Sensors

Purpose

The sensors allow C1C0 to take input from its surroundings and map out its environment when it encounters obstacles. The goal of the project this semester was to ultimately provide C1C0's Jetson with the data collected from the sensors and store that information so that it could be retrieved when it is needed, namely for path planning algorithms.

Progress

This semester, the firmware code was updated to include functionality for sending messages through one of the serial ports in order to send its sensor measurements to the Jetson using UART communication. An API was developed for taking this byte information and decoding it into understandable data and storing it on the Jetson, which can then be retrieved upon command. This API is now usable by the CS subteams to be used on their software.

LIDAR (Light Detection and Ranging) Sensor

The LIDAR sensor is located near the bottom front side of C1C0. It is utilized in helping C1C0 map out its environment. The sensor emits lasers which bounce off surrounding objects back to the sensor allowing it to calculate the distance between to the nearest object using the time it takes the beam to "bounce back". Our sensor constantly collects data and fills an array of size

Figure 46: LIDAR Sensor

360 which records both the angle at a particular moment in time (between 0 and 360) and the associated distance at that point in time. Once the data is requested, it is sent to the Jetson. This allows C1C0 to essentially visualize its surroundings in the plane at which the sensor is located. The LIDAR data is utilized by the CS subteam in path planning.

```
471  if (IS_OK(lidar.waitPoint())) {
472        uint16_t distance = (uint16_t) lidar.getCurrentPoint().distance; //distance value in mm unit
473        uint16_t angle    = (uint16_t) lidar.getCurrentPoint().angle; //angle value in degrees
474
475        //Serial.println("Angle:" + String(angle));
476        //Serial.println("Distance:" + String(distance));
477        if (lidar_array_index <= LIDAR_DATA_POINTS-1) {
478          LidarData[lidar_array_index*2] = angle;
479          LidarData[lidar_array_index*2+1] = distance;
480          lidar_array_index++;
481          }
482        }
483    else {
484
485      // try to detect RPLIDAR...
486      rplidar_response_device_info_t info;
487      if (IS_OK(lidar.getDeviceInfo(info, 100))) {
488         // detected...
489         lidar.startScan();
490         // start motor rotating at max allowed speed
491         analogWrite(RPLIDAR_MOTOR, 255);
492         delay(1000);
493      }
494    }
495
496    if (lidar_array_index == LIDAR_DATA_POINTS) {
497        convert_b16_to_b8(LidarData, lidar_databuffer,LIDAR_DATA_POINTS*2);
498        lidar_array_index=0;
499    }
500
```

Figure 47: Lidar Data Point Collection Code

Terabee Sensors

The terabee sensors are similar to the LIDAR sensors, but instead emit and receive infrared radiation in order to detect objects in its surroundings. These sensors are in units which are placed in fixed positions on C1C0 and are directed at specific angles. Two of the terabee sets

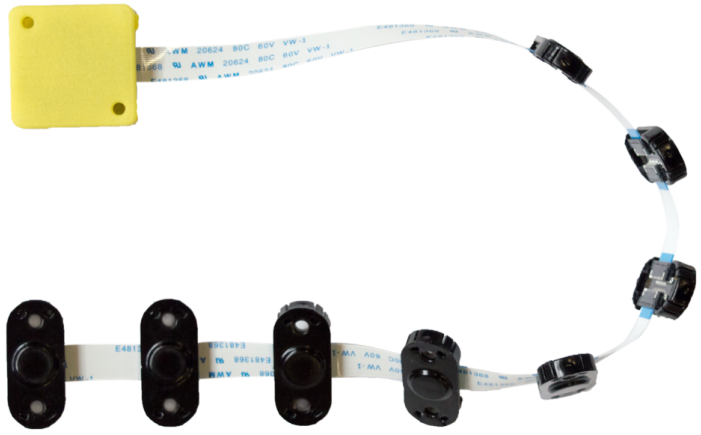

Figure 48: Terabee IR Sensor Set

(sixteen sensors in total) are placed radially around C1C0's head, and the third terabee set (eight sensors in total) is placed radially fourteen inches below the other two sets. This allows the terabee sensors to be used for avoiding collisions that the LIDAR may not be able to detect because of their different orientations on the robot body.

```
377   avail = Serial1.available();
378   if (avail > 0) {
379     if (state == MSG_INIT || state == MSG_BEGIN) {
380       find_msg(state, Serial1);
381     } else if (state == MSG_DATA) {
382       Serial1.readBytes(terabee1_databuffer, 16);
383       state = MSG_INIT;
384       convert_b8_to_b16(terabee1_databuffer, terabee1_data);
385 //      for (int i = 0; i < 8; i++) {
386 //        Serial.print("Sensor1 ");
387 //        Serial.print(i);
388 //        Serial.print(": ");
389 //        Serial.println(terabee1_data[i]);
390 //      }
391     }
392   }
```

