#include "main.h"

Controller master (CONTROLLER\_MASTER); *//master controller*

*//V5 Components*

Motor leftBase1(1, MOTOR\_GEARSET\_18, 0, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 1*

Motor leftBase2(2, MOTOR\_GEARSET\_18, 1, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 2*

Motor rightBase1(3, MOTOR\_GEARSET\_18, 0, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 3*

Motor rightBase2(4, MOTOR\_GEARSET\_18, 1, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 4*

Motor lift1(6, MOTOR\_GEARSET\_36, 0, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 6*

Motor lift2(7, MOTOR\_GEARSET\_36, 1, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 7*

Motor claw1(9, MOTOR\_GEARSET\_18, 0, E\_MOTOR\_ENCODER\_DEGREES); *//normal motor plugged into port 9*

*//Legacy Components*

ADIAnalogIn liftPot(2); *//potentiometer for lift in port 2*

ADIDigitalIn liftLimit(1); *//limit switch for lift in port 1*

ADIUltrasonic cubeSensor(3, 4); *//sonar sensor with ping in port 3 and echo in port 4*

ADIEncoder aftEnc(5, 6, 0); *//left wheel encoder in ports 5 and 6*

ADIEncoder yawEnc(7, 8, 0); *//perpendicular wheel encoder in ports 7 and 8*

#include "main.h"

int sgn(float x) { *//inputs a double and outputs its sign (+ or -) as an int*

return (x > 0) - (x < 0);

}

PID initPID(bool useP, bool useI, bool useD, float kP, float kI, float kD) { *//gives a created PID struct its values*

return { 0, 0, 0, 0, kP, kI, kD, useP, useI, useD };

}

float runPID(PID \*pid) {

pid->integral += pid->error; *//increment integral by current error*

pid->derivative = pid->error - pid->prevError; *//set derivative to the delta error*

pid->prevError = pid->error; *//set prevError to current error to be used for next iteration*

return (pid->error \* pid->kP \* (float)pid->useP) + (pid->integral \* pid->kI \* (float)pid->useI) + (pid->derivative \* pid->kD \* (float)pid->useD); *//calculate final value*

}

float findTheta(float x, float y) {

return atan2(y, x) \* 180 / 3.1415926535; *//returns the angle of the joystick in degrees*

}

float joyValRemap(float joyVal) {

float percentVal = 0.0;

if((float)joyVal <= -3.0) { *//threshold prevents accidental movement at low values*

percentVal = pow(-pow((float)joyVal, 2) - 6 \* (float)joyVal + 18327.7, 0.5) - 154.414; *//uses a circle to maximize the versatility of slow movement and still allow for fast movement*

}

else if((float)joyVal >= 3.0) {

percentVal = -pow(-pow((float)joyVal, 2) + 6 \* (float)joyVal + 18327.7, 0.5) + 154.414; *//uses the same circle but negated*

}

else {

percentVal = 0.0;

}

return percentVal;

}

float leftBaseRemap(float r, float theta) { *//remaps the joystick value to send the correct voltage to the left base motors*

theta += 45; *//shifts the angle by 45 degrees to accomodate for the angled motors on the x-drive*

float leftBaseVal;

if(theta == 0 || theta == 180) *//if the value is purely for the left base*

leftBaseVal = r;

else if(abs(cos(theta \* M\_PI / 180)) >= abs(sin(theta \* M\_PI / 180)))

leftBaseVal = r \* sgn(cos(theta \* M\_PI / 180)); *//makes the value equal to the x portion of the joystick vector*

else if(abs(cos(theta \* M\_PI / 180)) < abs(sin(theta \* M\_PI / 180)))

leftBaseVal = r \* cos(theta \* M\_PI / 180) / abs(sin(theta \* M\_PI / 180)); *//makes the value equal to a fraction of the joystick vector*

return leftBaseVal;

}

float rightBaseRemap(float r, float theta) {*//remaps the joystick value to send the correct voltage to the right base motors*

theta += 45; *//shifts the angle by 45 degrees to accomodate for the angled motors on the x-drive*

float rightBaseVal;

if(theta == 0 || theta == 180) *//if the value is purely for the left base*

rightBaseVal = r;

else if(abs(sin(theta \* M\_PI / 180)) >= abs(cos(theta \* M\_PI / 180)))

rightBaseVal = r \* sgn(sin(theta \* M\_PI / 180)); *//makes the value equal to the y portion of the joystick vector*

else if(abs(sin(theta \* M\_PI / 180)) < abs(cos(theta \* M\_PI / 180)))

rightBaseVal = r \* sin(theta \* M\_PI / 180) / abs(cos(theta \* M\_PI / 180)); *//makes the value equal to a fraction of the joystick vector*

return rightBaseVal;

