Mobility Chair

Jacob Rey

```
/*  Version Notes:

     Features Implemented:
      PID control and encoder feedback of two motors
      8 Directional Control (orthogonal and diagonals)
      Inputs are still sketched as physical buttons
      Therapist Override Switch
      LED outputs for control indicator/feedback
      3 Mode Speed Selection
      Battery Voltage sensing / low battery indicator
      Alarms (blinks of different duration for different meanings for:
      - Gyroscope / Tilt Detection
      - IR distance sensing
      - Proximity of Enclosure
      - Audible Feedback for motion

    Features Not Yet Implemented:
      - Seat presence / weight detection

    Bugs Found/Fixed:
      - N/A

*/

/*
  Code library for the MPU 9250 IMU was sourced from:
    Advanced_I2C.ino
    Brian R Taylor
    brian.taylor@bolderflight.com
    Copyright (c) 2017 Bolder Flight Systems
    https://github.com/bolderflight/MPU9250

    Permission is hereby granted, free of charge, to any person obtaining a copy of this software
    and associated documentation files (the "Software"), to deal in the Software without restriction,
    including without limitation the rights to use, copy, modify, merge, publish, distribute,
    sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is
    furnished to do so, subject to the following conditions:

    The above copyright notice and this permission notice shall be included in all copies or
    substantial portions of the Software.

    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING
    BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
    NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM,
    DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
```

```
//INCLUDE CODE LIBRARY////////////////////////////////////////////////////////////////////////////////////
  //Include the library for the MPU9250 and setup the accelerometer variables
    #include "MPU9250.h"
  //Setup variables for inertial measurment unit
    float Xdeg = 0;
    float Ydeg = 0;
    float Zdeg = 0;
    float Xrad = 0;
    float Yrad = 0;
    float Zmss = 0;
    float temp = 0;
    float Zmss_limit = 8.80;        // Limit of the tilt in the Z axis before shutoff
    const byte temp_limit = 49;     // Limit of the temperature in °C before shutoff (~120°F)
  //Setup I2C Comms for MPU9250 object on bus 0 with address 0x68
    MPU9250 IMU(Wire,0x68);     //NOTE: This sensor is physically connected to pins 20 and 21 (SDA and SCL).  It also requires 3.3V and GND.
    int status;

//PINOUT DEFINITIONS/////////////////////////////////////////////////////////////////////////////////////
//Control Inputs
const byte forwardButtonPin = 5;                    //Set the pin used for the input button
const byte leftButtonPin = 6;
const byte rightButtonPin = 7;
const byte reverseButtonPin = 8;
/* The direction inputs will all eventually be replaced by the I2C communication of the capacitive touch*/
const byte mode1Pin = 32;                           //Used for selecting mode 1 on rotary switch  (switch dim 3 to 4, also 7 to 8)
const byte mode2Pin = 33;                           //Used for selecting mode 2 on rotary switch  (switch dim 1 to 2, also 5 to 6)
const byte mode3Pin = 34;                           //Used for selecting mode 3 on rotary switch
//note, program needs to be simplified, switch only has 2 positions
const byte enableButtonPin = 2;                     //Used for button that enables motion control (controlled by supervisor)
//Status Indicators (Outputs)
const byte systemEnabledIndPin = 30;                //Used for LED lights that indicate motion control is activated
const byte alarmIndPin = 35;                        //Used for LED lights that indicate a low battery level
const byte leftIndPin = 26;                         //Used for LED lights that indicate left has been pressed
const byte rightIndPin = 27;                        //Used for LED lights that indicate right has been pressed
const byte forwardIndPin = 28;                      //Used for LED lights that indicate forward has been pressed
const byte reverseIndPin = 29;                      //Used for LED lights that indicate reverse has been pressed
const byte audioIndPin = 51;                        //Reserved for audio feedback
//Motor Outputs
const byte leftMotorPWMPin = 4;                     //Used for left motor PWM constrol signal
const byte leftMtrDirPin = 25;                      //Pin used to signal rotation of left motor
const byte rightMotorPWMPin = 3;                    //Used for right motor PWM constrol signal
const byte rightMtrDirPin = 24;                     //Pin used to signal rotation of left motor
```

```
//Encoder Inputs
const byte encoderLeftAPin = 14;                    //Encoder channel A for Left Motor - Note: This will be accessed by the interrupt only
const byte encoderLeftBPin = 15;                    //Encoder channel B for Left Motor
const byte encoderRightAPin = 16;                   //Encoder channel A for Right Motor
const byte encoderRightBPin = 17;                   //Encoder channel B for Right Motor
//Sensor Inputs
const byte sclPin = 21;                             //Used for the gyroscope and the capacitive touch
const byte sdaPin = 20;                             //Used for the gyroscope and the capacitive touch
const byte distIR1Pin = A0;                         //Used for the IR sensor
const byte distIR2Pin = A1;                         //Used for the second IR sensor
const byte proxEncPin = A3;                         //Used for the prox sensor that monitors the enclosure
const byte seatSensePin = A4;                       //Used for the sensor that detects weight of seat/passenger
const byte batteryVoltagePin = A5;                  //Used for the voltage sensor that detects low battery level
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//HARDWARE RELATED CONSTANTS/////////////////////////////////////////////////////////////////////////////
const int encoderResolution = 360;                  //Number of pulses for a single resolution (used to calculate RPM for troubleshooting/reference)
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//STATUS VARIABLES///////////////////////////////////////////////////////////////////////////////////////
//Sum of All Control Input Variables
byte controlState_crnt = 0;                             //Sum of all the control states, used to detect any change in input
byte controlState_prev = 0;
//Values for individual controls
boolean fwdBtnState_crnt = 1;                           // current state of the forward button
boolean fwdBtnState_prev = 1;                           // previous state of the forward button
boolean leftBtnState_crnt = 1;                          // These are all set to 1 by default because they are pullup inputs (1 is off)
boolean leftBtnState_prev = 1;
boolean rightBtnState_crnt = 1;
boolean rightBtnState_prev = 1;
boolean revBtnState_crnt = 1;
boolean revBtnState_prev = 1;
//Used in StopOverride
boolean systemEnableState = HIGH;                         // Used to track the state of the supervisor override button
volatile boolean enableBtnState_crnt = LOW;              // Used to toggle the state of the enable button
boolean enableBtnState_prev = LOW;                       // Used to toggle the state of the enable button
volatile unsigned long enableTime_crnt = 0;              // Used to debouce the enable button based on timing (compare current to previous to see duration in between)
unsigned long enableTime_prev = 0;                       // Used to debouce the
```

```
enable button based on timing
//Battery Level Indication and Constants
const int lowBatteryLevel = 125;                         // This should be adjusted to the lowest acceptable battery voltage level x100 (2200 estimated for 22V)
unsigned long lowBatteryTime_crnt = 0;
unsigned long lowBatteryTime_prev = 0;

