Samuel Caleb Martínez Hernández

Ing. Mecatrónica 8-A

UPZMG

Dinámica y Control de Robots

## Código

### *“serial_tx_cmdline.py”*

```
#!/usr/bin/Python (La versión correspondiente a Ubuntu)
import rospy
import sys
import serial
from std_msgs.msg import String
def tx_to_serial(device_name):
    print("Running serial_tx_cmdline node with device: " +
device_name)
    serial_timeout = 1
    rospy.init_node('serial_tx_cmdline', anonymous=False)
    pub = rospy.Publisher('serial_tx_cmdline', String,
queue_size=10)
    psoc_baud = 115200
    serial_port = serial.Serial(device_name, psoc_baud,
timeout=serial_timeout, exclusive=False)
    serial_port.reset_input_buffer()
    serial_port.reset_output_buffer()
    print("Opened port. Ctrl-Z to stop.")
    while not rospy.is_shutdown():
        try:
```

```python
            to_psoc = raw_input("Message to send over serial 
terminal: ")


            to_psoc += '\n'

            serial_port.write(to_psoc)

            pub.publish(to_psoc)

        except KeyboardInterrupt:

            rospy.signal_shutdown("Shutting down, Ctrl-Z 
received.")


if __name__ == '__main__':

    try:

        tx_to_serial(sys.argv[1])

    except rospy.ROSInterruptException:

        pass
```

