**Lesson 5 – Advanced Sensing**

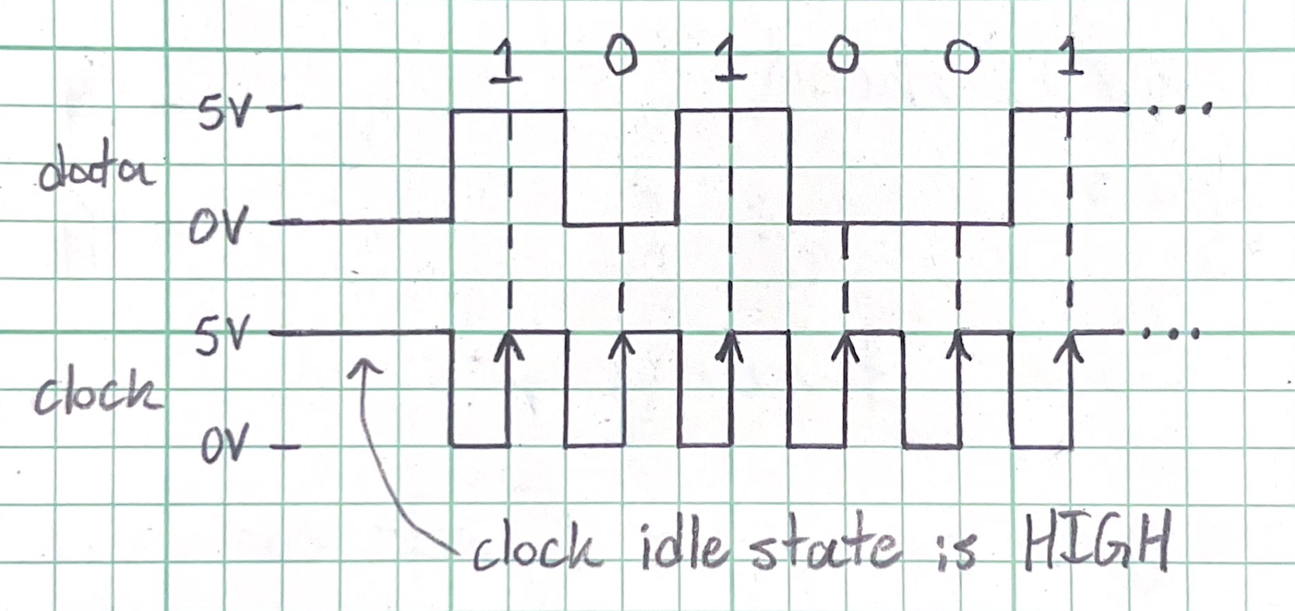
In this lesson we’ll learn how to use sensors that share large amounts of complex information using communication protocols. First, we’ll learn what a communication protocol is, then how to use a few of the most common protocols with Arduino. Finally, we’ll use a communication protocol to add an inertial measurement unit to our rover and build a new FSM to navigate around an obstacle.

**What is a Communication Protocol?**

Up to now we’ve thought primarily about a robot with one onboard processor (our Arduino) acting as the robot’s brain, managing a lot of other simpler components. Sometimes, though, this isn’t an accurate model of how robots work. Many complicated robots are networks of different processors each managing their own complicated tasks (e.g. one processor dedicated to motion planning and decision-making, one dedicated to sensing the environment, and another dedicated to controlling actuators). In this case, these processors need ways of sharing information so they can all work together in harmony. The techniques used to share information between processors are called communication protocols.

Fundamentally, digital communication between processors is all about connecting the output of one processor to the input of another processor, and then agreeing on the meanings of specific [logic levels](https://learn.sparkfun.com/tutorials/logic-levels) (HIGH vs LOW pin voltage) and timings. Digital data is represented with a [base-2 number system](https://learn.sparkfun.com/tutorials/binary), where each digit- or “bit”- can have a value of either 0 or 1. It’s easy enough to assign a value of 0 to a LOW pin voltage (0V) and a value of 1 to a HIGH pin voltage (5V, in the case of the Arduino). By setting our pin voltage LOW we can send a 0 and by setting it HIGH we can send a 1. The thing is [integers other than 0 and 1](https://www.purplemath.com/modules/numbbase.htm), [floating point numbers](https://towardsdatascience.com/binary-representation-of-the-floating-point-numbers-77d7364723f1), and [alphanumeric characters](https://www.geeksforgeeks.org/ascii-table/) are all represented with groups of bits, so we need to send more than one bit in a message. If we can agree on a timing convention to communicate when one bit ends and another begins, though, we can send one bit after another in a process called *serial* (or *bit-serial*)communication. Using serial communication, we’re able to send the groups of bits that build larger messages.

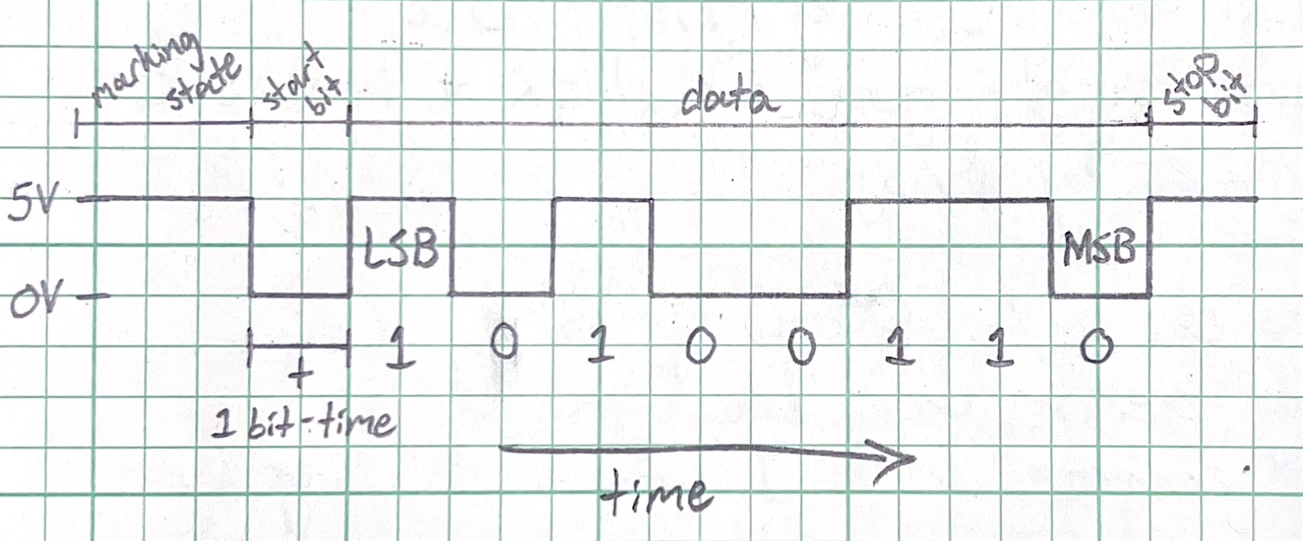
One such method of coordinating the timing involved in serial communication is known as *synchronous* serial communication. In this method of serial communication, the two processors are connected by a common ground (like always), a data wire for serial data, and a clock wire the sender uses to indicate when a bit of data on the data line is valid. Assuming our clock is *rising-edge active* (indicated by the arrow on the rising edge), each time we raise the logic level on our clock to HIGH, the receiver knows to read a bit from the data line.