//Enclosure Status Indicators
volatile boolean enclosureState_crnt = 0;                // Used by the falling interrupt created by prox sensor to detect enclosure lid open/close
volatile boolean enclosureState_prev = 0;
volatile unsigned long enclosureTime_crnt = 0;          //Used to debounce the enclosure prox sensor - bounce happens when voltage drops upon motor start
unsigned long enclosureTime_prev = 0;
//IR Distance Constant
const int IRdistMAX = 15;
//General Alarm Status Variables
byte alarmStatus = 0;

//Motor and Encoder Status Variables
boolean stateEncoderL_B = 0;                           // State of the quatrature channel for Left motor encoder (used to determin direction of rotation)
boolean stateEncoderR_B = 0;                           // State of the quatrature channel for Left motor encoder (used to determin direction of rotation)
volatile unsigned int encoderLeftValue_counter = 0;    //The current count of the left encoder (updated instantly by interrupt)
volatile unsigned int encoderRightValue_counter = 0;   //The current count of the right encoder (updated instantly by interrupt)
int encoderLeftValue_crnt = 0;                         //Count of left encoder after 50ms (updated on time basis)
int encoderLeftValue_prev = 0;                         //Last count of the left encoder
String encoderLeftDir_txt = "";
int encoderRightValue_crnt = 0;                        //Count of left encoder after 50ms (updated on time basis)
int encoderRightValue_prev = 0;                        //Last count of the left encoder
String encoderRightDir_txt = "";
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//TIMING VARIABLES///////////////////////////////////////////////////////////////////////////////////////
//time variables must use long type because they will overflow if other datatypes are used
unsigned long fwdPressedTime = 0;     // the moment when fwrd button was pressed
unsigned long fwdReleasedTime = 0;    // the moment when fwd button was released
unsigned long fwdTimeActive = 0;
unsigned long fwdTimeInactive = 0;

unsigned long leftPressedTime = 0;
unsigned long leftReleasedTime = 0;
unsigned long leftTimeActive = 0;
unsigned long leftTimeInactive = 0;

unsigned long rightPressedTime = 0;
unsigned long rightReleasedTime = 0;
unsigned long rightTimeActive = 0;
unsigned long rightTimeInactive = 0;

unsigned long revPressedTime = 0;
unsigned long revReleasedTime = 0;
unsigned long revTimeActive = 0;
unsigned long revTimeInactive = 0;

//System timers
unsigned long timeHeld = 0;           // the duration the button has currently been held (while active)
unsigned long timeReleased = 0;       // the duration of time since the button was released (while inactive)
unsigned long timeCurrent = 0;        // the length of time the button has been held
unsigned long shortTimer = 0;
unsigned long mediumTimer = 0;
unsigned long longTimer = 0;

//Warning Indicator timers
byte warningState = 0;                  // Set to an integer value by the various sensors to indicate which alarm is active
byte warningShort = 100;                // The off time of the alarm blink
byte warningLong = 300;                 // The on time of the alarm blink
unsigned long warningTimer_crnt = 0;
unsigned long warningTimer_prev = 0;

//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//TIMING CONSTANTS////////////////////////////////////////////////////////////////////////////////////////
/*Timers used to determine how often different subroutines are executed*/
const int shortInterval = 30;         // Refresh rate for items checked often, in ms (note if changed from 50 array values need to be recalculated)
const int mediumInterval = 50;
const int longInterval = 250;         // Refresh rate for items checked less often
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//MOTOR SPEED and DIRECTION VARIABLES///////////////////////////////////////////////////////////////////
int leftTargetSpeed_crnt = 0;                    // This value is dynamically adjusted based on the time the control inputs have been held/released
int leftTargetSpeed_prev = 0;
int rightTargetSpeed_crnt = 0;                   // This value is dynamically adjusted based on the time the control inputs have been held/released
int rightTargetSpeed_prev = 0;

const byte mode1Factor = 1;                     //Full Speed
const float mode2Factor = 0.67;                 //Medium Speed (default)
const float mode3Factor = 0.50;                 //Slow Speed
float modeFactorFinal = 1.00;
boolean mode1Status = 1;                        //Set to 1 by default to indicate "off" (representing pullup inputs)
boolean mode2Status = 1;
boolean mode3Status = 1;

//LeftMotor
//String leftMotorDirection = "";       //Used for troubleshooting detected rotation
//String rightMotorDirection = "";      //Used for troubleshooting detected rotation
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//MOTOR SPEED RELATED
```

```
CONSTANTS//////////////////////////////////////////////////////////////////////////
const int maxSpeed = 243;              //Maximum target speed in pulses per 50ms (based on expected pulses from encoder)
const int accelTime = 4000;            //Target time to reach maximum speed in ms (controls rate of acceleration - note, if this changes array must be recreated)
const int decelTime = 900;             //Target time to reach a stop in ms (rate of deceleration of the motor)
const int AccelSpeedValues[80] = {1, 1, 1, 1, 1, 2, 2, 2, 3, 4, 4, 6, 7, 9, 10, 12, 14, 17, 20, 22, 26, 29, 33, 36, 41, 45, 49, 54, 59, 64, 69, 75, 80, 86, 92, 98, 103, 109, 116, 122, 128, 134, 140, 146, 151, 157, 163, 168, 174, 179, 184, 189, 194, 198, 203, 207, 210, 214, 217, 221, 224, 226, 229, 231, 233, 235, 236, 237, 239, 240, 240, 241, 242, 242, 242, 242, 243, 243, 243, 243};
const byte AccelArrayCount = 80;       //Number of values contained in AccelSpeedValues[]
const int DecelSpeedValues[18] = {238, 232, 221, 207, 190, 169, 146, 122, 98, 75, 54, 37, 23, 12, 6, 2, 1, 0};
const byte DecelArrayCount = 18;       //Number of values contained in DecelSpeedValues[]
const byte revFactor = 2;              //This controls the speed of reverse relative to forward (divide by this value)
const byte diagTurnFactor1 = 5;        //These constants control the tightness of a diagonal turning radius
const byte diagTurnFactor2 = 2;        //Multiply by factor1, divide by factor 2 to achieve rounded factors (ex2.5)
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//PID CONTROL CONSTANTS///////////////////////////////////////////////////////////////////////////////////
float kp = 1.25; //These values will need some tuning
float ki = 0.01;
float kd = 0.1;
float PID_Lp, PID_Li, PID_Ld;
int PID_Ltotal = 0;
float PID_Rp, PID_Ri, PID_Rd;
int PID_Rtotal = 0;
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

//PID CONTROL VARIABLES///////////////////////////////////////////////////////////////////////////////////
int errorLeft_crnt;
int errorLeft_prev;
int errorRight_crnt;
int errorRight_prev;
int errorBetween_crnt;
int errorBetween_prev;
//\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\


//*************BEGIN PROGRAM EXECUTION********************************************************************
//********************************************************************************************************
//THIS SETS UP THE INPUT AND OUTPUTS AND SERIAL MONITOR COMMUNICATION
void setup() {
  //Start the serial monitor for troubleshooting and displaying status
  Serial.begin(115200);        // initialize serial communication
  //Serial.end();            // Uncomment to turn off serial comms
  Serial.println("Mobility Chair Startup Routine");

  //Start the I2C communication for the MPU9250
  while(!Serial) {}
    //start communication with IMU
    status = IMU.begin();                       // NOTE: The sensor zeros its orientation upon initialization.
