**Sr Design II – Communications Midterm Report**

**Table of Contents –**

1. **GPS**
2. **Distance**
3. **IMU**
4. **Ultrasonic**
5. **ESP32**
6. **CAN**
7. **ROS2**
8. **EE GANTT**

**List of Figures –** *(for EECOM)*

* **EECOM Figure 1:**

Sensor data monitoring while programming used through SerialPlot

* **EECOM Figure 2:**

Sensor breadboard for initial sensor coding, telemetry testing and wiring. A Raspberry Pi Pico W -MCU (green), Sparkfun XA1110 GPS (red), Adafruit BNO055 IMU (blue), and HC-SR04 ultrasonic sensor (metal).

* **EECOM Figure 3:**

Left is the CAM diagram for the ESP32 communications shield PCB. Center is the ESP32 pinout for the developer board used with this shield. Right is the final board and shield intended for use.

* **EECOM Figure 4:**

Motor controller used for mode testing with push-button enter and exit mode buttons and potentiometer with the MCP2515 CAN module for motor communication

* **EECOM Figure 5:**

Current high-level communication diagram. Transmission load is provided for each control block.

**I. GPS**:

Both a Sparkfun XA1110 and the ZED FP-9 are producing data for this project but not optimized for potential accuracy. SBAS and base station (collectively the differential GPS) will significantly improve results. The error of 2.5m for the XA1110 in good conditions could be reduced to 1m or less without need of expensive VBOX or RTK (Real Time Kinematics) GPS units, where 2.5 m is the max error. Despite the cost, RTK is the best solution for accurate results which will reduce confusion and allow the rover to have greater confidence in navigation. This should provide a large speed up in the competition without the risks or need to increase the speed of the rover itself and a greater likelihood of meeting the 3-meter error allowance for the competition. With enough satellites in view (SIV) and a Horizontal Dilution of Precision (HDOP) of 1 or less, the XA1110 provides the 2.5m accuracy required to achieve this, but with a large degree of time and energy for correction will be required. The physical communication line between the XA1110 and the ESP32 utilized I2C with support contained int the GPS libraries. The TinyGPS Plus library has geographical data for its point-to-point distance calculation proving to be far more accurate than using the Haversine formula alone.

Due to the need for higher accuracy, the team decided to upgrade to the Sparkfun ZED-FP9 which will allow accuracy of 10 centimeters (about 3.94 in) with the RTK enabled. If the accuracy is within the width of the rover itself, then the auto-navigation should have minimal errors to resolve for. Unfortunately using this module for the differential GPS may not be possible given the time remaining for this project, but it will still provide a large boost in accuracy. Furthermore, this would be a good project for the coming years to properly optimize the navigational system by using another u-Blox RTK GNSS (Global Navigation Satellite System) module to build this differential system properly with dead reckoning.

A picture containing diagram

Description automatically generated

EECOM Figure 1: Sensor data monitoring while programming used through SerialPlot

**II. Distance**:

The distance testing of the Ubiquiti 2.4GHz Rocket M2 radio system has proven to require more time and resources than originally speculated. Acquiring the ability to test the radio system wirelessly without the rover required voltage regulation of 24v on with a jack for DC-input POE injector in which is required for both radios to work. 15ft wires were used to get the antennas far enough from the cars to prevent reflection. This also required a temporary stand for the antennas, unfortunately one was too difficult to support properly and affordable, so cylindrical antenna had to be held. A splitter to a 9v regulator was used to attach the IP camera to the power supply used for the POE injector. The camera was connected to the access point since the rover will be the access point radio and results were read at the base station side. Due to limitations of needing to be near the car which produced safety concerns when parking off roadsides, uneven terrain, and other various issues. Testing was done outside the HUB and 330 meters (about 1082.68 ft) provided no change in quality and no signal dropped within line of sight. There was a 5 dB drop of 30 dB before signal is unusable, which indicates a kilometer should be achievable with this system. When the antenna was within 2 meters of the car, the signal was strong, but the video connection was inconsistent as expected with a surface which reflects radio signals.

**III. IMU**:

The Adafruit BNO055 9 axis IMU is connected via an I2C bus. The prototype shared the same bus with the GPS to produce the sensor graph from SerialPlot. Tilt results, heading, and vibration all proved reliable functionality. It should be noted that the sensor needs to be rolled around and gestured in a figure-8 pattern to initialize accurate readings upon reset. This sensor is essential for navigation reference. Pitch and roll functions are used directly from Adafruit’s BNO055 library. Heading is derived from the magnetometer data fused with the tilt angles to provide more stable heading when the rover is not level.

Raw accelerometer data is used to produce a vibration metric. The difference is taken between two readings at each axis and the time change is found in milliseconds. Differentiation of acceleration provides jerk; the magnitude of this three-dimensional vector allows substantial changes is acceleration such as being jostled or dropping abruptly will be distinctly visible. Due to the function being the slope of acceleration, values have a low maximum magnitude and are therefore linearly amplified for monitoring purposes. This metric could be good for slowing the vehicle when there is large turbulence to maintain the integrity of the rover.