Figure 49: Terabee Data Point Collection Code

The terabees and LIDAR both use UART in order to send its data to the Teensy when it is collected. Teensy only attempts to read data from the sensors when there are bytes waiting in the serial buffer to be read. This prevents the Teensy from attempting to read data when there is none available, causing an automatic reset of the hardware. Once there is data in the serial buffer, the Teensy will read out sixteen bytes from the serial buffer and store it into an array designated for that sensor. This corresponds to two-byte measurements received from eight of the terabees.

IMU (Inertial Measurement Unit) Sensor

The IMU sends data based on the sensor's movement, and will be mounted in C1C0's head. The IMU collects data based on both linear and angular accelerations, and the firmware uses the global positioning library to convert these collected values to record orientation using displacement values with respect to each three-dimensional axis. These values are the data that is sent to the Jetson, even though they are not measurements directly taken from the IMU.

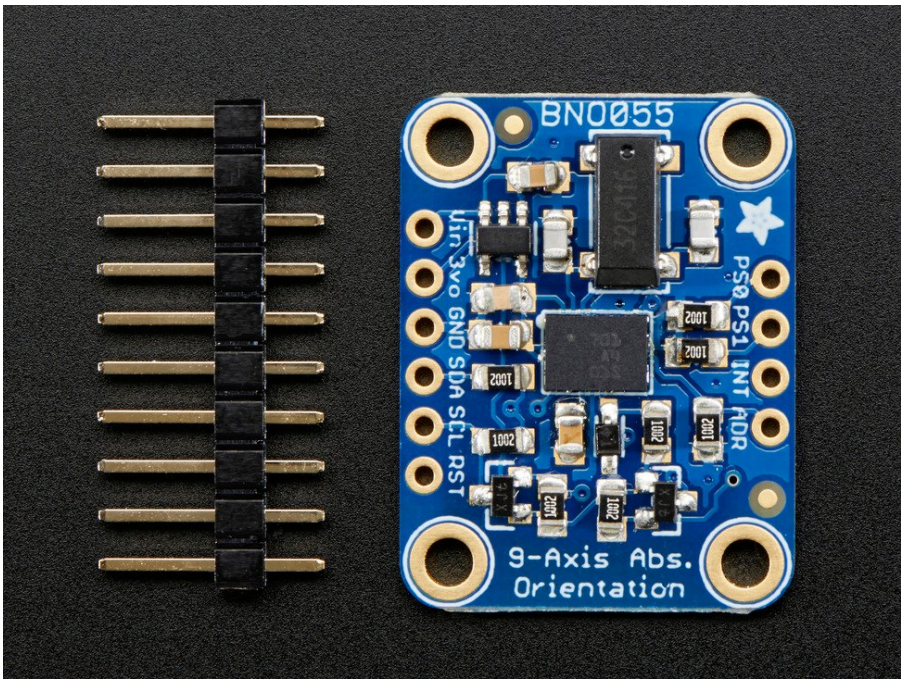

Figure 50: Inertial Measurement Unit

```
428
429         bno.getEvent(&event);
430         imu_data[0] = (int)event.orientation.x;
431         imu_data[1] = (int)event.orientation.y;
432         imu_data[2] = (int)event.orientation.z;
433         convert_b16_to_b8(imu_data, imu_databuffer, 6);
434
```

Figure 51: IMU Data Collection

This sensor uses I2C communication in order to send its data to Teensy, constantly updating its values. To obtain the most data points in a given period of time from the IMU, it would be optimal to place the data collection code in the open loop of the firmware code. However, because the data is not being sent in the open loop (which will be enumerated in the Teensy Code section), there is no reason to constantly update the IMU data arrays. In order to ensure that the IMU data that is sent to the Jetson is its most recent version, the data collection is placed right before it is sent within the Terabee 3 data collection scope. For the actual data collection of the values that are sent to the Jetson, the getEvent function from the global positioning library is used to store each x, y, and z value into the orientation fields of an event class. These are then casted from a floating point value to an integer (the reason for which will be explained in the Teensy Code section) and stored into the array designated for the IMU points. The three values are then converted into six one-byte integer values so that it can be sent in the serial buffer.