    if (status < 0) {
      Serial.println("IMU initialization unsuccessful");
      Serial.println("Check IMU wiring or try cycling power");
      Serial.print("Status: ");
      Serial.println(status);
      while(1) {}
    }
    // setting the accelerometer full scale range to +/-8G
    IMU.setAccelRange(MPU9250::ACCEL_RANGE_8G);
    // setting the gyroscope full scale range to +/-500 deg/s
    IMU.setGyroRange(MPU9250::GYRO_RANGE_500DPS);
    // setting DLPF bandwidth to 20 Hz
    IMU.setDlpfBandwidth(MPU9250::DLPF_BANDWIDTH_20HZ);
    // setting SRD to 19 for a 50 Hz update rate
    IMU.setSrd(19);
    Serial.println("IMU initialization success");
    Serial.println();
  //End MPU9250 Setup

  //Initialize the control input buttons
  pinMode(forwardButtonPin, INPUT_PULLUP);    // initialize the button pin as a input
  pinMode(leftButtonPin, INPUT_PULLUP);       // initialize the button pin as a input
  pinMode(rightButtonPin, INPUT_PULLUP);      // initialize the button pin as a input
  pinMode(reverseButtonPin, INPUT_PULLUP);    // initialize the button pin as a input
  pinMode(enableButtonPin, INPUT_PULLUP);

  //Initialize the mode selection inputs (rotary switch)
  pinMode(mode1Pin, INPUT_PULLUP);            //Used to set mode 1 (full speed)
  pinMode(mode2Pin, INPUT_PULLUP);            //Used to set mode 2 (medium speed)
  pinMode(mode3Pin, INPUT_PULLUP);            //Used to set mode 3 (slow speed)

  //Initialize the outputs for the motor controller
  pinMode(leftMotorPWMPin, OUTPUT);           //Used to control the speed of the left motor
  pinMode(leftMtrDirPin, OUTPUT);             //Used to control the direction of the left motor
  pinMode(rightMotorPWMPin, OUTPUT);          //Used to control the speed of the right motor
  pinMode(rightMtrDirPin, OUTPUT);            //Used to control the direction of the right motor

  //Initialize the inputs for the motor encoders
  pinMode(encoderLeftAPin, INPUT_PULLUP);     //Used to count pulses on the left encoder main channel (uses interrupt)
  pinMode(encoderLeftBPin, INPUT_PULLUP);     //Used to detect state of left encoder quadrature channel to determine direction
  pinMode(encoderRightAPin, INPUT_PULLUP);    //Used to count pulses on the right encoder main channel (uses interrupt)
  pinMode(encoderRightBPin, INPUT_PULLUP);    //Used to detect state of right encoder quadrature channel to determine direction

  //Initialize the system indicators outputs (lights/warnings)
  pinMode(systemEnabledIndPin, OUTPUT);
  pinMode(alarmIndPin, OUTPUT);
  pinMode(leftIndPin, OUTPUT);
  pinMode(rightIndPin, OUTPUT);
  pinMode(forwardIndPin, OUTPUT);
  pinMode(reverseIndPin, OUTPUT);
  pinMode(audioIndPin, OUTPUT);

  //Initialize the sensor related inputs
  pinMode(batteryVoltagePin, INPUT);
  pinMode(proxEncPin, INPUT);
  pinMode(distIR1Pin, INPUT);

  //SETUP INTERRUPTS
  attachInterrupt(digitalPinToInterrupt(encoderLeftAPin), readLeftMtrSpd, RISING);    //Whenever the Left encoder has a rising pulse call the routine
  attachInterrupt(digitalPinToInterrupt(encoderRightAPin), readRightMtrSpd, RISING);    //Whenever the Right encoder has a rising pulse call the routine
  attachInterrupt(digitalPinToInterrupt(enableButtonPin), StopOverrideISR, FALLING);    //Used for the therapist/supervisor enable/disable pushbutton momentary switch
  attachInterrupt(digitalPinToInterrupt(proxEncPin), enclosureSensorISR, CHANGE);    //Used to detect a change in the prox sensor that monitors the enclosure (was falling)

  //Flash some lights so that you know the system started successfully.
  startupFlash();

}
void startupFlash() {
  //Flash the control lights so that the user knows the system has been turned on.
  digitalWrite(leftIndPin, HIGH);
  delay(250);
  digitalWrite(forwardIndPin, HIGH);
  delay(250);
  digitalWrite(rightIndPin, HIGH);
  delay(250);
  digitalWrite(reverseIndPin, HIGH);
  delay(750);
  digitalWrite(leftIndPin, LOW);
  digitalWrite(rightIndPin, LOW);
  digitalWrite(forwardIndPin, LOW);
  digitalWrite(reverseIndPin, LOW);
}

/*-----------------------------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
/*--MAIN PROGRAM LOOP---MAIN PROGRAM LOOP---MAIN PROGRAM LOOP---MAIN PROGRAM LOOP---MAIN PROGRAM LOOP---MAIN PROGRAM LOOP------------------*/
//THIS IS THE MAIN PROGRAM LOOP (it calls all the subroutines)
void loop() {
  if (enableBtnState_crnt != enableBtnState_prev) {         // Check if a change has occurred to the enable/stop supervisor button
    StopOverrideHandler();                                  // If a change has occurred, go to the handler subroutine to enable or disable the system
  }

  if (enclosureState_crnt != enclosureState_prev) {
    enclosureProxHandler();
  }

  if (alarmStatus > 0) {
    systemEnableState = LOW;                                //Turn off the controls
```

```
if they are on
    digitalWrite(systemEnabledIndPin, systemEnableState);   //Turn off the indicator for the controls
    warningIndicator();                                     //Toggle the warning LED the appropriate number of times to indicate error

    switch (alarmStatus) {                                  //Print the cause of the alarm to the serial monitor
      case 1:
        Serial.println("BATTERY ALARM - EC01");
        break;
      case 2:
        Serial.println("ENCLOSURE ALARM - EC02");
        break;
      case 3:
        Serial.println("DROPOFF ALARM - EC03");
        break;
      case 4:
        Serial.println("TILT ALARM - EC04");
        break;
      case 5:
        Serial.println("TEMPERATURE ALARM - EC05");
        break;
    }
  }

  if (systemEnableState == HIGH) {                              // Only allow the controls to be used if system is enabled
    Serial.println("CONTROLS ENABLED");                         // Print to the serial monitor for troubleshooting
    controlLoop();                                              // Go to subroutine to update the control inputs as they occur

    //Update these functions only every 30ms
    if (millis() > shortTimer + shortInterval) {
      shortTimer = millis();                                  //Update the system timer every 30ms
      updateTimerValue();                                     //Duration controls pressed or released
      calcMtrSpd();                                           //Read the actual encoder speed value
      pidLoop();                                              //Motor PID control
    }

    //Update these functions every 50ms
    if (millis() > mediumTimer + mediumInterval) {
      mediumTimer = millis();                                 //Update the system timer every 50ms.
      updateTargetSpeed();                                    //Set the target for the PID control based on current speed and duration of control inputs.
      IRdistance();                                           //Monitor for sudden increase in distance that indicates a ledge is present.
      //Serial.println("MED INTERVAL");
    }

  } else {
    digitalWrite(leftMotorPWMPin, 0);                           // The system is disabled, turn off both motors
    digitalWrite(rightMotorPWMPin, 0);
    leftTargetSpeed_crnt = 0;                //Ensure that the target system speed is reset if it was interrupted while in motion
    rightTargetSpeed_crnt = 0;
    digitalWrite(forwardIndPin, LOW);         //Turn off the control LED indicator for visual feedback
    digitalWrite(leftIndPin, LOW);            //Turn off the control LED indicator
```

```
for visual feedback
    digitalWrite(rightIndPin, LOW);            //Turn off the control LED indicator for visual feedback
    digitalWrite(reverseIndPin, LOW);            //Turn off the control LED indicator for visual feedback
    digitalWrite(audioIndPin, LOW);          //Send a signal too the UNO board to turn off audio feedback
    Serial.println("SYSTEM DISABLED");                          // Print to the serial monitor for troubleshooting.
    Serial.println();
  }

  //Update these functions every 250ms
  if (millis() > longTimer + longInterval) {
    longTimer = millis();                                       // Update the system timer every 250ms.  Used for functions that do not need instant response.
    modeSelect();                                               // Determine which speed mode is active.
    batteryMonitor();                                           // Monitor the status of the battery for charge
    tiltSensor();                                               // Monitor tilt in the Z axis.  This sensor is setup to refresh at a maximum of 20Hz or 0.05s.