A picture containing text

Description automatically generated

EECOM Figure 2: Sensor breadboard for initial sensor coding, telemetry testing and wiring. A Raspberry Pi Pico W MCU (green), Sparkfun XA1110 GPS (red), Adafruit BNO055 IMU (blue), and HC-SR04 ultrasonic sensor(metal).

**IV. Ultrasonic Sensors**:

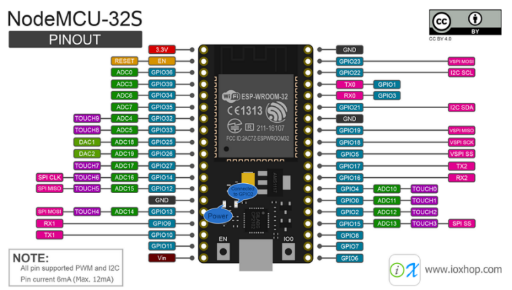
The Sparkfun ultrasonic sensors will be mounted to the front of the vehicle aimed 5 meters ahead of the rover to provide data on unseen obstacles. The data was generated and calculated using a 40kHz pulse. The raw data was very noisy with useless data and large variance, frequently defaulting to 10 meters. When the sensor does not detect an echo, it displays 10 meters. A threshold of 9 meters was placed where the old data is used instead of the null response of 10 meters. This is valuable since distances above 5 meters are not used. The mean and variance were taken over 10 points of data where if variance is equal to or greater than the mean then it defaults to previous raw data. Average update using this trust-based filtering was 10Hz.

**V. ESP32**

An MCU is required for operation of the AK60-6 motors, but the ESP32 was chosen for its wireless networking capabilities to create local control nodes. This prevents bottlenecking and ensures high reliability in manual control mode with minimal reliance on the TX2. The goal of the Esp32 network is to reduce cords where data cables are local to each motor controller, providing that a power cable should be the only cable running through the rover’s joints. This reduces risk of tugging on the wires, improving speed by allowing parallel signals to control the individual nodes and improves reliability by reducing the points in which the data needs to pass to pass for the manual mission. The wireless MCU network also increases ease in maintenance of the rover by reducing the number of cords to disconnect and reconnect when updating or servicing individual hardware that need to independently communicate between each other. A mesh network in which reports to the local antenna and then jumped to the base station antenna to ensure transmission over distance. IoT devices connected to the WAN will be static to prevent the overhead cost of DHCP since no device in the network needs to be dynamically detected due to all potentially connected devices being known.

The ESP32 and most MCUs, even various Arduinos, are not easily integrated with ROS through the micro-ROS. This makes wireless communication difficult with ROS, so a wired I2C bus from each controller to the TX2 is still an option, but an MQTT-ROS bridge node is attempted to be developed to get around this issue. Both ROS and MQTT are publish-subscribe protocols, but the goal is to use ROS for controls internal to the control side device and use MQTT for wireless communication with the peripheral devices.

Diagram, schematic

Description automatically generatedA picture containing text, electronics

Description automatically generated

EECOM Figure 3: Left is the CAM diagram for the ESP32 communications shield PCB. Center is the ESP32 pinout for the developer board used with this shield. Right is the final board and shield intended for use.

Due to heavy vibrations, tug on the connection, and the AK60-6 using a CAN bus with CAN-H and CAN-L ports, prototyping for a communications shield for the ESP32 devices has begun, initial design was made specifically for the NodeMCU-32S WROOM-based ESP32 development board but can be easily updated for other boards if necessary. The communication shield utilizes JST connectors for all wires and includes independent secure UART, I2C, SPI, GPIO, PWM, and ADC (Analog Digital Converter) connection ports. The can uses a smaller JSTPH connector. To make the ESP32 capable of communicating with CAN a transceiver and TJA1050 controller connected to the RX\_CAN and TX\_CAN ports. Unfortunately, changing the example C++ code provided by the manufacturer did not work with the motor. Multiple resources provide that the issue is that the TJA1050 requires 5v logic. The SN65HVD230 transceiver worked in combination with the MCP2515 CAN-SPI translator IC with few changes to the code provided in the manual. The total size of the shield design is 61.2 mm (about 2.41 in) long and 105 mm (about 4.13 in) wide to minimize the size of the controller nodes, but still contains extra space to provide versatility of drilling sensors such as the IMU directly to the controller’s board. Unfortunately, for motor controls the new transceiver module is large, and does not fit in the space allocated but should not pose a problem. The shield design was made in Altium Designer 23.1.1.