## Código de PsoC 5LP

***"main.c"***

```c
#define PWM_CW_MAX 400
#define PWM_CW_MIN 315

#define PWM_STOP 300

#define PWM_CCW_MAX 285
#define PWM_CCW_MIN 200

// Tolerance to stop moving motor
#define TICKS_STOP_QD 500

// For transmitting strings with other variables substituted in,
// (note: re-using variable names since out-of-scope of
uart_helper_fcns.)
#define TRANSMIT_LENGTH 128

// Include both the UART helper functions and the header
// that has the global variables we need.
// note that both of these should have include guards in them already
// so it's safe to include them directly here.
#include <project.h>
```

```c
#include <math.h>
#include <stdlib.h>
#include "stdio.h"
#include "uart_helper_fcns.h"
#include "data_storage.h"

// for send some debugging messages
char transmit_buffer[TRANSMIT_LENGTH];

// constants of proportionality are integers.

//float Kp_qd = 1;
//float Kp_qd = 0.1;
float Kp_qd = 0.001;
//float Kp_qd = 0;

//float Ki_qd = 1;
//float Ki_qd = 0.00001;
float Ki_qd = 0.00001;

//float Kd_qd = 1;
//float Kd_qd = 0.0005;
float Kd_qd = 0.009;

// A set of local variables for the calculated PWM signals.
// These are *signed*, so cannot be directly used with WriteCompare.
static int32 pwm_controls[NUM_MOTORS] = {0, 0, 0, 0};

void move_motor_1() {
    // Proportional term
    error[0] = current_control[0] - (QuadDec_Motor1_GetCounter());

    // Integral term: discretized integration = addition (scaled.)
    // Note that we have to prevent integer overflow here.
    if((integral_error[0] + error[0] >= INT32_LOWERBOUND) &&
(integral_error[0] + error[0] <= INT32_UPPERBOUND)){
        integral_error[0] += error[0];
    }

    // Derivative term. discretized derivative = subtraction.
    deriv_error[0] = error[0] - prev_error[0];


    // Calculate the control input.
    // This automatically casts the integral control input to an int from
a float.
    pwm_controls[0] = error[0] * Kp_qd + integral_error[0] * Ki_qd +
deriv_error[0] * Kd_qd;

    // Shift by *300* (determined by PWM clock and period right now) to
put PWM values in correct range for VESC
    int32 pwm_control_0 = pwm_controls[0] + 300;


    // Apply the PWM value. Five options:
    // 1) If we're within tolerance of the target, turn off the PWM.
    // 2) REMOVED
```

```
    // 3) If not within tolerance, lower bound with PWM_MIN.
    // 4) REMOVED
    // 5) If not within tolerance, upper bound with PWM_MAX.
    // 6) If greater than STOP but less than minimum value to move CW,
apply PWM_CW_MIN
    // 7) If less than STOP but greater than maximum value to move CCW,
apply PWM_CCW_MAX
    // 8) If not in either range, apply control input

    //sprintf(transmit_buffer, "Current control: %ld\r\n",pwm_control_0);
    //UART_PutString(transmit_buffer);

    // 1) Is absolute encoder value within tolerance?
    if (abs(error[0]) < TICKS_STOP_QD){
        PWM_1_WriteCompare(PWM_STOP);
        motor_1 = 0;
        // minor hack for now:
        // reset the integral terms, so this is a "stopping point"
        integral_error[0] = 0;
        // reset error, so this is a "stopping point"
        error[0] = 0;
    }
    // Otherwise if havent reached tolerance, do 2-5.
    else {
        // 5) Check if upper bounded.
        if (pwm_control_0 > PWM_CW_MAX) {
            PWM_1_WriteCompare(PWM_CW_MAX);
        }
        // 3) Check if lower bounded.
        else if (pwm_control_0 < PWM_CCW_MIN) {
            PWM_1_WriteCompare(PWM_CCW_MIN);
        }
        // 6) Check if below lower bound for CW.
        else if ((pwm_control_0 <= PWM_CW_MIN) && (pwm_control_0 >
PWM_STOP)) {
            PWM_1_WriteCompare(PWM_CW_MIN);
        }
        // 7) Check if above upper bound for CCW.
        else if ((pwm_control_0 < PWM_STOP) && (pwm_control_0 >=
PWM_CCW_MAX)) {
            PWM_1_WriteCompare(PWM_CCW_MAX);
        }
        // 8) otherwise, we know we're within the min to max.
        else {
             // This, right here, is the actual application of our
control signal.
            PWM_1_WriteCompare(pwm_control_0);
        }
    }

    // Finally, set the stored value for the next iteration's error term.
    // It's safest to do this all the way at the end.
    prev_error[0] = error[0];
}

void move_motor_2() {
    // MOTOR 2
```

```
    // Proportional term
    error[1] = current_control[1] - QuadDec_Motor2_GetCounter();

    // Integral term: discretized integration = addition to sum (scaled.)
    // Note that we have to prevent buffer overflow here.
    if((integral_error[1] + error[1] >= INT32_LOWERBOUND) &&
(integral_error[1] + error[1] <= INT32_UPPERBOUND)){
        integral_error[1] += error[1];
    }

    // Derivative term. discretized derivative = subtraction.