}

void moveStraight(float distance, float theta, int time) { *//PID for moving the base in any direction in a straight line*

theta += 45;

float distVal, diffVal, highVal, lowVal;

float leftVal, rightVal;

PID dist = initPID(1, 0, 1, 0.4, 0, 1); *//kP = 0.4, kD = 1*

PID diff = initPID(1, 0, 0, 0.1, 0, 1); *//kP = 0.1*

resetBaseMotorEnc();

for(int i = 0; i < time; i ++) {

dist.error = distance - pow(pow(getLeftBaseEnc(), 2) + pow(getRightBaseEnc(), 2), 0.5); *//uses the hypotenuse of the right triangle to find absolute distance traveled*

distVal = runPID(&dist) > 190 ? 190 : runPID(&dist); *//limits the distVal*

diff.error = -getRightBaseEnc() \* cos(theta \* M\_PI / 180) + getLeftBaseEnc() \* sin(theta \* M\_PI / 180); *//uses the ideal distances vs the actual distances of the motors to fix the angle of movement*

diffVal = runPID(&diff); *//updates the diffVal*

highVal = distVal + diffVal > 200 ? 200 : distVal + diffVal; *//finds the high and low values*

lowVal = distVal - diffVal > 200 ? 200 : distVal - diffVal;

if(abs(cos(theta \* M\_PI / 180)) > abs(sin(theta \* M\_PI / 180))) { *//makes one side run at full speed and the other side run at a fraction of that speed*

leftVal = lowVal \* sgn(cos(theta \* M\_PI / 180));

rightVal = sin(theta \* M\_PI / 180) / abs(cos(theta \* M\_PI / 180)) \* highVal;

}

else if(abs(cos(theta \* M\_PI / 180)) < abs(sin(theta \* M\_PI / 180))) {

leftVal = cos(theta \* M\_PI / 180) / abs(sin(theta \* M\_PI / 180)) \* lowVal;

rightVal = highVal \* sgn(sin(theta \* M\_PI / 180));

}

std::cout << "distance: " << pow(pow(getLeftBaseEnc(), 2) + pow(getRightBaseEnc(), 2), 0.5) << " | dist error: " << dist.error << " | diff error: " << diff.error << " | leftVal: " << leftVal << " | rightVal: " << rightVal << " | time: " << i << "\n";

runLeftBase1(leftVal);

runLeftBase2(leftVal);

runRightBase1(rightVal);

runRightBase2(rightVal);

delay(1);

}

runLeftBase1(0);

runLeftBase2(0);

runRightBase1(0);

runRightBase2(0);

}

void turn(float theta, int time) { *//uses a PID control loop to turn to a desired relative angle*

float setPoint = -theta \* 5.4; *//modifies angle to fit the motor encoder values*

float turnVal;

PID turn = initPID(1, 0, 0, 0.45, 0, 0); *//kP = 0.45*

resetBaseMotorEnc();

for(int i = 0; i < time; i++) { *//time limit for the PID to run*

turn.error = setPoint - getBaseMotorEnc(); *//updates the error for the turn PID*

turnVal = runPID(&turn); *//updates the value for the motors, reference misc.cpp*

runLeftBase1(turnVal);

runLeftBase2(-turnVal);

runRightBase1(turnVal);

runRightBase2(-turnVal);

*//std::cout << leftBase1.get\_position() << " | " << leftBase2.get\_position() << " | " << rightBase1.get\_position() << " | " << rightBase2.get\_position() << " | " << turnVal << "\n";*

std::cout << getBaseMotorEnc() << " | " << setPoint << " | " << turnVal << "\n";

delay(1);

}

}

bool manual = false, manualUsed = false, shiftUp = false, shiftDown = false, reset = false;

int height = 0, liftSetPoint;

int aboveCube[5] = {1110, 1560, 1910, 2320, 2930}; *//array of potentiometer values that correspond with heights above certain number of cubes*

void liftCtrl(void\* param) {

PID lift = initPID(1, 1, 0, 0.2, 0.0001, 0); *//kP = 0.2, kI = 0.0001*

liftSetPoint = aboveCube[height];

float liftVal;

while(true) { *//runs continuously the whole match*

if(!manual) {

if(reset) {

reset = false;

height = 0; *//changes the height of the lift to above 0 cubes*

liftSetPoint = aboveCube[height];

