```
//
//
// All code for multi-bot is contained here (minus auton routines)
// Created by Sam Burton
// Contributors:
//
// To Do:
// Test everything
// Calibrate arm/wrist/flipper/flywheel/drive variables & #defines
// Calculate flywheel speed table
// Choose controller mapping & change #defines - 1st version done
// Write vision sensor code (next comp)
// Write IR code
// Write LiDAR code (next comp)
//
//


#include "main.h"


using namespace pros;


#define TURBO E_MOTOR_GEARSET_06
#define SPEED E_MOTOR_GEARSET_18
#define TORQUE E_MOTOR_GEARSET_36
#define RAW E_MOTOR_ENCODER_DEGREES

#define FLYWHEEL 1
#define ARM 2

#define HIGH 1
#define MIDDLE 2



///////////////////////////////////////////////////////////////////////
 /////
// Controller Mapping
// #defines for controller buttons      // CONTROLLER BUTTON

#define BTN_TOGGLE DIGITAL_DOWN
#define BTN_ABORT DIGITAL_UP
#define BTN_CHOOSE_AUTON DIGITAL_X

// Flywheel
#define BTN_FIRE_HIGH DIGITAL_L1
#define BTN_FIRE_LOW DIGITAL_L2
#define BTN_INTAKE_IN DIGITAL_R1
#define BTN_INTAKE_OUT DIGITAL_R2
#define BTN_FIRE_BOTH DIGITAL_B
#define BTN_TOGGLE_COAST DIGITAL_A
#define BTN_TOGGLE_INTAKE DIGITAL_Y

// Arm
#define BTN_WRIST_UP DIGITAL_RIGHT
#define BTN_WRIST_DOWN DIGITAL_LEFT
#define BTN_ARM_UP DIGITAL_X
#define BTN_ARM_DOWN DIGITAL_B
#define BTN_FLIPPER_LEFT DIGITAL_Y
#define BTN_FLIPPER_RIGHT DIGITAL_A
#define BTN_FLIP DIGITAL_R1
#define BTN_WRIST DIGITAL_R2
#define BTN_ARM_HIGH DIGITAL_L1
#define BTN_ARM_LOW DIGITAL_L2



// #defines for arm positions           // CALCULATE
#define FLIP_POS1 1                     // 1:1 Ratio, 0°
#define FLIP_POS2 180                   // 1:1 Ratio, 180°
#define WRIST_BACK_POS (200*3)          // 1:3 Ratio, 200°
#define WRIST_BACKWARD_DROP_POS (-45*3) // 1:3 Ratio, -65°
#define WRIST_FORWARD_POS (105*3)       // 1:3 Ratio, 100°
#define WRIST_FORWARD_DROP_POS (35*3)   // 1:3 Ratio, 65°
#define WRIST_VERTICAL_POS 1            // 1:3 Ratio, 0°
#define ARM_POS_HIGH (135*5)            // 1:5 Ratio, 135°
#define ARM_POS_LOW (85*5)              // 1:5 Ratio, 90°
#define ARM_POS_DOWN 1                  // 1:5 Ratio, 0°

// #defines for tuning
// Flywheel
#define FLYWHEEL_AIM_RANGE 5            // fire ball when within x
 degrees of flag

// Arm - higher value is more gentle seek
#define armSeekRate 1
#define wristSeekRate 0.25
#define wristSeekSlow 8
#define flipperSeekRate 1

// Gyro Stuff
#define CSCALE 0.9876     //Clockwise scale adjustments to counteract
 rotation errors
#define ASCALE 0.9486    //Anti-clockwise scale adjustments to
 counteract rotation errors
#define GYRO_PORT 1

#include "BallBotAutons.h"
```

```
Controller controller(E_CONTROLLER_MASTER);     // Controller object

//////////////////////////////////////////////////////////////////////
 /////////
// Motors
// Motor name(port, gearing, reversed?, encoder_units);
// Drive Motors
Motor drive_left_1(1, SPEED, 0, RAW);
Motor drive_left_2(2, SPEED, 1, RAW);
Motor drive_left_3(3, SPEED, 0, RAW);
Motor drive_right_1(4, SPEED, 1, RAW);
Motor drive_right_2(5, SPEED, 0, RAW);
Motor drive_right_3(6, SPEED, 1, RAW);
// Flywheel Motors
Motor flywheel_1(17, TURBO, 1, RAW);
Motor flywheel_2(16, TURBO ,0, RAW);
// Intake
Motor intake_in(18, SPEED, 1, RAW);
Motor intake_out(19, SPEED, 1, RAW);
// Arm Motors
Motor arm_1(7, SPEED, 0, RAW);
Motor arm_2(20, SPEED, 1, RAW);
// Flipper
Motor wrist(11, SPEED, 1, RAW);
Motor flip(14, SPEED, 0, RAW);
// Gyro Sensor
ADIGyro sensor_gyro(1, GYRO_PORT);
// Inner Intake Button
ADIDigitalIn bumper_in (2);

///////////////////////////////////////////////////////////////
// Drive tuning variables (CALIBRATE)
// Drive
double deadZone = 10;
double ticksPerTile = 640;
double minForward = 40;
double driveLerp = 0.1;
// Turn
double turnAccepted = 1;
double pulsePause = 10;
double pulseTime = 5;
double minSpeed = 25;
double maxTurn = 127;
double turnRate = 150;
double ticksPerDegree = 90;
// Tracking
double trackingTicksPerTile = ticksPerTile;
double trackingTicksPerDegree = ticksPerDegree;

// Drive control variables
// Drive
double autoMode = DRIVEMODE_USER;
bool autonComplete = false;
double autoTime = 0;
bool speedOverride = false;
double rightRunSpeed = 0;
double leftRunSpeed = 0;
bool drivingToPos = false;
double autoTimeOut = 0;
double targetDistance = 0;
double autoSpeed = 0;
bool usingGyro = true;
double currentDist = 0;
double recordedTime = 0;
double recordedDistLeft = 0;
double recordedDistRight = 0;
double lastRightEnc = 0;
double lastLeftEnc = 0;
// Turn
double targetDirection = 0;
double turnMode = 0;
double direction = 0;
// Position Tracking
double targetX = 0;
double targetY = 0;
double targetS = 0;
double yPosition = 0;
double xPosition = 0;


// Auton Routines
extern int autonSelect;
extern double defaultAuton[];
extern double redAuton[];
extern double blueAuton[];


double gyroDirection = 0;
bool hasInitialised = false;

// Declare and initialize any global flags we need:
// Control mode
int controlMode = FLYWHEEL;

// Auton Control
double* autonCommand = &defaultAuton[0];    // default auto routine
bool nextCommand = true;

// For Flywheel
int autoFireState = -1;          // -1 for neutral, 1 for
 'aim&spin&fire', 2 for 'spin & fire', 3 for 'fire!'
