```
// Code for MultiBot

#include "main.h"

using namespace pros;

// Do we want to use serial gyro?
// Comment this entire line if no
//#define USE_SERIAL_GYRO 1

// Un-comment to skip calibration for timed skills run
// #define PRACTICE_SKILLS

// When defined, camera will use the green flags to determine what
 flags to aim for
#define USE_GREEN_FLAGS 1

// Defines for different gearings for motors
#define TURBO E_MOTOR_GEARSET_06
#define SPEED E_MOTOR_GEARSET_18
#define TORQUE E_MOTOR_GEARSET_36
#define DEGREES E_MOTOR_ENCODER_DEGREES

// Defines for control modes flywheel/arm
#define FLYWHEEL 1
#define ARM 2

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

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

// Flywheel mode
#define BTN_FIRE_HIGH DIGITAL_L1
#define BTN_FIRE_LOW DIGITAL_L2
#define BTN_INTAKE_IN DIGITAL_RIGHT
#define BTN_INTAKE_OUT DIGITAL_LEFT
#define BTN_FIRE_BOTH DIGITAL_B
#define BTN_TOGGLE_COAST DIGITAL_A
#define BTN_TOGGLE_INTAKE DIGITAL_Y
#define BTN_TOGGLE_SCRAPER DIGITAL_X

// Arm mode
#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 auto-stack miniroutines
#define HIGH_STACK_START 1
#define KNOCK_HIGH_START 500
#define LOW_STACK_START 1000


///////////////////////////////////
// Tuning Parameters
// #defines For Tuning

// Arm - higher value is more gentle seek
#define armSeekRate 0.125           // Was 0.25
#define wristSeekRate 0.25
#define wristSeekSlow 8
#define flipperSeekRate 1
#define scraperSeekRate 0.25


// Gyro Correction Values
#ifdef USE_SERIAL_GYRO
#define CSCALE 1                // Clockwise scale adjustments to
 counteract rotation errors
#define ASCALE 1                // Anti-clockwise scale adjustments to
 counteract rotation errors
#else
#define CSCALE 0.9876           // Red Bot Clockwise scale adjustments
 to counteract rotation errors
#define ASCALE 0.9486           // Red Bot Anti-clockwise scale
 adjustments to counteract rotation errors
#define CSCALEBLUE 0.9293       // Blue Bot Clockwise scale
 adjustments to counteract rotation error
#define ASCALEBLUE 0.9293       // Blue Bot Anti-clockwise scale
 adjustments to counteract rotation errors
#endif
```

```
#define GYRO_PORT 1

// Vision Sensor Stuff
#define DEFAULT 0                       // Choose colour of flag based
 on auton mode
#define MAX_FLAG_WIDTH 150              // Widest object camera will
 recognise
#define MAX_FLAG_HEIGHT 500             // Tallest object camera will
 recognise
#define MIN_FLAG_Y -200                 // Lowest camera will
 recognise object
#define AIM_ACCEPT 5                    // Stop auto-aiming within x
#define FLAG_OFFSET -10                 // Value to add/subtract to
 angle to hit flag closer to edge
#define FLYWHEEL_AIM_RANGE 5            // fire ball when within x
 degrees of flag
#define VISION_SEEK_RATE 3              // How fast to turn to aim,
 bigger = slower

#define BLUE_CODE_ID 11                 // Colour code ids for blue
 flag
#define RED_CODE_ID 19                  // Colour code ids for red
 flag

#include "BallBotAutons.h"              // File with autonomous
 routine steps

// Defines for auton routine numbers
#define REDAUTON 0
#define BLUEAUTON 1
#define SKILLSAUTON 2
#define REDBACKAUTON 3
#define BLUEBACKAUTON 4

Controller controller(E_CONTROLLER_MASTER);     // Controller object

