Adding New Sensors

Contents

Introduction

This document will walk the reader through the process of adding new sensors to the OpenExo system via an example using gyroscopic data from Arduino Nanos. These Nanos will send their gyroscope data via I2C to the Teensy 4.1 located on the exoskeleton's PCB.

After reading this document the reader will gain familiarity with how to wire a new sensor to the exoskeleton, establishing communication between the sensor and the Teensy, and how to handle communication when the sensor needs to send data that is specific to its side of the exoskeleton.

Justification for the Sensors Being Added

The sensors being integrated here are two Arduino Nano 33 BLE Sense Rev 2s. These Nanos will be mounted on the carbon fiber uprights on either side of the exoskeleton and will send angular velocity data using the Nanos' onboard gyroscope. The Nanos will send that data via I2C to the Teensy 4.1 on the PCB of the exoskeleton. To make use of this data, code will be added to the Controller.cpp and Controller.h files in the software. The code in these two files will allow the exoskeleton to access this data and use it to calculate motor commands. However, using the data to calculate motor commands will be omitted here to keep the guide simple and relevant to the task of connecting new sensors. How the data factors into the motor commands is up to the reader to determine, as it will vary based on sensor type and desired torque profile.

I2C Considerations

The communication protocol used here is I2C, facilitated by the Arduino Wire library. A complication that arises when using the Wire library is that it expects the data being sent and received to be formatted as bytes. Thus, we'll need to convert the float values that we get from the gyroscopes to a vector of bytes, send that vector of bytes via the Wire library, and then convert the vector of bytes back to float values. The details of this process will be covered below.

Wiring

This section outlines the process of finding available pins on the PCB and wiring the sensors appropriately.

Finding Pins

Below is an overhead-view schematic of the exoskeleton's PCB with the available pinouts annotated.

