables to plug the Vcc and Gnd pins into the 5V bus. We’ll use a third jumper cable to plug Echo into A29 on the breadboard and a fourth jumper cable to plug Trig into A30 on the breadboard.



**Figure 4-6:** The HC-SR04 and jumper cables plugged into the mini breadboard



**Figure 4-7:** Jumpers from Echo and Trig plugged into A29 and A30 on the breadboard

We need to send a signal to the sensor on one pin (trig) and receive a signal on another pin (echo). Like always, that means we need to declare and set up these pins. Here is the declaration, which goes at the very top to make these global variables:

// declare echo and trig pins for ultrasonic sensor

const int echo = 11; // Echo -> D11

const int trig = 12; // Trig -> D12

And here is the pin setup, which we put inside “void setup()”:

//--------------------setup sensor pins--------------------

pinMode(echo, INPUT);

pinMode(trig, OUTPUT);

Now we need to write some code that measures the time required for a pulse sent on the trig pin to bounce off an obstacle and return to the echo pin. We’ll write this code inside a function called get\_distance() to keep our main loop tidy. The first step is to generate a pulse on the trig pin. We can use the digitalWrite() and delayMicroseconds() functions to raise the trig pin to 5V for 10ms:

// send out an ultrasonic pulse thats 10ms long

digitalWrite(trig, HIGH);

delayMicroseconds(10);

digitalWrite(trig, LOW);

Now we need to determine the time it takes for that pulse to reflect off an object and bounce back to the sensor. Luckily, there’s a native Arduino function called [pulseIn()](https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/) we can use to do this:

// use pulseIn function to see how long it takes for the pulse to return to the sensor

echo\_time = pulseIn(echo, HIGH);

Finally, now that we have the time it takes the signal to travel to and back from the obstacle, we can use the formula from figure 4-1 to calculate the distance to the obstacle.

// calculate distance using formula from ToF sensor section

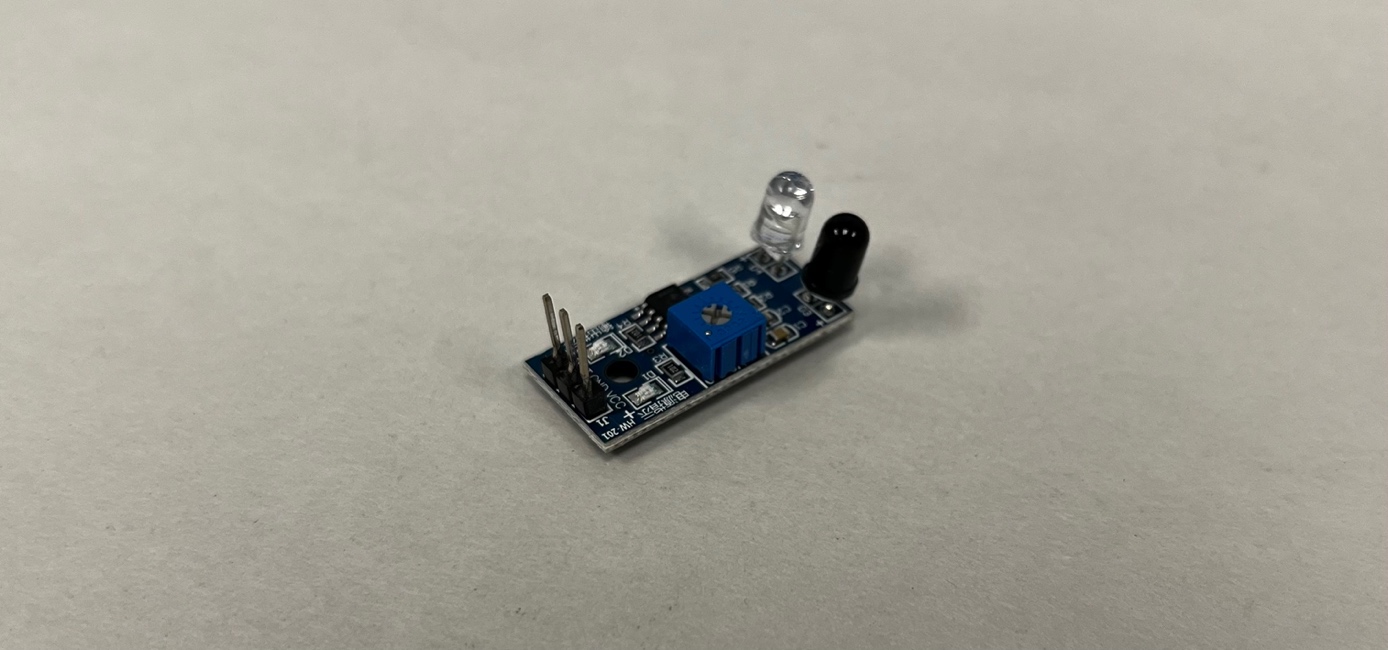
return calculated\_distance = (echo\_time / 2) / (346 \* 10);