//////////////////////////////////////////////////////////////////////
 /////////
// Motors
// Motor name(port, gearing, reversed?, encoder_units);

// Drive Motors
Motor drive_left_1(1, SPEED, 0, DEGREES);
Motor drive_left_2(2, SPEED, 1, DEGREES);
Motor drive_left_3(3, SPEED, 0, DEGREES);
Motor drive_right_1(4, SPEED, 1, DEGREES);
Motor drive_right_2(5, SPEED, 0, DEGREES);
Motor drive_right_3(6, SPEED, 1, DEGREES);
// Flywheel Motors
Motor flywheel_1(19, TURBO, 1, DEGREES);
Motor flywheel_2(10, TURBO ,0, DEGREES);
```

```
// Intake
Motor intake_in(8, SPEED, 1, DEGREES);
Motor intake_out(9, SPEED, 1, DEGREES);
// Arm Motors
Motor arm_1(7, TORQUE, 0, DEGREES);
Motor arm_2(18, TORQUE, 1, DEGREES);
// Flipper
Motor wrist(17, SPEED, 1, DEGREES);
Motor flip(20, SPEED, 0, DEGREES);
// Skills Scraper
Motor scraper(13,TORQUE,0,DEGREES);

// Gyro Sensor
ADIGyro sensor_gyro(1, GYRO_PORT);  // A
// Other sensors
ADIDigitalIn upper_IR (2);  // B
ADIDigitalIn lower_IR (3);  // C
ADIDigitalIn left_IR (4);   // D
ADIDigitalIn right_IR (5);  // E
ADIDigitalOut ballLight(6); // F
ADIUltrasonic sonar (7,8);  // G,H
// Vision sensor
Vision camera(16);


// 1 Metric F. Ton of globals
///////////////////////////////////////////////////////////////
// Drive tuning variables
// Drive
double deadZone = 10;                  // Ignore controller inputs if
 less than this
double ticksPerTile = 640;             // Encoder ticks/tile
double minForward = 40;                // Speed at which to change
 strength of angle correction whilst driving forward
double driveLerp = 0.1;                // Value for dampening of drive
// Turn
double turnAccepted = 1;               // Range within which we accept a
 turn
double pulsePause = 10;                // How long to wait between turn
 pulses
double pulseTime = 7;                  // How long each pulse should be
double minSpeed = 20;                  // Speed less-than-which to switch
 to pulse mode
double maxTurn = 127;                  // Max speed to turn at
double turnRate = 100;                 // Tuning value for turn : Smaller
 = faster
double ticksPerDegree = 10;            // Encoder ticks/degree
// Tracking
double trackingTicksPerTile = 640;  // Encoder ticks/tile used for
 position tracking
```

```
double trackingTicksPerDegree = 10; // Encoder ticks/degree used for
 position tracking

// Drive control variables
// Driving
double autoMode = DRIVEMODE_USER;   // Mode for the drive, can be
 user, time, distance, turn
bool autonComplete = false;         // Has the drive move completed?
double autoTime = 0;                // How long to drive for
bool speedOverride = false;         // Do we want to override the
 drive & force a speed
double rightRunSpeed = 0;           // If yes, right side speed is
double leftRunSpeed = 0;            // And left side speed is
bool drivingToPos = false;          // Do we want to drive to an (x,y)
 position
double autoTimeOut = 0;             // How long should the bot try
 before giving up
double targetDistance = 0;          // How far should the bot drive
double autoSpeed = 0;               // How fast should the bot drive
bool usingGyro = true;              // Should the bot use the gyro to
 turn, false = encoder based turns
double currentDist = 0;             // How far through the distance
 drive has the bot gotten?