    //Serial.println("LONG INTERVAL");
  }
}
/*-ISR-------------------------------------------------------------------------------------------------------------------------------------*/
/*-----ISR---------------------------------------------------------------------------------------------------------------------------------*/
/*---------ISR-----------------------------------------------------------------------------------------------------------------------------*/
/*INTERRUPT SERVICE FUNCTIONS HERE*/
//These functions determine the encoder counts and handle the therapist override switch

//Interrupt Routine for reading the left encoder
void readLeftMtrSpd() {
  // Increment value for each pulse from encoder, this is triggered by an interrupt
  encoderLeftValue_counter++;
  stateEncoderL_B = digitalRead(encoderLeftBPin);
}

//Interrupt Routine for reading the right encoder
void readRightMtrSpd() {
  // Increment value for each pulse from encoder, this is triggered by an interrupt
  encoderRightValue_counter++;
  stateEncoderR_B = digitalRead(encoderRightBPin);
}

//Interrupt Routine for the therapist stop button
void StopOverrideISR() {
  enableTime_crnt = millis();                       //Record the time the stop/enable button was pressed
  enableBtnState_crnt = !enableBtnState_crnt;
}

//Interrupt Routine for the proximity sensor checking the enclosure lid
void enclosureSensorISR() {
  enclosureTime_crnt = millis();
  enclosureState_crnt = !enclosureState_crnt;
  //Serial.println("ENCLOSURE STATE CHANGE");
  /*for (byte i = 0; i < 10; i++) {
    Serial.println("ENCLOSURE STATE CHANGED ΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔΔ");
  }
  */
}
/*-CONTROL LOOP----------------------------------------------------------------------------------------------------------------------------*/
/*-----CONTROL LOOP------------------------------------------------------------------------------------------------------------------------*/
/*---------CONTROL LOOP--------------------------------------------------------------------------------------------------------------------*/
//This function reads the control inputs
void controlLoop() {
  //Not timer dependent

  //Read the control input pins
  fwdBtnState_crnt = digitalRead(forwardButtonPin);         // read the forward button input
  leftBtnState_crnt = digitalRead(leftButtonPin);           // read the left button input
  rightBtnState_crnt = digitalRead(rightButtonPin);         // read the right button input
  revBtnState_crnt = digitalRead(reverseButtonPin);         // read the reverse button input

  controlState_crnt = fwdBtnState_crnt + leftBtnState_crnt + rightBtnState_crnt + revBtnState_crnt;

  //If a change in the control has occurred (first time pressed or released), keep track of the time
  /*NOTES: May be able to streamline this with math instead of reading each state*/
  if (controlState_crnt != controlState_prev) {
    updateStartStopTimes();
  }

  fwdBtnState_prev = fwdBtnState_crnt;                  // save fwdbtn state for next loop so that a change can be detected each iteration
  leftBtnState_prev = leftBtnState_crnt;
  rightBtnState_prev = rightBtnState_crnt;
  revBtnState_prev = revBtnState_crnt;
  controlState_prev = controlState_crnt;

  //Write the motor controller values

  //Left Motor Values
  byte leftMotorPWM_value = abs(PID_Ltotal);           //PWM value can only be positive (but direction is controlled by sign)
  if (PID_Ltotal > 0) {                                 //Set the motor direction based on sign of PWM value
    digitalWrite(leftMtrDirPin, HIGH);                  //Set the motor direction
  } else {
    digitalWrite(leftMtrDirPin, LOW);
  }
  analogWrite(leftMotorPWMPin, leftMotorPWM_value);      //Set the motor speed based on the target calculated (first pass is zero)

  //Right Motor Values
  byte rightMotorPWM_value = abs(PID_Rtotal);
  if (PID_Rtotal > 0) {                                 //Set the motor direction based on sign of PWM value
    digitalWrite(rightMtrDirPin, HIGH);                  //Set the motor direction
  } else {
    digitalWrite(rightMtrDirPin, LOW);
  }
  analogWrite(rightMotorPWMPin, rightMotorPWM_value);      //Set the motor speed based on the target calculated (first pass is zero)
}
/*-MODE SELECTION------------------------------------------------------------------------------------------------------------------------*/
/*----------------MODE SELECTION---------------------------------------------------------------------------------------------------------*/
/*-------------------------------MODE SELECTION------------------------------------------------------------------------------------------*/
void modeSelect() {
  //Timer dependent because it is not expected to change frequently
  //Mode inputs will be pullup so LOW will indicate the selection
  mode1Status = digitalRead(mode1Pin);    //Slow speed
  //mode2Status = digitalRead(mode2Pin);  //Medium Speed (This pin was not physically implemented because the switch is only 2 position.)
  mode3Status = digitalRead(mode3Pin);    //Full Speed

  if (mode1Status == LOW) {
    modeFactorFinal = mode3Factor;
    Serial.println("MODE 3 - Slow Speed");
  }
  else if (mode3Status == LOW) {
    modeFactorFinal = mode1Factor;
    Serial.println("MODE 1 - Full Speed");
  } else {
    modeFactorFinal = mode2Factor;
    Serial.println("MODE 2 - Medium Speed");
  }
}
/*-BATTERY VOLTAGE MONITOR-----------------------------------------------------------------------------------------------------------------*/
/*-----------BATTERY VOLTAGE MONITOR-------------------------------------------------------------------------------------------------------*/
/*---------------------BATTERY VOLTAGE MONITOR---------------------------------------------------------------------------------------------*/
void batteryMonitor() {
  float batteryPinADC = analogRead(batteryVoltagePin);
  unsigned int batteryVoltage = batteryPinADC * 2500 / 1024;
  Serial.println();
  Serial.print("Battery Voltage Level (x100) = ");
  Serial.print(batteryVoltage);
  Serial.println();

  if (batteryVoltage < lowBatteryLevel) {
    lowBatteryTime_crnt = millis();
    if (lowBatteryTime_crnt - lowBatteryTime_prev > 1000) {
      Serial.println("LOW BATTERY LEVEL");
      alarmStatus = 1;
    }
    lowBatteryTime_prev = lowBatteryTime_crnt;
  }
  Serial.println();
}
/*-IR DISTANCE SENSOR----------------------------------------------------------------------------------------------------------------------*/
/*-----------IR DISTANCE
```

```
SENSOR-----------------------------------------------------------------------------------------------------------------*/
/*---------------------IR DISTANCE SENSOR----------------------------------------------------------------------------------------------*/
void IRdistance() {
  float volts = analogRead(distIR1Pin) * 0.003125; // value from sensor * (5/1024)
  int distance = 13 * pow(volts, -1); // worked out from datasheet graph

  if (distance <= 100) {
    Serial.print("Distance: ");
    Serial.print(distance);   // print the distance
    Serial.println(" inches");
  }

  if (distance > IRdistMAX) {
    Serial.println("DROPOFF DETECTED ERROR");
    alarmStatus = 3;
  }
  Serial.println();
}
/*-MPU9250 INERTIAL SENSOR-----------------------------------------------------------------------------------------------------------------------*/
/*-----------MPU9250 INERTIAL SENSOR-------------------------------------------------------------------------------------------------------*/
/*---------------------MPU9250 INERTIAL SENSOR-------------------------------------------------------------------------------------------*/
void tiltSensor() {
  // Read the sensor from the MPU9250
  IMU.readSensor();

  // Display the data from the Z axis
  Zmss = IMU.getAccelZ_mss();
  temp = IMU.getTemperature_C();

  Serial.println("IMU Data");
  Serial.print("X: ");
  Serial.print(IMU.getAccelX_mss(),3); //3 decimal places
  Serial.print("\t");
  Serial.print("Y: ");
  Serial.print(IMU.getAccelY_mss(),3);
  Serial.print("\t");
  Serial.print("Z: ");
  Serial.print(Zmss,3);
  Serial.print("\t");
  Serial.print("Temperature: ");
  Serial.print(temp,3);
  Serial.println("°C");

  // If the tilt from the Z axis is greater than the limit, flag an error condition.
  if (abs(Zmss) < Zmss_limit) {
    Serial.println("TILT DETECTED ERROR");
    alarmStatus = 4;
  }

  // If the temperature inside the enclosure reaches the max temp shut motors off.