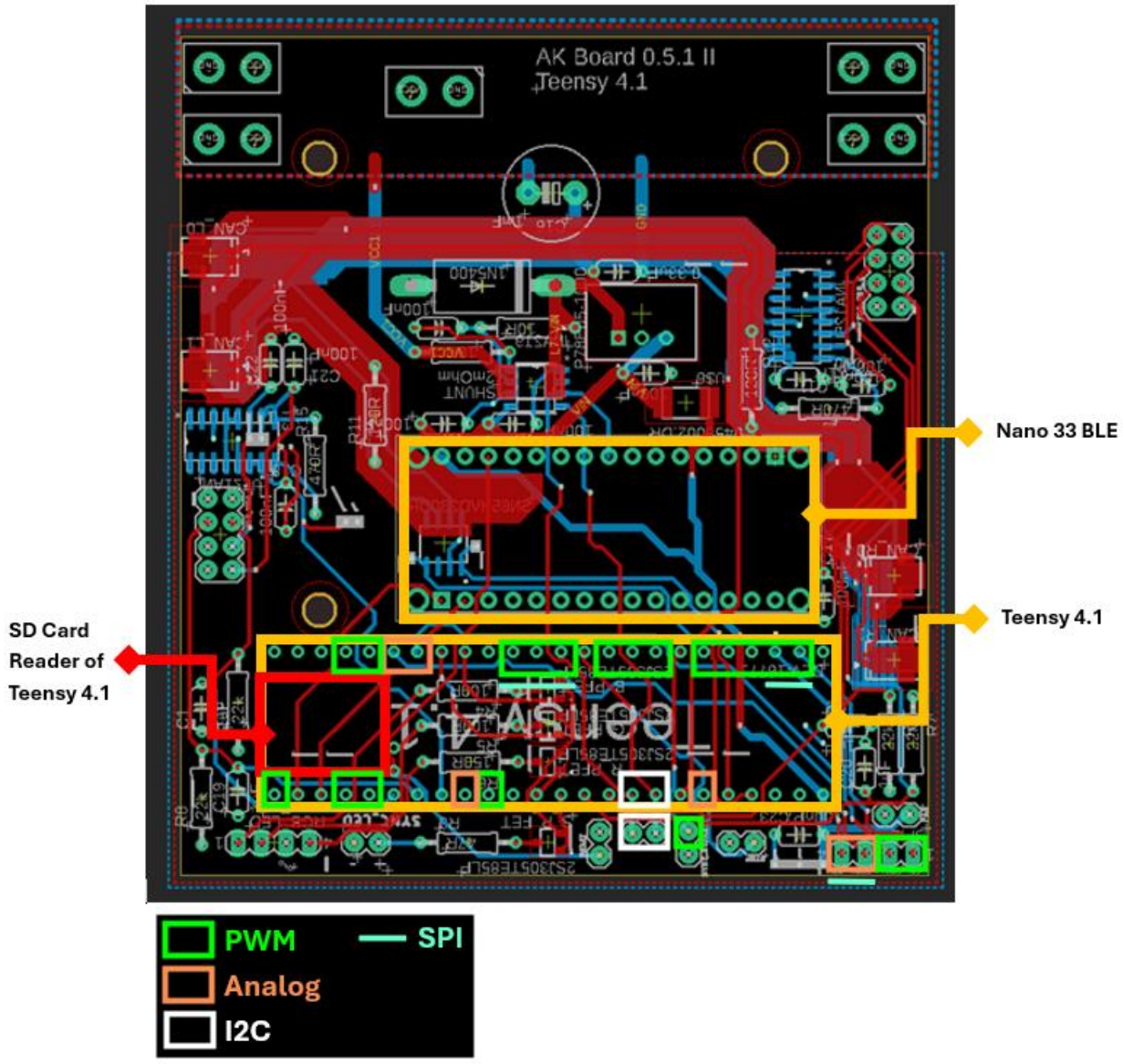


Figure 1: PCB Layout

As mentioned previously, the Nanos will be communicating with the Teensy via I2C. Thus, we'll need to select one of the available I2C pinouts labeled in the image above. In this case, we will use the two I2C pins outlined on the Teensy board itself (rather than on the PCB). Referencing PJRC's pinout diagram for the Teensy 4.1, we find that the two pins just selected correspond to pins 18 and 19, with pin 18 corresponding to the SDA line and pin 19 to the SCL line (please review the basics of I2C communication if you are unfamiliar with that

terminology). Furthermore, we again reference PJRC's pinout diagram to find the 3.3V and GND pins. Make sure to double check "Board.h" in the software to ensure these pins are available for use.

Similarly, we must now to determine the pins to be used on the Nano side. Referencing the pinout diagram for the Arduino Nano 33 BLE Sense Rev 2, we find that the A4 and A5 pins correspond to SDA and SCL respectively. Furthermore, we again note the location of the VIN and GND pins.

Soldering

Now that we know the location of the pins we need, we can solder wires accordingly. Below are images of the soldered wires. Note that the wires change color between the PCB and the nano. On the PCB, grey and purple correspond to ground and power respectively, and the green and yellow wires are for the I2C lines. On the Nanos, red and black correspond to power and ground while green and yellow again are the I2C lines.