**Figure 5-1:** Inter-processor communication using a clock wire to synchronize the message

Another method of coordinating serial communication timing is called *asynchronous serial communication*. Asynchronous serial communication does away with the clock wire in favor of stricter structuring of each message and each processor maintaining its own internal clock to determine when the data is valid. This determination of validity is based on the concept of *bit-time*, which is the duration of time a single bit of data is present on the data line. The communicating processors must agree on the bit-time to send and receive a clear message.

For asynchronous serial communication, the data line is set to HIGH in its idle state (referred to as its *marking state*). The beginning of a message is indicated when the data line drops from the marking state at HIGH to the *spacing state* at LOW, which marks the beginning of the *start bit*. The start bit lasts 1 bit-time, after which the sender transmits all the bits in the message, each lasting 1 bit-time. Typically, these messages will be 8 bits long, though seven is also common. The first bit transferred is the [*least significant bit*](https://en.wikipedia.org/wiki/Bit_numbering) (LSB), while the final bit is the [*most significant bit*](https://en.wikipedia.org/wiki/Bit_numbering)(MSB). After the MSB, the *stop bit* is sent to signal the end of the message. After the stop bit has been sent, the data line is again free to send another message.

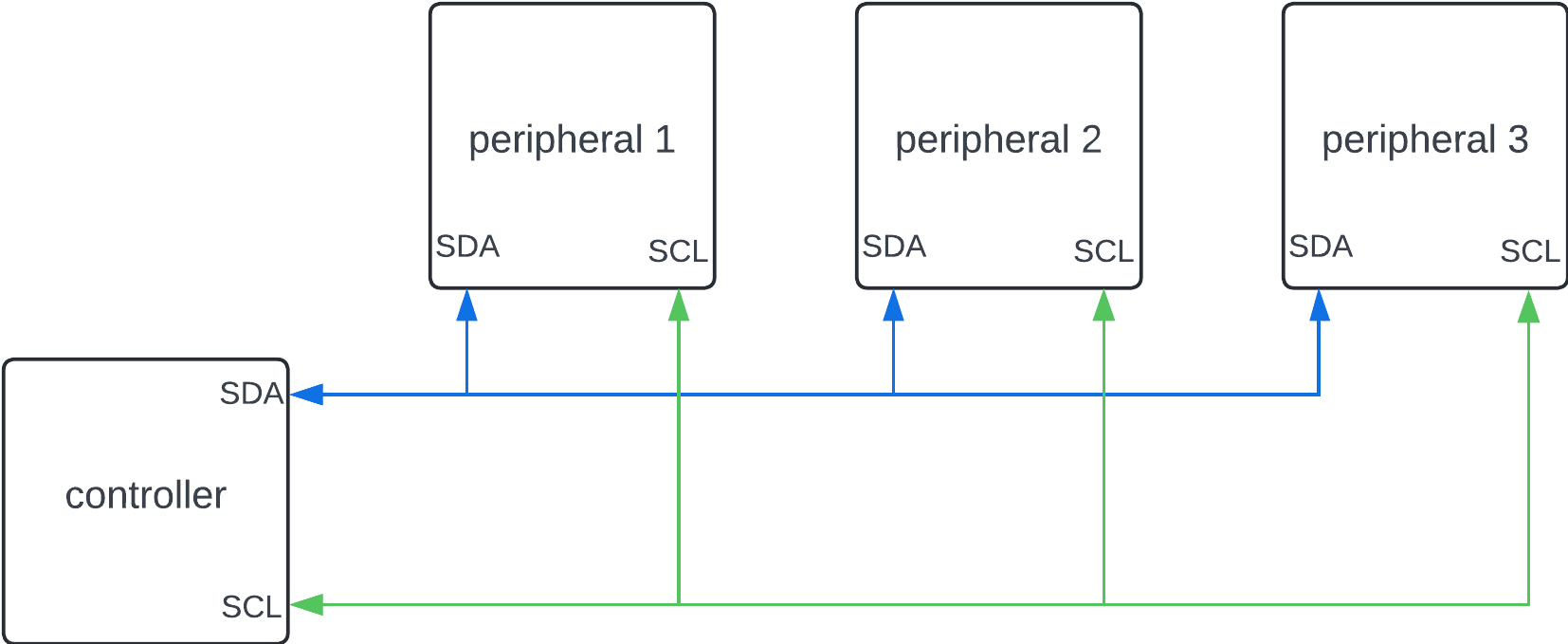


**Figure 5-2:** Inter-processor communication without a clock wire

These are simply the fundamentals of how serial communication protocols work. The communication protocols we’ll discuss in the remainder of this lesson build on these foundations to make data exchange fast, efficient, and reliable, and suit the needs of different users. We’ll discuss these differences and learn the basic tools to use each of these protocols in an Arduino-based project.