  if (temp > temp_limit) {
    Serial.println("OVER MAX TEMPERATURE ERROR");
    alarmStatus = 5;
  }
  Serial.println();
```

```
}
/*-START_STOP_TIMES-----------------------------------------------------------------------------------------------------------------------*/
/*-----START_STOP_TIMES-------------------------------------------------------------------------------------------------------------------*/
/*---------START_STOP_TIMES---------------------------------------------------------------------------------------------------------------*/
//THIS UPDATES THE START AND STOP TIMES STORED AS VARIABLES IN THE GLOBAL SCOPE
void updateStartStopTimes() {
  // Not timer dependent, is called by controlLoop()
  //Runs the instant any motion control is pressed or released, "PULLUP" so LOW means pressed

  //Forward control
  if (fwdBtnState_crnt != fwdBtnState_prev) {
    if (fwdBtnState_crnt == LOW) {
      fwdPressedTime = millis();                //Stores the instant forward button was pressed
      fwdTimeInactive = 0;                      //Clears the inactivity timer
      digitalWrite(forwardIndPin, HIGH);        //Turn on the control LED indicator for visual feedback
      digitalWrite(audioIndPin, HIGH);          //Send a signal too the UNO board to play audio feedback
    } else {                                    // the button was just released as the other state is HIGH
      fwdReleasedTime = millis();               //Stores the instant the forward button was released
      fwdTimeActive = 0;                        //Clears the activity timer
      digitalWrite(forwardIndPin, LOW);         //Turn off the control LED indicator for visual feedback
      digitalWrite(audioIndPin, LOW);          //Send a signal too the UNO board to turn off audio feedback
    }
  }

  //Left Control
  if (leftBtnState_crnt != leftBtnState_prev) {
    if (leftBtnState_crnt == LOW) {
      leftPressedTime = millis();               //Stores the instant forward button was pressed
      leftTimeInactive = 0;                     //Clear inactivity timer
      digitalWrite(leftIndPin, HIGH);           //Turn on the control LED indicator for visual feedback
      digitalWrite(audioIndPin, HIGH);          //Send a signal too the UNO board to play audio feedback
    } else {                                    // the button was just released as the other state is HIGH
      leftReleasedTime = millis();              //Stores the instant the button was released
      leftTimeActive = 0;                       //Clears the activity timer
      digitalWrite(leftIndPin, LOW);            //Turn off the control LED indicator for visual feedback
      digitalWrite(audioIndPin, LOW);          //Send a signal too the UNO board to turn off audio feedback
    }
  }

  //Right Control
  if (rightBtnState_crnt != rightBtnState_prev) {
    if (rightBtnState_crnt == LOW) {
      rightPressedTime = millis();              //Stores the instant forward button was pressed
      rightTimeInactive = 0;                    //Clear inactivity timer
      digitalWrite(rightIndPin, HIGH);          //Turn on the control LED indicator for visual feedback
      digitalWrite(audioIndPin, HIGH);          //Send a signal too the UNO board to play audio feedback
    } else {                                    // the button was just released as the other state is HIGH
      rightReleasedTime = millis();             //Stores the instant the button was released
      rightTimeActive = 0;                      //Clears the activity timer
      digitalWrite(rightIndPin, LOW);           //Turn off the control LED indicator for visual feedback
      digitalWrite(audioIndPin, LOW);          //Send a signal too the UNO board to turn off audio feedback
    }
  }

  //Reverse Control
  if (revBtnState_crnt != revBtnState_prev) {
    if (revBtnState_crnt == LOW) {
      revPressedTime = millis();                //Stores the instant forward button was pressed
      revTimeInactive = 0;                      //Clear inactivity timer
      digitalWrite(reverseIndPin, HIGH);        //Turn on the control LED indicator for visual feedback
      digitalWrite(audioIndPin, HIGH);          //Send a signal too the UNO board to play audio feedback
    } else {                                    // the button was just released as the other state is HIGH
      revReleasedTime = millis();               //Stores the instant the forward button was released
      revTimeActive = 0;                        //Clears the activity timer
      digitalWrite(reverseIndPin, LOW);         //Turn off the control LED indicator for visual feedback
      digitalWrite(audioIndPin, LOW);          //Send a signal too the UNO board to turn off audio feedback
    }
  }
}
/*--CONTROL TIME VALUES---------------------------------------------------------------------------------------------------------------------*/
/*------CONTROL TIME VALUES-----------------------------------------------------------------------------------------------------------------*/
/*----------CONTROL TIME VALUES-------------------------------------------------------------------------------------------------------------*/
void updateTimerValue() {
  //Timer Dependent
  timeCurrent = millis();
  // This function runs ALL THE TIME but only calculates a time while the forward button is actively being pressed and for one cycle after its release
  //FORWARD
  if (fwdBtnState_crnt == fwdBtnState_prev && fwdBtnState_crnt == 0) {  // FWD Button is active and loop has passed at least once
    fwdTimeActive = timeCurrent - fwdPressedTime;
    //fwdReleasedTime = 0;
    Serial.print("The FWD button has been held for");
    Serial.print('\t');
    Serial.print(fwdTimeActive);
    Serial.println(" ms.");
  }
  if (fwdBtnState_crnt == fwdBtnState_prev && fwdBtnState_crnt == 1) { //FWD Button is inactive and loop has passed at least once
```

```
    fwdTimeInactive = timeCurrent - fwdReleasedTime;
    Serial.print("The FWD button has been released for");
    Serial.print('\t');
    Serial.print(fwdTimeInactive);
    Serial.println(" ms.");
  }
  //LEFT
  if (leftBtnState_crnt == leftBtnState_prev && leftBtnState_crnt == 0) {  // LEFT Button is active and loop has passed at least once
    leftTimeActive = timeCurrent - leftPressedTime;
    //leftReleasedTime = 0;
    Serial.print("The LEFT button has been held for");
    Serial.print('\t');
    Serial.print(leftTimeActive);
    Serial.println(" ms.");
  }
  if (leftBtnState_crnt == leftBtnState_prev && leftBtnState_crnt == 1) { //LEFT Button is inactive and loop has passed at least once
    leftTimeInactive = timeCurrent - leftReleasedTime;
    Serial.print("The LEFT button has been released for");
    Serial.print('\t');
    Serial.print(leftTimeInactive);
    Serial.println(" ms.");
  }
  //RIGHT
  if (rightBtnState_crnt == rightBtnState_prev && rightBtnState_crnt == 0) { //Right Button is active and loop has passed at least once
    rightTimeActive = timeCurrent - rightPressedTime;
    //rightReleasedTime = 0;
    Serial.print("The RIGHT button has been held for");
    Serial.print('\t');
    Serial.print(rightTimeActive);
    Serial.println(" ms.");
  }
  if (rightBtnState_crnt == rightBtnState_prev && rightBtnState_crnt == 1) { //right Button is inactive and loop has passed at least once
    rightTimeInactive = timeCurrent - rightReleasedTime;
    Serial.print("The RIGHT button has been released for");
    Serial.print('\t');
    Serial.print(rightTimeInactive);
    Serial.println(" ms.");
  }
  //REVERSE
  if (revBtnState_crnt == revBtnState_prev && revBtnState_crnt == 0) {  //Reverse Button is active and loop has passed at least once
    revTimeActive = timeCurrent - revPressedTime;
    //revReleasedTime = 0;
    Serial.print("The REV button has been held for");