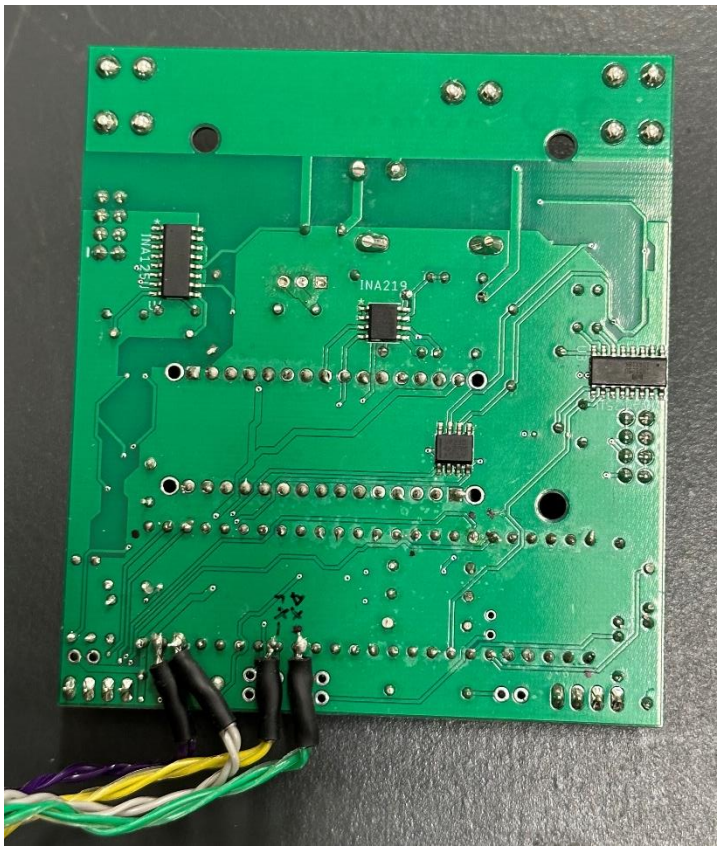


Figure 2: PCB Wiring

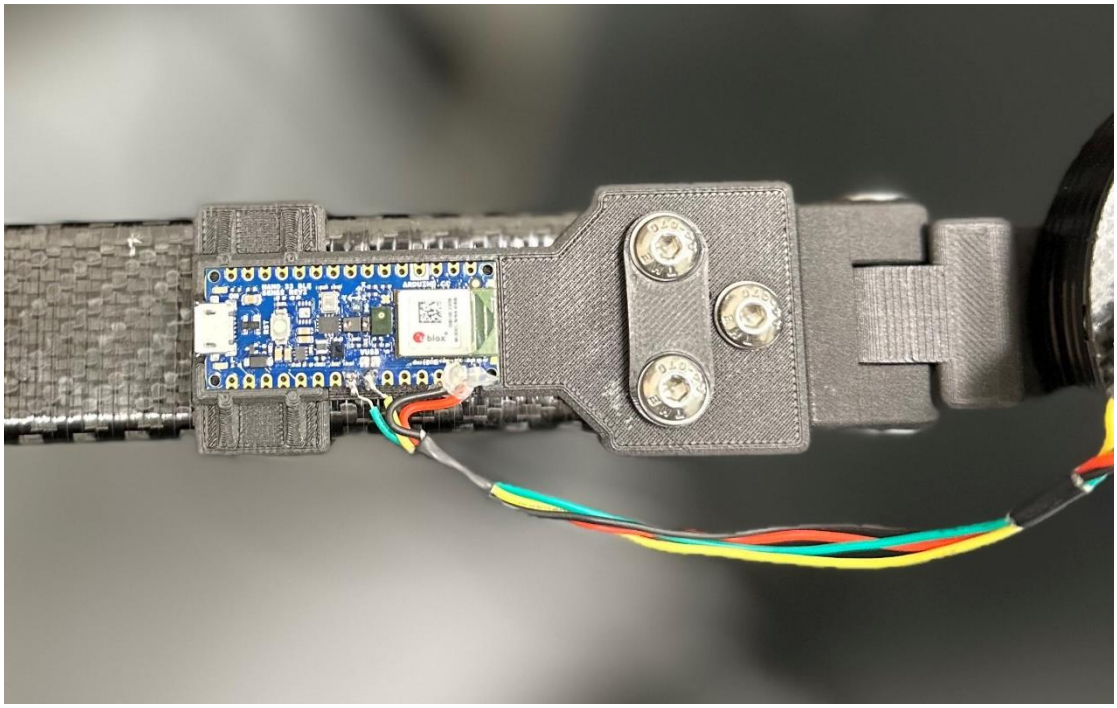


Figure 3: Nano Wiring

In the image, the PCB has two of each wire on each pin. This is because, as mentioned previously, there are two Nanos, one for either side of the exoskeleton. I2C can handle multiple devices on the same I2C line, provided the devices (i.e., the Nanos) have different addresses. With the Nanos, we can manually set the address in the code. However, if the reader wishes to use a different sensor, they should ensure that they can either set the I2C address manually, or that the two devices they are using have different addresses. If there is only one device on the I2C line however, this is not an issue.

It is good practice to use a multimeter to verify that everything is wired properly and that there are no shorts.

