```c
#include <tistdtypes.h>
#include <coecsl.h>
#include "user_includes.h"
#include "math.h"

// These two offsets are only used in the main file user_CRSRobot.c  You just need to create them here and find the correct offset and then these offset will adjust the encoder readings
float offset_Enc2_rad = -0.415; //-0.37;
float offset_Enc3_rad = 0.233; //0.27;

// Your global variables.
float px = 0; // x coordinate of end effector
float py = 0; // y coordinate of end effector
float pz = 0; // z coordinate of end effector
float theta1m_IK = 0; // theta1 motor from inverse kinematics
float theta2m_IK = 0; // theta2 motor from inverse kinematics
float theta3m_IK = 0; // theta3 motor from inverse kinematics
long mycount = 0;
long counter = 0;
int cycle = 0;

// Omega calculations
float Theta1_old = 0;
float Omega1_old1 = 0;
float Omega1_old2 = 0;
float Omega1 = 0;
float e1_old = 0;
float integral1_old = 0;

float Theta2_old = 0;
float Omega2_old1 = 0;
float Omega2_old2 = 0;
float Omega2 = 0;
float e2_old = 0;
float integral2_old = 0;

float Theta3_old = 0;
float Omega3_old1 = 0;
float Omega3_old2 = 0;
float Omega3 = 0;
float e3_old = 0;
float integral3_old = 0;

// Velocity calculations
float x_old = 0;
float vx_old1 = 0;
float vx_old2 = 0;
float vx = 0;
//float x_e_old = 0;
//float integral1_old = 0;

float y_old = 0;
float vy_old1 = 0;
float vy_old2 = 0;
float vy = 0;
//float y_e_old = 0;
//float integral2_old = 0;

```

```
float z_old = 0;
float vz_old1 = 0;
float vz_old2 = 0;
float vz = 0;
//float z_e_old = 0;
//float integral3_old = 0;

// Constants
float dt = 0.001;
float threshold1 = 0.01;
float threshold2 = 0.06;
float threshold3 = 0.02;

//Inverse Dynamics Gains
float kp1 = 300.0;
float kd1 = 4;
float ki1 = 420;

float kp2 = 5000;
float kd2 = 250;
float ki2 = 280;

float kp3 = 7000;
float kd3 = 300;
float ki3 = 260;

//PD Gains
float fkp1 = 300;
float fkp2 = 350;
float fkp3 = 300;

float fkd1 = 4;
float fkd2 = 4;
float fkd3 = 4;

float theta_dotdot = 0.0;

float e1 = 0;
float e2 = 0;
float e3 = 0;

float x_desired = 0;
float y_desired = 0;
float z_desired = 0;
//Positive and negative viscous coefficients, and positive and negative coulomb (static) coefficients for each joint
float Viscouspos1=0.17, Viscouspos2=0.23, Viscouspos3=0.1922;
float Viscousneg1=0.17, Viscousneg2=0.287, Viscousneg3=0.2132;
float Coulombpos1=0.40, Coulombpos2=0.40, Coulombpos3=0.40;
float Coulombneg1=-0.35, Coulombneg2=-0.40, Coulombneg3=-0.50;

float slope1 = 3.6;
float slope2 = 3.6;
float slope3 = 3.6;

//Without Mass
float p1 = 0.0300;
float p2 = 0.0128;
float p3 = 0.0076;
float p4 = 0.0753;
```

```
float p5 = 0.0298;

float a_theta2;
float a_theta3;

float sintheta2;
float costheta2;
float sintheta3;
float costheta3;
float g = 9.81;


float J1 = 0.0167;
float J2 = 0.03;
float J3 = 0.0128;

float ff = 1;

float a0,a1,a2,a3;
float tau1cur,tau2cur,tau3cur;

float desired_1, desired_2, desired_3;
float desired_dot_1, desired_dot_2, desired_dot_3;
float desired_doubledot_1, desired_doubledot_2, desired_doubledot_3;

//Lab 4 Starter Code
float cosq1 = 0;
float sinq1 = 0;
float cosq2 = 0;
float sinq2 = 0;
float cosq3 = 0;
float sinq3 = 0;
float JT_11 = 0;
float JT_12 = 0;
float JT_13 = 0;
float JT_21 = 0;
float JT_22 = 0;
float JT_23 = 0;
float JT_31 = 0;
float JT_32 = 0;
float JT_33 = 0;
float cosz = 0;
float sinz = 0;
float cosx = 0;
float sinx = 0;
float cosy = 0;
float siny = 0;