    deriv_error[1] = error[1] - prev_error[1];

    // Calculate the control input.
    // This automatically casts the integral control input to an int from
a float.
    pwm_controls[1] = error[1] * Kp_qd + integral_error[1] * Ki_qd +
deriv_error[1] * Kd_qd;
    // Shift by *300* (determined by PWM clock and period right now) to
put PWM values in correct range for VESC
    int32 pwm_control_1 = pwm_controls[1] + 300;


    // Apply the PWM value. Five options:
    // 1) If we're within tolerance of the target, turn off the PWM.
    // 2) If not within tolerance, check if first application of control:
apply "init" to break static friction
    // 3) If not within tolerance, lower bound with PWM_MIN.
    // 4) If not within tolerance and input less than max, apply the
calculated input.
    // 5) If not within tolerance, upper bound with PWM_MAX.

        // 1) Is absolute encoder value within tolerance?
    if (abs(error[1]) < TICKS_STOP_QD){
        PWM_2_WriteCompare(PWM_STOP);
        motor_2 = 0;
        // minor hack for now:
        // reset the integral terms, so this is a "stopping point"
        integral_error[1] = 0;
        // reset error, so this is a "stopping point"
        error[1] = 0;
    }
    // Otherwise if havent reached tolerance, do 2-5.
    else {
        // 5) Check if upper bounded.
        if (pwm_control_1 > PWM_CW_MAX) {
            PWM_2_WriteCompare(PWM_CW_MAX);
        }
        // 3) Check if lower bounded.
        else if (pwm_control_1 < PWM_CCW_MIN) {
            PWM_2_WriteCompare(PWM_CCW_MIN);
        }
        // 6) Check if below lower bound for CW.
        else if ((pwm_control_1 <= PWM_CW_MIN) && (pwm_control_1 >
PWM_STOP)) {
            PWM_2_WriteCompare(PWM_CW_MIN);
```

```
        }
        // 7) Check if above upper bound for CCW.
        else if ((pwm_control_1 < PWM_STOP) && (pwm_control_1 >=
PWM_CCW_MAX)) {
            PWM_2_WriteCompare(PWM_CCW_MAX);
        }
        // 8) otherwise, we know we're within the min to max.
        else {
             // This, right here, is the actual application of our
control signal.
            PWM_2_WriteCompare(pwm_control_1);
        }
    }

    // Finally, set the stored value for the next iteration's error term.
    // It's safest to do this all the way at the end.
    prev_error[1] = error[1];
}

void move_motor_3() {
    // Proportional term
    error[2] = current_control[2] - QuadDec_Motor3_GetCounter();

    // Integral term: discretized integration = addition (scaled.)
    // Note that we have to prevent integer overflow here.
    if((integral_error[2] + error[2] >= INT32_LOWERBOUND) &&
(integral_error[2] + error[2] <= INT32_UPPERBOUND)){
        integral_error[2] += error[2];
    }

    // Derivative term. discretized derivative = subtraction.
    deriv_error[2] = error[2] - prev_error[2];

    // Calculate the control input.
    // This automatically casts the integral control input to an int from
a float.
    pwm_controls[2] = error[2] * Kp_qd + integral_error[2] * Ki_qd +
deriv_error[2] * Kd_qd;
    // Shift by *300* (determined by PWM clock and period right now) to
put PWM values in correct range for VESC
    int32 pwm_control_2 = pwm_controls[2] + 300;


    // Apply the PWM value. Five options:
    // 1) If we're within tolerance of the target, turn off the PWM.
    // 2) If not within tolerance, check if first application of control:
apply "init" to break static friction
    // 3) If not within tolerance, lower bound with PWM_MIN.
    // 4) If not within tolerance and input less than max, apply the
calculated input.
    // 5) If not within tolerance, upper bound with PWM_MAX.

        // 1) Is absolute encoder value within tolerance?
    if (abs(error[2]) < TICKS_STOP_QD){
        PWM_3_WriteCompare(PWM_STOP);
        motor_3 = 0;
        // minor hack for now:
```

```
        // reset the integral terms, so this is a "stopping point"
        integral_error[2] = 0;
        // reset error, so this is a "stopping point"
        error[2] = 0;
    }
    // Otherwise if havent reached tolerance, do 2-5.
    else {
        // 5) Check if upper bounded.
        if (pwm_control_2 > PWM_CW_MAX) {
            PWM_3_WriteCompare(PWM_CW_MAX);
        }
        // 3) Check if lower bounded.
        else if (pwm_control_2 < PWM_CCW_MIN) {
            PWM_3_WriteCompare(PWM_CCW_MIN);
        }
        // 6) Check if below lower bound for CW.
        else if ((pwm_control_2 <= PWM_CW_MIN) && (pwm_control_2 >
PWM_STOP)) {
            PWM_3_WriteCompare(PWM_CW_MIN);
        }
        // 7) Check if above upper bound for CCW.
        else if ((pwm_control_2 < PWM_STOP) && (pwm_control_2 >=
PWM_CCW_MAX)) {
            PWM_3_WriteCompare(PWM_CCW_MAX);
        }
        // 8) otherwise, we know we're within the min to max.