**The Inter-Integrated Circuit Bus**

The [Inter-Integrated Circuit Bus (I2C)](https://learn.sparkfun.com/tutorials/i2c/all) is a synchronous serial communication protocol that has become popular because it is fast, can link more devices than asynchronous protocols, and requires fewer wires than other synchronous protocols. To connect devices over I2C, the SDA pin on each device must be connected to an SDA bus (“bus” is a term for a common wire) and the SCL pin on each device must be connected to an SCL bus. It’s important to remember all these devices [need to be connected to a common ground](https://forum.arduino.cc/t/common-ground-and-why-you-need-one/626215), but we’ll omit that from our diagrams for simplicity.



**Figure 5-3:** The basic I2C circuit schematic

Any I2C message is broken up into frames. The first is a 7-bit address frame, which the controller uses to name which peripheral device it intends to communicate with. After the address frame comes one or more 8-bit data frames, depending on the message length. The controller may send these data frames to the peripheral or request them from the peripheral. The controller, though, decides when communication happens and which devices it communicates with.

Arduino microcontrollers offer some [convenient tools for working with I2C communication](https://docs.arduino.cc/learn/communication/wire/#introduction). For one thing, the above diagram isn’t quite an accurate representation of an I2C bus. The SDA and SCL buses each idle at a HIGH logic level. This is accomplished using [pull-up resistors](https://learn.sparkfun.com/tutorials/pull-up-resistors/all), which are sometimes included in diagrams of an I2C network. Arduino devices, though, have built in pull-up resistors, so we don’t have to think much about them when assembling our I2C circuit.

Another great tool Arduino offers for I2C communication is the [Wire library](https://www.arduino.cc/reference/en/language/functions/communication/wire/). The Wire library provides a variety of different functions to make communication between devices over I2C simple for Arduino projects. This article offers a bunch of [examples that illustrate how to use these functions](https://docs.arduino.cc/learn/communication/wire/#examples). Later in this lesson, though, we’ll use a sensor with its own custom I2C library. Custom libraries are common among these breakout sensors that are largely designed for hobby use. These custom libraries are generally built on top of the Wire library and add just a few functions to make interfacing with the sensor easier. We’ll explore this more after we discuss two other serial communication protocols.

**The Serial Peripheral Interface**

The [Serial Peripheral Interface (SPI)](https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all) is another synchronous serial communication protocol common in the hobby electronics world. SPI requires more hardware than I2C to work properly but offers even higher communication speed than I2C and much higher speed than asynchronous methods can offer. But before we get started discussing the SPI circuit, we should have a quick note on communication protocol terminology.

The use of controller/peripheral to describe the devices linked by serial communications protocols is gaining popularity because the old terms are offensive. We’ll use these new terms, but because this is a recent shift you’ll probably see the old terms as you browse documentation and explore other resources. To avoid confusion, you may need to be familiar with these old terms:

|  |  |
| --- | --- |
| **Obsolete Terms (Master/Slave)** | **New Terms (Controller/Peripheral)** |
| Master in, slave out (MISO) | Controller in, peripheral out (CIPO or POCI) |
| Master out, slave in (MOSI) | Controller out, peripheral in (COPI or PICO) |
| Slave select (SS) | Chip Select (CS) |

An SPI circuit is a bit more complicated than an I2C circuit. Just like I2C (or any synchronous serial protocol) there will be a clock line (called SCK). Unlike I2C, though, communication from the peripheral to the controller (CIPO) is performed on a different bus than controller to peripheral (COPI). Finally, the chip select (CS) pins are used to identify which peripheral the controller wants to talk to in a multi-peripheral system. An SPI peripheral knows to communicate when it’s CS pin is pulled low. When its CS pin is high, it ignores the controller. This makes addressing peripherals simple, but it means the controller must have as many CS pins as peripherals it communicates with.



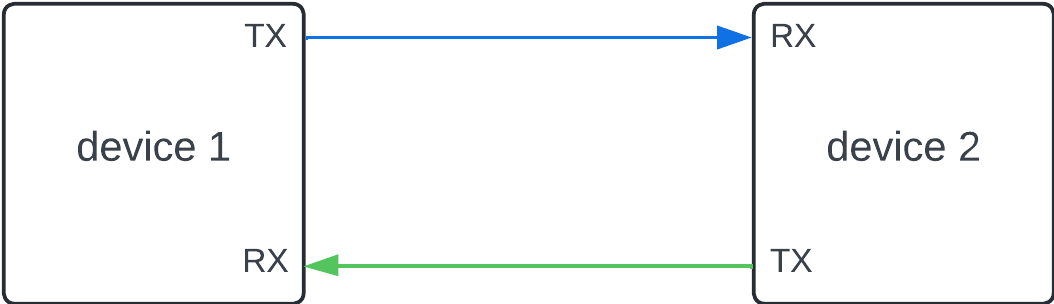
**Figure 5-4:** The basic SPI circuit schematic

The actual message that SPI sends can be a lot simpler than I2C, but that's because the hardware required to send that message is more complex to make up that functionality.

Just like with I2C, it’s possible to [create your own serial communication program](https://docs.arduino.cc/tutorials/generic/introduction-to-the-serial-peripheral-interface/), but Arduino provides the [SPI library](https://www.arduino.cc/reference/en/language/functions/communication/spi/) for SPI communication. This library includes [functions to make SPI communication simple](https://www.circuitbasics.com/how-to-set-up-spi-communication-for-arduino/) for the Arduino programmer.

**The Universal Asynchronous Receiver-Transmitter**

Finally, we get to a serial protocol that's a bit different than anything else we’ve discussed. The [Universal Asynchronous Receiver-Transmitter (UART)](https://docs.arduino.cc/learn/communication/uart/#rxtx-pin-examples) protocol is an asynchronous serial protocol, unlike the previous two synchronous protocols we’ve discussed. Remember, that means that UART forgoes a shared clock wire to keep the communicating devices in sync with each other, and instead relies on strict standardization and timing of the messages it sends. This makes UART communication slower than the previous two communication protocols we’ve looked at, but it does simplify serial communication from a hardware perspective:



**Figure 5-5:** The basic UART circuit schematic

That’s all the wiring UART serial communication requires– just two wires (and a common ground between the devices, AS ALWAYS). Unfortunately, UART is best for connecting only two devices, so this easy protocol isn’t great for building the larger networks of devices we can create with I2C or SPI. It is still a useful protocol and when using the [Serial class](https://www.arduino.cc/reference/en/language/functions/communication/serial/), you may recognize some of the commands we use as the same commands that interact with the Arduino IDE serial monitor. This is because the Arduino IDE uses UART to upload compiled code to an Arduino device and receive data from the device to print to the serial monitor.