Code

In this section, we'll write a basic script to verify that everything is working properly. If the reader is following along, it is assumed at this point that they have downloaded OpenExo. The code we'll be writing will be located in:

OpenExo/ExoCode/src/Controller.cpp (and Controller.h).

It is also assumed that the reader has already written the code to implement their own controller, for straightforward instructions for doing so, see:

OpenExo/Documentation/AddingNew/AddingNewController.md.

Nano Code

First, we'll write the code that will go on the Nanos. Since we are using I2C, we'll need to decide which device will be the primary. In this case, the Teensy will be the primary and the Nanos will be the secondary (peripherals).

With that decided, we'll begin writing the code for the Nanos. I will only outline the code for the Nano on the left leg of the exoskeleton, as the code for the right Nano will be identical except in a few cases. When the code is side-specific, this will be explicitly highlighted.

To begin, we'll need to make two new Arduino files (one for each Nano) and name them something appropriate. For example, in this instance we named them "nanoGyroLeft.ino" and "nanoGyroRight.ino".

Global Variables

Now that the files are made, we'll want to include the Wire library so that we can use I2C (Line 2 below). Next, we'll define the I2C addresses of the Nanos. They need to be different so the Teensy can tell them apart. It can be any number, but I'll use 8 for the right Nano and 9 for the left (Line 4 below). Furthermore, since I'm using the Nano 33 BLE Sense Rev 2, I'll need to include the library that allows me to make use of the on-board gyroscope (<Arduino_BMI270_BMM150.h>; Line 6 below).

```
1  // include Arduino library for I2C
2  #include <Wire.h>
3  // define peripheral I2C Address
4  #define PERIPHERAL_ADDR 9
5  // include library for the IMU inside the nano 33 sense rev2
6  #include <Arduino_BMI270_BMM150.h>
7
8  void setup() {
9  }
10
11 void loop() {
12 }
```

Next, we'll want to establish a union to store the data we want to send (see AddingNewSensor.md in the OpenExo documentation for more information on Unions). A union is used because it allows the same data to be accessed in different ways. In our case, we get float data from the gyroscope, but we need the data to be formatted as a vector of bytes to send it over I2C. So, we'll use a union to enable us to do so easily. The union is initialized globally and the syntax for doing so is shown in the image below.

```
8   union {
9     float angularVel;         // The float representation of the data being stored.
10    byte angularVelBytes[4];  // The vector of bytes representing the data.
11  }Data;                      // Data is the name of the union.
```

Float-type data consists of four bytes. As such, we will need to initialize our byte-type variable as a vector of four bytes so that it can properly store the gyroscope data, hence the [4] at the end of "angularVelBytes".

Void Setup()

Now we'll move into void setup(), shown below. We will initialize I2C communication with "Wire.begin(PERIPHERAL_ADDR)". Remember that we defined PERIPHERAL_ADDR above. It is 8 for the right Nano and 9 for the left. Specifying the address tells the wire library that the device is a secondary, not the primary.

"Wire.onRequest(sendEvent)" tells the Nano to execute the function "sendEvent()" when it receives a request from the Teensy. Send Event will be defined last in the code.

"IMU.begin()" simply initializes the IMU on-board the Nano.

```
13  void setup() {
14    Wire.begin(PERIPHERAL_ADDR);  // initialize i2c communication as a peripheral
15    Wire.onRequest(sendEvent);  // upon request of the main i2c device, initiate the sendEvent function
16    Serial.begin(9600);
17    IMU.begin();
18  }
```

Void Loop()

Here, we will repeatedly record the gyroscope reading about the z-axis into the float-type variable in the union.

```
20  void loop() {
21    float x, y, z;
22    if (IMU.gyroscopeAvailable()) {
23      IMU.readGyroscope(x, y, z);
24      Data.angularVel = z;
25    })
26  }
```

Void sendEvent()

Finally, we'll define the function used to send our gyroscope data upon request of the Teensy. As mentioned previously, we'll need to send the data as a vector of bytes. Because we used a union, this is as easy as simply accessing the byte-type variable of the union that we had

just stored our gyroscope reading into. We also need to specify the size of the data being sent, which we established previously is four bytes.

```
28  void sendEvent() {
29    Wire.write(Data.angularVelBytes, sizeof(Data.angularVelBytes)); // sends the angle reading over i2c as bytes
30  }
```

Teensy Code

In this section, we will cover the code to be uploaded to the Teensy 4.1. What we'll need to do on the Teensy is receive the data coming via I2C from the Nanos and convert it back to float data. Additionally, since the exoskeleton runs a unique instance of the code for the left and right side, we'll want to ensure that we're receiving the data from the Nanos in a manner that agrees with that structure. This will be elaborated upon below. Again, it is assumed that the reader has already written the outline for their own controller based on the instructions in AddingNewController.md.