        else {
             // This, right here, is the actual application of our
control signal.
            PWM_3_WriteCompare(pwm_control_2);
        }
    }

    // Finally, set the stored value for the next iteration's error term.
    // It's safest to do this all the way at the end.
    prev_error[2] = error[2];
}

void move_motor_4() {
    // Proportional term
    error[3] = current_control[3] - QuadDec_Motor4_GetCounter();

    // Integral term: discretized integration = addition (scaled.)
    // Note that we have to prevent integer overflow here.
    if((integral_error[3] + error[3] >= INT32_LOWERBOUND) &&
(integral_error[3] + error[3] <= INT32_UPPERBOUND)){
        integral_error[3] += error[3];
    }

    // Derivative term. discretized derivative = subtraction.
    deriv_error[3] = error[3] - prev_error[3];

    // Calculate the control input.
    // This automatically casts the integral control input to an int from
a float.
    pwm_controls[3] = error[3] * Kp_qd + integral_error[3] * Ki_qd +
deriv_error[3] * Kd_qd;
```

```
        // Shift by *300* (determined by PWM clock and period right now)
to put PWM values in correct range for VESC
    int32 pwm_control_3 = pwm_controls[3] + 300;


    // Apply the PWM value. Five options:
    // 1) If we're within tolerance of the target, turn off the PWM.
    // 2) If not within tolerance, check if first application of control:
apply "init" to break static friction
    // 3) If not within tolerance, lower bound with PWM_MIN.
    // 4) If not within tolerance and input less than max, apply the
calculated input.
    // 5) If not within tolerance, upper bound with PWM_MAX.

        // 1) Is absolute encoder value within tolerance?
    if (abs(error[3]) < TICKS_STOP_QD){
        PWM_4_WriteCompare(PWM_STOP);
        motor_4 = 0;
        // minor hack for now:
        // reset the integral terms, so this is a "stopping point"
        integral_error[3] = 0;
        // reset error, so this is a "stopping point"
        error[3] = 0;
    }
    // Otherwise if havent reached tolerance, do 2-5.
    else {
        // 5) Check if upper bounded.
        if (pwm_control_3 > PWM_CW_MAX) {
            PWM_4_WriteCompare(PWM_CW_MAX);
        }
        // 3) Check if lower bounded.
        else if (pwm_control_3 < PWM_CCW_MIN) {
            PWM_4_WriteCompare(PWM_CCW_MIN);
        }
        // 6) Check if below lower bound for CW.
        else if ((pwm_control_3 <= PWM_CW_MIN) && (pwm_control_3 >
PWM_STOP)) {
            PWM_4_WriteCompare(PWM_CW_MIN);
        }
        // 7) Check if above upper bound for CCW.
        else if ((pwm_control_3 < PWM_STOP) && (pwm_control_3 >=
PWM_CCW_MAX)) {
            PWM_4_WriteCompare(PWM_CCW_MAX);
        }
        // 8) otherwise, we know we're within the min to max.
        else {
             // This, right here, is the actual application of our
control signal.
            PWM_4_WriteCompare(pwm_control_3);
        }
    }

    // Finally, set the stored value for the next iteration's error term.
    // It's safest to do this all the way at the end.
    prev_error[3] = error[3];
}
```

```c
CY_ISR(timer_handler) {
    if (tensioning == 1) {
        if (fabs(tension_control) == 1) {
            move_motor_1();
        }
        else if (fabs(tension_control) == 2) {
            move_motor_2();
        }
        else if (fabs(tension_control) == 3) {
            move_motor_3();
        }
        else if (fabs(tension_control) == 4) {
            move_motor_4();
        }
    }

    if (controller_status == 1) {
        move_motor_1();
        move_motor_2();
        move_motor_3();
        move_motor_4();

    }
    Timer_ReadStatusRegister();
}

int main(void) {

    // Enable interrupts for the chip
    CyGlobalIntEnable;
    __enable_irq();

    // Start the interrupt handlers / service routines for each
interrupt:
    // UART, main control loop, encoder counting.
    // These are found in the corresponding helper files (declarations in
.h, implementations in .c)
    isr_UART_StartEx(Interrupt_Handler_UART_Receive);
    isr_Timer_StartEx(timer_handler);

    // For the quadrature (encoder) hardware components
    QuadDec_Motor1_Start();
    QuadDec_Motor2_Start();
    QuadDec_Motor3_Start();
    QuadDec_Motor4_Start();

    PWM_1_Start();
    PWM_1_WriteCompare(0);
    PWM_2_Start();
    PWM_2_WriteCompare(0);
    PWM_3_Start();
    PWM_3_WriteCompare(0);
    PWM_4_Start();
    PWM_4_WriteCompare(0);

    Timer_Start();
    UART_Start();
```

```c
    // Print a welcome message. Comes from uart_helper_fcns.
    UART_Welcome_Message();

    for(;;)
    {
        // Nothing to do. Entirely interrupt driven! Hooray!
    }
}

                            /* [] END OF FILE */
```