```
#pragma config(Sensor, in1,    left_line_sensor, sensorLin
#pragma config(Sensor, in2,    right_line_sensor, sensorLi
#pragma config(Sensor, in3,    claw_potent,    sensorPoten
#pragma config(Sensor, dgtl1,  right_encoder,  sensorQuadE
#pragma config(Sensor, dgtl3,  left_encoder,   sensorQuadE
#pragma config(Motor,  port1,           claw_1,        tmo
#pragma config(Motor,  port2,           right_front,   tmo
#pragma config(Motor,  port3,           left_front,    tmo
#pragma config(Motor,  port4,           arm_left_1,    tmo
#pragma config(Motor,  port5,           arm_left_2,    tmo
#pragma config(Motor,  port6,           arm_right_1,   tmo
#pragma config(Motor,  port7,           arm_right_2,   tmo
#pragma config(Motor,  port8,           right_back_middle,
#pragma config(Motor,  port9,           left_back_middle,
#pragma config(Motor,  port10,          claw_2,        tmo
//*!!Code automatically generated by 'ROBOTC' configuratio

#include "robotc_base_include.c"

// utilitys for numbers
// unfortunately we do not have the safety of C++ template

#define max(x, a) (x > a ? x : a)
#define min(x, a) (x < a ? x : a)
#define clamp(x, a, b) (min(b, max(a, x)))

// These values are exposed here to allow helper functions
float requested_left = 0.0;
float requested_right = 0.0;

float current_left = 0.0;
float current_right = 0.0;

float integral_left;
float integral_right;

float last_error_left;
float last_error_right;
```

```
float max_speed = 60.0;

void get_requested_delta(float *left, float *right) {
    int r = (vexRT[Ch1] - vexRT[Ch3]);
    int l = (vexRT[Ch1] + vexRT[Ch3]);

    *left  = abs(l) > 20 ? l : 0;
    *right = abs(r) > 20 ? r : 0;
}

float step_pid(float constant_p,
               float constant_i,
               float constant_d,
               float requested,
               float current_value,
               float *last_error,
               float *integral) {
    float error = current_value - requested;

    float derivitive = error - *last_error;
    *last_error      = error;

    if (constant_i != 0 && abs(error) < 50) {
        *integral = *integral + error;
    } else {
        *integral = 0;
    }

    return constant_p * error +
           constant_i * *integral +
           constant_d * derivitive;
}

task pid_loop() {
    // clear out the encoder
    SensorValue[left_encoder]  = 0;
    SensorValue[right_encoder] = 0;
```

```
    current_left  = 0;
    current_right = 0;

    requested_left  = 0;
    requested_right = 0;

    last_error_left  = 0;
    last_error_right = 0;

    integral_left  = 0;
    integral_right = 0;

    bool reset = true;

    while (true) {
        current_left  = SensorValue[left_encoder];
        current_right = SensorValue[right_encoder];

        // Since pid is only used in autonomous this code
        /*
        // get new usercontrol values here
        float delta_requested_left  = 0;
        float delta_requested_right = 0;

        get_requested_delta(&delta_requested_left, &delta_


        if (delta_requested_left == 0 && delta_requested_r
            if (reset) {
                requested_left             = 0;
                requested_right            = 0;
                SensorValue[left_encoder]  = 0;
                SensorValue[right_encoder] = 0;

                last_error_left  = 0;
                last_error_right = 0;
```

```
                reset = false;
            }
        } else {
            requested_left += delta_requested_left / 20;
            requested_right += delta_requested_right / 20;

            reset = true;
        }
        */

        float new_left  = step_pid(1, 0.0, 1,
                                   requested_left,
                                   current_left,
                                   &last_error_left,
                                   &integral_left);
        float new_right = step_pid(1, 0.0, 1,
                                   requested_right,
                                   current_right,
                                   &last_error_right,
                                   &integral_right);

        if (abs(new_left) < 20) new_left = 0;
        if (abs(new_right) < 20) new_right = 0;

        writeDebugStreamLine("r: %.2f %.2f c: %.2f %.2f e:


        new_left  = clamp(new_left, -max_speed, max_speed)
        new_right = clamp(new_right, -max_speed, max_speed

        motor[right_front] = -new_right;
        motor[right_back_middle] = -new_right;

        motor[left_front] = -new_left;
        motor[left_back_middle] = -new_left;
    }
}
```

```
void init() {
    stop_tasks_between_mode = true;
}

void forward(float amount) {
    requested_right -= amount;
    requested_left += amount;
}
void turn(float amount) {
    requested_right += amount;
    requested_left += amount;
}

bool at_dest(float threshold) {
    if(abs(requested_left - current_left) < threshold &&
           abs(requested_right - current_right) < threshol
        integral_left = 0;
        integral_right = 0;
        return true;
    }

    return false;
}

void wait_for_dest(int timeout_ticks) {
    int curr = timeout_ticks;