As you may have already figured out, the [Serial class](https://www.arduino.cc/reference/en/language/functions/communication/serial/) provides built in UART functionality for Arduino projects. Because it’s a class we use it a bit differently than the functions provided by the previous libraries we’ve looked at. The Serial class comes from a pre-installed [Arduino library called HardwareSerial](https://github.com/arduino/ArduinoCore-avr/blob/master/cores/arduino/HardwareSerial.h), which [automatically creates the objects Serial and Serial1](https://stackoverflow.com/questions/22104059/where-is-arduino-serial-created) from the serial class. You can use the Serial object to talk to another UART device over the USB port (usually your development computer) and the Serial1 object to talk to another UART device you’ve plugged into the TX and RX pins on your Arduino.

Because timing is so critical in asynchronous communication, each time you use UART communication, you need to tell your device how fast to expect communication to take place. The speed of UART communication is described by a *baud rate*, which we’ll specify in the setup() function with the command

Serial.begin(9600);

Remember Serial refers to the object that communicates over the USB port. If you want to talk to another device over the TX/RX pins, you’ll need to call the begin() method for the Serial1 object. The argument (input) to the begin() method is going to be the baud rate; 9600 and 115200 are common baud rates for communication with the Arduino IDE.

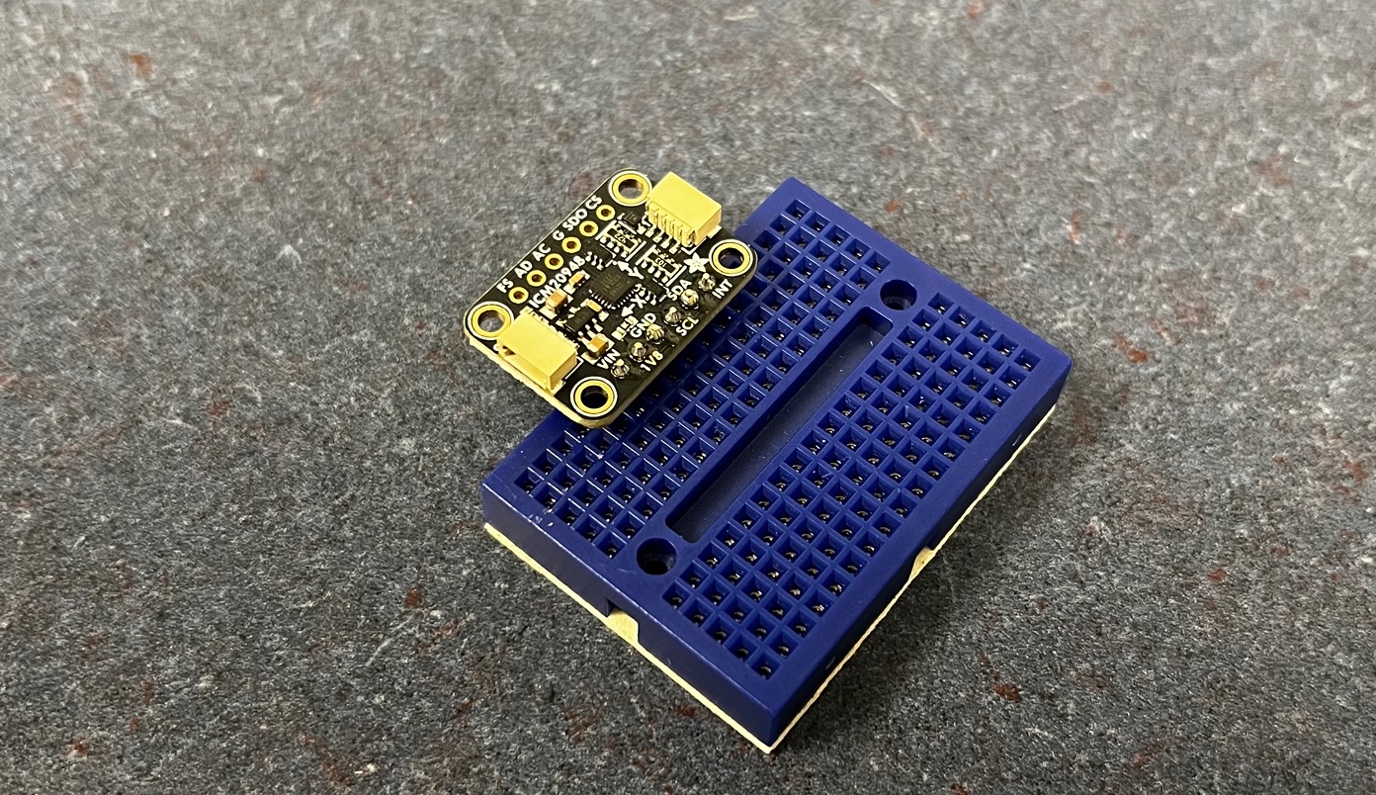
As long as we get that set up properly, we’re free to [use other methods provided by the Serial class](https://docs.arduino.cc/learn/communication/uart/#transmit--receive-messages) such as read(), write(), and println() to communicate with other devices over UART.

**I2C Communication with an Inertial Measurement Unit**

Communication protocols give us the ability to network processors which results in robots that run fast and are easy to modify and repair. We’ll practice using a communication protocol– I2C in this case– by connecting an inertial measurement unit (IMU) to our Arduino and using it to help navigate around an obstacle.

An IMU is a device that detects orientation and movement using a suite of accelerometers, gyroscopes, and sometimes magnetometers. Each of these sensors have different strengths and weaknesses, but by using [*sensor fusion*](https://en.wikipedia.org/wiki/Sensor_fusion#:~:text=Sensor%20fusion%20is%20the%20process,these%20sources%20were%20used%20individually.) techniques to combine the data from each sensor, it’s possible to track the IMU’s motion with more accuracy and precision than you could with any individual sensor. That, though, is beyond the scope of this workshop; we’ll use individual sensors on the IMU to try and give our rover the abilities of a compass and a level.