First, the reader will want to open Controller.h and Controller.cpp and navigate to the portions of code relevant to their controller. In my case, the controller I created is called "Angle" as can be seen in the snippet of code from Controller.h below, which is where we will begin.

Controller.h

The reader's controller should look something like the image below.

```
427  class Angle : public _Controller
428  {
429  public:
430      Angle(config_defs::joint_id id, ExoData* exo_data);
431      ~Angle() {};
432
433      float calc_motor_cmd();         /* Function that calculates the motor command. */
434  };
```

To start, we'll define a union identical to the one on the Nanos.

```
class Angle : public _Controller
{
public:
    Angle(config_defs::joint_id id, ExoData* exo_data);
    ~Angle() {};

    union {
        float angularVel;           // The float representation of the data being stored
        byte angularVelBytes[4];    // The vector of bytes representing the data
    }Data;                          // Data is the name of the union

    float calc_motor_cmd();         /* Function that calculates the motor command. */

};
```

Additionally, we'll define the I2C addresses for the right and left nano and a couple other variables (specific to the controller being designed) to help with the code we'll write in Controller.cpp.

```
class Angle : public _Controller
{
public:
    Angle(config_defs::joint_id id, ExoData* exo_data);
    ~Angle() {};

    union {
        float angularVel;           // The float representation of the data being stored
        byte angularVelBytes[4];    // The vector of bytes representing the data
    }Data;                          // Data is the name of the union

    int const rightNano = 8;
    int const leftNano = 9;
    double prevtime = 0;
    float hipAngularVel = 0;
    float cmd = 0;

    float calc_motor_cmd();         /* Function that calculates the motor command. */
};

```

Now that we've got things initialized, we can move on to Controller.cpp, where we will read the incoming I2C data from the Nanos.

Controller.cpp

The reader's controller should look something like the image below.

```
2361  Angle::Angle(config_defs::joint_id id, ExoData* exo_data)
2362      : _Controller(id, exo_data)
2363  {
2364  #ifdef CONTROLLER_DEBUG
2365      Serial.println("Angle::Angle");
2366  #endif
2367  }
2368
2369  float Angle::calc_motor_cmd()
2370  {
2371      cmd = 0;
2372      return cmd;
2373
2374  }
```