}

if(!manualUsed) { *//if the lift is at a certain value of the aboveCube array*

if(shiftUp) {

shiftUp = false;

if(height < 4) *//if the lift is still able to move up*

height++;

liftSetPoint = aboveCube[height]; *//update the liftSetPoint*

}

else if(shiftDown) {

shiftDown = false;

if(height > 0) *//if the lift is still able to move down*

height--;

liftSetPoint = aboveCube[height]; *//update the liftSetPoint*

}

}

else if(manualUsed) { *//if the lift needs to move to the nearest value of the aboveCube array*

if(shiftDown) {

for(int i = 0; i < 5; i++) { *//finds the closest height that is still less than the current value*

if(liftSetPoint < aboveCube[i])

height = i;

break;

}

liftSetPoint = aboveCube[height];

manualUsed = false;

}

else if(shiftUp) {

for(int i = 4; i >= 0; i--) { *//finds the closest height that is still greater than the current height*

if(liftSetPoint > aboveCube[i])

height = i;

break;

}

liftSetPoint = aboveCube[height];

manualUsed = false;

}

}

if(lift2.is\_over\_temp() || lift2.is\_over\_current()) *//prevents the motors from burning out or overheating*

            lift2.set\_voltage\_limit(0);

        else

            lift2.set\_voltage\_limit(12000);

lift.error = liftSetPoint - liftPot.get\_value(); *//updates the error for the liftPID*

liftVal = runPID(&lift); *//updates the value for the liftPID, reference misc.cpp*

runLift(-liftVal);

std::cout << liftSetPoint << " | " << liftPot.get\_value() << " | " << lift.error << "\n";

}

else if(manual) { *//allows the driver to control the lift completely*

liftSetPoint = liftPot.get\_value();

if(lift2.is\_over\_temp() || lift2.is\_over\_current())

            lift2.set\_voltage\_limit(0); *//prevents overheating or burning out of motors*

        else

            lift2.set\_voltage\_limit(12000);

shiftUp = false;

shiftDown = false;

manualUsed = true;

}

delay(1);

}

}

void autostack(int cubes) {

int sonarDist = 0;

if(!manualUsed) {

int heightF = height - cubes;

while(!cubeSensor.get\_value() == 0) { *//uses a P loop to move to the correct distance from the cube*

runLeftBase1((sonarDist - cubeSensor.get\_value()) \* 0);

runLeftBase2((sonarDist - cubeSensor.get\_value()) \* 0);

runRightBase1((sonarDist - cubeSensor.get\_value()) \* 0);

runRightBase2((sonarDist - cubeSensor.get\_value()) \* 0);

}

runClaw(-100); *//opens claw*

delay(300);

liftSetPoint = aboveCube[heightF >= 0 ? heightF : 0]; *//checks if the lift is high enough above the next cube to stack*

while(abs(liftPot.get\_value() - liftSetPoint) > 10) { *//runs the P loop again while the lift is reaching to correct height*

if(!cubeSensor.get\_value() == 0) {

runLeftBase1((sonarDist - cubeSensor.get\_value()) \* 0);

runLeftBase2((sonarDist - cubeSensor.get\_value()) \* 0);

runRightBase1((sonarDist - cubeSensor.get\_value()) \* 0);

runRightBase2((sonarDist - cubeSensor.get\_value()) \* 0);

}

}

runClaw(100); *//closes the claw on the whole stack*

delay(300);