Wiring

Figure 52 displays a wiring diagram for the sensors and their connections to Teensy's breakout board (omitting the sets of pins that are unused on the board), and Figure 53 displays an example of the sensors wiring to the PCB, using only the third terabee set. The code written on the firmware is specific to each sensor given its own respective TX/RX slot. The wires for each sensor are relatively short, which was not an issue for the purposes of testing the API and the firmware. However, as the sensors are starting to be mounted onto C1C0 at varying distances, the wires for each sensor will need to be extended.

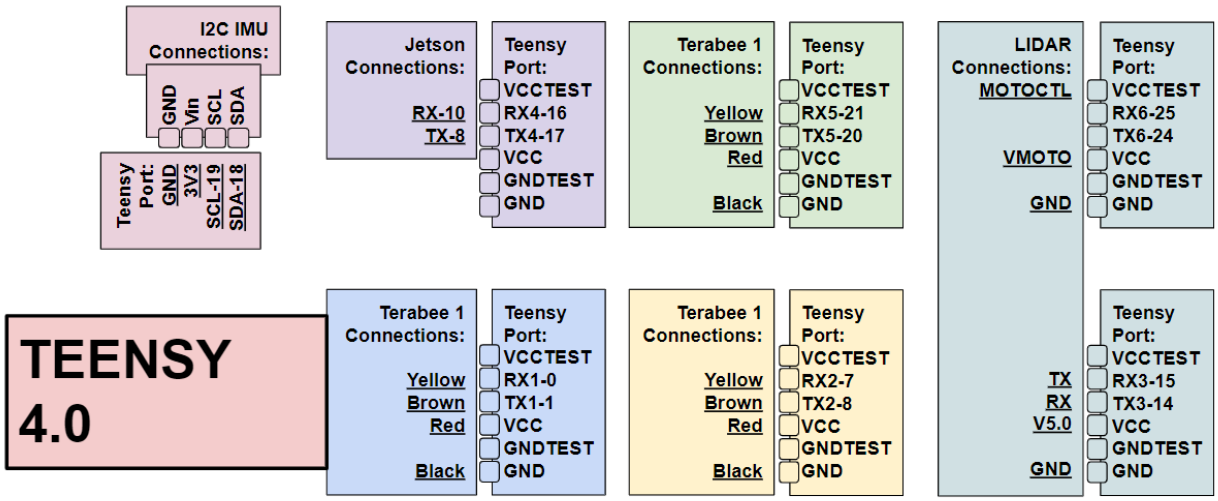

Figure 52: Sensors and Teensy Wiring Connections

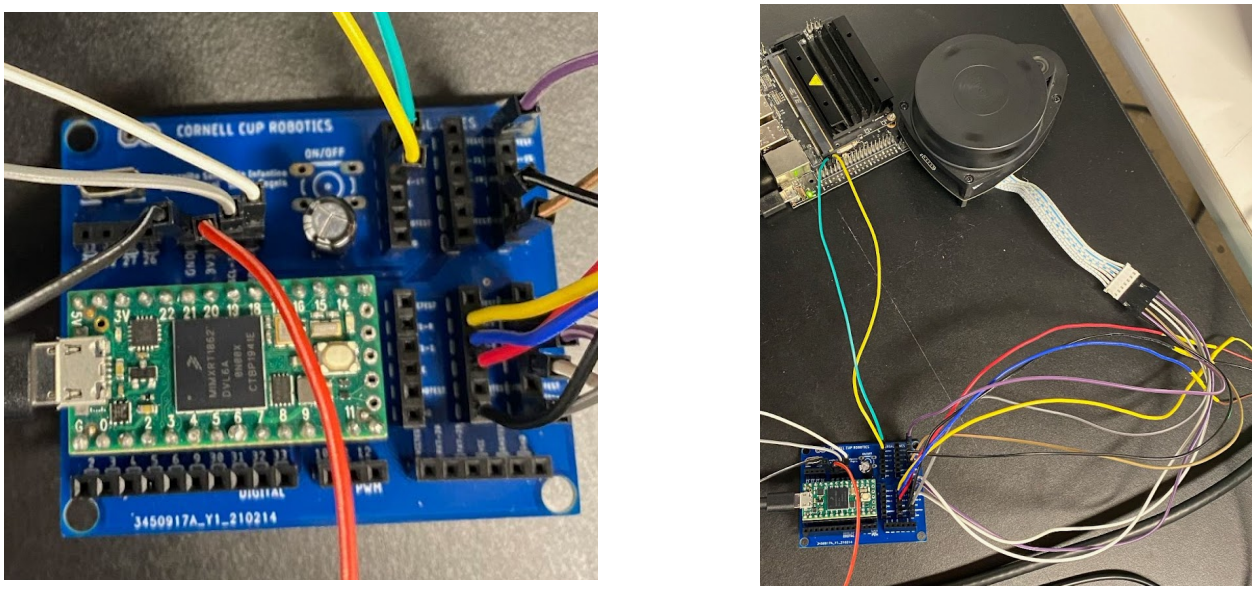

Figure 53: Wiring Example for Teensy Breakout

More information about the Teensy breakout board can be found in section 5.2 of the C1C0 Fall 2020 Final Documentation:
https://docs.google.com/document/d/1ZgagdO0gPmTSg7pqps35l8OJpdnSuMHlxYUMDMlNR1Y/edit

Teensy Code

Because R2Protocol is used for sending and receiving the sensor data, the messages from each sensor are sent in their respective packets labeled with their sensor type. These packets are sent to the Jetson in sequence with each other so that the software on the Jetson can correctly decode the byte information. Additionally, since the data must be sent to the Jetson in a consistent sequential order, and the sensors each read their data at different rates, the data sent cannot just send its packet every time the data is updated. Instead, the code forces the packets to be sent at the same time the third terabee sensor is updated. This specific sensor was chosen because having the data being sent outside of the sensor's scope will cause the data to be sent too fast with respect to the bytelength of the messages. Having the packets sent with the LIDAR updates seemed to be too slow, so the terabee's timing was a fair middle-ground.

Normally, to calibrate the IMU it must be rotated in a variety of ways and calibration code must be run. Calibration was excluded from the firmware code since the IMU will be attached to C1C0 and will not be able to move in the ways that calibration requires. This calibration will not be a problem because position calculations will be made relative to the IMUs initial position. To avoid further problems due to the IMU's sensitivity, decimal values (which are irrelevant) were floored and converted to integers to simplify the data sending process.

The latest commit of the firmware code can be found here:
https://github.com/cornell-cup/c1c0-ece/blob/master/teensy_sensors/teensy_sensors.ino

Sensors API