    while(!at_dest(20) && curr >= 0) {
        sleep(25);
        --curr;
        //clearLCDLine(0);
        //displayLCDNumber(0, 0, curr, 2);
    }

    if(curr == 0) {
        displayLCDString(0, 0, "MISSED!!");
        requested_left = current_left;
        requested_right = current_right;
        last_error_left = 0;
```

```
        last_error_right = 0;
    }
}

// positive for up, negative for down
void arm(int power) {
    motor[arm_left_1] = -power;
    motor[arm_left_2] = power;

    motor[arm_right_1] = -power;
    motor[arm_right_2] = power;
}

void claw(int power) {
    motor[claw_1] = power;
    motor[claw_2] = -power;
}

task auton() {
    // Make sure the pid loop is restarted properly
    // This will get rid of residue
    stopTask(pid_loop);
    startTask(pid_loop);

    // Clear other sensors that we might use in the future
    SensorValue[claw_potent] = 0;
    SensorValue[left_line_sensor] = 0;
    SensorValue[right_line_sensor] = 0;

    // potential for new
    int auton_option = 0;

    bLCDBacklight = true;
    clearLCDLine(0);

    switch(auton_option) {
    case 0:
        // Stack a cone onto the static goal
```

```
        // move cone forward and then drive backwards
        forward(100);
        wait_for_dest(200);
        forward(-100);
        wait_for_dest(200);

        // Lift the arm up so that the claw can be closed
        arm(127);
        sleep(300);
        arm(0);
        // claw close
        claw(127);
        sleep(500);
        // arm down
        claw(0);
        arm(-127);
        sleep(700)
        arm(0);

        // drive forwards so that the claw is around the c
        forward(100);
        wait_for_dest(1000);

        // claw close
        claw(127);
        sleep(800);
        claw(30);

        // arm up
        arm(127);
        sleep(800);
        arm(0);

        // forward
        forward(250);
        wait_for_dest(500);
```

```
        // wait for the rocking to subside before lowering
        sleep(1300);

        // arm down
        arm(-127);
        sleep(400);

        // claw open
        claw(-127);
        sleep(400);
        arm(0);
        claw(0);

        // drive back
        forward(-200);
        wait_for_dest(100);

        // close the claw so that it doesnt get stuck on t
        claw(127);
        sleep(400);
        claw(0);

        // turn first away from the mobile goal
        turn(200);
        wait_for_dest(300);
        // and move forwards
        forward(250);
        wait_for_dest(300);

        // then turn inwards towards the centre of the pit
        turn(-150);
        wait_for_dest(300);
        // and forwards
        forward(600);
        wait_for_dest(400);

        break;
    }
```

```
    // Stop pid and stop all motors to prevent drifting ar
    stopTask(pid_loop);
    all_motors_off();
}

// states for the claw
enum {
    claw_open = 0,
    claw_closed = 1,
}

int requested_claw = claw_closed;

task user_control() {
    SensorValue[claw_potent] = 0;

    SensorValue[left_line_sensor] = 0;
    SensorValue[right_line_sensor] = 0;

    sleep(500);

    while (true) {
        // usercontrol movement
        // we dont run pid in drivercontrol as the driftin
        // movement
        int r = (vexRT[Ch3] - vexRT[Ch1]);
        int l = (vexRT[Ch3] + vexRT[Ch1]);

        // account for deadzone
        if (abs(l) < 20) l = 0;
        if (abs(r) < 20) r = 0;

        motor[right_front] = -r;
        motor[right_back_middle] = -r;

        motor[left_front] = l;
        motor[left_back_middle] = l;
```

```
        // move the arm
        int arm_sign = 0;
        if(vexRT[Btn5U] == true) {
            arm_sign = 1;
        } else if(vexRT[Btn5D] == true) {
            arm_sign = -1;
        }

        motor[arm_left_1] = 127 * -arm_sign;
        motor[arm_left_2] = 127 * arm_sign;

        motor[arm_right_1] = 127 * -arm_sign;
        motor[arm_right_2] = 127 * arm_sign;

        // move the claw
        if(vexRT[Btn6U] == true) {
            requested_claw = claw_closed;
        } else if(vexRT[Btn6D] == true) {
            requested_claw = claw_open;
        }

        if(requested_claw == claw_open) {
            if(SensorValue[claw_potent] > 2900) { // close
                motor[claw_1] = -127;
            } else if(SensorValue[claw_potent] > 2800) { /
                motor[claw_1] = -30;
            } else {
                motor[claw_1] = 0;
            }
        } else if(requested_claw == claw_closed) {
            if(SensorValue[claw_potent] < 3000) { // open
                motor[claw_1] = 127;
            } else if(SensorValue[claw_potent] < 3100) { /
                motor[claw_1] = 30;
            } else if(SensorValue[claw_potent] < 3250) { /
                motor[claw_1] = 30;
            } else {
```

```
                motor[claw_1] = 0; // we missed...
            }
        }


        /*
        // Test code for the line sensors
        if(SensorValue[left_line_sensor] < 2700 && SensorV
            bLCDBacklight = true;
        } else {
            bLCDBacklight = false;
        }
        */
    }

}
```