//Rotational matrix
float R11 = 0;
float R12 = 0;
float R13 = 0;
float R21 = 0;
float R22 = 0;
float R23 = 0;
float R31 = 0;
float R32 = 0;
float R33 = 0;
//Transpose of the rotational matrix
float RT11 = 0;
float RT12 = 0;
```

```
float RT13 = 0;
float RT21 = 0;
float RT22 = 0;
float RT23 = 0;
float RT31 = 0;
float RT32 = 0;
float RT33 = 0;

float thetax = 0;
float thetay = 0;
float thetaz = 60;

float kpx = 0.5;
float kpy = 0.5;
float kpz = 0.5;
float kdx = 0.025;
float kdy = 0.025;
float kdz = 0.025;

//Friction Multiplication factor
float fzcmd = 8;
//Robots torque constant
float kt = 6;
//Gravity Compensation in Z direction
float grav_comp = 5;

// Straight line trajectory
float vel = 0.5;
float xa = 8;
float ya = 8;
float za = 10;
float xb = 25.32;
float yb = 18;
float zb = 10;
float t_start = 0;

// Gains in N-frame
float Kpx_n = 1.3;
float Kpy_n = 1.3;
float Kpz_n = 1.3;
float Kdx_n = 0.03;
float Kdy_n = 0.03;
float Kdz_n = 0.025;


#pragma DATA_SECTION(whattoprint, ".my_vars")
float whattoprint = 0.0;

#pragma DATA_SECTION(whatnottoprint, ".my_vars")
float whatnottoprint = 0.0;

#pragma DATA_SECTION(theta1array, ".my_arrs")
float theta1array[100];

#pragma DATA_SECTION(theta2array, ".my_arrs")
float theta2array[100];

long arrayindex = 0;

float printtheta1motor = 0;
```

```
float printtheta2motor = 0;
float printtheta3motor = 0;

// Assign these float to the values you would like to plot in Simulink
float Simulink_PlotVar1 = 0;
float Simulink_PlotVar2 = 0;
float Simulink_PlotVar3 = 0;
float Simulink_PlotVar4 = 0;


void inverseKinematics(float px, float py, float pz) {
    // Calculate DH angles from end effector position
    //Function evaluates inverse kinematics for the robot, inputs are px, py, pz
    float theta1 = atan2(py, px); // DH theta 1
    float z = pz - 10;
    float beta = sqrt(px*px + py*py);
    float L = sqrt(z*z + beta*beta);
    float theta3 = acos((L*L - 200)/200); // DH theta 3
    float theta2 = -theta3/2 - atan2(z, beta); // DH theta 2


    // Convert DH angles to motor angles
    theta1m_IK = theta1;
    theta2m_IK = (theta2 + PI/2);
    theta3m_IK = (theta3 + theta2m_IK - PI/2);
}

// Omega estimation
void omega(float theta1motor, float theta2motor, float theta3motor) {
    //Function updates omega values given theta1motor, theta2motor and theta3motor
    Omega1 = (theta1motor - Theta1_old)/0.001;
    Omega1 = (Omega1 + Omega1_old1 + Omega1_old2)/3.0;

    Theta1_old = theta1motor;

    Omega1_old2 = Omega1_old1;
    Omega1_old1 = Omega1;

    Omega2 = (theta2motor - Theta2_old)/0.001;
    Omega2 = (Omega2 + Omega2_old1 + Omega2_old2)/3.0;

    Theta2_old = theta2motor;

    Omega2_old2 = Omega2_old1;
    Omega2_old1 = Omega2;

    Omega3 = (theta3motor - Theta3_old)/0.001;
    Omega3 = (Omega3 + Omega3_old1 + Omega3_old2)/3.0;

    Theta3_old = theta3motor;

    Omega3_old2 = Omega3_old1;
    Omega3_old1 = Omega3;
}

//Velocity Updates
void velocity(float x, float y, float z) {
    //Function updates velocity given x, y, z
    vx = (x - x_old)/0.001;
    vx = (vx + vx_old1 + vx_old2)/3.0;
```

```
    x_old = x;

    vx_old2 = vx_old1;
    vx_old1 = vx;

    vy = (y - y_old)/0.001;
    vy = (vy + vy_old1 + vy_old2)/3.0;

    y_old = y;

    vy_old2 = vy_old1;
    vy_old1 = vy;

    vz = (z - z_old)/0.001;
    vz = (vz + vz_old1 + vz_old2)/3.0;

    z_old = z;

    vz_old2 = vz_old1;
    vz_old1 = vz;
}