```

```
int targetFlag = 1;             // 1 for low, 2 for high, 3 for high
 then low

// For Intake
bool forceIntake = false;
double intakeSpeedOuter = 0;    // speed for outer intake
double intakeSpeedInner = 0;    // speed for inner intake
int runTillBall = 0;            // 0 = nothing, 1 = run till 1 ball
 in, 2 = run for two balls

// For cap mechanisms
double armSeek = -1;
double wristSeek = -1;
double flipperSeek = -1;

double armPos = 0;
double flipperPos = 0;
double wristPos = 0;

double armOffset = 0;
double flipperOffset = 0;
double wristOffset = 0;

int stackTarget = -1;
int stackStep = -1;

// Array for flywheel speed lookup'
// Distance (tiles), low flag speed (rpm), high flag speed (rpm)
// For each distance we record flywheel speeds needed for hitting
 high/low flags

#define FLYWHEEL_SPEED_RANGE 30          // fire ball when within x
 rpm of target speed
#define flywheelSlowSpeed 50
#define flywheelFastSpeed 127

double flyWheelDefaultSpeed = 80;    // set speed for fixed-dist
 fireing
double flyWheelSpeeds[2][3] = {                 // CALIBRATE & add
 more
    {-100, 0, 0},   // to catch errors
    {0, 400, 500},
};
int flyWheelSpeedsDefinition = 4;   // number of entries
double autoFireTimeout = -1;


void setArmPos(double pos) {
    // set all motor encoders to 0
    arm_1.tare_position();
    arm_2.tare_position();
    // set position
    armOffset = pos;
    armPos = pos;
}
void setFlipperPos(double pos) {
    // set all motor encoders to 0
    flip.tare_position();
    // set position
    flipperOffset = pos;
    flipperPos = pos;
}
void setWristPos(double pos) {
    // set all motor encoders to 0
    wrist.tare_position();
    // set position
    wristOffset = pos;
    wristPos = pos;
}


void initAll() {        // called when robot activates & start of
 auton
    if (!hasInitialised) {
        // First time / manual init...
        // eg. calibrate gyro
        controller.print(0,0,"Calibrating");
        sensor_gyro = ADIGyro(1, GYRO_PORT);
        pros::delay(3000);
    }
    hasInitialised = true;
    // Every time init...
    // eg. tare arm position
    arm_1.tare_position();
    arm_2.tare_position();
    wrist.tare_position();
    flip.tare_position();
    controller.print(0,0,"                          ");
}


double processEntry() {
    autonCommand++;
    return *autonCommand;
}


//////////////////////////////////////////////////////////////////////
 ///////////
// Gyro Stuff (To move into own file)
// gyroDirection will be updated with 'more accurate' gyro value
gyros gyro1,gyro2;
```

```
short gyroinit=0;

void resetGyro()
{
    sensor_gyro.reset();
    gyro1.truedir=0;
    gyro2.truedir=0;
    gyro1.last=0;
    gyro2.last=0;
    gyroDirection=0;
}

void setGyro(double dir)
{
    gyro1.truedir=dir;
    gyro2.truedir=dir;
    //    gyro1.last=dir;
    //    gyro2.last=dir;
    gyroDirection=dir;
}

void checkGyro(gyros *gyro)
{
    float currentGyro;                                          //gyro
     position
    float tempAngle;                                             //
     temporary angle variable

    currentGyro=sensor_gyro.get_value();    //read hardware gyro value
    tempAngle=currentGyro-gyro->last;            //what is the delta
     change in the gyro this loop?
    tempAngle=-tempAngle;
    gyro->last=currentGyro;                                 //store
     current gyro value for comparison next time

    if (abs(tempAngle)>2500)                //huge delta so probably
     wrapped
    {
        if (tempAngle>0) {tempAngle=tempAngle-3600;}    //get true
         delta change taking...