//\*10 is to scale units appropriately since pulseIn returns in microseconds

The get\_distance() function should now return the distance to an obstacle in meters when called. Uncomment “[Serial.println](https://www.arduino.cc/reference/en/language/functions/communication/serial/println/)(get\_distance());” and open the serial monitor. Upload the code and experiment with placing objects in front of your robot to see how the ultrasonic sensor behaves.

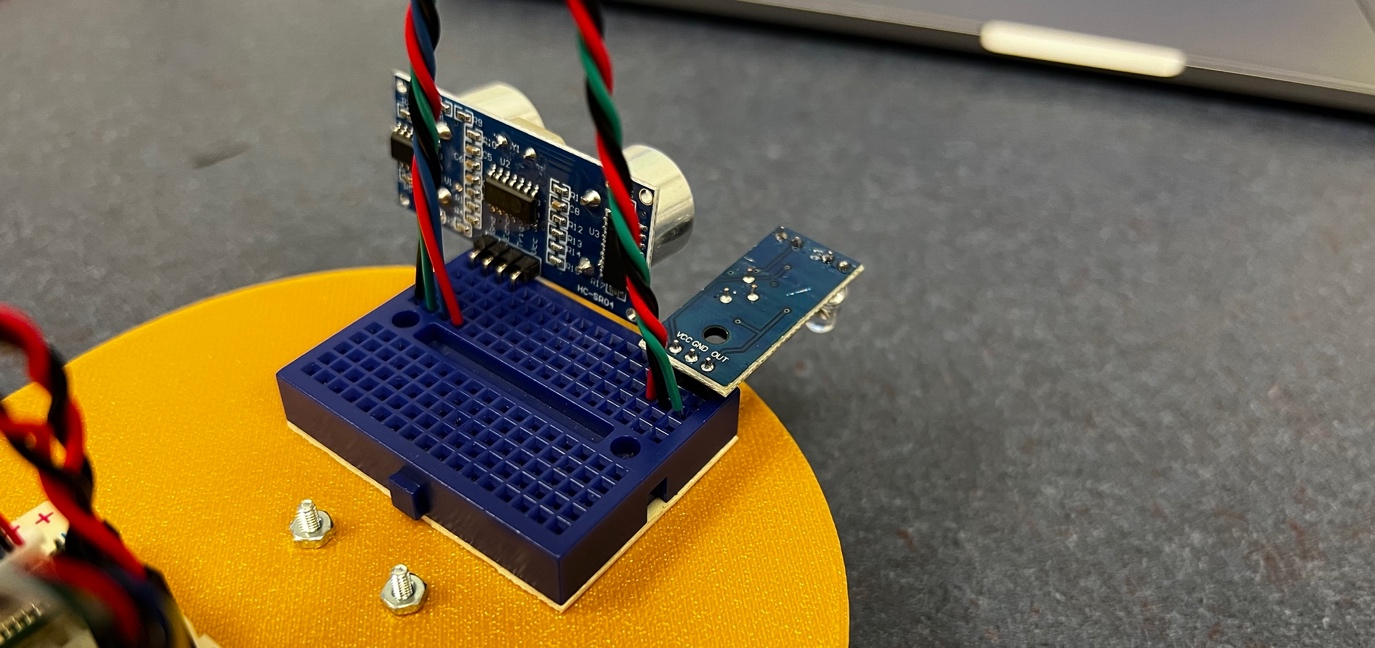
**Integrating the Infrared Light Sensor**

The infrared light sensor is even simpler than the ultrasonic time-of-flight sensor to use. First, use your fingers to bend the IR LED and Phototransistor to be perpendicular to the sensor board. Then use pliers and do the same with the header pins. Then insert the sensor, facing down, into the top right corner of the mini breadboard.