**VI. CAN**

The AK60-6 motors use CAN as previously mentioned. To ease the difficulty of communications with the CAN bus, an MCP2515 SPI-to-CAN adapter is used as previously mentioned. The second prototype was produced and fixed various flaws in the original. Explain about benefits of CAN, and cord requirements. Custom cords and JST fittings have been acquired to extend the bus and provide more reliable connections. The controller used has a 120-ohm termination option as required for physical CAN communications. The communication is over a 1MHz line for the MIT mode which is the highest frequency for CAN communications. The command protocol for CAN requires addressing and packing 8-byte commands with a header in which are provided in the user manual. The mode selections provided are servo and MIT Control. The MIT control protocol provides a simplified instruction set for the motor where current control is not required and allows for simple control of torque, velocity and position with onboard encoder correction held within each motor’s driver. Velocity mode will be used for manual control and position is proposed to be used for most of the autonomous missions, thus both protocols will be utilized when transmitting commands from the ROS nodes to the MCU. Due to difficulty working with the shift registers to acquire data, power and current information will be more difficult to obtain from the MCU connection due to difficult due to a less than ideal explanation provided in the manual for the read data bit masking needs relative the current mode and errors masking this could lead to difficult to notice false readings. A device for motor testing has this ability, the r-Link, in which connects the motor to the computer for diagnostics, debugging, settings, and calibration allows for these readings with the limitation that it prevents the ability to acquire this data in real world conditions in which the rover will encounter.

A picture containing electronics

Description automatically generated

EECOM Figure 4: Motor controller used for mode testing with push-button enter and exit mode buttons and potentiometer with the MCP2515 CAN module for motor communication.

**VII. ROS2**

ROS is used to publish and subscribe data between control nodes to tilt/vibration, IP/USB video, ultrasonic, GPS, encoders, and motor actuation. To implement the communication the boards will utilize the micro-ROS library and client sharing topics with the autonomous section, led by the CS, to simplify communications with the components with hopes of integrating MQTT for remote communications. This is due to extreme compatibility limits between micro-ROS and commercially available microcontrollers. Libraries are not available for this functionality, so we will be required to develop our own bridge nodes to transform the data into ROS messages. One major difference between the methods used by the manual controls and autonomous controls is the data will need to be positional for some of the autonomous tasks, but strictly velocity control for manual unless joint kinematics can be incorporated. Another difference being that commands are provided to the ESP32 nodes by the Yahboom NVIDIA Jetson TX2 for the autonomous mission, where all other missions the vehicle is controlled from the base station. This means that in case of difficulties applying the ROS control system, a few simple python control nodes using MQTT should be manageable to produce for manual controls to achieve base functionality.

Diagram

Description automatically generated

EECOM Figure 5: Current high-level communication diagram. Transmission load is provided for each control block.

**VIII. EE GANTT**

We were hoping to have been constructing and assembling by this time, but various complications have surfaced through the project to cause our progress relative to the GANTT to be a month delayed. Between motor shipping times, finding the correct hardware, obtaining a method with equipment for testing, and getting all the initial hardware setup has caused the project to be delayed. Given the delays we will not be in position to compete in the URC challenge, but the electrical team should be able to produce a functional manual control system by the end of the project given that a large amount of the remaining work is incorporating currently functional components, debugging, and optimizing the system.