        else {tempAngle=tempAngle+3600;}                //...into
         account wrap
    }
    //tempAngle now holds correct delta change between old and new
     gyro angles

    //        if (abs(ang2)<JITTER)
     {SensorValue[gyro]=lastgyro;}    //tiny delta change so overwrite
     hardware gyro with lastgyro (removes jitter)
    //        else
    //        if (abs(ang2)>JITTER)
    if (tempAngle>0)    //anti-clockwise rotation
    {
        gyro->truedir=gyro->truedir+(tempAngle*gyro->ascale);
         //update ?tempDir? if anti-clockwise rotation and scale by
         Anti-Clockwise scale
        if (gyro->truedir<0) {gyro->truedir=gyro->truedir+3600;}
         //wrap
    }
    else
    {
        gyro->truedir=gyro->truedir+(tempAngle*gyro->cscale);
         //update ?tempDir? if clockwise rotation and scale by
         Clockwise scale
        if (gyro->truedir>=3600) {gyro->truedir=gyro->truedir-3600;}
         //wrap
    }
    //truedir ends up as positive float >=0 and <3600 to be used in
     rest of code
}

void run_gyro(void* params)
{
    if (gyroinit==0)
    {
        gyroinit=1;

        gyro1.port=GYRO_PORT;
        gyro1.truedir=0;
        gyro1.last=sensor_gyro.get_value();
        gyro1.ascale=ASCALE;
        gyro1.cscale=CSCALE;

        gyro2.port=GYRO_PORT;//////SHOULD BE GYROB
        gyro2.truedir=0;
        gyro2.last=sensor_gyro.get_value();
        gyro2.ascale=ASCALE;
        gyro2.cscale=CSCALE;

        gyroDirection=0;
    }

    while(true)
    {
        checkGyro(&gyro1);
        checkGyro(&gyro2);

        //        gyroDirection=gyro1.truedir;

        //find average of the two angles
```

```
        if (gyro1.truedir>gyro2.truedir)
        {
            float tempAngle=gyro1.truedir;
            gyro1.truedir=gyro2.truedir;
            gyro2.truedir=tempAngle;            //swap order so that
             gyro2 always larger
        }
        if (gyro2.truedir-gyro1.truedir>1800) gyro2.truedir-
         =3600;    //big difference so fix wrap
        gyroDirection=(gyro2.truedir+gyro1.truedir)/2;        //
         average them
        if (gyroDirection<0) gyroDirection+=3600;        //unwrap
         negative case
        pros::delay(20);
    }
}
//////////////////////////////////////////////////////////////////////
 ////////////
// End of gyro stuff
//

double getLeftEnc() {
    return ( drive_left_1.get_position() + drive_left_2.get_position()
     + drive_left_3.get_position() ) / 3;
}

double getRightEnc() {
    return ( drive_right_1.get_position() +
     drive_right_2.get_position() + drive_right_3.get_position() ) /
     3;
}


//////////////////////////////////////////////////////////////////////
 //////
// Drive auton functions
//
void driveStop() {
    autoTime = 0;
    autoMode = DRIVEMODE_USER;
    autoSpeed = 0;
    speedOverride = false;
    drivingToPos = false;
}

void driveTime(double s, double d, double t) {
    // speed, direction, distance, time
    autoSpeed = s;
    autoMode = DRIVEMODE_TIME;
    autoTimeOut = t*1000;
    targetDirection = d;
    recordedTime = pros::millis();
}

void driveDist(double s, double dir, double dist, double t = 10) {
    // speed, direction, distance, timeout
    autoSpeed = s;
    targetDirection = dir;
    autoMode = DRIVEMODE_DIST;
    autoTimeOut = t*1000;
    recordedTime = pros::millis();
    recordedDistLeft = getLeftEnc();
    recordedDistRight = getRightEnc();

    if (s > 0) {
        targetDistance = (dist * ticksPerTile) + (recordedDistRight +
         recordedDistLeft)/2;
    }
    else {
        targetDistance = (-dist * ticksPerTile) + (recordedDistRight +
         recordedDistLeft)/2;
    }
}

void driveCustom(double s, double d, double t = 10) {
    // speed, direction, timeout
    recordedTime = pros::millis();
    autoSpeed = s;
    autoMode = DRIVEMODE_CUSTOM;
    autoTimeOut = t*1000;
    targetDirection = d;
}

void turnTo(double a, double t = -1) {
    // angle, timeout
    recordedTime = pros::millis();
    targetDirection = a;
    autoTimeOut = t*1000;
    autoMode = DRIVEMODE_TURN;
    turnMode = TURNMODE_GYRO;
}

void turnRelative(double a, double t = -1) {
    // angle, timeout
    recordedTime = pros::millis();
    targetDirection = direction + a;
    autoTimeOut = t*1000;
    autoMode = DRIVEMODE_TURN;
    turnMode = TURNMODE_GYRO;
}

void turnRelativeEncoder(double a, double t = -1) {
```

```
    // angle, timeout
    recordedTime = pros::millis();
    targetDirection = direction + a;
    autoTimeOut = t*1000;
    autoMode = DRIVEMODE_TURN;
    turnMode = TURNMODE_ENCODER;
    recordedDistLeft = getLeftEnc();
    recordedDistRight = getRightEnc();
    targetDistance = (a * ticksPerDegree) + (recordedDistRight -
     recordedDistLeft)/2;
}


///////////////////////////////////////////////////////////////////////
 ////////////
// Position Tracking stuff
//

void setPosition(double x, double y, double d) {
    xPosition = x;
    // lastRightEnc = getRightEnc();
    yPosition = y;
    // lastLeftEnc = getLeftEnc();
    direction = d;
}

void trackPosition() {
    double leftEnc = getLeftEnc();      // get encoder values from
     motors
    double rightEnc = getRightEnc();
    double leftDiff = leftEnc - lastLeftEnc;   // Find encoder changes
    double rightDiff = rightEnc- lastRightEnc;

    double angleChange = (rightDiff - leftDiff)/2;  // Find angle
     change
    angleChange *= trackingTicksPerDegree;

    double distChange = (leftDiff + rightDiff)/2;   // Find lin. dist
     change
    distChange *= trackingTicksPerTile;

    direction += angleChange;   // Find cumulative direction
    xPosition += distChange * cos(direction * M_PI / 180);  // find
     cumulative xPos
    yPosition += distChange * sin(direction * M_PI / 180);  // find
     cumulative yPoS

    lastLeftEnc = leftEnc;  // remember last values for next
     comparison
    lastRightEnc = rightEnc;
}
```

```
void turnToPoint(double x, double y, double t = -1) {
    double dx = x - xPosition;
    double dy = y - yPosition;
    double dir = atan(dy/dx);
    turnTo(dir);
}
void driveTo(double s, double x, double y, double t = 10) {
    targetX = x;
    targetY = y;
    targetS = s;
    double dx = x - xPosition;
    double dy = y - yPosition;
    double dir = atan(dy/dx);
    double dist = hypot(x,y);
    driveDist(s, dir, dist, t);
    drivingToPos = true;
}

///////////////////////////////////////////////////////////////////////
 ////////////
// Drive task
// Interprets user input & auton commands and sends to drive motors

void run_drive(void* params) {

    double currentTime = 0;
    double leftPower = 0;
    double rightPower = 0;
    double leftSpeed = 0;
    double rightSpeed = 0;

    double lastAngle = 0;
    double turnPulse = 0;

    double slewRate = 2;

    int turnGoodCount = 0;

    while (true) {

        //trackPosition();        // keep track of where we are on the
         field        // CHANGE

        if (usingGyro) {
            direction = gyroDirection/10;  // gyroDirection is updated
             by gyro code, direction is used by drive code
        }
        else {
            // maybe using compass/encoders?