//Trajectory Generation
float cubic_func(float a0, float a1, float a2, float a3, float t) {
    return a0*t*t*t + a1*t*t + a2*t + a3;
}
float dot(float a0, float a1, float a2, float t) {
    return 3*a0*t*t + 2*a1*t + a2;
}
float doubledot(float a0, float a1, float t) {
    return 6*a0*t + 2*a1;
}

void lab(float theta1motor,float theta2motor,float theta3motor,float *tau1,float *tau2,float *tau3, int error) {

    //Forward Kinematics
    float x = 10*cos(theta1motor)*(cos(theta3motor) + sin(theta2motor));
    float y = 10*sin(theta1motor)*(cos(theta3motor) + sin(theta2motor));
    float z = 10*cos(theta2motor) - 10*sin(theta3motor) + 10;

    float time = counter*0.001;
    //Coefficients for different time steps
    if (counter < 330){
        a0 = -27.826474107465835; a1 = 13.774104683195590; a2 = 0; a3 = 0.25;
    }
    if (counter >= 330 && counter < 4000){
        a0 = 0; a1 = 0; a2 = 0; a3 = 0.75;
    }
    if (counter >= 4000 && counter < 4330){
        a0 = 27.826474107496760; a1 = -347.6917939731718; a2 = 1445.863594625530; a3 = -2000.530017811164;
    }
    if (counter >= 4330 && counter < 8000){
        a0 = 0; a1 = 0; a2 = 0; a3 = 0.25;
    }
    //Desired Trajectories
    desired_1 = cubic_func(a0, a1 , a2, a3, time);
```

```
    desired_dot_1 = dot(a0, a1, a2, time);
    desired_doubledot_1 = doubledot(a0, a1, time);

    desired_2 = cubic_func(a0, a1 , a2, a3, time);
    desired_dot_2 = dot(a0, a1, a2, time);
    desired_doubledot_2 = doubledot(a0, a1, time);

    desired_3 = cubic_func(a0, a1 , a2, a3, time);
    desired_dot_3 = dot(a0, a1, a2, time);
    desired_doubledot_3 = doubledot(a0, a1, time);

    //Step Trajectory to 10, 10, 10 without velocity
    //Desired Position
//    float xd = 10;
//    float xd_dot = 0;
//
//    float yd = 10;
//    float yd_dot = 0;
//
//    float zd = 10;
//    float zd_dot = 0;


//Straight line Trajectory for Lab3
// set desired points
    float xd = xb;
    float yd = yb;
    float zd = zb;
    float xd_dot = 0;
    float yd_dot = 0;
    float zd_dot = 0;
    float t_total = sqrt((xb-xa)*(xb-xa) + (yb-ya)*(yb-ya) + (zb-za)*(zb-za)) / vel;

    //Calculate desired x,y,z position as a linear function of time
    if (time >= t_start && time <= t_start + t_total) {
        xd = (xb-xa)*(time - t_start)/t_total + xa;
        yd = (yb-ya)*(time - t_start)/t_total + ya;
        zd = (zb-za)*(time - t_start)/t_total + za;
    }

// Calculate/Update omegas/velocities
    omega(theta1motor, theta2motor, theta3motor);
    velocity(x, y, z);

// position error
    float x_error = xd - x;
    float y_error = yd - y;
    float z_error = zd - z;
    float xd_error = xd_dot - vx;
    float yd_error = yd_dot - vy;
    float zd_error = zd_dot - vz;

//   Desired theta - current theta
    float e1 = desired_1 - theta1motor;
    float e2 = desired_2 - theta2motor;
    float e3 = desired_3 - theta3motor;

    // Integral approximation
    float integral1 = integral1_old + (e1 + e1_old) * dt / 2;
    float integral2 = integral2_old + (e2 + e2_old) * dt / 2;
```

```
    float integral3 = integral3_old + (e3 + e3_old) * dt / 2;

    // Prevents integral windup
    if (fabs(e1) > threshold1) {
        integral1 = 0;
        integral1_old = 0;
    }

    if (fabs(e2) > threshold2) {
            integral2 = 0;
            integral2_old  = 0;
    }

    if (fabs(e3) > threshold3) {
            integral3 = 0;
            integral3_old = 0;