    Serial.print('\t');
    Serial.print(revTimeActive);
    Serial.println(" ms.");
  }
  if (revBtnState_crnt == revBtnState_prev && revBtnState_crnt == 1) { //Reverse Button is inactive and loop has passed at least once
    revTimeInactive = timeCurrent - revReleasedTime;
    Serial.print("The REV button has been released for");
    Serial.print('\t');
    Serial.print(revTimeInactive);
    Serial.println(" ms.");
  }
}
/*-----------------------------------------------------------------------------------------------------------------------------------------*/

```

```
/*-----------------------------------------------------------------------------------------------------------------------------------------*/

/*-----------------------------------------------------------------------------------------------------------------------------------------*/
void calcMtrSpd() {
  //Motor/Encoder Speed
  encoderLeftValue_prev = encoderLeftValue_crnt;              //Update the previous value with the current value
  encoderLeftValue_crnt = encoderLeftValue_counter;           //Store how many pulses occurred for calculation
  encoderLeftValue_counter = 0;                               //Reset the encoder value each loop (after 50ms)
  //Right Encoder Speed
  encoderRightValue_prev = encoderRightValue_crnt;            //Update the previous value with the current value
  encoderRightValue_crnt = encoderRightValue_counter;         //Store how many pulses occurred for calculation
  encoderRightValue_counter = 0;                              //Reset the encoder value each loop (after 50ms)

  //Motor/Encoder Direction
  if (stateEncoderL_B == LOW) {                               //Determine the direction the LEFT motor is spining based on quadrature signal
    encoderLeftDir_txt = "CW ";
  } else {
    encoderLeftValue_crnt = encoderLeftValue_crnt * -1;     //Set the encoder value to negative if CCW rotation
    encoderLeftDir_txt = "CCW";
  }

  if (stateEncoderR_B == LOW) {                               //Determine the direction the RIGHT motor is spining based on quadrature signal
    encoderRightDir_txt = "CW ";
  } else {
    encoderRightValue_crnt = encoderRightValue_crnt * -1;     //Set the encoder value to negative if CCW rotation
    encoderRightDir_txt = "CCW";
  }
  //Display the speed in the serial monitor for troubleshooting and tuning
  if (encoderLeftValue_prev != encoderLeftValue_crnt || encoderRightValue_prev != encoderRightValue_crnt) {       //Only update the serial monitor if there was a change
    Serial.print("PULSES per 50ms[L/R]: ");
    Serial.print('\t');
    Serial.print(encoderLeftValue_crnt);
    Serial.print('\t');
    Serial.print(encoderLeftDir_txt);
    Serial.print(" / ");
    Serial.print(encoderRightValue_crnt);
    Serial.print('\t');
    Serial.println(encoderRightDir_txt);
  }
}

/*-----------------------------------------------------------------------------------------------------------------------------------------*/

/*-----------------------------------------------------------------------------------------------------------------------------------------*/

/*-----------------------------------------------------------------------------------------------------------------------------------------*/
void pidLoop() {
  //Set a maximum PWM value for this subroutine (normally 255, but for
demonstration purposes backed off to avoid battery fault)
  const unsigned char maxPWM = 130;

  //Get the error for each motor setpoint
  errorLeft_crnt = leftTargetSpeed_crnt - encoderLeftValue_crnt;
  errorRight_crnt = rightTargetSpeed_crnt - encoderRightValue_crnt;

  //Calculate Proportional Term
  PID_Lp = kp * (float)errorLeft_crnt;
  PID_Rp = kp * (float)errorRight_crnt;

  //Calculate Derivative Term
  int errorLeft_diff = errorLeft_crnt - errorLeft_prev;
  PID_Ld = kd * (((float)errorLeft_crnt - (float)errorLeft_prev) / (float)shortInterval);

  int errorRight_diff = errorRight_crnt - errorRight_prev;
  PID_Rd = kd * (((float)errorRight_crnt - (float)errorRight_prev) / (float)shortInterval);

  //Calculate Left Integral Term
  if (-3 < errorLeft_crnt && errorLeft_crnt < 3) {
    PID_Li = PID_Li + (ki * (float)errorLeft_crnt);

  } else {

    PID_Li = 0;  //When both motors are added, this is probably the place to compare their speed

  }
  //Calculate Right Integral Term
  if (-3 < errorRight_crnt && errorRight_crnt < 3) {
    PID_Ri = PID_Ri + (ki * (float)errorRight_crnt);

  } else {

    PID_Ri = 0;  //When both motors are added, this is probably the place to compare their speed

  }

  //Sum for total PID control signal
  PID_Ltotal = PID_Lp + PID_Li + PID_Ld;
  PID_Rtotal = PID_Rp + PID_Ri + PID_Rd;

  //The motor PWM value cannot exceed maxPWM (if it does it is maxed out)
  if (PID_Ltotal > maxPWM) {
    PID_Ltotal = maxPWM;
  }
  if (PID_Ltotal < -maxPWM) {
    PID_Ltotal = -maxPWM;
  }

  if (PID_Rtotal > maxPWM) {
    PID_Rtotal = maxPWM;
  }
  if (PID_Rtotal < -maxPWM) {
    PID_Rtotal = -maxPWM;
  }

  Serial.print("PID Values [L/R]: ");
  Serial.print('\t');
  Serial.print(PID_Ltotal);
  Serial.print(" / ");
  Serial.println(PID_Rtotal);
```

```
}
/*--STOP BUTTON HANDLER---------------------------------------------------------------------------------------------------------------------*/
/*----------STOP BUTTON HANDLER-------------------------------------------------------------------------------------------------------------*/
/*------------------STOP BUTTON HANDLER-----------------------------------------------------------------------------------------------------*/
void StopOverrideHandler() {
  if (enableTime_crnt - enableTime_prev > 350) {     //If the interrupt has been toggled within 250ms its probably a switch bounce, ignore it
    systemEnableState = !systemEnableState;          //Only change the system state if signals are recieved more than 350ms appart
    digitalWrite(systemEnabledIndPin, systemEnableState);

    //Clear any existing system alarms.
    alarmStatus = 0;
    digitalWrite(alarmIndPin, LOW);

    boolean proxValue = digitalRead(proxEncPin);      //Read the proximity sensor
    if (proxValue == LOW) {                           //If the sensor output is high the lid is open
      enclosureState_prev = enclosureState_crnt;      //Clear the enclosure alarm if it has been shut.
    } else {
      alarmStatus = 2;
    }
  }
  enableTime_prev = enableTime_crnt;
  enableBtnState_prev = enableBtnState_crnt;
  ///* Troubleshooting
  Serial.print("STOP");
  Serial.print('\t');
  Serial.print("STOP");
  Serial.print('\t');
  Serial.print("STOP");
  Serial.print('\t');
  Serial.print("STOP");
  Serial.print('\t');
  Serial.print("STOP");
  Serial.print('\t');
  Serial.print("STOP");
  Serial.print('\t');
  Serial.println("STOP - - - - - - - - - -");
  Serial.print("System state is: ");
  Serial.println(systemEnableState);
  //*/
}
/*--ENCLOSURE PROX HANDLER------------------------------------------------------------------------------------------------------------------*/
/*----------ENCLOSURE PROX HANDLER----------------------------------------------------------------------------------------------------------*/
/*------------------ENCLOSURE PROX HANDLER--------------------------------------------------------------------------------------------------*/
/*
   NOTES:
   The proximity sensor input and output is 5V which cannot be powered or directly read by the DUE.
   A logic level converter must be used to step 5V down to 3V.
*/
```

```
void enclosureProxHandler() {

  if (enclosureTime_crnt - enclosureTime_prev > 100) {     //If the interrupt has been toggled within 350ms its probably a ground bounce, ignore it
                                                           //enclosureTime_crnt is updated by the enclosureISR when proxEncPin changes.