double recordedTime = 0;            // What time did the bot start the
 move
double recordedDistLeft = 0;        // Where was the left encoder at
 start of move
double recordedDistRight = 0;       // Where was the right encoder at
 start of move
double lastRightEnc = 0;            // Where was the left encoder last
 frame
double lastLeftEnc = 0;             // Where was the right encoder
 last frame
bool usingSonarDist = false;        // Do we want to use sonar for
 distance, false = encoder based distance
double cmPerTile = 610;             // Centimeters per tile for
 ultrasonic rangefinder (sonar)
// Turning
double targetDirection = 0;         // Direction we want the bot to
 face (degrees)
double turnMode = 0;                // Turn mode, can be encoder of
 gyro
double direction = 0;               // Direction the bot thinks it's
 facing
// Position Tracking
// Some variables are similar to those above, this is so we can have
 different parameters for position tracking
double targetX = 0;                 // X position we want to get to
double targetY = 0;                 // Y position we want to get to
double targetS = 0;                 // Speed we want to drive at
double yPosition = 0;               // Y position the bot thinks it's
 at
double xPosition = 0;               // X position the bot thinks it's
 at
bool trackWithGyro = true;          // Do we want to track direction
 with gyro, false = encoder based direction tracking
double trackingDirection = 0;       // Direction the bot thinks its
 facing (for tracking)


// Auton Routines - defined in BallBotAutons.h
extern int autonSelect;
extern double defaultAuton[];
extern double redAuton[];
extern double blueAuton[];
extern double skills[];
extern double redBackAuton[];
extern double blueBackAuton[];

double gyroDirection = 0;               // Direction from the gyro
bool hasInitialised = false;            // Have we initialised the bot

// 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
bool fireBoth = false;          // Are we doing a double shot

// For Intake
bool forceIntake = false;       // Do we want to force the intake to
 run at a speed
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
// Where we want each mechanism to seek to, -1 = don't seek
double armSeek = -1;
double wristSeek = -1;
double flipperSeek = -1;
double scraperSeek = -1;
```

```
// Where each mechanism thinks it's at
double armPos = 0;
double flipperPos = 0;
double wristPos = 0;
double scraperPos = 0;

// Offset values to tare position without taring motors
double armOffset = 0;
double flipperOffset = 0;
double wristOffset = 0;
double scraperOffset = 0;

int stackTarget = -1;           // Target pole to stack on (high or
 low)
int stackStep = -1;             // Step to do in the auto stack
 miniroutine
double lastAutonTime = 0;       // How long autonomous mode took last
 run


#define FLYWHEEL_SPEED_RANGE 15             // fire ball when within x
 rpm of target speed
#define flywheelSlowSpeed 50                // Speed to run flywheel
 when too fast
#define flywheelFastSpeed 127               // Speed to run flywheel
 when too slow

double flywheelRunSpeed = 0;            // How fast should the
 flywheel run
double flyWheelDefaultSpeed = 450;      // Set speed for fixed-dist
 fireing
bool coast = false;                     // Should the flywheel
 continue to run when not firing
double defaultFlywheelDistance = 1;     // Default distance for the
 flywheel if sonar not working
bool flipCapWIntake = false;            // Should we run the intake
 backwards to flip a cap

// 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
double flyWheelSpeeds[12][3] = {                 // CALIBRATE & add
 more
    // Dist, Low Flag Speed, High Flag Speed
    {-100,  0,  0},   // to catch errors
    {0,     0,  0},
    {1,     450,  450},
};
```

```
int flyWheelSpeedsDefinition = 12;      // Number of entries in speed
 table
double autoFireTimeout = -1;            // If positive, how long to
 try to fire before giving up


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


// Signatures to feed to the vision sensor
pros::vision_signature_s_t GREEN_SIG;
pros::vision_signature_s_t RED_SIG;
pros::vision_signature_s_t BLUE_SIG;

pros::vision_color_code_t BLUE_CODE;
pros::vision_color_code_t RED_CODE;

// Function to set vision sensor parameters
// Values generated in VEX Vision Utility
```

```
void calibrateVision() {

    // Calibration for Blue Bot