**Figure 4-8:** The IR sensor with pins, LED, and phototransistor bent perpendicular to the board

The infrared sensor has 3 pins: VCC, GND, and OUT. Just like before we’ll plug VCC and GND into the 5V bus on the breadboard. Then we’ll plug the OUT into J30 on the breadboard.



**Figure 4-9:** The IR sensor and jumpers plugged into the mini breadboard

Of course we’re going to declare the output pin at the top:

// declare pin for infrared sensor

const int ir = 13; // OUT -> D13

And set the pin mode in void setup():

pinMode(ir, INPUT);

And now we’ll go to our function, get\_line(). You’ll notice this function returns a new type of variable– [a Boolean](https://www.arduino.cc/reference/en/language/variables/data-types/bool/). A Boolean can be one of two values: it can be either true or false. In our case we want the function to tell us whether the sensor sees a line, so it’ll return true if it sees the line and false if it doesn’t.

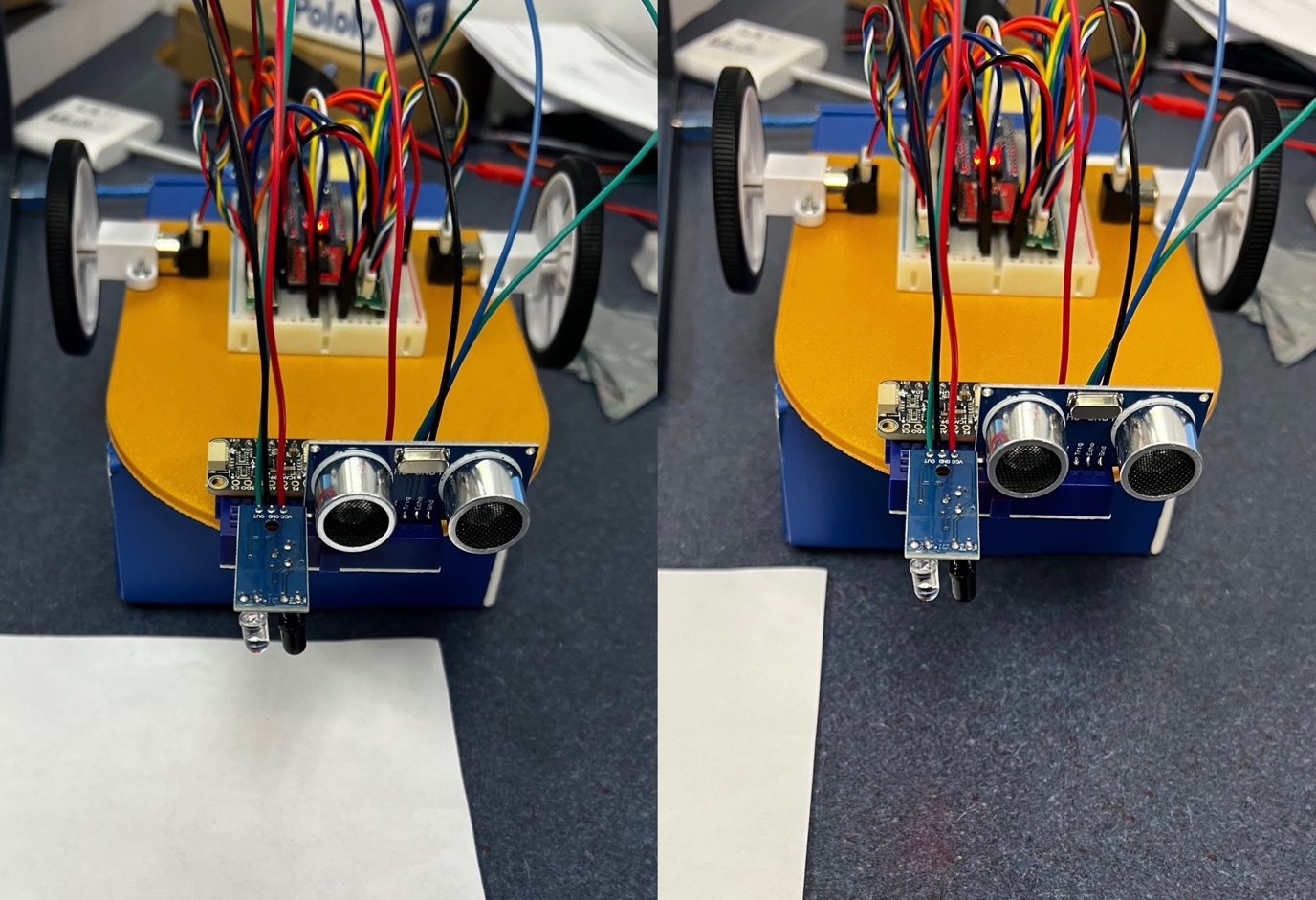
We can use digitalRead to check the value of the IR sensor output. If the IR sensor sees a white line, it’ll set the output pin to 0V and digitalRead will return a value of LOW. Otherwise, the IR sensor output pin will be at 5V and digitalRead will return HIGH. Conveniently, the value of HIGH and LOW [can be substituted for TRUE and FALSE (due to how these variables are represented in binary code)](https://forum.arduino.cc/t/interchanging-high-low-with-true-false/146611). All we need in this function, then, is one line of code:

return !digitalRead(ir);

The get\_line() function should return TRUE (1) when it sees a white line and FALSE (0) when it doesn’t. Uncomment “Serial.println(get\_line());” and play with the sensor to see what it can and can’t detect. There is a potentiometer on the sensor you can turn with a screwdriver, try adjusting this and see what happens. Adjust this potentiometer to tune your sensor until it reliably detects a line.

It’s important to note the IR sensor won’t work properly unless the 12V LiPo battery is plugged in. It's okay to have the battery plugged in while the Arduino is connected to your computer as long as you double check your electrical (especially your ground) connections.

One issue you may encounter is your sensor returning FALSE when it should return TRUE and vice versa. This could happen depending on the design of your IR sensor. The exclamation point in the above line of code is the logical “not” operator; it essentially gives you the opposite of whatever you put in front of it. If your sensor returns the opposite values you expect, try deleting the exclamation point in the above line of code.



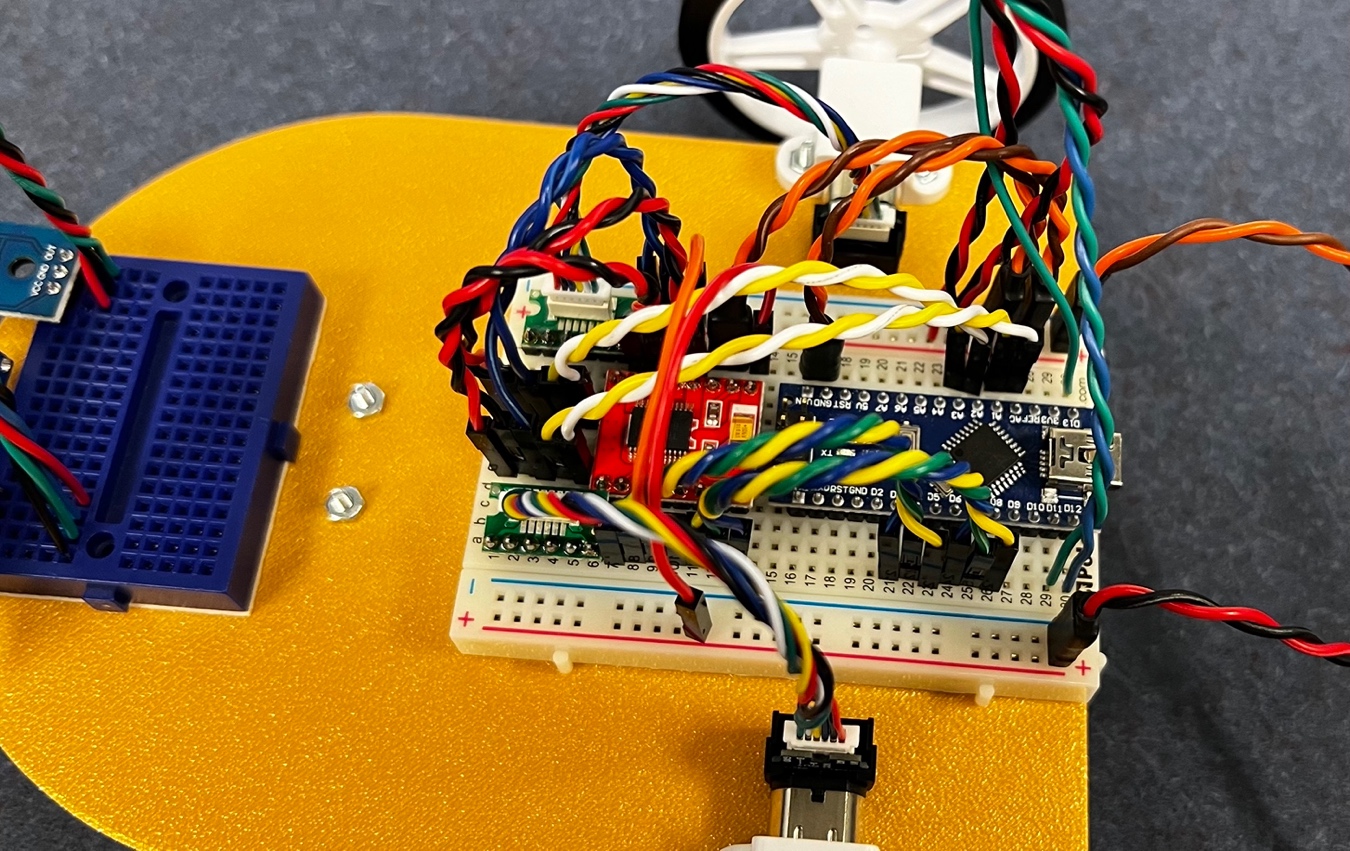
**Figure 4-10:** Using a sheet of paper to test and tune the IR sensor