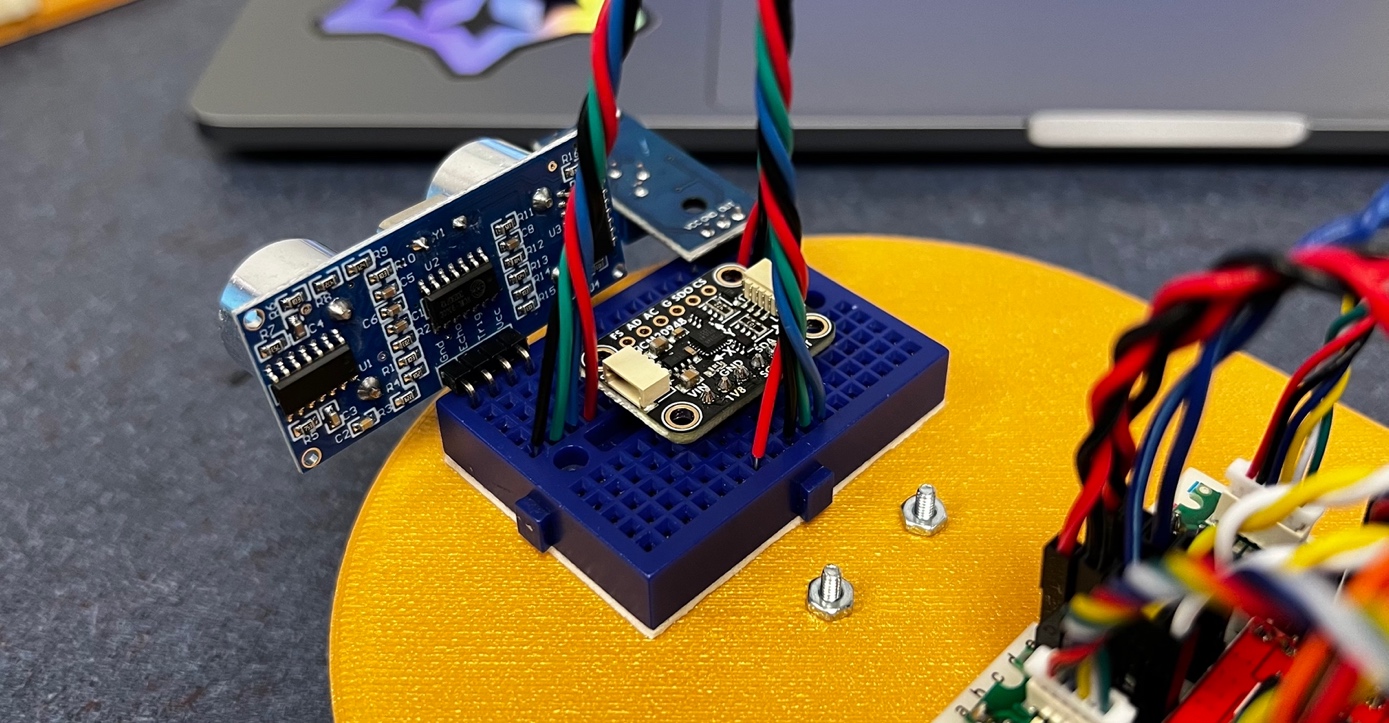
Before installing the IMU, you’ll need to solder on some header pins so we can press it into a breadboard. By now, that process should be familiar. In this case we don’t need (nor want) to attach *all* the header pins. Our IMU comes with the option to use different protocols to talk to it, but we plan to use I2C so we only need to feed it 5V power and connect the SCL and SDA pins. Snap off six header pins, and solder them *only* to the side of the IMU board with VIN, 1V8, GND, SCL, SDA, and INT pins.



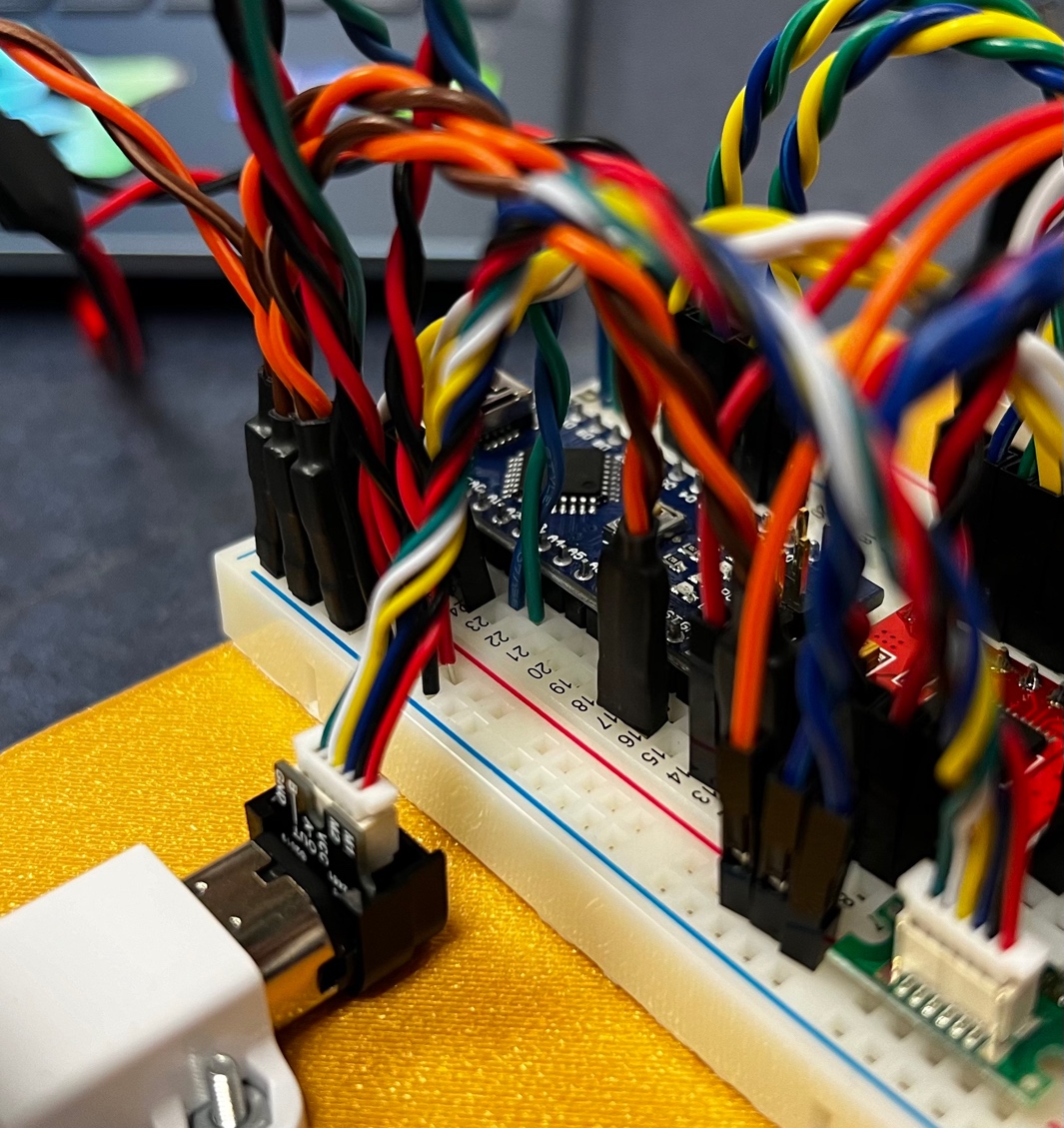
**Figure 5-6:** The IMU with 6 header pins soldered onto one side only