    if (autonSelect == BLUEBACKAUTON || autonSelect == REDBACKAUTON) {

        // Create signatures for calibration

        BLUE_SIG =
        pros::Vision::signature_from_utility(BLUE_FLAG, -3181, 233,
         -1474, 3431, 11679, 7554, 1, 1);

        RED_SIG =
        pros::Vision::signature_from_utility(RED_FLAG, 2725, 5285,
         4006, -459, 1, -228, 1.2, 1);

        GREEN_SIG =
        pros::Vision::signature_from_utility(GREEN_FLAG, -1945, -577,
         -1261, -6207, -5107, -5657, 2.5, 1);

        camera.set_signature(BLUE_FLAG, &BLUE_SIG);
        camera.set_signature(RED_FLAG, &RED_SIG);
        camera.set_signature(GREEN_FLAG, &GREEN_SIG);

        BLUE_CODE = camera.create_color_code(BLUE_FLAG,GREEN_FLAG);
        RED_CODE = camera.create_color_code(RED_FLAG,GREEN_FLAG);

        camera.set_exposure(79);
    }

    // Calibration for Red Bot
    else if (autonSelect == BLUEAUTON || autonSelect == REDAUTON){

        // Create signatures for calibration
        BLUE_SIG =
        pros::Vision::signature_from_utility(BLUE_FLAG, -3181, 233,
         -1474, 3431, 11679, 7554, 1, 1);

        RED_SIG =
        pros::Vision::signature_from_utility(RED_FLAG, 1165, 4187,
         2676, -383, 167, -108, 1, 1);

        GREEN_SIG =
        pros::Vision::signature_from_utility(GREEN_FLAG, -1945, -577,
         -1261, -6207, -5107, -5657, 4.8, 1);

        camera.set_signature(BLUE_FLAG, &BLUE_SIG);
        camera.set_signature(RED_FLAG, &RED_SIG);
        camera.set_signature(GREEN_FLAG, &GREEN_SIG);

        BLUE_CODE = camera.create_color_code(BLUE_FLAG,GREEN_FLAG);
        RED_CODE = camera.create_color_code(RED_FLAG,GREEN_FLAG);

        camera.set_exposure(78);

    }

    // Calibration for Skills
    else {
        // Create signatures for calibration
        BLUE_SIG =
        pros::Vision::signature_from_utility(BLUE_FLAG, -3181, 233,
         -1474, 3431, 11679, 7554, 1, 1);

        RED_SIG =
        pros::Vision::signature_from_utility(RED_FLAG, 1165, 4187,
         2676, -383, 167, -108, 0, 1);

        GREEN_SIG =
        pros::Vision::signature_from_utility(GREEN_FLAG, -1945, -577,
         -1261, -6207, -5107, -5657, 2.7, 1);

        camera.set_signature(BLUE_FLAG, &BLUE_SIG);
        camera.set_signature(RED_FLAG, &RED_SIG);
        camera.set_signature(GREEN_FLAG, &GREEN_SIG);

        BLUE_CODE = camera.create_color_code(BLUE_FLAG,GREEN_FLAG);
        RED_CODE = camera.create_color_code(RED_FLAG,GREEN_FLAG);

        camera.set_exposure(78);
    }

    camera.set_zero_point(pros::E_VISION_ZERO_TOPLEFT);


}


// Initialize function
void initAll() {        // called when robot activates & start of
 auton
    if (!hasInitialised) {
        // First time / manual init...
        // Calibrate gyro, tare motors, start tasks
        controller.print(0,0,"Calibrating");
        sensor_gyro = ADIGyro(1, GYRO_PORT);
        // Tare motor positions
        arm_1.tare_position();
        arm_2.tare_position();
        wrist.tare_position();
        flip.tare_position();
```

```
#ifndef USE_SERIAL_GYRO     // We don't need to calibrate if using the
 fancy gyro
        pros::delay(4000);
#endif
        // Start the tasks
        pros::Task flywheelTask (run_flywheel);
        pros::Task armTask (run_arm);
        pros::Task driveTask (run_drive);
        pros::Task gyroTask (run_gyro);
        pros::Task screenTask (run_screen);

#ifdef USE_SERIAL_GYRO
        pros::Task serialTask (serialRead);
#endif

        controller.print(0,0,"            ");
    }
    // Remember we've calibrated already, so we don't do it again
    hasInitialised = true;
    calibrateVision();
}

// Increments autonomous routine pointer & returns next command
double processEntry() {
    autonCommand++;
    return *autonCommand;
}


//////////////////////////////////////////////////////////////////////
 ////////////
// Serial Stuff
//

// Include sstream for serial parsing
#include <sstream>


// Prototypes for hidden vex functions to bypass PROS bug
extern "C" int32_t vexGenericSerialReceive( uint32_t index, uint8_t
 *buffer, int32_t length );
extern "C" void vexGenericSerialEnable(  uint32_t index, uint32_t
 nu );
extern "C" void vexGenericSerialBaudrate(  uint32_t index, uint32_t
 rate );

// Port to use for serial data
#define SERIALPORT 14           // Port 15
// Variable to put the gyro value into
double gyroValue = 0;
```

```
// Currently reads serial data & parses for gyro value
// Can be expanded to look for lidar distance, etc.