    boolean proxValue = digitalRead(proxEncPin);           //Read the proximity sensor

    if (proxValue == HIGH) {                               //If the sensor output is high the lid is open
      Serial.println();
      Serial.println("ENCLOSURE OPEN");
      Serial.println("ENCLOSURE OPEN");
      Serial.println("ENCLOSURE OPEN");
      Serial.println("ENCLOSURE OPEN");
      Serial.println("ENCLOSURE OPEN");
      Serial.println();
      alarmStatus = 2;                                    //Set the warning state flag to enclosure open
      enclosureTime_prev = 0;
      enclosureTime_crnt = 1;
    }
    if (proxValue == LOW) {
      Serial.println();
      Serial.println("ENCLOSURE CLOSED");
      Serial.println("ENCLOSURE CLOSED");
      Serial.println("ENCLOSURE CLOSED");
      Serial.println("ENCLOSURE CLOSED");
      Serial.println("ENCLOSURE CLOSED");
      Serial.println();
    }
  }
  enclosureTime_prev = enclosureTime_crnt;
  //enclosureState_prev = enclosureState_crnt;            //Prevent re-executing this loop until the system has been reset

}
/*--WARNING INDICATOR------------------------------------------------------------------------------------------------------------------------*/
/*----------WARNING INDICATOR----------------------------------------------------------------------------------------------------------------*/
/*------------------WARNING INDICATOR--------------------------------------------------------------------------------------------------------*/
void warningIndicator() {
  warningTimer_crnt = millis();

  switch (alarmStatus) {                                  //Blink the indicator a specific amount of times to indicate the alarm type
    case 1:  //Battery Alarm
      if (warningTimer_crnt > warningTimer_prev && warningTimer_crnt < warningTimer_prev + 250) {
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 250) {
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1500) {
        warningTimer_prev = warningTimer_crnt;
      }
      break;

    case 2:   //Enclosure Alarm
      if (warningTimer_crnt > warningTimer_prev && warningTimer_crnt < warningTimer_prev + 250) {         // On if 0<timer<250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 250 && warningTimer_crnt < warningTimer_prev + 500) {   // Off if 250<timer<500
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 500 && warningTimer_crnt < warningTimer_prev + 750) {   // On if 500<timer<750
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 750) {
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)                  // Off if timer>750
      }
      if (warningTimer_crnt > warningTimer_prev + 1750) {        // Reset sequence after 1000ms delay
        warningTimer_prev = warningTimer_crnt;
      }
      break;

    case 3:   //Drop off alarm (IR sensor)
      if (warningTimer_crnt > warningTimer_prev && warningTimer_crnt < warningTimer_prev + 250) {         // On if 0<timer<250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 250 && warningTimer_crnt < warningTimer_prev + 500) {   // Off if 250<timer<500
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 500 && warningTimer_crnt < warningTimer_prev + 750) {   // On if 500<timer<750
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 750 && warningTimer_crnt < warningTimer_prev + 1000) {   // Off if 750<timer<1000
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1000 && warningTimer_crnt < warningTimer_prev + 1250) {   // On if 1000<timer<1250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1250) {
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)                  // Off if timer>1250
      }
      if (warningTimer_crnt > warningTimer_prev + 2250) {        // Reset sequence after 1000ms delay
        warningTimer_prev = warningTimer_crnt;
      }
      break;

    case 4:   //Tilt alarm (accelerometer)
```

```
      if (warningTimer_crnt > warningTimer_prev && warningTimer_crnt < warningTimer_prev + 250) {         // On if 0<timer<250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 250 && warningTimer_crnt < warningTimer_prev + 500) {   // Off if 250<timer<500
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 500 && warningTimer_crnt < warningTimer_prev + 750) {   // On if 500<timer<750
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 750 && warningTimer_crnt < warningTimer_prev + 1000) {   // Off if 750<timer<1000
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1000 && warningTimer_crnt < warningTimer_prev + 1250) {   // On if 1000<timer<1250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1250 && warningTimer_crnt < warningTimer_prev + 1500) {   // Off if 1250<timer<1500
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1500 && warningTimer_crnt < warningTimer_prev + 1750) {   // On if 1500<timer<1750
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1750) {
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)                  // Off if timer>1750
      }
      if (warningTimer_crnt > warningTimer_prev + 2750) {                                // Reset sequence after 1000ms delay
```

```
        warningTimer_prev = warningTimer_crnt;
      }
      break;

    case 5:   //Temperature sensor (included w/mpu9250)
      if (warningTimer_crnt > warningTimer_prev && warningTimer_crnt < warningTimer_prev + 250) {         // On if 0<timer<250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 250 && warningTimer_crnt < warningTimer_prev + 500) {   // Off if 250<timer<500
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 500 && warningTimer_crnt < warningTimer_prev + 750) {   // On if 500<timer<750
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 750 && warningTimer_crnt < warningTimer_prev + 1000) {   // Off if 750<timer<1000
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1000 && warningTimer_crnt < warningTimer_prev + 1250) {   // On if 1000<timer<1250
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1250 && warningTimer_crnt < warningTimer_prev + 1500) {   // Off if 1250<timer<1500
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 1500 && warningTimer_crnt < warningTimer_prev + 1750) {   // On if 1500<timer<1750
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 2000 && warningTimer_crnt < warningTimer_prev + 2250) {   // Off if 2000<timer<2250
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 2250 && warningTimer_crnt < warningTimer_prev + 2500) {   // On if 2250<timer<2500
        digitalWrite(alarmIndPin, HIGH);   // turn the LED on (HIGH is the voltage level)
      }
      if (warningTimer_crnt > warningTimer_prev + 2500) {
        digitalWrite(alarmIndPin, LOW);   // turn the LED on (HIGH is the voltage level)                  // Off if timer>2500
      }
      if (warningTimer_crnt > warningTimer_prev + 3500) {                                                  // Reset sequence after 1000ms delay
        warningTimer_prev = warningTimer_crnt;
      }
      break;
  }


}
/*--TARGET SPEED---------------------------------------------------------------------------------------------------------------------------*/
/*------TARGET SPEED-----------------------------------------------------------------------------------------------------------------------*/
/*----------TARGET SPEED-------------------------------------------------------------------------------------------------------------------*/
void updateTargetSpeed() {
  //Timer dependent, called by main Void loop every 50ms


  //Store all the active timers into an array so that the maximum one can be selected.
  unsigned long maxTimerArray[] = {fwdTimeActive, leftTimeActive, rightTimeActive, revTimeActive};
  const String indexedButtons[] = {"FWD", "LEFT", "RIGHT", "REV"};
  byte maxIndex = 0;

  //Iterate through each item in the array to select the maximum timer value
  for (byte i = 0; i < 4; i++) {
    if (maxTimerArray[i] >= maxTimerArray[maxIndex]) {
      maxIndex = i;
    }
  }
  Serial.print("The max active timer value is: ");      //The very first iteration will be irrelvant (always REV because it is last element of array) and the target speed should be set to zero
  Serial.print('\t');                                   //After controls are input the system will be able to determine which input was selected first.
  Serial.println(indexedButtons[maxIndex]);             //This is required to determine which direction to twist for holding left + right simultaneously.

  //Check control state to see how many control inputs are active
  switch (controlState_crnt) {
    case 4:   // No controls are active, target should be zero or decelerate to zero if needed
      // Left and right will be checked for both positive and negative motion to see if gradual decel is required
      //LEFT TARGET SPEED SETTING
      if (leftTargetSpeed_crnt > 0) {                                 // Only decelerate if one target has not already reached zero.
        for (int i = 0; i < DecelArrayCount; i++) {                   //i is the number of elements in the decelSpeedValue array
          if (leftTargetSpeed_crnt > DecelSpeedValues[i]) {           //Set the value only if it is greater than the next index in the array, which is stepped through from largest to smallest
            leftTargetSpeed_crnt = DecelSpeedValues[i];
            break;
          }
        }
      } else if (leftTargetSpeed_crnt < 0) {                         //The motor must be moving backwards if negative target speed
        for (int i = 0; i < DecelArrayCount; i++) {                  //i is the
```

```
number of elements in the decelSpeedValue array
          if (-leftTargetSpeed_crnt > DecelSpeedValues[i]) {
            leftTargetSpeed_crnt = -DecelSpeedValues[i];
            break;
          }
        }
      } else {                                                       // If value is not >0 or <0 must be =0
        leftTargetSpeed_crnt = 0;
      }

      //RIGHT TARGET SPEED SETTING
      if (rightTargetSpeed_crnt > 0) {                               // Only decelerate if one target has not already reached zero.