**APPENDIX –**

* **Sensor code**
* #include <SparkFun\_I2C\_GPS\_Arduino\_Library.h> //Use Library Manager or download here: https://github.com/sparkfun/SparkFun\_I2C\_GPS\_Arduino\_Library
* I2CGPS myI2CGPS; //Hook object to the library
* //TINY GPS++ IS BUGGY, UNINSTALL AND REINSTALL LIBRARY IF NECESSARY
* #include <TinyGPS++.h> //From: https://github.com/mikalhart/TinyGPSPlus
* TinyGPSPlus gps; //Declare gps object
* #include <Arduino.h>
* #include "WiFi.h"
* #include <PubSubClient.h>
* #include <string.h>
* #include <time.h>
* #include "WiFi\_Logins.h"
* #include <Wire.h>
* #include <Adafruit\_Sensor.h>
* #include <Adafruit\_BNO055.h>
* #include <utility/imumaths.h>
* Adafruit\_BNO055 bno = Adafruit\_BNO055(55);
* #define SSID  WIFI\_NETWORK\_B
* #define WIFI\_PW  WIFI\_PASSWORD\_B
* const char\* device\_name = "gps\_XA1110";
* WiFiClient XA1110;
* PubSubClient client(XA1110);
* bool pinFlag = false;
* bool rebooted = true; // initialize functions/conditions on reboot
* float pitch = 0;
* float roll = 0;
* float yaw = 0;
* double pinLat, pinLng, dist\_b, distance;
* float yaw\_old;
* float jerkMag;
* float z\_vibeOld,y\_vibeOld,x\_vibeOld;
* //The following tells the TinyGPS library to scan for the PMTK001 sentence
* //This sentence is the response to a configure command from the user
* //Field 1 is the packet number, 2 indicates if configuration was successful
* TinyGPSCustom configureCmd(gps, "PMTK001", 1); //Packet number
* TinyGPSCustom configureFlag(gps, "PMTK001", 2); //Success/fail flag
* String configString;
* int ta = 0;
* int tb = 0;
* const int trigPin = 26;
* const int echoPin = 27;
* long duration;
* int objDistance[10]; //setup for 10 point variance check to avoid noise of 10m (says 10m when unsure)
* int objDist;//initialize final container for object distance
* int objDistOld=100;//initialize to expected distance (mounted) of 1m
* int stdDev;
* int distMean;
* boolean debug = false; //Keeps track of the enable/disable of debug printing within the GPS lib
* void sub2topics(){
* client.subscribe("set/pin0");
* client.subscribe("pinLat");//subscriibe(topic,qos)
* client.subscribe("pinLng");//subscriibe(topic,qos)
* client.subscribe("cats");//testing strings out after length unsigned int to int comparison in collect payload
* }
* void checkConnection() {
* WiFi.mode(WIFI\_STA);
* if (WiFi.status() != WL\_CONNECTED)
* {
* while (WiFi.status() != WL\_CONNECTED)
* {
* WiFi.begin(SSID, WIFI\_PW);//check for known network HOME
* Serial.print(".");
* delay(2000);
* }
* }
* }
* void checkBroker(){
* if(client.state()!=0){
* client.setServer(mqtt\_broker, mqtt\_port);
* client.setCallback(callback);
* if(client.connect (device\_name,NULL,NULL)) {
* Serial.println ("Connected to MQTT Broker");
* }
* else {
* Serial.print("MQTT Broker connection failed");
* Serial.println (client.state());
* delay(200);
* }
* sub2topics();
* }
* }
* void load\_settings(){
* int count = 320000; // gives a few seconds to collect data on reset
* Serial.println(" .....GATHERING SETTINGS BEFORE INITIALIZATION..... ");
* while(count>0){
* client.loop();
* count -=1;
* }
* }
* void publishDouble(double sensor\_data,const char\* topic\_name, int precision) {
* double reading = sensor\_data;
* constexpr size\_t BUFFER\_SIZE = sizeof(double)+3; //1 char for the sign, 1 char for the decimal dot, 4 chars for the value & 1 char for null termination
* char buffer[BUFFER\_SIZE];
* dtostrf(reading, BUFFER\_SIZE - 1 /\*width, including the decimal dot and minus sign\*/, precision /\*precision\*/, buffer);
* client.publish(topic\_name, buffer, BUFFER\_SIZE); //notice we're using the overload where you specify the length of the buffer, as we know it and it saves a call to strlen
* }
* void callback(char\* topic, byte\* payload, unsigned int length) {
* //Serial.print(topic);
* //Serial.print(" ");
* String msg;
* int i=0;
* for (i;i<length;i++) {
* //Serial.print((char)payload[i]);
* msg += String((char)payload[i]);
* }
* //prints msg for all topics but datatype is float
* Serial.println(msg);
* //for topics that return float values
* //Serial.println(msg.toFloat());
* //delay(750);
* if (String(topic) == ("set/pin0")) {
* pinLat = gps.location.lat();
* pinLng = gps.location.lng();
* pinFlag = true;
* }
* else if (String(topic) == ("pinLng")) {
* pinLng = msg.toDouble();
* Serial.print("\n");
* Serial.print("Pinned Longitude: ");
* Serial.println(pinLng);
* Serial.print("\n");
* }
* else if (String(topic) == ("pinLat")) {
* pinLat = msg.toDouble();
* Serial.print("\n");
* Serial.print("Pinned Latitude: ");
* Serial.println(pinLat);
* Serial.print("\n");
* }
* }
* void setup()
* {
* delay(1000);
* //use serial with previous I2C read
* Serial.begin(115200);
* Serial.println("GPS Configuration");
* if (myI2CGPS.begin() == false)
* {
* Serial.println("Module failed to respond. Please check wiring.");
* while (1); //Freeze!
* }
* Serial.println("GPS module found!");
* //Configure to 115200 baud to then set 10hz read
* //FIGURE OUT HOW TO FIX THIS ERROR!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* //BREAK DOWN SPARKFUNS BEGIN SERIAL FUNCTION????