}

}

void opcontrol() {

    std::uint\_least32\_t now = millis();

    float r, theta, leftTransVal, rightTransVal, turnVal;

    bool shiftUpAtck = true, shiftDownAtck = true, resetAtck = true, resetLift = false;

    while(true) {

*//std::cout << getBaseMotorEnc() << "\n";*

        r = pow(pow(master.get\_analog(E\_CONTROLLER\_ANALOG\_LEFT\_X), 2) + pow(master.get\_analog(E\_CONTROLLER\_ANALOG\_LEFT\_Y), 2), 0.5) > 127 ? 127 : pow(pow(master.get\_analog(E\_CONTROLLER\_ANALOG\_LEFT\_X), 2) + pow(master.get\_analog(E\_CONTROLLER\_ANALOG\_LEFT\_Y), 2), 0.5); *//finds the absolute value of the joystick regardless of angle*

        r = joyValRemap(r); *//remaps the joystick value to a more optimized array of values*

        theta = findTheta(master.get\_analog(E\_CONTROLLER\_ANALOG\_LEFT\_X), master.get\_analog(E\_CONTROLLER\_ANALOG\_LEFT\_Y));

        leftTransVal = 0.75 \* leftBaseRemap(r, theta); *//uses trigonometry to find the value the left motors should run at*

        rightTransVal = 0.75 \* rightBaseRemap(r, theta); *//uses trigonometry to find the value the right motors should run at*

        turnVal = joyValRemap(master.get\_analog(E\_CONTROLLER\_ANALOG\_RIGHT\_X)); *//uses the right joystick's value to turn*

        turnVal \*= 3 / 4; *//limits the voltage for turning to 75% of max voltage*

        runLeftBase1(leftTransVal + turnVal);

        runLeftBase2(leftTransVal - turnVal);

        runRightBase1(rightTransVal + turnVal);

        runRightBase2(rightTransVal - turnVal);

        if(master.get\_digital(E\_CONTROLLER\_DIGITAL\_L1) && !master.get\_digital(E\_CONTROLLER\_DIGITAL\_R1)) {

            manual = true;

            runLift(100);

        }

        else if(master.get\_digital(E\_CONTROLLER\_DIGITAL\_R1) && !master.get\_digital(E\_CONTROLLER\_DIGITAL\_L1) && !liftLimit.get\_value()) {

            manual = true;

            runLift(-100);

        }

        else { *//if R1 and L1 are both pressed or not pressed*

            manual = false;

            runLift(0);

        }

        if(!master.get\_digital(E\_CONTROLLER\_DIGITAL\_UP)) *//switch to allow holding the button to toggle once*

            resetAtck = false;

        else if(!resetAtck) {

            reset = true;

            resetAtck = true;

        }

        if(reset && !liftLimit.get\_value()) *//if reset is desired, the lift will move down until the button is triggered*

            runLift(-100);

        else if(reset && liftLimit.get\_value()) {

            runLift(0);

            reset = false;

        }

*/\*if(!master.get\_digital(E\_CONTROLLER\_DIGITAL\_L1))*

*shiftUpAtck = false;*

*else if(!shiftUpAtck) {*

*shiftUp = true;*

*shiftUpAtck = true;*

*}*

*if(!master.get\_digital(E\_CONTROLLER\_DIGITAL\_R1))*

*shiftDownAtck = false;*

*else if(!shiftDownAtck) {*

*shiftDown = true;*

*shiftDownAtck = true;*

*}\*/*

        if(master.get\_digital(E\_CONTROLLER\_DIGITAL\_R2) && !master.get\_digital(E\_CONTROLLER\_DIGITAL\_L2))

            runClaw(100);

        else if(master.get\_digital(E\_CONTROLLER\_DIGITAL\_L2) && !master.get\_digital(E\_CONTROLLER\_DIGITAL\_R2))

            runClaw(-100);

        else

            runClaw(0);

        if(leftBase1.is\_over\_temp() || leftBase1.is\_over\_current())

            leftBase1.set\_voltage\_limit(0);

        else

            leftBase1.set\_voltage\_limit(12000);

        if(leftBase2.is\_over\_temp() || leftBase2.is\_over\_current())

            leftBase2.set\_voltage\_limit(0);

        else

            leftBase2.set\_voltage\_limit(12000);

        if(rightBase1.is\_over\_temp() || rightBase1.is\_over\_current())

            rightBase1.set\_voltage\_limit(0);

        else

            rightBase1.set\_voltage\_limit(12000);

        if(rightBase2.is\_over\_temp() || rightBase2.is\_over\_current())

            rightBase2.set\_voltage\_limit(0);

        else

            rightBase2.set\_voltage\_limit(12000);

        if(claw1.is\_over\_temp() || claw1.is\_over\_current())

            claw1.set\_voltage\_limit(0);

        else

            claw1.set\_voltage\_limit(12000);

        std::cout << liftLimit.get\_value() << "\n";

        Task::delay\_until(&now, 10);

    }

}