    }

    //Mode = 0: Feed forward + acc;
    //Mode = 1: Feed forward + PD
    //Mode = 2: Task Space PD controller
    //Mode = 3: Simple Impedance Control
    int mode = 3;
    if (mode == 0){
        // acceleration of joint 2/3
            float D1 = p1;
            float D2 = -p3*sin(theta3motor-theta2motor);
            float D3 = -p3*sin(theta3motor-theta2motor);
            float D4 = p2;
            float C1 = 0;
            float C2 = -p3*cos(theta3motor-theta2motor)*Omega3;
            float C3 = p3*cos(theta3motor-theta2motor)*Omega2;
            float C4 = 0;
            float G1 = -p4*g*sin(theta2motor);
            float G2 = -p5*g*cos(theta3motor);


            a_theta2 = desired_doubledot_2 + kp2*(e2) + kd2*(desired_dot_2-Omega2);
            a_theta3 = desired_doubledot_2 + kp3*(e3) + kd3*(desired_dot_3-Omega3);

            *tau1 = J1*desired_doubledot_2 + kp1*(e1)+kd1*(desired_dot_2-Omega1);
            *tau2 = (D1*a_theta2+D2*a_theta2)+(C1*Omega2+C2*Omega3)+G1;
            *tau3 = (D3*a_theta2+D4*a_theta3)+(C3*Omega2+C4*Omega3)+G2;
    }
    else if(mode == 1){//Feed Forward + PD
        *tau1 = 0.0167 * desired_doubledot_1 + fkp1 * e1 + fkd1 * (desired_dot_1 - Omega1); // + ki1 * integral1;
        *tau2 = 0.03 * desired_doubledot_2 + fkp2 * e2 + fkd2 * (desired_dot_2 - Omega2); // + ki2 * integral2;
        *tau3 = 0.0128 * desired_doubledot_3 + fkp3 * e3 + fkd3 * (desired_dot_3 - Omega3);
    }
    else if(mode == 2){//Task Space PD controller
        float fx = kpx*(xd - x) + kdx*(xd_dot - vx);
        float fy = kpy*(yd - y) + kdy*(yd_dot - vy);
        float fz = kpz*(zd - z) + kdz*(zd_dot - vz);

        //Compute Jacobian Transpose
        // Jacobian Transpose
        cosq1 = cos(theta1motor);
```

```
        sinq1 = sin(theta1motor);
        cosq2 = cos(theta2motor);
        sinq2 = sin(theta2motor);
        cosq3 = cos(theta3motor);
        sinq3 = sin(theta3motor);
        JT_11 = -10*sinq1*(cosq3 + sinq2);
        JT_12 = 10*cosq1*(cosq3 + sinq2);
        JT_13 = 0;
        JT_21 = 10*cosq1*(cosq2 - sinq3);
        JT_22 = 10*sinq1*(cosq2 - sinq3);
        JT_23 = -10*(cosq3 + sinq2);
        JT_31 = -10*cosq1*sinq3;
        JT_32 = -10*sinq1*sinq3;
        JT_33 = -10*cosq3;
        //Simple Impedance Control
        float x_grav_comp = 0;
        float y_grav_comp = 0.0254*JT_23*fzcmd/kt;
        float z_grav_comp = 0.0254*JT_33*(fzcmd + grav_comp)/kt;
        *tau1 = JT_11*fx + JT_12*fy + x_grav_comp;
        *tau2 = JT_21*fx + JT_22*fy + JT_23*fz + y_grav_comp;
        *tau3 = JT_31*fx + JT_32*fy + JT_33*fz + z_grav_comp;

    }

    else if(mode == 3){//Simple Impedance Control
        //Lab 4 = Impedence Controls, all the variables are used here, provided in the starter code
        //Compute Jacobian Transpose
        // Jacobian Transpose

        //The first section of code defines any terms used in lab 4, primarily the matrix components. Because the majority of the terms below are sinusoidal, calculating them once per loop and then calling the saved value is far more efficient than calling a cosine or sine every time a term is used. In the torque equations, the forward kinematics, jacobian, and rotation matrices are defined and stored to be called later.
        //Saves the cos and sin values of the three joint angles so they don't have to be recalculated each time they're called during the function's run.