* //configString = myI2CGPS.createMTKpacket(251, ",115200");
* //myI2CGPS.sendMTKpacket(configString);
* //configString = myI2CGPS.createMTKpacket(220, ",100");
* //myI2CGPS.sendMTKpacket(configString);
* WiFi.begin(SSID,WIFI\_PW);//check for known network HOME
* delay(1000);
* Serial.println (" ");
* Serial.println ("Connecting to AP");
* checkConnection();
* Serial.println ("Connected to WiFi AP!!");
* Serial.print ("Got an IP address :");
* Serial.println (WiFi.localIP());
* checkBroker();
* load\_settings();
* //myI2CGPS.enableDebugging(); //Turn on printing of GPS strings
* myI2CGPS.disableDebugging(); //Turn off printing of GPS strings
* debug = false;
* /\* Initialise the sensor \*/
* if(!bno.begin())
* {
* /\* There was a problem detecting the BNO055 ... check your connections \*/
* Serial.print("Ooops, no BNO055 detected ... Check your wiring or I2C ADDR!");
* while(1);
* }
* delay(1000);
* bno.setExtCrystalUse(true);
* //pins for ultrasonic
* pinMode(trigPin, OUTPUT); // Sets the trigPin as an Output
* pinMode(echoPin, INPUT); // Sets the echoPin as an Input
* }
* void ultraSonic(){
* //ultrasonic sensor
* //signal acquisition from tutorial code at https://howtomechatronics.com/tutorials/arduino/ultrasonic-sensor-hc-sr04/
* //filtering was needed after using this method
* // Clears the trigPin
* digitalWrite(trigPin, LOW);
* delayMicroseconds(2);
* // Sets the trigPin on HIGH state for 10 micro seconds
* digitalWrite(trigPin, HIGH);
* delayMicroseconds(10);
* digitalWrite(trigPin, LOW);
* // Reads the echoPin, returns the sound wave travel time in microseconds
* duration = pulseIn(echoPin, HIGH);
* // Calculating the distance
* // Prints the distance on the Serial Monitor
* //Serial.print("Object Distance: ");
* //10-point data collection
* objDistance[9] = objDistance[8];
* objDistance[8] = objDistance[7];
* objDistance[7] = objDistance[6];
* objDistance[6] = objDistance[5];
* objDistance[5] = objDistance[4];
* objDistance[4] = objDistance[3];
* objDistance[3] = objDistance[2];
* objDistance[2] = objDistance[1];
* objDistance[1] = objDistance[0];
* objDistance[0] = duration \* 0.034 / 2;
* //filtering unwated values that run high when sensor is unsure filtered
* //trust based filter. Low confidence due to high variation or numbers greater than usable range. Prevents slow change in response and distortion from averaging filters
* //using standard deviation vs mean
* int distTemp = objDistance[0]; // prevents it from posting while taking average. Initialize with useful value
* //10 is used since false signals dont appear to propigate more than 6 times in a row
* int points = 7; // THIS SLOWS THE RESONIVENESS WITH MORE POINTS TO CONdSIDER... MAKES IT MORE "PICKY"
* for (int ii = 1; ii<points;ii++){//starts at 1 to initialize withuseful value
* distTemp += objDistance[ii];
* }
* distMean = distTemp/points;
* int dev = pow(objDistance[0]-distMean,2); // initialize with a useful value
* for (int ii = 1; ii<points; ii++){
* dev += pow(objDistance[ii]-distMean,2);
* }
* stdDev = pow(dev/points, 0.5);//calculate standard deviation
* //THIS SLOWS RESPONSIVENESS THE MOST, RAISE THRESHOLD IF TOO SLOW
* int thresh = distMean\*2;//fairly trusting
* if ((stdDev<thresh) && (objDistance[0]<900)){//note that values above 900 imply no signal returned so prevent storing above this value and just repeats previous known value if above 900
* objDist = objDistance[0];//use raw reading when std is not too high (low variation vs mean) cant use mean since it would drive value high or low with false readings
* }
* else {objDist = objDistOld;}//if std deviation is high versus the mean then only use old/trusted values
* }
* void IMU(){
* imu::Vector<3> mag = bno.getVector(Adafruit\_BNO055::VECTOR\_MAGNETOMETER);
* imu::Vector<3> acc = bno.getVector(Adafruit\_BNO055::VECTOR\_ACCELEROMETER);
* sensors\_event\_t event;
* bno.getEvent(&event);
* /\* Get a new sensor event \*/
* int averages = pow(2,14);
* for (int ii = 0; ii<averages;ii++){
* pitch += event.orientation.y;
* roll += event.orientation.z;
* }
* pitch /=averages;
* roll /=averages;
* float phiRad=roll/360\*(2\*M\_PI);
* float thetaRad=-pitch/360\*(2\*M\_PI);
* float magx,magy,magz;
* magx = mag.x();
* magy = mag.y();
* magz = mag.z();
* float Xm=magx\*cos(thetaRad) - mag.y()\*sin(phiRad)\*sin(thetaRad) + magz\*cos(phiRad)\*sin(thetaRad);
* float Ym=magy\*cos(phiRad) + magz\*sin(phiRad);
* //float yaw=atan2(Ym,Xm);//as radians for filtering!!!
* yaw=atan2(Ym,Xm)/(2\*3.14)\*360;
* //filter (CAUTOIN OF SLOWING UPDATE TO LOWER THAN RATE OF ROVER!!!)
* //float yaw\_new = 0.3\*(yaw) + 0.7\*(yaw\_old); // old will become more reliable than new data so new data has a low weighing on the filter
* //yaw\_old = yaw\_new; // uses filtered value to update old and not raw data
* tb = millis();
* float z\_vibe = ((acc.z()-9.8)\*1000); //keeps data sitting around 0
* float y\_vibe = ((acc.y())\*1000); //keeps data sitting around 0
* float x\_vibe = ((acc.x())\*1000); //keeps data sitting around 0
* // jerk=accel\*d/dt
* double dt = tb-ta/1000.00; // discrete time interval in seconds