            // direction = compassDirection
```

```
}

// This is where the fun begins

double forward = 0;
double turn = 0;

// Calculate useful information
currentTime = pros::millis();           // current time to
 determine if timed out

// find where encoders are right now
double currentDistLeft = getLeftEnc();
double currentDistRight = getRightEnc();
currentDist = (currentDistRight + currentDistLeft)/2;

if (controller.get_digital(BTN_ABORT)) {    // if user wants
 to abort, stop auton move
    autoMode = DRIVEMODE_USER;
}

// auto functions
if (autoMode != DRIVEMODE_USER) {   // If auton is asking for
 drive to move

    if (drivingToPos) {         // keep calculating new angle
     & distance to stay on-target
        // Must write position tracking algorythm first
        driveTo(targetS, targetX, targetY);
    }

    forward = autoSpeed;        // autoSpeed is speed asked
     for, forward will be sent to drive motors

    if (autoMode == DRIVEMODE_TURN)  {  // if we are only
     turning, make translational speed 0
        forward = 0;
        autoSpeed = 0;
    }

    if (autoMode == DRIVEMODE_DIST) {   // If auto move should
     end with a distance
        double slowDown = (targetDistance - currentDist) /
         (0.75 * ticksPerTile);

        forward *= slowDown;

        if (autoSpeed > 0 && forward < minForward) forward =
         minForward;
        if (autoSpeed < 0 && forward > minForward) forward = -
         minForward;
```

```
        if (forward > 127) forward = 127;   // Cap max and min
         speed
        if (forward < -127) forward = -127;

        // Terminate contition for distance
        if (autoSpeed > 0) {
            if (currentDist > targetDistance) autonComplete =
             true;
        }
        else {
            if (currentDist < targetDistance) autonComplete =
             true;
        }
    }

    if (currentTime > autoTimeOut + recordedTime &&
     autoTimeOut > 0) {     // If auton move has timed out,
     stop driving
        autonComplete = true;
        std::cout << "Time Out - ";
    }

    // Turn code
    double driveMag = autoSpeed;
    double seek = targetDirection;
    double angle = 0;

    if (turnMode == TURNMODE_GYRO) {
        angle = seek - direction;
    }
    else if (turnMode == TURNMODE_ENCODER) {
        angle = (recordedDistRight - recordedDistLeft)/2;
        angle -= (currentDistRight - currentDistLeft)/2;
        angle /= ticksPerDegree;
    }

    if (angle < 0) angle += 360;
    if (angle > 180) angle -= 360;

    angle /= (2 * turnRate);
    angle *= 127;
    if (driveMag < minSpeed) {
        angle *= 2;
    }

    if (angle < -maxTurn) angle = maxTurn;
    if (angle > maxTurn) angle = maxTurn;

    if (driveMag > minSpeed) {
        if (angle < 0) {
```

```
        if (angle > -2) {
            angle = 0;
        }
        else if (angle > -4) {
            angle = -4;
        }
    }
    else {
        if (angle < 2) {
            angle = 0;
        }
        else if (angle < 4) {
            angle = 4;
        }
    }
}
else {
    turn = angle;
    angle = abs(angle);
    if (angle < minSpeed) {
        if (((lastAngle > 0) && (turn < 0)) || ((lastAngle
         < 0) && (turn > 0))) {
            angle = 0;
        }
        else {
            if (angle > minSpeed/5) {
                angle = minSpeed;
            }
            else {
                turnPulse++;
                if (turnPulse < pulseTime) {
                    angle = minSpeed;
                }
                else {
                    angle = 1;
                    if (turnPulse > pulsePause) {
                        turnPulse = 0;
                    }
                }
            }
        }
    }
    if (turn < 0) angle *= -1;
}

turn = angle;

if (autoSpeed == 0 || autoMode == DRIVEMODE_TURN) {
    if (abs(direction - targetDirection) < turnAccepted) {
        turnGoodCount++;
        if (turnGoodCount > 3)
                autonComplete = true;
        }
        else {
            turnGoodCount = 0;
        }
    }

    lastAngle = angle;

}
// Auto-move is complete, so stop moving
if (autonComplete) {
    autonComplete = false;
    autoMode = DRIVEMODE_USER;
    forward = 0;
    turn = 0;
    autoSpeed = 0;
    drivingToPos = false;
    nextCommand = true;
    std::cout << "Drive Move Done: " << currentTime <<
     std::endl;
}

// User controls
if (autoMode == DRIVEMODE_USER) {

    // Tank controls
    leftSpeed = controller.get_analog(ANALOG_LEFT_Y);
    rightSpeed = controller.get_analog(ANALOG_RIGHT_Y);

    if (abs(leftSpeed) < deadZone) leftSpeed = 0;
    if (abs(rightSpeed) < deadZone) rightSpeed = 0;
}
else {
    leftSpeed = forward - turn;
    rightSpeed = forward + turn;
}

// Constant-speed override
if (speedOverride) {
    leftSpeed = leftRunSpeed;
    rightSpeed = rightRunSpeed;
}

// dampen motors so they don't spike current
rightPower = rightPower + ( (rightSpeed - rightPower) /
 slewRate );
leftPower = leftPower + ( (leftSpeed - leftPower) /
 slewRate );

// std::cout << "gyro: " << gyroDirection << std::endl;
```

```