**Integrating the Motor Encoders**

Integrating motor-driven quadrature encoders can be challenging because the speed at which the encoder pins rise and fall may be faster than the speed at which a microcontroller loops through a program, which can cause the microcontroller to miss counts. There are [hardware-based solutions to this problem](https://forum.digikey.com/t/design-a-robust-quadrature-encoder-isr-based-program-for-arduino-motor-control/39913), and it can be an educational problem to solve on your own. In this lesson, though, we’ll use libraries to handle the challenging tasks of recording encoder counts. To do this, open the library manager in the Arduino IDE and install the QuadratureEncoder and enableInterrupt libraries.

Once you’ve completed that, we’ll connect the quadrature encoders. The first thing we need to do is feed the sensor power. Do this by plugging two jumpers into the 5V rail and then plugging one into E3 and the other into F3. We don’t need to use separate jumpers for ground in this case because the ground pins (on E6 and J6) are connected to ground pins on the motor driver, so they’re already linked to the common ground. Next let's connect the left encoder to the Arduino. Grab 1 yellow jumper and 1 white jumper and twist them together. Plug one end of the yellow jumper into E4 and the other end into J26, then one end of the white jumper into E5 and the other end into J27. To connect the right encoder, get another pair of yellow and white jumpers and twist them together. This time, plug one end of the yellow jumper into F4 and the other end into J24. Then plug one end of the white jumper into F5 and the other end into J25.

These jumpers are admittedly a pain to insert, you may find it easier to remove certain nearby jumpers and then reinsert them after you’ve inserted these sets of yellow and white jumpers.



**Figure 4-11:** The encoder 5V wires (twisted blue jumpers) and signal wires (twisted yellow/white jumpers) plugged in

Once we’ve got the encoder plugged in, we’ll go back to our code. We need to include the QuadratureEncoder library near the top of our script:

// include the QuadratureEncoder library

#include <QuadratureEncoder.h>

We’ll declare our encoder pins for ease of use and [use the encoders class provided by QuadratureEncoder to create some encoder objects](https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP) for each of our motor encoders:

// declare a and b pins for encoders (can be analog pins)

// right encoder

const int a\_r = A3; // A -> A3

const int b\_r = A2; // B -> A2

// left encoder

const int a\_l = A1; // A -> A1

const int b\_l = A0; // B -> A0

// lets also create our encoder objects

Encoders right\_encoder(a\_r, b\_r);

Encoders left\_encoder(a\_l, b\_l);

Now let's go work on our get\_odom() function. The first thing we need to do is get our encoder counts, using the getEncoderCount method that the Encoders class (from the *QuadratureEncoders* library) provides. Some of these words may be unfamiliar depending on your programming background, but they all come from a concept called [Object Oriented Programming (OOP)](https://www.geeksforgeeks.org/introduction-of-object-oriented-programming/). OOP is a large topic, but here is an (unfortunately quite reductive) summary to make some of this language less confusing:

A *class* is user-defined data type. We’ve used data types such as ints, floats, bools, and more. A specific instance of a data type is called a variable, and it has some name and value that we give it. A specific instance of a class is called an *object*, and it has a name and values (yes, often more than one value) that we assign it. Then there are *methods*, which we use a lot like the functions we’ve been writing, except methods are provided by a class. When we call a method, we first refer to an object we’ve created, and then to the method we want to call. That method then performs some action or calculation, often using values assigned to that object and *arguments* (inputs) we provide to it.