        for (int i = 0; i < DecelArrayCount; i++) {                  //i is the number of elements in the decelSpeedValue array
          if (rightTargetSpeed_crnt > DecelSpeedValues[i]) {
            rightTargetSpeed_crnt = DecelSpeedValues[i];
            break;
          }
        }
      } else if (rightTargetSpeed_crnt < 0) {                        // The motor must be moving backwards if negative targetspeed
        for (int i = 0; i < DecelArrayCount; i++) {                  //i is the number of elements in the decelSpeedValue array
          if (-rightTargetSpeed_crnt > DecelSpeedValues[i]) {
            rightTargetSpeed_crnt = -DecelSpeedValues[i];
            break;
          }
        }
      } else {                                                       // If value is not >0 or <0 must be =0
        rightTargetSpeed_crnt = 0;
      }
      //Write the target value to the serial monitor
      Serial.println();
      Serial.print("The target velocity is L/R:");
      Serial.print('\t');
      Serial.print(leftTargetSpeed_crnt);
      Serial.print(" / ");
      Serial.println(rightTargetSpeed_crnt);
      Serial.println();
      break;

    case 3:   // A single control input is active --------------------------------------------------------------------------------------

      switch (maxIndex) {   //Sub switch case determined by which single control is active
        case 0: // FORWARD only is active (drive both motors)
          for (int i = 0; i < AccelArrayCount; i++) {
            if (leftTargetSpeed_crnt < AccelSpeedValues[i]*modeFactorFinal) {   // This checks if the system was already moving instead of accelerating from a stop
              leftTargetSpeed_crnt = AccelSpeedValues[i] * modeFactorFinal;
            }
            if (rightTargetSpeed_crnt < AccelSpeedValues[i]) {   // This checks if the system was already moving instead of accelerating from a stop
              rightTargetSpeed_crnt = AccelSpeedValues[i];
              break;//forloop
            }
          }

          break;

        case 1: // LEFT only is active (drive right motor)
          for (int i = 0; i < AccelArrayCount; i++) {
            if (rightTargetSpeed_crnt < AccelSpeedValues[i]*modeFactorFinal) {   // This checks if the system was already moving instead of accelerating from a stop
              rightTargetSpeed_crnt = AccelSpeedValues[i] * modeFactorFinal;
              break;
            }
          }
          break;
        case 2: // RIGHT only is active (drive left motor)
          for (int i = 0; i < AccelArrayCount; i++) {
            if (leftTargetSpeed_crnt < AccelSpeedValues[i]*modeFactorFinal) {   // This checks if the system was already moving instead of accelerating from a stop
              leftTargetSpeed_crnt = AccelSpeedValues[i] * modeFactorFinal;
              break;
            }
          }
          break;
        case 3: // REVERSE only is active (drive both motors, backwards and divided by a factor)
          for (int i = 0; i < AccelArrayCount; i++) {
            if (abs(leftTargetSpeed_crnt) < AccelSpeedValues[i] / revFactor * modeFactorFinal) { // This checks if the system was already moving instead of accelerating from a stop
              leftTargetSpeed_crnt = -AccelSpeedValues[i] / revFactor * modeFactorFinal;       // abs() is used because target will be negative.  Negative is used to designate reverse to the PID loop.
            }
            if (abs(rightTargetSpeed_crnt) < AccelSpeedValues[i] / revFactor * modeFactorFinal) { // This checks if the system was already moving instead of accelerating from a stop
              rightTargetSpeed_crnt = -AccelSpeedValues[i] / revFactor * modeFactorFinal;
              break;//for loop exit
            }
          }
          break;
      }

      //Write the target value to the serial monitor
      Serial.println();
      Serial.print("The target velocity is L/R:");
      Serial.print('\t');
      Serial.print(leftTargetSpeed_crnt);
      Serial.print(" / ");
      Serial.println(rightTargetSpeed_crnt);
      Serial.println();
      break;
    case 2:   // Two control inputs are active simultaneously ---------------------------------------------------------------------------
      //Diagonal Left Forward
      if (fwdBtnState_crnt == 0 && leftBtnState_crnt == 0) {
        for (int i = 0; i < AccelArrayCount; i++) {
          if (rightTargetSpeed_crnt < AccelSpeedValues[i]*modeFactorFinal) {   // This checks if the system was already moving instead of accelerating from a stop
            rightTargetSpeed_crnt = AccelSpeedValues[i] * modeFactorFinal;
            leftTargetSpeed_crnt = rightTargetSpeed_crnt * diagTurnFactor1 / diagTurnFactor2;   //Turn the left wheel proportionally slower than the right in order to turn left
            break;//for loop exit
          }
        }
      }

      //Diagonal Right Forward
      if (fwdBtnState_crnt == 0 && rightBtnState_crnt == 0) {
        for (int i = 0; i < AccelArrayCount; i++) {
          if (leftTargetSpeed_crnt < AccelSpeedValues[i]*modeFactorFinal) {   // This checks if the system was already moving instead of accelerating from a stop
            leftTargetSpeed_crnt = AccelSpeedValues[i] * modeFactorFinal;
            rightTargetSpeed_crnt = leftTargetSpeed_crnt * diagTurnFactor1 / diagTurnFactor2;   //Turn the right wheel proportionally slower than the left in order to turn right
            break;//for loop exit
          }
        }
      }
      //Diagonal Left Reverse
      if (revBtnState_crnt == 0 && leftBtnState_crnt == 0) {
        for (int i = 0; i < AccelArrayCount; i++) {
          if (abs(rightTargetSpeed_crnt) < AccelSpeedValues[i] / revFactor * modeFactorFinal) { // This checks if the system was already moving instead of accelerating from a stop
            rightTargetSpeed_crnt = -AccelSpeedValues[i] / revFactor * modeFactorFinal;
            leftTargetSpeed_crnt = rightTargetSpeed_crnt * diagTurnFactor1 / diagTurnFactor2;   //Turn the right wheel proportionally slower than the left in order to turn right
            break;//for loop exit
          }
        }
      }
      //Diagonal Right Reverse
      if (revBtnState_crnt == 0 && rightBtnState_crnt == 0) {
        for (int i = 0; i < AccelArrayCount; i++) {
          if (abs(leftTargetSpeed_crnt) < AccelSpeedValues[i] / revFactor * modeFactorFinal) { // This checks if the system was already moving instead of accelerating from a stop
            leftTargetSpeed_crnt = -AccelSpeedValues[i] / revFactor * modeFactorFinal;
            rightTargetSpeed_crnt = leftTargetSpeed_crnt * diagTurnFactor1 / diagTurnFactor2;   //Turn the right wheel proportionally slower than the left in order to turn right
            break;//for loop exit
          }
        }
      }
      //Twist Conditions (left and right control simultaneous)
      if (leftBtnState_crnt == 0 && rightBtnState_crnt == 0) {
        //Twist Left
        if (maxIndex == 2) {  //Right button was applied first
          for (int i = 0; i < AccelArrayCount; i++) {
            if (rightTargetSpeed_crnt < AccelSpeedValues[i] / revFactor * modeFactorFinal) {           // This checks if the system was already moving instead of accelerating from a stop
              rightTargetSpeed_crnt = AccelSpeedValues[i] / revFactor * modeFactorFinal; //Set speed according to mode selected
              leftTargetSpeed_crnt = -rightTargetSpeed_crnt;
```

```
//Turn the left wheel the opposite direction but same speed as right wheel to pivot in place
              break;//for loop exit
            }
          }
        } else {             //Left button was applied first, maxIndex must be equal to 1
          for (int i = 0; i < AccelArrayCount; i++) {
            if (leftTargetSpeed_crnt < AccelSpeedValues[i] / revFactor * modeFactorFinal) {           // This checks if the system was already moving instead of accelerating from a stop
              leftTargetSpeed_crnt = AccelSpeedValues[i] / revFactor * modeFactorFinal; //Set final speed according to mode selected
              rightTargetSpeed_crnt = -leftTargetSpeed_crnt; //Turn the right wheel the opposite direction but same speed as left wheel to pivot in place
              break;//for loop exit
            }
          }
        }
      }

      //Write the target value to the serial monitor
      Serial.println();
      Serial.print("The target velocity is L/R:");
      Serial.print('\t');
      Serial.print(leftTargetSpeed_crnt);
      Serial.print(" / ");
      Serial.println(rightTargetSpeed_crnt);
      Serial.println();
      break;
    case 1:   // Three control inputs are active simultaneously ------------------------------------------------------------------------
              // There is no forseeable condition in which the user could intentionally do this, turn the motors off.
      leftTargetSpeed_crnt = 0;
      rightTargetSpeed_crnt = 0;
      break;
    case 0:   // All 4 directional control inputs are active ----------------------------------------------------------------------------
              // There is no forseeable condition in which the user could intentionally do this, turn the motors off.
      leftTargetSpeed_crnt = 0;
      rightTargetSpeed_crnt = 0;
      break;
  }
}
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
/*-----------------------------------------------------------------------------------------------------------------------------------------*/
```