Once you’ve soldered on the IMU pins, place your IMU, with those pins towards the back of the rover, into the mini breadboard. Make sure it is towards the back of the mini breadboard, but with enough space that you can insert jumper wires behind it.

To connect the IMU you’ll need 4 jumper wires. Plug two in right behind the VIN and GND pins on the IMU and plug those into the 5V bus on the large breadboard. Take the other two and plug them right behind the SCL and SDA pins. Take the wire behind the SCL pin and plug it into I22 on the large breadboard. Then take the wire behind the SDA pin and plug it into I23 on the large breadboard. These correspond to the SDA and SCL pins on the Arduino Nano; we’ve just made the I2C circuit shown in figure 5-3.



**Figure 5-7:** The IMU and its jumper wires placed on the mini breadboard



**Figure 5-8:** The SDA and SCL jumper wires nestled in I22 and I23 against the Arduino

Now that the IMU is installed, we need to write some code to talk to it. The first step in this process is to get all the libraries required to work with the IMU. This IMU is called [the ICM20948](https://learn.adafruit.com/adafruit-tdk-invensense-icm-20948-9-dof-imu/arduino), and Adafruit produces some libraries made specifically for that sensor which make it easy to use. Open the “lesson\_5\_advanced\_sensing.ino” code file. In the library manager of the Arduino IDE, look up and download the “Adafruit ICM20X” library. This is an example of a custom I2C device library built on top of the Wire library which [makes using an I2C device even simpler](https://github.com/adafruit/Adafruit_ICM20X/blob/master/examples/adafruit_icm20948_test/adafruit_icm20948_test.ino) than it usually would be, and we’ll use it to easily access data from our IMU chip. If the IDE asks to also install the “Adafruit BusIO” and “Adafruit Unified Sensor” libraries, let it. Otherwise, look up and install these libraries yourself. Then, include the proper libraries at the top of this code file to get started. This is also a good place to create the sensor object which provides us with the methods we need to interact with the sensor.

// include communications libraries

#include <Adafruit\_ICM20X.h>

#include <Adafruit\_ICM20948.h>

#include <Adafruit\_Sensor.h>

#include <Wire.h> // A4 is SDA, A5 is SCL

// create icm object from ICM20948 class

Adafruit\_ICM20948 icm;

Now on to the setup() function. We’ll give ourselves some troubleshooting tools in this code with some print statements so we can track the startup process of our IMU in the serial monitor. We’ll need to begin UART serial communication with our Arduino to allow this, and that means setting a baud rate.

//--------------------set up ICM20948 IMU--------------------

Serial.begin(115200);

while (!Serial) {

delay(10);

}

The while loop isn’t strictly necessary, but it gives our serial bus time to start up before we do anything else. Also notice we set the serial baud rate to 115200. When we open the serial monitor to view data, we need to specify the baud rate in a drop-down menu on the right side of the monitor, otherwise everything we see will be gibberish.

Next, we’ll try to start our I2C communication with the IMU chip. All we need to do is call icm.begin\_I2C to attempt to initialize our device. It isn’t strictly necessary, but we’ll also add some simple error handling logic that lets us know if initialization is unsuccessful and enters an infinite loop so the program doesn’t progress any further.

Serial.println("Adafruit ICM20948 test");

// try to initialize

if (!icm.begin\_I2C()) {

Serial.println("failed to find ICM20948 chip");

while (1) { // enter infinite loop if chip is not found

delay(10);

}

}

Serial.println("ICM20948 found");

After making the necessary additions to the setup() function to use our IMU, we’ll create a new function that gets the magnetometer data from the sensor and converts that data into an orientation in degrees. We’ll call this function get\_angle\_mag(), and it’ll return that orientation as a float datatype. The first step is to call the method that gets us new data from the IMU.

// get a new normalized sensor event

sensors\_event\_t accel;

sensors\_event\_t gyro;

sensors\_event\_t mag;