        // Send speeds to drive motors
        drive_left_1.move_voltage(leftPower * 12000 / 127);
        drive_left_2.move_voltage(leftPower * 12000 / 127);
        drive_left_3.move_voltage(leftPower * 12000 / 127);
        drive_right_1.move_voltage(rightPower * 12000 / 127);
        drive_right_2.move_voltage(rightPower * 12000 / 127);
        drive_right_3.move_voltage(rightPower * 12000 / 127);

        pros::delay(10);   // don't hog cpu
    }
}


///////////////////////////////////////////////////////////////////////
 //////////////////////////////
// Flywheel
// Read flywheel motors to get its speed
//
double getFlywheelSpeed() {
    return (flywheel_1.get_actual_velocity() +
     flywheel_2.get_actual_velocity() ) / 2;
}
// Read serial input for IR sensors and Lidar distance - NEEDS
 IMPLEMENTATION
bool getInnerSensor() {
    return bumper_in.get_value();
}
bool getOuterSensor() {
    return true;
}
double getDistance() {
    return 0;
}
// Read vision sensor to get angle needed to turn
double getRelativeAngle() {

}

void run_flywheel(void* params) {
    // Declare any local variables
    bool ballIsIn = false;
    bool ballWasIn = false;
    bool justToggledAutoBall = false;
    bool coast = false;
    bool toggledCoast = false;

    while (true) {

        // Set intake motor speeds to 0
        if (!forceIntake) {
            intakeSpeedInner = 0;
            intakeSpeedOuter = 0;
        }

        // keep flywheel at default speed
        double targetSpeed = flyWheelDefaultSpeed;
        if (!coast) targetSpeed = 0;

        ballIsIn = getInnerSensor();

        if (autoFireState != -1) {  // Auto fire

            // Move flipper out of way
            if (flipperPos > (FLIP_POS1 + FLIP_POS2)/2) {
                flipperSeek = FLIP_POS2;
            }
            else {
                flipperSeek = FLIP_POS1;
            }

            // Check vision sensor to determine necessary turn
            double relativeAngle = 0;
            if (autoFireState <= 1) {
                // Read sensor and find relative angle
                relativeAngle = getRelativeAngle();

            }

            // Check lidar / ultrasonic for distance
            double distance = -1;
            if (autoFireState <= 2) {
                // Read sensor and find distance
                distance = getDistance();

            }

            // Lookup distance in flywheelSpeeds table, & interpolate
             to find speed
            targetSpeed = flyWheelDefaultSpeed;
            if (autoFireState <= 2 && distance != -1) {
                int index = -1;
                for (int i = 1; i < flyWheelSpeedsDefinition; i++) {
                    // Look for speed too high
                    if (flyWheelSpeeds[i][0] >= distance) {
                        index = i;
                        break;
                    }
                }
                if (index == -1) {
                    // Further than furthest in table
```

```
        if (targetFlag == 1) targetSpeed =
         flyWheelSpeeds[flyWheelSpeedsDefinition-1][1];
        else targetSpeed =
         flyWheelSpeeds[flyWheelSpeedsDefinition-1][2];
    }
    else {
        // Interpolate for correct speed
        // find how similar distance is to each value
        double distDiff = (distance -
         flyWheelSpeeds[index-1][0]);
        distDiff /= (flyWheelSpeeds[index][0] -
         flyWheelSpeeds[index-1][0]);
        double speedDiff;
        // Find how similar speed should be to each value
         & set target speed
        if (targetFlag == 1) {
            speedDiff = (flyWheelSpeeds[index][1] -
             flyWheelSpeeds[index-1][1]);
            targetSpeed = speedDiff*distDiff +
             flyWheelSpeeds[index-1][1];
        }
        else {
            speedDiff = (flyWheelSpeeds[index][2] -
             flyWheelSpeeds[index-1][2]);
            targetSpeed = speedDiff*distDiff +
             flyWheelSpeeds[index-1][2];
        }
    }
}

// Read vision sensor & ask drive to turn appropriately
if (abs(relativeAngle) > 0)
 turnRelative(relativeAngle,autoFireTimeout);

bool fireBall = false;
// Check current speed of flywheel & if aimed
if ( (abs(getFlywheelSpeed() - targetSpeed) <
 FLYWHEEL_SPEED_RANGE) && (abs(relativeAngle) <
 FLYWHEEL_AIM_RANGE) ) {
    // Set flag for fireing ball
    fireBall = true;
}

// Check if ball is in ready position
// read sensors to check if ball is in


if (!ballIsIn) {    // Ball is not close yet
    intakeSpeedOuter = 127;
    intakeSpeedInner = 127;
}
    if (fireBall) {         // aimed and running correct speed
        // Run intake motor
        intakeSpeedInner = 127;
        intakeSpeedOuter = 100;
    }
    if (ballWasIn && !ballIsIn) {   // ball has left
        // Clear flags
        fireBall = false;
        if (targetFlag == 3) {  // wanted to shoot high then
         low
            targetFlag = 1;
        }
        else {
            autoFireState = -1;
            targetSpeed = flyWheelDefaultSpeed;
            driveStop();
        }
    }

}   // end of auto-fire


// Check controller buttons...