* float dAz = z\_vibe - z\_vibeOld;// using absolute value to just find change in accel
* float dAy = y\_vibe - y\_vibeOld;// using absolute value to just find change in accel
* float dAx = x\_vibe - x\_vibeOld;// using absolute value to just find change in accel
* float jerkz = dAz/dt;// find change. z should always be + if right side up
* //jerkz \*= 1000; //use gain to magnify data for visualization on combined graph
* float jerky = dAy/dt;// find change. y should always be + if right side up
* //jerky \*= 1000; //use gain to magnify data for visualization on combined graph
* float jerkx = dAx/dt;// find change. x should always be + if right side up
* //jerkx \*= 1000; //use gain to magnify data for visualization on combined graph
* jerkMag = sqrt(pow(jerkz,2)+pow(jerky,2)+pow(jerkx,2));
* ta = tb;
* z\_vibeOld = z\_vibe;
* y\_vibeOld = y\_vibe;
* x\_vibeOld = x\_vibe;
* /\*
* //prevent small changes an constants from showing
* if (jerk<10){
* z\_vibe = 0; //use this threshold only if change in acceleration does not prove to be useful
* }
* \*/
* }
* void loop()
* {
* checkConnection();
* checkBroker();
* client.loop();
* IMU();
* ultraSonic();
* while (myI2CGPS.available()) //available() returns the number of new bytes available from the GPS module
* {
* gps.encode(myI2CGPS.read()); //Feed the GPS parser
* }
* if (gps.time.isUpdated()) //Check to see if new GPS info is available
* {
* displayInfo();
* }
* }
* void displayInfo(){
* //NEED TO ADD CALIBRATION FOR THE COMPASS TO WORK!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
* /\* Display the floating point data \*/
* Serial.print(pitch, 4);
* Serial.print(",");
* Serial.print(roll, 4);
* Serial.print(",");
* //Serial.print(Xm, 4);
* //Serial.print(",");
* //Serial.print(Ym, 4);
* //Serial.print(",");
* Serial.print(yaw, 4); // converts to degrees for display only
* //conditional ensures that hdop is less than 1 which should ensure <2.5m accuracy when reporting
* double hdop = gps.hdop.value()/100.0;
* if (pinFlag){
* publishDouble(pinLat, "pinLat",6);
* publishDouble(pinLng, "pinLng",6);
* //pinFlag is deactivated in initial distance calc if LPF is active
* pinFlag = false; //deactive reboot condition flag
* }
* double lat1 = pinLat;
* double lat2 = gps.location.lat();
* double long1 = pinLng;
* double long2 = gps.location.lng();
* /\*
* //slow rate of change in GPS but not slower than the rover would travel
* if(rebooted||pinFlag){
* //initial distance calc
* distance = 6378100.0 \* acos((sin(lat1) \* sin(lat2)) + cos(lat1) \* cos(lat2) \* cos(long2 - long1)); //convert to meters using cylindrical coordinates
* rebooted = false;//deactive reboot condition flag
* pinFlag = false; //deactive reboot condition flag
* }
* else {
* dist\_b = 6378100.0 \* acos((sin(lat1) \* sin(lat2)) + cos(lat1) \* cos(lat2) \* cos(long2 - long1)); //convert to meters using cylindrical coordinates
* distance = 0.99\*(distance) + 0.01\*(dist\_b); //"LPF" that slows rapid changes in position a setting of 0.9 and 0.1 at 1Hz means proper update time for object come to a halt is 10 seconds
* Serial.print("Distance from Pin: ");
* Serial.println(distance);
* publishDouble(distance, "distance",2);
* }
* \*/
* //deactivaate below if LPF is used and remove move pinFlag from publish condition
* //distance = 6378100.0 \* acos((sin(lat1) \* sin(lat2)) + cos(lat1) \* cos(lat2) \* cos(long2 - long1)); //convert to meters using cylindrical coordinates
* distance = TinyGPSPlus::distanceBetween(lat2, long2, lat1, long1);
* lat1 = pinLat\*(M\_PI/180);
* lat2 = lat2\*(M\_PI/180);
* long1 = pinLng\*(M\_PI/180);
* long2 = long2\*(M\_PI/180);
* //distance = 0.99\*(distance) + 0.01\*(dist\_b); //"LPF" that slows rapid changes in position a setting of 0.9 and 0.1 at 1Hz means proper update time for object come to a halt is 10 seconds
* //Serial.print("Distance from Pin: ");
* //Serial.println(distance);
* publishDouble(distance, "distance", 2);
* //Serial.println("HDOP: ");
* //Serial.println(hdop);
* /\*
* if (gps.time.hour() < 10) Serial.print("0");
* Serial.print(gps.time.hour());
* Serial.print(":");
* if (gps.time.minute() < 10) Serial.print("0");
* Serial.print(gps.time.minute());
* Serial.print(":");
* if (gps.time.second() < 10) Serial.print("0");
* Serial.print(gps.time.second());
* \*/
* //Serial.print(" Loc:");
* Serial.print(",");
* Serial.print(gps.location.lat(), 6);
* Serial.print(",");
* Serial.print(gps.location.lng(), 6);
* //Serial.print(" SIV:");
* Serial.print(",");
* Serial.print(gps.satellites.value());
* //Serial.print(" HDOP:");
* Serial.print(",");
* Serial.print(gps.hdop.value()/100.0, 2); //TinyGPS reports DOPs in 100ths
* //using raw acceleration data to obtain vibrational noise data
* //find the rate of change in vertical acceleration data to quantify vibrations
* //maybe do a for loop that counts multiple changes of a certain magnitude threshold before throwing errors to user or slowing the vehicle
* Serial.print(",");
* Serial.print(jerkMag\*1000);//Jerk (derivative of accel) is plotted to find density of change in acceleration. Multiple highs suggest rapid change in axial acceleration providing vibrational denisty