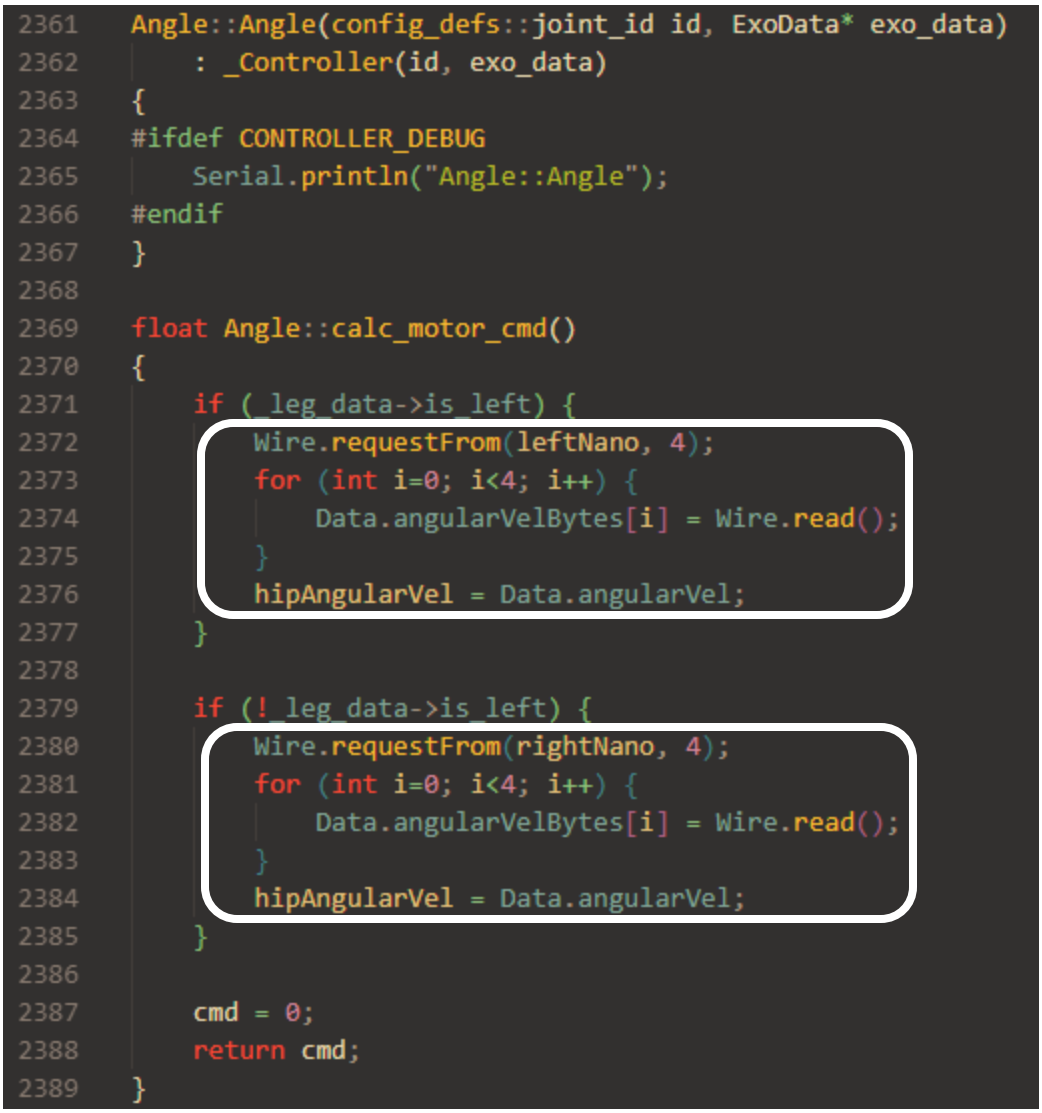
Within calc_motor_cmd() is where we will write our code to receive the angular velocity data from the Nanos. Notice that the cmd variable is set to zero at the moment. This is because we do not want to send torque to the motors. The motors should not run unless connected to the python GUI, which we will not be doing here, but to ensure that nothing unexpected happens it is best to not alter this variable for the time being.

We'll start by adding if statements so that we can have the right Nano's data be received only for the instance of code running for the right side of the exoskeleton and similarly for the left Nano. Essentially, what we will be saying is "if the current instance of the code is for the left side of the exoskeleton, then pull data from the left nano. If the current instance is for the right side, then pull data from the right Nano."

```
2361  Angle::Angle(config_defs::joint_id id, ExoData* exo_data)
2362      : _Controller(id, exo_data)
2363  {
2364  #ifdef CONTROLLER_DEBUG
2365      Serial.println("Angle::Angle");
2366  #endif
2367  }
2368
2369  float Angle::calc_motor_cmd()
2370  {
2371      if (_leg_data->is_left) {
2372
2373      }
2374
2375      if (!_leg_data->is_left) {
2376
2377      }
2378
2379      cmd = 0;
2380      return cmd;
2381  }
```

This is the only place we'll need to specify the side. We will not be writing a full controller here, but if we were, the code corresponding to the calculation of the motor command would be written outside of the if statements. Again, the if statements are simply to ensure that the data from the left Nano is read by the left-side instance of the code and similarly for the right Nano.