// Set flags for preset flywheel speeds & auto-aim-fire
// If manual intake buttons pressed, override intake speeds
if (controlMode == FLYWHEEL) {
    if (controller.get_digital(BTN_FIRE_LOW)) { // auto fire
     low
        autoFireState = 1;
        autoFireTimeout = -1;
        targetFlag = 1;
    }
    if (controller.get_digital(BTN_FIRE_HIGH)) { // auto fire
     high
        autoFireState = 1;
        targetFlag = 2;
        autoFireTimeout = -1;
    }
    if (controller.get_digital(BTN_FIRE_BOTH)) { // auto fire
     both
        autoFireState = 1;
        targetFlag = 3;
        autoFireTimeout = -1;
    }
    /*if (controller.get_digital(BTN_FIRE_PRESET)) { // auto
     fire preset
     autoFireState = 3;
     autoFireTimeout = -1;
```

```
        }*/

        if (controller.get_digital(BTN_INTAKE_IN)) { // manual run
         intake in
            intakeSpeedInner = 127;
            intakeSpeedOuter = 127;
            runTillBall = 0;
            forceIntake = false;
        }
        if (controller.get_digital(BTN_INTAKE_OUT)) { // manual
         run intake out
            intakeSpeedInner = -127;
            intakeSpeedOuter = -127;
            runTillBall = 0;
            forceIntake = false;
        }
        if (controller.get_digital(BTN_TOGGLE_INTAKE)) { // toggle
         auto ball intake
            if (!justToggledAutoBall) {
                if (runTillBall) runTillBall = 0; else runTillBall
                 = 2;
            }
            justToggledAutoBall = true;
        }
        else {
            justToggledAutoBall = false;
        }
        if (controller.get_digital(BTN_TOGGLE_COAST)) {
            if (!toggledCoast) {
                coast = !coast;
            }
            toggledCoast = true;
        }
        else {
            toggledCoast = false;
        }
    }
    if (controller.get_digital(BTN_ABORT)) {     // cancel auto
     functions
        autoFireState = -1;
        runTillBall = 0;
        forceIntake = false;
    }

    if (runTillBall) {
        if (!getInnerSensor()) {     // ball is not all the way in
            intakeSpeedOuter = 127;
            intakeSpeedInner = 127;
        }
        else if (!getOuterSensor() && (runTillBall == 2)) {   // 1
         ball is in, but not 2
                intakeSpeedOuter = 127;
            }
        }


        // Math for the flywheel
        double flywheelCurrSpeed = 0;
        double flywheelSpeed = 0;
        flywheelCurrSpeed = ( flywheel_1.get_actual_velocity() +
         flywheel_2.get_actual_velocity() ) / 2;


        if (targetSpeed > 0) {
            if (flywheelCurrSpeed > targetSpeed) {  // Too fast
                flywheelSpeed = flywheelSlowSpeed;  // So run slow
            }
            if (flywheelCurrSpeed <= targetSpeed) { // Too slow
                flywheelSpeed = flywheelFastSpeed;  // So run fast
            }
        }

        if (targetSpeed == flyWheelDefaultSpeed) {
            flywheelSpeed = flyWheelDefaultSpeed;
        }

        // Set motors on flywheel
        flywheel_1.move_voltage(flywheelSpeed * 12000 / 127);
        flywheel_2.move_voltage(flywheelSpeed * 12000 / 127);

        // Send speeds to intake motors
        intake_in.move_voltage(intakeSpeedInner*12000 / 127);
        intake_out.move_voltage(intakeSpeedOuter*12000 / 127);

        // Remember ball info for fireing
        ballWasIn = ballIsIn;


        pros::delay(20);   // don't hog cpu
    }
}

void run_arm(void* params) {
    bool justFlipped = false;
    bool justShifted = false;
    bool shifted = false;
    bool justToggledMode = false;
    bool justArmToggled = false;
    bool justWristToggled = false;
    bool slowSeek = false;

    while (true) {
```

```
double armSpeed = 0;             // Start with zero speeds
double wristSpeed = 0;
double flipperSpeed = 0;

flipperPos = flip.get_position();       // Find current
 positions
wristPos = -wrist.get_position();
armPos = (arm_1.get_position() + arm_2.get_position()) / 2;

// std::cout << "F: " << flipperPos << " W: " << wristPos << "
 A: " << armPos << std::endl;

// If we want to stack something, follow the steps
switch (stackStep) {
    case 1:

        break;
    case 2:

        break;
    default:
        stackStep = -1;
        break;
}

// Read button toggle between flywheel & arm control
if (controller.get_digital(BTN_TOGGLE)) {
    if (!justToggledMode) {
        controller.rumble(".");
        if (controlMode == FLYWHEEL) {
            controlMode = ARM;
        }
        else if (controlMode == ARM) {
            controlMode = FLYWHEEL;
        }

    }
    justToggledMode = true;
}
else {
    justToggledMode = false;
}



// Check controller inputs
if (controlMode == ARM) {

    if (controller.get_digital(BTN_FLIP)) {     // Auto flip
     (180°)
        stackStep = -1;
        if (!justFlipped) {
            if (flipperPos > (FLIP_POS1 + FLIP_POS2)/2) {
                flipperSeek = FLIP_POS1;
            }
            else {
                flipperSeek = FLIP_POS2;
            }
        }
        justFlipped = true;
    }
    else {
        justFlipped = false;
    }


    // Manual Overrides
    if (controller.get_digital(BTN_ARM_DOWN)) {
        armSpeed = -100;
        armSeek = -1;
        stackStep = -1;
    }
    if (controller.get_digital(BTN_ARM_UP)) {
        armSpeed = 100;
        armSeek = -1;
        stackStep = -1;
    }
    if (controller.get_digital(BTN_WRIST_DOWN)) {
        wristSpeed = -100;
        wristSeek = -1;
        stackStep = -1;
    }
    if (controller.get_digital(BTN_WRIST_UP)) {
        wristSpeed = 100;
        wristSeek = -1;
        stackStep = -1;
    }
    if (controller.get_digital(BTN_FLIPPER_LEFT)) {
        flipperSpeed = -25;
        flipperSeek = -1;
        stackStep = -1;
    }
    if (controller.get_digital(BTN_FLIPPER_RIGHT)) {
        flipperSpeed = 25;
        flipperSeek = -1;
        stackStep = -1;
    }

    if (controller.get_digital(BTN_WRIST)) {
        if (!justWristToggled) {
            if (wristSeek != WRIST_VERTICAL_POS) {
```

```
                wristSeek = WRIST_VERTICAL_POS;
            }
            else {
                slowSeek = true;
                if (armSeek == ARM_POS_LOW) {
                    wristSeek = WRIST_FORWARD_DROP_POS;
                }
                else if (armSeek == ARM_POS_HIGH) {
                    wristSeek = WRIST_BACKWARD_DROP_POS;
                }
                else {
                    slowSeek = false;
                    wristSeek = WRIST_FORWARD_POS;
                }
            }
        }
        justWristToggled = true;
    }
    else {
        justWristToggled = false;
    }
    if (controller.get_digital(BTN_ARM_HIGH)) {
        slowSeek = false;
        if (!justArmToggled) {
            if (armSeek == ARM_POS_HIGH) armSeek =
             ARM_POS_DOWN;
            else armSeek = ARM_POS_HIGH;
        }
        justArmToggled = true;
    }
    else if (controller.get_digital(BTN_ARM_LOW)) {
        slowSeek = false;
        if (!justArmToggled) {
            if (armSeek == ARM_POS_LOW) armSeek =
             ARM_POS_DOWN;
            else armSeek = ARM_POS_LOW;
        }
        justArmToggled = true;
    }
    else {
        justArmToggled = false;
    }

}
if (controller.get_digital(BTN_ABORT)) {                //
 Stop all auton functions!