* publishDouble(hdop,"hdop",2);
* //From ultrasonic reading
* Serial.print(",");
* Serial.println(objDist);
* /\* DEBUG ULTRASONIC FILTERING
* Serial.println("STD: ");
* Serial.println(stdDev);
* Serial.println("Mean: ");
* Serial.println(distMean);
* Serial.println("Accepted Distance: ");
* Serial.println(objDist);
* delay(2000);
* \*/
* objDistOld = objDist;//retain used value in for use when redout is unreliable
* delay(100);
* }
* **Motor code**
* #include <mcp\_can.h>
* #include <SPI.h>
* long unsigned int rxId;
* // the cs pin of the version v1.4 is default to D10
* // Set CS pin
* #define P\_MIN -12.5f
* #define P\_MAX 12.5f
* #define V\_MIN -40.0f
* #define V\_MAX 40.0f
* #define KP\_MIN 0.0f
* #define KP\_MAX 500.0f
* #define KD\_MIN 0.0f
* #define KD\_MAX 5.0f
* #define T\_MIN -15.0f
* #define T\_MAX 15.0f
* // Set Values
* float p\_in= 0.0f;
* float v\_in= 0.0f;
* float kp\_in= 0.00f;
* float kd\_in= 0.01f;
* float t\_in= 0.0f;
* //measured values - responses from the motor
* float p\_out = 0.0f;  // actual position
* float v\_out = 0.0f;  // actual velocity
* float t\_out = 0.0f;  // actual torque
* const int SPI\_CS\_PIN =5;      //pin 5 for esp32 nodemcu
* const int start = 33; //pin used for on button
* const int stop = 27; //pin used for off button
* bool onButton = 0;
* bool offButton = 0;
* bool onFlag = false;
* MCP\_CAN CAN(SPI\_CS\_PIN);               //Set CS pin
* //initialize adc potentiometer
* adc\_attenuation\_t att = ADC\_11db;
* #define adc\_pin   35
* #define rezESP32  8 //bitwidth of adc reading. Reduced bitwidth (12->8) as 256bits in hopes to stabilize reading
* int vel\_old = 0.0;
* void setup() {
* // put your setup code here, to run once:
* Serial.begin(115200);
* delay(1000);
* while(CAN\_OK !=CAN.begin( CAN\_1000KBPS)){
* Serial.println("CAN BUS Shield init fail");
* Serial.println("Init CAN BUS Shield again");
* delay(1000);
* }
* Serial.println("CAN BUS Shield init ok!");
* delay(100);
* //Zero();
* //ExitMotorMode();
* pinMode(start,INPUT);
* pinMode(stop,INPUT);
* analogReadResolution(rezESP32);
* analogSetAttenuation(att);//ADC\_0db,ADC\_2\_5db,ADC\_6db,ADC\_11db
* adcAttachPin(adc\_pin);//ADC GPIO STARTS AT 3 FOR ESP32-S3
* delay(10);
* }
* long previousMillis =0;
* boolean newData = false;
* void loop() {
* //note that a pulldown is used to hold zero state
* //WILL TOGGLE ON/OFF IF LOGIC PIN NOT CONNECTED!!
* onButton = digitalRead(start);
* if (onButton){
* Serial.println("Entering MIT Mode");
* EnterMotorMode();
* delay(500);//prevent double press
* onButton = 0;
* onFlag = true;
* }
* offButton = digitalRead(stop);
* if (offButton){
* Serial.println("Exiting Motor Mode");
* v\_in = 0; //send velocity to 0 before exiting
* pack\_cmd();
* delay(1000); //wait for motor to stop
* ExitMotorMode();
* delay(500);//prevent double press
* offButton = 0;
* onFlag = false;
* }
* char rc;
* float vel = potentCMD(); // read from potentiometer
* if (/\*int(floor(vel) != vel\_old)  &&\*/ onFlag){ //onflag prevents v\_in from updating when not in motor mode
* vel\_old = int(floor(vel)); // update vel\_old here before changing vel in conditionals below
* v\_in = vel\_old; //sends to global for pack\_cmd()
* //use these prints for serial monitor
* //Serial.print("Vel Input: ");
* //Serial.print(int(vel));
* //Serial.println(v\_in);
* pack\_cmd();
* }
* // receive CAN
* if(CAN\_MSGAVAIL == CAN.checkReceive()){ // check if data coming{
* unpack\_reply();
* }
* //Serial.print(millis()-previousMillis);
* previousMillis=millis();
* delay(100);
* }
* void EnterMotorMode(){
* byte buf[8];
* buf[0]=0xFF;
* buf[1]=0xFF;
* buf[2]=0xFF;
* buf[3]=0xFF;
* buf[4]=0xFF;
* buf[5]=0xFF;
* buf[6]=0xFF;
* buf[7]=0xFC;
* CAN.sendMsgBuf(0x05,0,8, buf);
* }
* void ExitMotorMode(){
* byte buf[8];
* buf[0]=0xFF;
* buf[1]=0xFF;
* buf[2]=0xFF;
* buf[3]=0xFF;
* buf[4]=0xFF;
* buf[5]=0xFF;
* buf[6]=0xFF;
* buf[7]=0xFD;
* CAN.sendMsgBuf(0x05,0,8, buf);
* }
* void Zero(){
* byte buf[8];
* buf[0]=0xFF;
* buf[1]=0xFF;
* buf[2]=0xFF;
* buf[3]=0xFF;
* buf[4]=0xFF;
* buf[5]=0xFF;
* buf[6]=0xFF;
* buf[7]=0xFE;
* CAN.sendMsgBuf(0x05,0,8, buf);
* }
* void pack\_cmd(){
* byte buf[8];
* //Serial.print("Velocity Sent: ");
* //Serial.println(v\_in);
* // limit data to  be within bounds ///
* float p\_des= constrain(p\_in,P\_MIN,P\_MAX);
* float v\_des= constrain(v\_in,V\_MIN,V\_MAX);
* float kp= constrain(kp\_in,KP\_MIN,KP\_MAX);
* float kd= constrain(kd\_in,KD\_MIN,KD\_MAX);
* float t\_ff= constrain(t\_in,T\_MIN,T\_MAX);
* unsigned int p\_int = float\_to\_uint(p\_des,P\_MIN,P\_MAX,16);
* unsigned int v\_int = float\_to\_uint(v\_des,V\_MIN,V\_MAX,12);
* unsigned int kp\_int = float\_to\_uint(kp,KP\_MIN,KP\_MAX,12);
* unsigned int kd\_int = float\_to\_uint(kd,KD\_MIN,KD\_MAX,12);
* unsigned int t\_int = float\_to\_uint(t\_ff,T\_MIN,T\_MAX,12);
* //pack ints into can buffer ///
* buf[0]=p\_int >>8;
* buf[1]=p\_int & 0xFF;
* buf[2]=v\_int >> 4;
* buf[3]=((v\_int&0xF)<< 4)| (kp\_int >>8);
* buf[4]= kp\_int &0xFF;
* buf[5]= kd\_int >>4;
* buf[6]= ((kd\_int&0xF)<< 4)| (t\_int >>8);
* buf[7]= t\_int &0xFF;
* CAN.sendMsgBuf(0x05,0,8,buf);