The sensor API is primarily utilized by the CS subteam in gathering data from the sensors for path planning operations and calculations. The API allows the Jetson to open its serial port to start receiving messages, as well as close the serial port when it is not being used. It also provides functionality for reading the information sent from the Teensy and decoding the byte information and storing them into their respective arrays. The API decodes the information by parsing the byte array it receives and looking for the data's label to determine the type of sensor the information is from. Based on the type of sensor, the API will take the byte array that corresponds to the sensor and convert it into integers (or a tuples of integers in the case of the LIDAR). These measurements stored in global fields are constantly updated as time passes, and

the API provides functions for obtaining each of the data arrays. The following sections provide descriptions of key functions used in the API.

The latest version of the API which the CS teams use can be found here:
https://github.com/cornell-cup/c1c0-ece/blob/master/TEST_API.py

Decode Arrays

The decoding function first starts by starting a while loop, which will be exited if the data read is correct. The while loop parameter is first initialized to be false. Then, the global variables for the arrays that hold the sensor measurements are initialized to be empty to get rid of its old data. Once the data is read from the serial port, R2Protocol is used in order to decode the data.

```
86          good_data = False
87          print("GET ARRAY FUNCTION")
88          while(not good_data):
89                  terabee_array_1 = []
90                  terabee_array_2 = []
91                  terabee_array_3 = []
92                  ldr_array = []
93                  imu_array = []
94
95                  print("IN LOOOP")
96                  ser_msg = ser.read(TOTAL_BYTES) #IMU +IR1+IR2+IR3+LIDAR+ENCODING
97                  #print(ser_msg)
98                  print("GOT MESSAGE")
99                  mtype, data, status = r2p.decode(ser_msg)
```

Figure 54: Decode Arrays Function Serial Read

Because the information sent to the Jetson is in a sequence of arrays with information of different types, there are sometimes cases when the Jetson begins to read the bytes from the serial port from a location that is not the beginning of the message initially sent from the Teensy (which is designated as the first terabee sensor's message). Normally, R2Protocol uses a status parameter that determines if the data is correctly read. However, with the sensors, because there are three different types of sensors with different data sizes, the API cannot rely on the status parameter to determine a correct data read. Instead, the type parameter of the R2Protocol decode function must be considered. If the message type is one of IR, IR2, IR3, LDR, or IMU, (which is determined by the first four bytes of the serial message) the decoding functions for each case can be called and the while loop parameter is set to True so that the loop does not run again. However, if the first four bytes do not match one of the types, this means that the Jetson is