    wristSeek = -1;
    armSeek = -1;
    flipperSeek = -1;
    stackStep = -1;
}
```

```
        // If we need to seek, then tell the arm, wrist, and flipper
         (lerp code)
        if (armSeek > 0) {
            armSpeed = (armSeek - armPos) / armSeekRate;
            if (armSpeed > 100) armSpeed = 100;
            if (armSpeed < -100) armSpeed = -100;
        }
        if (wristSeek != -1) {

            double actualWristSeek = wristSeek + ( armPos * 3 / 5 );

            if (actualWristSeek < 0) actualWristSeek = 0;
            if (actualWristSeek > 800) actualWristSeek = 800;

            double wSR = 1;
            if (slowSeek) wSR = wristSeekSlow;

            wristSpeed = -(actualWristSeek - wristPos) /
             (wristSeekRate * wSR);
            if (wristSpeed > 100) wristSpeed = 100;
            if (wristSpeed < -100) wristSpeed = -100;
        }
        if (flipperSeek > 0) {
            flipperSpeed = (flipperSeek - flipperPos) /
             flipperSeekRate;
            if (flipperSpeed > 100) flipperSpeed = 100;
            if (flipperSpeed < -100) flipperSpeed = -100;
        }


        // Finally, send values to motors
        flip.move_voltage(flipperSpeed * 12000 / 127);
        wrist.move_voltage(wristSpeed * 12000 / 127);
        arm_1.move_voltage(armSpeed * 12000 / 127);
        arm_2.move_voltage(armSpeed * 12000 / 127);

        pros::delay(20);   // don't hog cpu
    }
}


void run_auton() {

    initAll();

    // Start task
    pros::Task flywheelTask (run_flywheel);
    pros::Task armTask (run_arm);
    pros::Task driveTask (run_drive);
```

```
pros::Task gyroTask (run_gyro);

int driveMode = 0;
double pauseTime = 0;
// Set pointer to chosen auton routine
if (autonSelect == 0) autonCommand = &redAuton[0];
if (autonSelect == 1) autonCommand = &blueAuton[0];

// First entry is always starting direction,
setGyro((*autonCommand) * 10);
//drive.setDirection(*autonCommand);
direction = *autonCommand;

double lidarDist = 0;
nextCommand = true;
std::cout << " Auton Begun - ";

double pauseTimeOut = 0;

while (true) {

    // Auton table decipherer - switch statement
    // Commands will set flags / call object funtions
    // Need commands for:
    // DRIVE (Time, distance, lidar)
    // TURN (Abs & relative)
    // PAUSE (Time, till ball shot)
    // SETGYRO
    // FIRE (Auto aim, high & low)
    // INTAKE (Time / Until ball enters)
    double ds,dd,dt;
    if (nextCommand) {
        std::cout << "Next Command: " << pros::millis() <<
         std::endl;
        nextCommand = false;
        switch ((int)processEntry()) {
            case PAUSE:
                pauseTimeOut = -1;
                pauseTime = processEntry();
                std::cout << "Pause" << std::endl;
                if (pauseTime > 0) pauseTime = (pauseTime * 1000)
                 + pros::millis();
                if (pauseTime < 0) pauseTimeOut = (processEntry()
                 * 1000) + pros::millis();
                break;
            case DRIVE:
                ds = processEntry();
                dd = processEntry();
                dt = processEntry();
                if (dt < 0) {
                    if (dt == DISTANCE) {
                        driveMode = dt;

                         driveDist(ds,dd,processEntry(),processEnt
                         ry());
                        std::cout << "Drive Distance" <<
                         std::endl;
                    }
                    else if (dt == LIDAR) {
                        driveMode = dt;
                        lidarDist = processEntry();         //
                         target lidar value
                        driveCustom(ds,dd,processEntry());  //
                         custom drive with timeout
                        std::cout << "Drive Lidar" << std::endl;
                    }
                    else {
                        driveMode = dt;
                        driveCustom(ds,dd,processEntry());
                        std::cout << "Drive Custom" << std::endl;
                    }
                }
                else {
                    driveMode = dt;
                    driveTime(ds,dd,dt);
                    std::cout << "Drive Time" << std::endl;
                }
                break;
            case TURN:
                turnTo(processEntry(), processEntry());
                std::cout << "Turn" << std::endl;
                break;
            case TURN_REL:
                turnRelative(processEntry(), processEntry());
                std::cout << "Turn Relative" << std::endl;
                break;
            case TURN_ENC:

                 turnRelativeEncoder(processEntry(),processEntry()
                 );
                std::cout << "Turn Relative w/ Encoders" <<
                 std::endl;
                break;
            case SET_GYRO:
                setGyro(processEntry() * 10);
                std::cout << "Set Gyro" << std::endl;
                break;
            case FIRE_PRESET:
                autoFireState = 3;
                autoFireTimeout = processEntry();
                std::cout << "Fire Preset" << std::endl;
                nextCommand = true;
```

```
        break;
    case FIRE_AIM:
        autoFireTimeout = processEntry();
        autoFireState = 1;
        targetFlag = processEntry();
        std::cout << "Fire Aim" << std::endl;
        nextCommand = true;
        break;
    case INTAKE_ON:
        runTillBall = 2;
        std::cout << "Intake On" << std::endl;