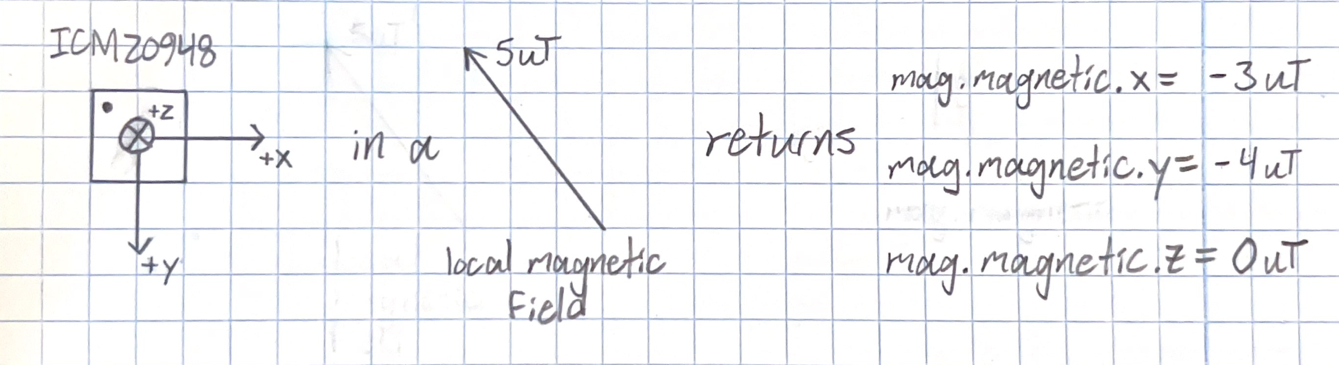
sensors\_event\_t temp;

icm.getEvent(&accel, &gyro, &temp, &mag);

Now that we have this data, we need to figure out a math equation we can plug our magnetometer data into which will spit out our robot’s angle. To do this, let's consider what our magnetometer data means.

Figure 13 in the [ICM20948 datasheet](https://invensense.tdk.com/wp-content/uploads/2024/03/DS-000189-ICM-20948-v1.6.pdf) shows us the axes of the magnetometer compared to the housing of the chip. When looking from above such that the dot on the ICM20948 chip is in the top left corner, the x-axis points to the right, the y-axis points straight down, and the z-axis points [away from you (into the page)](https://physics.stackexchange.com/questions/302386/confusion-regarding-plane-of-the-paper) following the [right-hand-rule](https://en.wikipedia.org/wiki/Right-hand_rule).

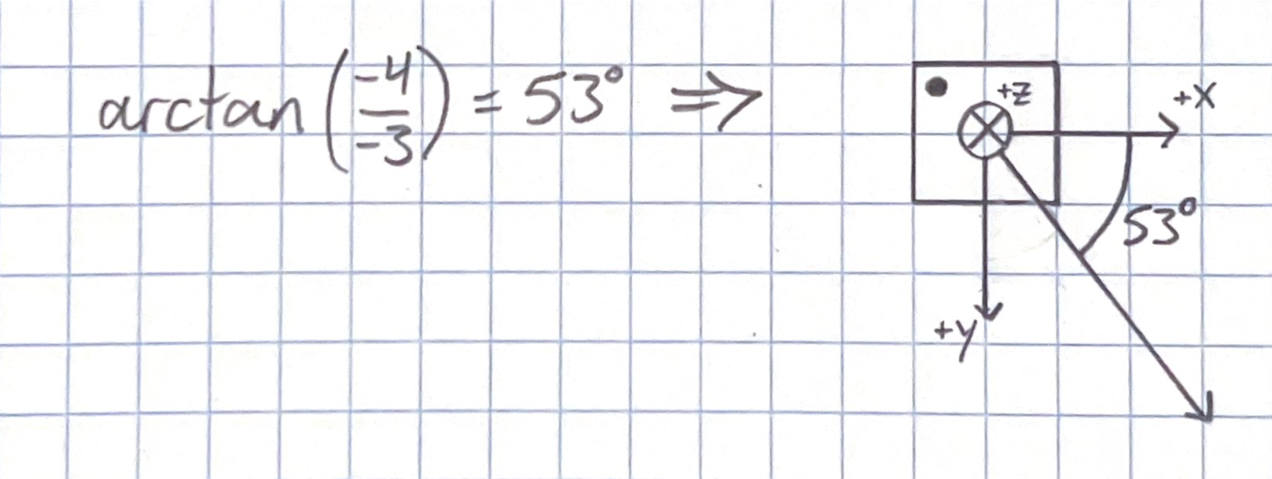
What the magnetic field sensor does is measure magnetic field strength in [micro Teslas](https://en.wikipedia.org/wiki/Tesla_(unit)) along the x, y, and z axes of the magnetometer. This gives you the [vector components](https://www.varsitytutors.com/hotmath/hotmath_help/topics/components-of-a-vector) of the local magnetic field.



**Figure 5-9:** Example of how the magnetometer would describe a certain magnetic field

The vector components of the magnetic field are useful to us because we can use the [trigonometric function arctan(y/x)](https://en.wikipedia.org/wiki/Inverse_trigonometric_functions) to find the angle of the magnetic field vector. If we can determine the direction of the magnetic field of the earth relative to our robot, we can compare the orientation of our robot to something that doesn’t move ([quickly, at least](https://en.wikipedia.org/wiki/Magnetic_declination)).

So, lets plug y = -4 and x = -3 into arctan and see what happens:



**Figure 5-10:** Arctan() alone is not sufficient for finding the angle of a vector

That’s strange, when we use arctan to find the direction of the magnetic field vector, it gives us a value that's completely opposite that 5uT local magnetic field we measured above (remember the direction of the vector is measured from the x-axis and in the direction of RRH). How can we explain this?

For one thing, when we have two negative vector components, the negatives cancel, and it appears to the arctan function just the same as having two positive vector components. A similar thing happens when you have only one negative vector component. In that case, either vector component could be negative, but arctan will not discriminate between the different possibilities. Essentially, arctan can’t tell the difference between [graph quadrants](https://www.cuemath.com/geometry/quadrant/) 1 and 4 and 2 and 3, so it assumes your vector is in quadrants 1 and 4. What we need is a version of arctan that *can* discriminate between these various cases.

Luckily, Arduino provides the [atan2() function](https://en.wikipedia.org/wiki/Atan2) for this exact situation. This function calculates the output of arctan, and then considers the signs of the vector components to decide which quadrant the vector is actually in. We’ll use atan2() for our get\_angle() function. A few final details are atan2() outputs an angle in radians, so we’ll use a conversion factor Arduino provides called RAD\_TO\_DEG to get angle in degrees. We’ll also input the negative of the y component to atan2(), which essentially rotates the axes of our magnetometer so the y-axis points towards the front of the rover.