        cosq1 = cos(theta1motor);
        sinq1 = sin(theta1motor);
        cosq2 = cos(theta2motor);
        sinq2 = sin(theta2motor);
        cosq3 = cos(theta3motor);
        sinq3 = sin(theta3motor);

        //Jacobian Transpose for the CRS robot, separated into its constituent parts from matrix form
        JT_11 = -10*sinq1*(cosq3 + sinq2);
        JT_12 = 10*cosq1*(cosq3 + sinq2);
        JT_13 = 0;
        JT_21 = 10*cosq1*(cosq2 - sinq3);
        JT_22 = 10*sinq1*(cosq2 - sinq3);
        JT_23 = -10*(cosq3 + sinq2);
        JT_31 = -10*cosq1*sinq3;
        JT_32 = -10*sinq1*sinq3;
        JT_33 = -10*cosq3;

        //zxy Rotation and Transpose
```

```
        //The values of cos and sin of the rotation about each axis are stored so they don't have to be recalculated every time the function is called.
        cosz = cos(thetaz);
        sinz = sin(thetaz);
        cosx = cos(thetax);
        sinx = sin(thetax);
        cosy = cos(thetay);
        siny = sin(thetay);

        //The given equations for the rotation matrix's transpose, divided into its constituent parts
        RT11 = R11 = cosz*cosy-sinz*sinx*siny;
        RT21 = R12 = -sinz*cosx;
        RT31 = R13 = cosz*siny+sinz*sinx*cosy;
        RT12 = R21 = sinz*cosy+cosz*sinx*siny;
        RT22 = R22 = cosz*cosx;
        RT32 = R23 = sinz*siny-cosz*sinx*cosy;
        RT13 = R31 = -cosx*siny;
        RT23 = R32 = sinx;
        RT33 = R33 = cosx*cosy;

        ///The torque equations are calculated with the addition of a rotation matrix in this section of code. The rotation matrix's purpose is to change the direction in which a force can be applied to the end effector while keeping the trajectory in the world frame. MATLAB was used to multiply the matrices by hand.
        *tau1 = (JT_11*R11 + JT_12*R21 + JT_13*R31)*(Kdx_n*R11*xd_error + Kpx_n*R11*x_error + Kdx_n*R21*yd_error + Kpx_n*R21*y_error + Kdx_n*R31*zd_error + Kpx_n*R31*z_error) + (JT_11*R12 + JT_12*R22 + JT_13*R32)*(Kdy_n*R12*xd_error + Kpy_n*R12*x_error + Kdy_n*R22*yd_error + Kpy_n*R22*y_error + Kdy_n*R32*zd_error + Kpy_n*R32*z_error) + (JT_11*R13 + JT_12*R23 + JT_13*R33)*(Kdz_n*R13*xd_error + Kpz_n*R13*x_error + Kdz_n*R23*yd_error + Kpz_n*R23*y_error + Kdz_n*R33*zd_error + Kpz_n*R33*z_error);
        *tau2 = (JT_21*R11 + JT_22*R21 + JT_23*R31)*(Kdx_n*R11*xd_error + Kpx_n*R11*x_error + Kdx_n*R21*yd_error + Kpx_n*R21*y_error + Kdx_n*R31*zd_error + Kpx_n*R31*z_error) + (JT_21*R12 + JT_22*R22 + JT_23*R32)*(Kdy_n*R12*xd_error + Kpy_n*R12*x_error + Kdy_n*R22*yd_error + Kpy_n*R22*y_error + Kdy_n*R32*zd_error + Kpy_n*R32*z_error) + (JT_21*R13 + JT_22*R23 + JT_23*R33)*(Kdz_n*R13*xd_error + Kpz_n*R13*x_error + Kdz_n*R23*yd_error + Kpz_n*R23*y_error + Kdz_n*R33*zd_error + Kpz_n*R33*z_error);
        *tau3 = (JT_31*R11 + JT_32*R21 + JT_33*R31)*(Kdx_n*R11*xd_error + Kpx_n*R11*x_error + Kdx_n*R21*yd_error + Kpx_n*R21*y_error + Kdx_n*R31*zd_error + Kpx_n*R31*z_error) + (JT_31*R12 + JT_32*R22 + JT_33*R32)*(Kdy_n*R12*xd_error + Kpy_n*R12*x_error + Kdy_n*R22*yd_error + Kpy_n*R22*y_error + Kdy_n*R32*zd_error + Kpy_n*R32*z_error) + (JT_31*R13 + JT_32*R23 + JT_33*R33)*(Kdz_n*R13*xd_error + Kpz_n*R13*x_error + Kdz_n*R23*yd_error + Kpz_n*R23*y_error + Kdz_n*R33*zd_error + Kpz_n*R33*z_error);
    }

     //  Friction Compensation