        nextCommand = true;
        break;
    case INTAKE_OFF:
        runTillBall = 0;
        std::cout << "Intake Off" << std::endl;
        nextCommand = true;
        break;
    case ARMSEEK:
        armSeek = processEntry();
        std::cout << "Arm Seek" << std::endl;
        nextCommand = true;
        break;
    case WRISTSEEK:
        wristSeek = processEntry();
        std::cout << "Wrist Seek" << std::endl;
        nextCommand = true;
        break;
    case FLIPPERSEEK:
        flipperSeek = processEntry();
        std::cout << "Flipper Seek" << std::endl;
        nextCommand = true;
        break;
    case FLIP:
        std::cout << "Flip" << std::endl;
        if (flipperPos > (FLIP_POS1 + FLIP_POS2)/2) {
            flipperSeek = FLIP_POS1;
        }
        else {
            flipperSeek = FLIP_POS2;
        }
        nextCommand = true;
        break;
    case STACK_LOW:
        stackTarget = LOW;
        stackStep = 1;
        std::cout << "Low Stack" << std::endl;
        nextCommand = true;
        break;
    case STACK_HIGH:
        stackTarget = HIGH;
        stackStep = 1;
        std::cout << "High Stack" << std::endl;
        nextCommand = true;
        break;
    case STACK_LOW_FROM:
        stackTarget = LOW;
        stackStep = processEntry();
        std::cout << "Stack Low From..." << std::endl;
        nextCommand = true;
        break;
    case STACK_HIGH_FROM:
        stackTarget = HIGH;
        std::cout << "Stack high from..." << std::endl;
        nextCommand = true;
        stackStep = processEntry();
        break;
    case END:
        std::cout << "Auton Finished: " << pros::millis()
         << std::endl;
        break;
    case STOP_FLYWHEEL:
        autoFireState = -1;
        std::cout << "Stop Flywheel" << std::endl;
        nextCommand = true;
        break;
    default:
        break;
    }
}


// Auton command termination code
// Decide if we should move to the next command
// eg. checking timers for pause, flags for shooting balls,
 etc.
bool terminateDrive = false;

if (driveMode == LIDAR) {
    // Check if close enough going forward
    if (ds > 0 && getDistance() <= lidarDist) terminateDrive =
     true;
    // Check if far enough going backward
    if (ds < 0 && getDistance() >= lidarDist) terminateDrive =
     true;
}

if (pauseTimeOut > 0 && pauseTime < 0) {
    if (pros::millis() > pauseTimeOut) {
        pauseTime = 0;
        nextCommand = true;
        pauseTimeOut = 0;
```

```
            std::cout << "Pause Finished Timeout- " <<
             pros::millis() << std::endl;
        }
    }

    if (pauseTime > 0) {
        if (pros::millis() > pauseTime) {
            pauseTime = 0;
            nextCommand = true;
            std::cout << "Pause Finished - " << pros::millis() <<
             std::endl;
        }
    }
    else {
        if (pauseTime == FIRED && autoFireState == -1) {
            nextCommand = true;
            pauseTime = 0;
            std::cout << "Pause Finished - " << pros::millis() <<
             std::endl;
        }
        if (pauseTime == GOTBALL && getInnerSensor()) {
            nextCommand = true;
            pauseTime = 0;
            std::cout << "Pause Finished - " << pros::millis() <<
             std::endl;
        }
        if (pauseTime == GOTBALLS && getInnerSensor() &&
         getOuterSensor()) {
            nextCommand = true;
            pauseTime = 0;
            std::cout << "Pause Finished - " << pros::millis() <<
             std::endl;
        }
    }

    if (terminateDrive) {
        std::cout << "Stop Drive" << std::endl;
        driveMode = 0;
        driveStop();
        nextCommand = true;
    }


    pros::delay(20);   // let other tasks use cpu
  }
}


/**
 * Runs the operator control code. This function will be started in
  its own task
 * with the default priority and stack size whenever the robot is
  enabled via
 * the Field Management System or the VEX Competition Switch in the
  operator
 * control mode.
 *
 * If no competition control is connected, this function will run
  immediately
 * following initialize().
 *
 * If the robot is disabled or communications is lost, the
 * operator control task will be stopped. Re-enabling the robot will
  restart the
 * task, not resume it from where it left off.
 */
void opcontrol() {

    // Start task
    pros::Task flywheelTask (run_flywheel);
    pros::Task armTask (run_arm);
    pros::Task driveTask (run_drive);
    pros::Task gyroTask (run_gyro);

    bool justToggledAuto = false;

    while (true) {

        std::cout << "Sensor: " << sensor_gyro.get_value() << " Gyro:
         " << gyroDirection << " Direction: " << direction <<
         std::endl;

        if (autonSelect == 0)
            pros::lcd::print(0, "RED RED RED RED RED RED RED RED RED
             RED RED RED RED RED RED RED");
        else if (autonSelect == 1)
            pros::lcd::print(0, "BLUE BLUE BLUE BLUE BLUE BLUE BLUE
             BLUE BLUE BLUE BLUE BLUE BLUE");

        pros::lcd::print(2, "Direction: %f", direction);

        if ( controller.get_digital(BTN_ABORT) &&
         controller.get_digital(BTN_CHOOSE_AUTON) ) {
            if (!justToggledAuto) {
                autonSelect++;
                if (autonSelect > NUMBER_AUTONS - 1) {
                    autonSelect = 0;
                }
            }
            justToggledAuto = true;
```

```
        }
        else {
            justToggledAuto = false;
        }

        pros::delay(20);
    }
}
```