* }
* void unpack\_reply(){
* byte len=0;
* byte buf[8];
* CAN.readMsgBuf(&len,buf); //&rxId was added from mcp\_can example
* unsigned long canId=rxId; //changed from CAN.getCanId()
* ///unpack ints from can buffer///
* unsigned int id = buf[0];
* unsigned int p\_int = (buf[1]<<8)| buf[2];
* unsigned int v\_int =(buf[3]<<4)|(buf[4]>>4);
* unsigned int i\_int = ((buf[4]&0xF )<<8)| buf[5];
* /// convert uints to floats ///
* p\_out = uint\_to\_float(p\_int,P\_MIN,P\_MAX, 16);
* v\_out = uint\_to\_float(v\_int,V\_MIN,V\_MAX, 12);
* t\_out = uint\_to\_float(i\_int,T\_MIN,T\_MAX, 12);
* Serial.print("position: ");
* Serial.println(p\_out);
* Serial.print("velocity: ");
* Serial.println(v\_out);
* Serial.print("torque: ");
* Serial.println(t\_out);
* Serial.println("");
* }
* unsigned int float\_to\_uint(float x, float x\_min, float x\_max, float bits){
* //convert a float to an unsigned int, given range and number of bits  ///
* float span = x\_max-x\_min;
* float offset = x\_min;
* unsigned int pgg=0;
* if(bits ==12){
* pgg= (unsigned int)((x-offset)\*4095.0/span);
* }
* if(bits ==16){
* pgg= (unsigned int)((x-offset)\*65535.0/span);
* }
* return pgg;
* }
* float potentCMD(){
* float adc\_readout =0.0;
* int averages = 1024;
* //average for reading stabilizaiton
* for (int ii =0; ii<averages;ii++){
* //accumulate readings
* adc\_readout += analogRead(adc\_pin);
* }
* //average the accumulation
* adc\_readout /= averages;
* //shift to make middle the zero point
* adc\_readout -= (pow(2,rezESP32)/2);
* //normalize
* adc\_readout /= (pow(2,rezESP32)/2);
* //expand to max and min of motor controller at 30 or 50 depending
* adc\_readout \*= V\_MAX;
* //Serial.print(F("Vel Input: "));
* //Serial.println(adc\_readout);
* //delay(500);
* return adc\_readout;
* }
* float uint\_to\_float(unsigned int x\_int, float x\_min,float x\_max, int bits){
* /// converts unsined int to float, given range and numbers of bits///
* float span = x\_max - x\_min;
* float offset = x\_min;
* float pgg=0;
* if (bits ==12){
* pgg=((float)x\_int)\*span/4095.0 + offset;
* }
* if (bits ==16){
* pgg=((float)x\_int)\*span/65535.0 + offset;
* }
* return pgg;
* }