    //If omega greater than 0.1 (Max Velocity) for joint 1
    if (Omega1 > 0.1) {
        *tau1 = *tau1 + 0.6*(Viscouspos1 * Omega1 + Coulombpos1) ;
        }
    //If omega greater than -0.1 (Min Velocity) for joint 1
    else if (Omega1 <-0.1) {
        *tau1 = *tau1 +0.6*(Viscousneg1 * Omega1 + Coulombneg1);
    }
    //If omega between min and max velocity for joint 1
    else {
        *tau1 = *tau1 + 0.6*(slope1*Omega1);
```

```
    }

    //If omega greater than 0.05(Max Velocity) for joint 2
    if (Omega2 > 0.05) {
        *tau2 = *tau2 + 0.6*(Viscouspos2 * Omega2 + Coulombpos2);
    }
    //If omega greater than -0.05 (Min Velocity) for joint 2
    else if (Omega2 <-0.05) {
        *tau2 = *tau2 + 0.6*(Viscousneg2 * Omega2 + Coulombneg2);
    }
    //If omega between min and max velocity for joint 2
    else {
        *tau2 = *tau2 + 0.6*(slope2*Omega2);
    }
    //If omega greater than -0.05 (Max Velocity) for joint 3
    if (Omega3 > 0.05) {
        *tau2 = *tau2 + 0.6*(Viscouspos3 * Omega3 + Coulombpos3) ;
    }
    //If omega greater than -0.05 (Min Velocity) for joint 3
    else if (Omega3 <-0.05) {
        *tau2 = *tau2 + 0.6*(Viscousneg3 * Omega3 + Coulombneg3);
    }
    //If omega between min and max velocity for joint 3
    else {
        *tau2 = *tau2 + 0.6*(slope3*Omega3);
    }

    //Prevents integral windup (Max: 5 Min: -5)
    if (*tau1 >= 5) {
        *tau1 = 5;
        integral1 = integral1_old;
    } else if (*tau1 < -5) {
        *tau1 = -5;
        integral1 = integral1_old;
    }

    if (*tau2 >= 5) {
        *tau2 = 5;
        integral2 = integral2_old;
    } else if (*tau2 < -5) {
        *tau2 = -5;
        integral2 = integral2_old;
    }

    if (*tau3 >= 5) {
        *tau3 = 5;
        integral3 = integral3_old;
    } else if (*tau3 < -5) {
        *tau3 = -5;
        integral3 = integral3_old;
    }

    e1_old = e1;
    e2_old = e2;
    e3_old = e3;
    integral1_old = integral1;
    integral2_old = integral2;
    integral3_old = integral3;


```

```
    // save past states
    if ((mycount%50)==0) {

        theta1array[arrayindex] = theta1motor;
        theta2array[arrayindex] = theta2motor;

        if (arrayindex >= 100) {
            arrayindex = 0;
        } else {
            arrayindex++;
        }

    }

    if ((mycount%500)==0) {
        if (whattoprint > 0.5) {
            serial_printf(&SerialA, "I love robotics\n\r");
        } else {
            printtheta1motor = theta1motor;
            printtheta2motor = theta2motor;
            printtheta3motor = theta3motor;
            SWI_post(&SWI_printf); //Using a SWI to fix SPI issue from sending too many floats.
        }

        GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Blink LED on Control Card
        GpioDataRegs.GPBTOGGLE.bit.GPIO60 = 1; // Blink LED on Emergency Stop Box
    }



    Simulink_PlotVar1 = desired_1;  //yellow
    Simulink_PlotVar2 = theta1motor; //blue
    Simulink_PlotVar3 = theta2motor; //orange
    Simulink_PlotVar4 = theta3motor; //green
    mycount++;
    counter++;
}

void printing(void){
    // Printing (theta1, theta2, theta3, , py, pz) and then on a new line (theta1_IK, theta2_IK, theta3_IK)
   serial_printf(&SerialA, "px: %.2f, py: %.2f, pz: %.2f \n\r", px, py, pz);
}


```