When we get our encoder counts from our left\_encoder and right\_encoder objects, we’ll first refer to that object and then call a method that returns those encoder counts.

// get encoder counts using getEncoderCount method from the Encoders class

long left\_encoder\_count = left\_encoder.getEncoderCount();

long right\_encoder\_count = right\_encoder.getEncoderCount();

Now that we’ve got our encoder counts, we’ve got to relate those to angular wheel positions in radians. For that, we need to determine the *counts-per-revolution* (CPR) of our encoders. This is simple; our [motor datasheet](https://www.pololu.com/product-info-merged/5225) tells us the encoders count 12 CPR at the gearbox. To relate that to our actual wheel revolutions, we need to multiply that gearbox CPR by the gear ratio (297.92:1, again from the motor datasheet), which gets us 3575.04 CPR at the wheel. Now we want to relate encoder counts to the angular position of the wheels in radians. With some [dimensional analysis](https://www.chem.tamu.edu/class/fyp/mathrev/mr-da.html), we find that for each 2\*pi radians travelled, we’d register 3575.04 encoder counts. Now we can put this into code to determine the angular position of our wheels in radians based on our encoder counts:

// find the angular position of our wheels

float left\_wheel\_pos = left\_encoder\_count \* ((2 \* 3.14) / 3575.04);

float right\_wheel\_pos = right\_encoder\_count \* ((2 \* 3.14) / 3575.04);

Finally, we need a function to find the distance travelled by our robot from the angular positions of our wheels. Back in lesson one we found the inverse and forward kinematic equations of our robot. We used the inverse kinematics to determine wheel speeds from desired velocities. Now we’ll use forward kinematics to determine our robot’s linear position from its wheel speeds. This equation relates the wheel speeds of our differential drive bot to its linear speed:

Position (p) is simply the integral of velocity, so taking the integral of the above function gets us:

All we need to do is insert the function for position into our code and return the value it spits out.

// calculate the linear position of our robot from the angular position of the wheels

float odom = (r / 2) \* (left\_wheel\_pos + right\_wheel\_pos);

return odom;

You know the drill by now. There are a few lines of code in the main loop that test get\_odom(). Uncomment them, watch the serial monitor, and see what happens. Take the time to be sure this function operates correctly. If get\_odom() returns a negative distance or the distance it returns is much smaller than the real distance the rover drove, then the signal (yellow/white) wires for one or both encoders may be reversed. Try switching the yellow/white wires and see how that changes things.

Once you’ve verified each of our sensors are integrated properly, it's time to move on to the Finite State Machine.

**Programming a Finite State Machine**

There are a few different ways to program an FSM. Given that we’ve seen if statements before, that might seem like the obvious way to build our state machine– use many nested if statements to check the state. This syntax, however, can become cumbersome as state machines get more complicated. Another common way to program FSMs, which often makes for neater code, is to use [the switch:case](https://www.arduino.cc/reference/en/language/structure/control-structure/switchcase/). We’ll use this expression to build our FSM (here [are](https://forum.arduino.cc/t/yet-another-finite-state-machine-introduction/1198997)–[some](https://www.digikey.com/en/maker/tutorials/2023/how-to-program-an-arduino-finite-state-machine)–[references](https://forum.arduino.cc/t/state-machines-a-short-tutorial/580593)).