Now we'll fill out the if statements with the code to read from the Nanos.



```
2361  Angle::Angle(config_defs::joint_id id, ExoData* exo_data)
2362      : _Controller(id, exo_data)
2363  {
2364  #ifdef CONTROLLER_DEBUG
2365      Serial.println("Angle::Angle");
2366  #endif
2367  }
2368
2369  float Angle::calc_motor_cmd()
2370  {
2371      if (_leg_data->is_left) {
2372          Wire.requestFrom(leftNano, 4);
2373          for (int i=0; i<4; i++) {
2374              Data.angularVelBytes[i] = Wire.read();
2375          }
2376          hipAngularVel = Data.angularVel;
2377      }
2378
2379      if (!_leg_data->is_left) {
2380          Wire.requestFrom(rightNano, 4);
2381          for (int i=0; i<4; i++) {
2382              Data.angularVelBytes[i] = Wire.read();
2383          }
2384          hipAngularVel = Data.angularVel;
2385      }
2386
2387      cmd = 0;
2388      return cmd;
2389  }
```

As can be seen in the image, we have the Teensy send a request to the Nanos for data with a size of four bytes (Line 2372). Immediately after, the Teensy then reads the incoming bytes and one-by-one appends them to the vector of bytes that we had defined previously in the union in Controller.h (Lines 2373-2375). Once all four bytes have been appended, we then set a variable equal to the float version of the data just received (Line 2376).

We will now use serial prints to verify that the data is being received properly. It is worth double checking that the reader's active controller set in the SD card on the Teensy is the one they intend.

```
2369  float Angle::calc_motor_cmd()
2370  {
2371      if (_leg_data->is_left) {
2372          Wire.requestFrom(leftNano, 4);
2373          for (int i=0; i<4; i++) {
2374              Data.angularVelBytes[i] = Wire.read();
2375          }
2376          hipAngularVel = Data.angularVel;
2377          Serial.print(hipAngularVel);
2378          Serial.print("\t ");
2379      }
2380
2381      if (!_leg_data->is_left) {
2382          Wire.requestFrom(rightNano, 4);
2383          for (int i=0; i<4; i++) {
2384              Data.angularVelBytes[i] = Wire.read();
2385          }
2386          hipAngularVel = Data.angularVel;
2387          Serial.print(hipAngularVel);
2388          Serial.println();
2389      }
2390
2391      cmd = 0;
2392      return cmd;
2393  }
```

Because the serial prints are within the if statements, we will get a reading from the right and left Nano despite using only one variable; it's simply that in the code running for the right side of the exoskeleton that variable is associated with the data coming from I2C address 8, and for the left side it's associated with I2C address 9. And because the code runs for one side and then the other, we're able to format the serial prints as seen in the image. This particular snippet of code hopefully gives the reader a better understanding of how the exoskeleton runs.

Because continuously requesting and reading data from the Nanos can cause some lag in the exoskeleton when connected to the GUI, we will decrease the rate at which the Teensy samples the Nanos with the following addition (Line 2371 below).

```
2369  float Angle::calc_motor_cmd()
2370  {
2371      if (millis() - prevtime >= 100) {
2372          if (_leg_data->is_left) {
2373              Wire.requestFrom(leftNano, 4);
2374              for (int i=0; i<4; i++) {
2375                  Data.angularVelBytes[i] = Wire.read();
2376              }
2377              hipAngularVel = Data.angularVel;
2378              Serial.print(hipAngularVel);
2379              Serial.print("\t ");
2380          }
2381
2382          if (!_leg_data->is_left) {
2383              Wire.requestFrom(rightNano, 4);
2384              for (int i=0; i<4; i++) {
2385                  Data.angularVelBytes[i] = Wire.read();
2386              }
2387              hipAngularVel = Data.angularVel;
2388              Serial.print(hipAngularVel);
2389              Serial.println();
2390          }
2391          prevtime = millis();
2392      }
2393
2394      cmd = 0;
2395      return cmd;
2396  }
```

Conclusion

In summary, we have added two new sensors, wired them, programmed them to send data over I2C, programmed the exoskeleton to receive that I2C data, and have written the code in such a way that the sensors send their data to their respective side-specific instance of the code.

Hopefully it is easy to see how this can be further extended to use this data to send motor commands (the variable can be treated as any other variable, such as FSR data in the Franks_Collins controller).