reading the byte array from the middle of a sensor measurement. To solve this, the serial buffer must be cleared so that the next packet of information sent is read from a desired location in the byte array.

```
110                 if (mtype == b'IR\x00\x00'):
111                         decode_from_ir(data)
112                         good_data = True
113
114                 elif (mtype == b'IR2\x00'):
115                         decode_from_ir2(data)
116                         good_data = True
117
118                 elif (mtype == b'IR3\x00'):
119                         decode_from_ir3(data)
120                         good_data = True
121
122                 elif (mtype == b'LDR\x00'):
123                         decode_from_ldr(data)
124                         good_data = True
125
126
127                 elif (mtype == b'IMU\x00'):
128                         decode_from_imu(data)
129                         good_data = True
130
131                 else:
132                         print("NO GOOD")
133                         ser.reset_input_buffer()
```

Figure 55: Conditional Statements in Decode Arrays

Decode From Sensor

The API includes five different functions that cover each case for when the Jetson begins reading the serial message from each sensor type. Figure 55 shows the function for when the Jetson begins reading from the first Terabee (the most common case). The function takes in the input parameter that is passed through the decode_arrays which contains the byte array for each sensor measurement, as well as the encoding bytes for the arrays. This function goes through the byte array and splices it based on the data length, and then moves past the encoding bytes that lie in between sensor measurements to obtain the next sensor data. The function then calls the functions for each sensor type that fills a target array with the data obtained. These functions first convert the bytes into two-byte integers, then append the new values into the global target array. The LIDAR array holds 360 tuples of two integers, the terabee arrays hold eight integers, and the IMU array holds three integers.

```
136 def decode_from_ir(data):
137         """
138         Description: Function that decodes the data if the received mtype
139         is that corresponding to Terabee 1.
140         Returns: Nothing
141         """
142         terabee1_data = data[:TERABEE_DATA_LEN]
143         terabee2_data = data[TERABEE_DATA_LEN + ENCODING_BYTES:TERABEE_DATA_LEN + ENCODING_BYTES + TERABEE_DATA_LEN]
144         terabee3_data = data[TERABEE_DATA_LEN*2 + ENCODING_BYTES*2:TERABEE_DATA_LEN*2 + ENCODING_BYTES*2 + TERABEE_DATA_LEN]
145         ldr_data      = data[TERABEE_DATA_LEN*3 + ENCODING_BYTES*3:TERABEE_DATA_LEN*3 + ENCODING_BYTES*3 + LIDAR_DATA_LEN]
146         imu_data      = data[TERABEE_DATA_LEN*3 + LIDAR_DATA_LEN + ENCODING_BYTES*4:TOTAL_BYTES]
147
148         terabee_array_append(terabee1_data, terabee_array_1)
149         terabee_array_append(terabee2_data, terabee_array_2)
150         terabee_array_append(terabee3_data, terabee_array_3)
151         lidar_tuple_array_append(ldr_data, ldr_array)
152         imu_array_append(imu_data, imu_array)
```

Figure 56: Decoding Function From Terabee 1

Get Arrays

The data that is decoded gets stored in global variables that store the arrays for the sensor measurements. As long as the decode_arrays function continues to be called, these arrays will be updated. The get_arrays function allows these global arrays to be called.

```
60          if array_type == "TB1":
61                  return terabee_array_1
62          elif array_type == "TB2":
63                  return terabee_array_2
64          elif array_type == "TB3":
65                  return terabee_array_3
66          elif array_type == "LDR":
67                  return ldr_array
68          elif array_type == "IMU":
69                  return imu_array
```

Figure 57: Get Arrays Return Conditions

Future Work

Continuing working on this project will include working alongside CS subteams to fix any bugs that they find in the API or the Teensy code, as well as optimizing the code to improve efficiency and speed. Such optimizations can include changing the baud rate of the data being sent to the Jetson and the amount of data to be sent at one time. One optimization already made was changing the number of datapoints that the LIDAR sends from 50 to 360 points, which allowed for a full 360 degree coverage in the smallest amount of time. Additionally, because the Jetson will need to send and receive data from multiple microcontrollers, the Jetson will need to

communicate messages with the Teensy to indicate whether or not the Teensy should send the sensor information. This must be done so that the Teensy does not flood the Jetson's serial buffer with information when the Jetson is running processes that involve data from different sources. Because this feature is a result of multiple microcontrollers sharing serial ports with the Jetson, cooperation with the bus protocol will be essential for implementation.