First, we need a way of expressing our states in code. Using the [enumeration (enum) expression](https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-enum#:~:text=Enum%2C%20which%20is%20also%20known,from%20the%20set%20of%20values.) to do this helps to make very nice, readable code. Enum is a quick way to create your own datatype which can have a finite set of custom values that you specify. The compiler assigns numerical values to these custom values, but as we read and write code, we see our assigned values which makes the code easier to understand. We’ll create a datatype with enum to represent our states, and then declare a few variables of that type to use in our FSM:

//--------------------set up FSM--------------------

enum STATE {follow\_right, turn\_right, follow\_left, turn\_left, stop};

STATE last\_state;

STATE current\_state = follow\_right; // give it a value so we can enter the switch:case on first loop

STATE next\_state = current\_state; // give this a value so our switch:case behaves

Now we’ll use the switch:case in the main loop to create the structure of our FSM. Each time we pass through the main loop, “switch (current\_state)” will be evaluated once, and it will send our program to the case matching whatever the current state is. That means we need one case for each of the states in our FSM:

switch (current\_state) {

case follow\_right :

break;

case turn\_right :

break;

case follow\_left :

break;

case turn\_left :

break;

case stop :

break;

}

Now, we need to do two things in each of our cases: check for events that would cause a transition to a new state and perform the set of actions corresponding to our current state. Let's consider first our follow\_right case. There are two events that would transition us to a new state: our odometer hits 5m, or we come within 10cm of a box. We’ll use if statements to see if one of these conditions has been met, and if so, move us to the appropriate next state (according to the STD in figure 4-4):

case follow\_right :

// check for events

if (get\_odom() > 5) { // our odometer hits 5m

next\_state = stop;

break;

}

if (get\_distance() < 0.1) { // we come w/in 10cm (0.1m) of a box

next\_state = turn\_right;

break;

}

If no events have occurred, then we’ll perform the actions associated with the follow\_right state, which means we need to follow a white line to the left of the robot. The way we do this is by moving forward slowly and turning right if we see the line and left if we don’t see the line. We can perform these actions with some if statements:

case follow\_right :

// check for events

if (get\_odom() > 5) { // our odometer hits 5m

next\_state = stop;

break;

}

if (get\_distance() < 0.1) { // we come w/in 10cm (0.1m) of a box

next\_state = turn\_right;

break;

}

// perform set of actions

if (get\_line()) { // if robot sees the line

motor\_controller(0.1, -1); // slowly forward, turning right

}

if (!get\_line()) { // if robot DOES NOT see the line

motor\_controller(0.1, 1); // slowly forward, turning left

}

break;

Now let's consider the turn\_right case. Like before, we first check for events. As our robot comes up to the box it enters the turn\_right state, where it turns away from the line. One way to know we’ve turned 180 degrees and are now going the other direction is we can see the line again. So, the event that moves us on to follow\_left is seeing the line. If we don’t see the line, we want to keep turning in place. What if, though, when we get close to the box our IR sensor just happens to be above the line? We’d see the line and immediately enter the follow\_left state without turning around. We need some sort of [*debouncer*](https://www.circuitbasics.com/how-to-use-switch-debouncing-on-the-arduino/) to prevent this from happening. A simple way to do this is to make sure that the first time we loop through turn\_right, we wait long enough to clear the line before checking if we see the line. We can do this with an if and delay statement, and by putting our actions before our event checker in this state.

case turn\_right :

// perform set of actions

motor\_controller(0, -3);

if (last\_state != turn\_right) { // debouncer to make sure we actually turn around

delay(500); // will need to play with this value for effective debouncing

}

// check for events

if (get\_line()) {

next\_state = follow\_left;

break;

}

break;

The last case we’ll look at is the stop state. The event that takes us out of this state is a timer going off, and the path we take out of the state depends on the path we took into the state. The easiest way to accomplish all this is to simply perform the actions associated with the state, set a 3 second delay, and then set next state based on what our last state was. No need to loop through this state.

case stop :

// perform set of actions

motor\_controller(0, 0);

right\_encoder.setEncoderCount(0);

left\_encoder.setEncoderCount(0);

delay(3000);

// set next\_state

if (last\_state == follow\_right) {

next\_state = turn\_right;

}

if (last\_state == follow\_left) {

next\_state = turn\_left;

}

break;

Finally, after we’ve gone through our switch statement, we need to update our STATE variables. Last\_state needs to be set to our current\_state, and current\_state needs to be set to our next\_state. Then we’re ready for the next loop.

// update states

last\_state = current\_state;

current\_state = next\_state;

Now try programming the follow\_left and turn\_left cases yourself. You should be able to recycle code from the follow\_right and turn\_right cases, but think carefully about the variable names and directions involved. If your code doesn’t follow the line properly, those would be good things to check.

Once you’ve got the rover following a line in between two boxes, make sure to commit and push your new code to GitHub the same way we did in the previous lesson. In the next lesson, we’ll learn about a fundamental tool that gives us access to more advanced sensors and robotic capabilities.