// determine angular position

float angle = atan2(-mag.magnetic.y, mag.magnetic.x) \* RAD\_TO\_DEG;

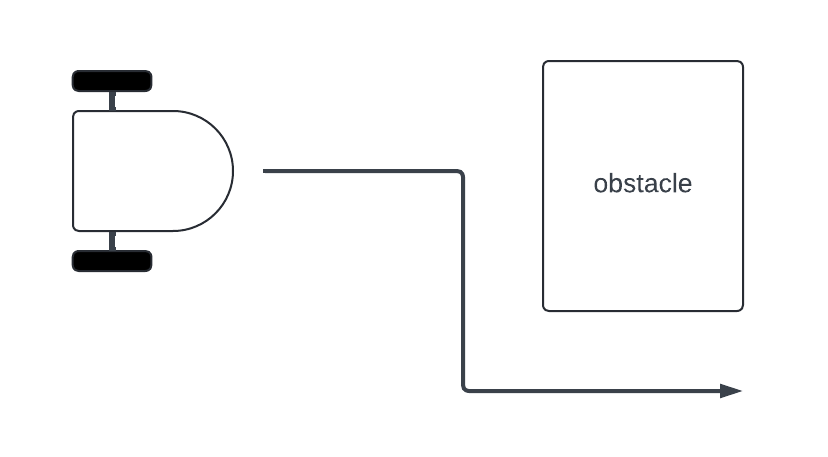
return angle;

Complete get\_angle\_mag() and test it out. How accurate do the values you’re getting seem? What might cause the behavior of these values? Are there any ways we could improve the performance of this sensor?

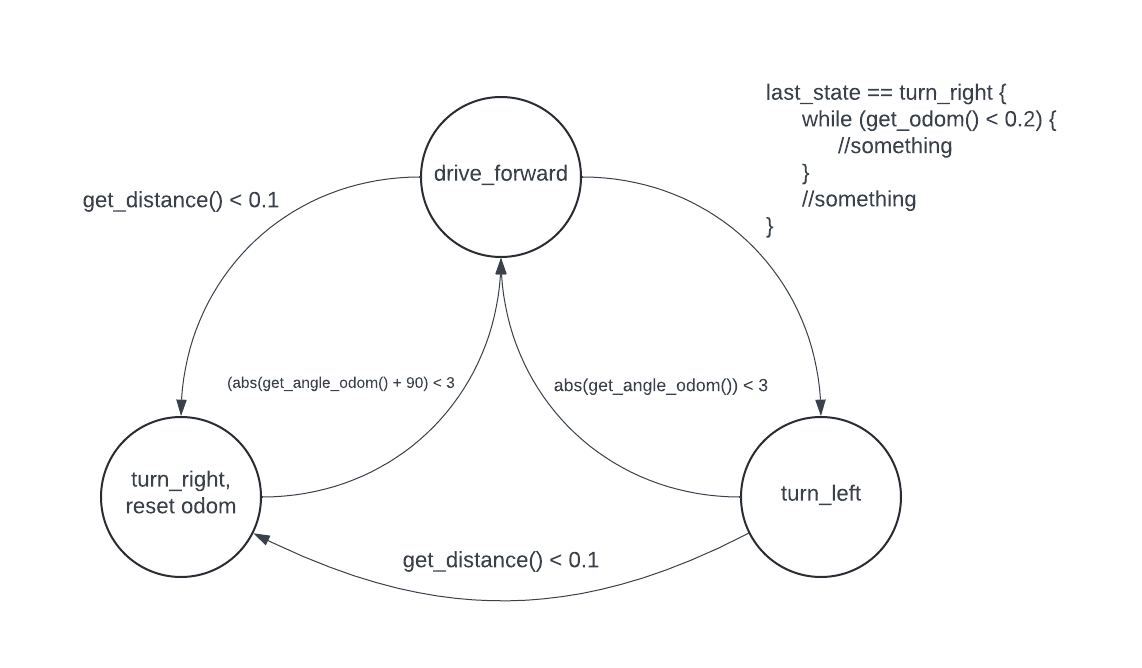
We can use similar code to find the [robot's pitch](https://simple.wikipedia.org/wiki/Pitch,_yaw,_and_roll) by using the accelerometer to detect gravity. Give it a shot yourself by filling in the get\_pitch() function! Heres a hint: accel.acceleration.x will return the acceleration measured by the accelerometer in its x-axis.

**Navigating Around an Obstacle**

Before we wrap up, let's use our now improved sensing capability to create a rover which can navigate around obstacles and keep moving in a set direction. A simple way to do this is to have our rover, when it gets close to an obstacle, turn right, drive a certain distance, and then turn back to its set direction to see if the obstacle is still there. If it is, the rover will turn right and drive again; if not then the rover will continue forward until it sees another obstacle.

**Figure 5-11:** The goal for our rover

We can use a finite state machine to solve this problem, just like in the last lesson. The following state transition diagram shows an FSM structure that can perform this task:



**Figure 5-12:** A state-transition diagram that can achieve our rover’s goal

This time, it’s up to you to program the event checkers and actions that will drive this FSM. Reference this diagram and the FSM we built in the previous lesson for help. The switch case structure we’ll need is shown below:

void loop() {

// put your main code here, to run repeatedly:

// Put your FSM in here:

switch (current\_state) {

case drive\_forward :

// check for events

// perform actions

break;

case turn\_right :

// check for events

// perform actions

break;

case turn\_left :

// check for events

// perform actions

break;

}

last\_state = current\_state;

current\_state = next\_state;

}

Use the above switch case structure and fill in the event checkers and actions with your own code to finish this FSM. Depending on your comfort level with the programming we’ve done so far, this could be a very challenging task. If you feel stuck, check the state-transition diagram and the code we used in the last lesson, or collaborate with your peers.

To make this task a bit more approachable, the code already includes a function called get\_angle\_odom(). Get\_angle\_odom() operates almost identically to the get\_odom() function we wrote in previous lessons, but it uses forward kinematics to find the *angular* position of the robot rather than the *linear* position. It's easier to use this function rather than get\_angle\_mag() because of the unreliability of the magnetometer you likely observed earlier.

Once you’ve gotten your rover to navigate around an obstacle, make sure to add and commit your changes to your repository and push those changes to GitHub. Good work, this has been a tough challenge! This wraps up the hands-on work for this workshop. The final lesson will be an overview of more advanced topics you can explore if you feel ready to tackle harder problems.