void serialRead(void* params) {

    // Start serial on desired port
    vexGenericSerialEnable( SERIALPORT - 1, 0 );

    // Set BAUD rate
    vexGenericSerialBaudrate( SERIALPORT - 1, 115200 );

    // Let VEX OS configure port
    pros::delay(10);

    // Serial message format:
    // D[LIDAR DIST]I[IR DATA]A[GYRO ANGLE]E
    // Example Message:
    // D50.2I128A12.32E

    while (true) {

        // Buffer to store serial data
        uint8_t buffer[256];
        int len = 256;

        // Get serial data
        int32_t nRead = vexGenericSerialReceive(SERIALPORT - 1,
         buffer, len);

        // Now parse the data
        if (nRead >= 9) {

            // Stream to put the characters in
            std::stringstream myStream("");
            bool recordAngle = false;

            // Go through characters
            for (int i = 0; i < nRead; i++) {
                // Get current char
                char thisDigit = (char)buffer[i];

                // If its special, then don't record the value
                if (thisDigit == 'D' || thisDigit == 'I' || thisDigit
                 == 'A')
                    recordAngle = false;

                // Finished recieving angle, so put into variable
                if (thisDigit == 'E') {
                    recordAngle = false;
                    myStream >> gyroValue;
                }
```

```
            // If we want the digits, put them into stream
            if (recordAngle)
                myStream << (char)buffer[i];

            // If the digit is 'A', then the following data is the
             angle
            if (thisDigit == 'A')
                recordAngle = true;

        }

    }

    // Delay to let serial data arrive
    pros::delay(10);

}

}


///////////////////////////////////////////////////////////////////////
 ///////////
// Gyro Stuff
// gyroDirection will be updated with 'more accurate' gyro value
// Built for additional gyro, but we only use 1
gyros gyro1,gyro2;

short gyroinit=0;

// Function to reset the gyros
void resetGyro() {
    sensor_gyro.reset();
    gyro1.truedir=0;
    gyro2.truedir=0;
    gyro1.last=0;
    gyro2.last=0;
    gyroDirection=0;
}

// Function to set the gyros to a value
void setGyro(double dir) {
    gyro1.truedir=dir;
    gyro2.truedir=dir;
    gyroDirection=dir;
}

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


#ifdef USE_SERIAL_GYRO
    currentGyro = gyroValue*10;                         // Read
     hardware gyro value from serial
#else
    currentGyro = sensor_gyro.get_value();              // Read
     hardware gyro value from analog
#endif

    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 (tempAngle>0) {  // Anti-clockwise rotation
        gyro->truedir=gyro->truedir+(tempAngle*gyro->ascale);
         // Multiply difference by correction value
        if (gyro->truedir<0) {gyro->truedir=gyro->truedir+3600;}
         // Wrap
    }
    else {
        gyro->truedir=gyro->truedir+(tempAngle*gyro->cscale);
         // Multiply difference by correction value
        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();
    if (autonSelect == REDBACKAUTON || autonSelect ==
     BLUEBACKAUTON) {
        gyro1.ascale=ASCALEBLUE;
        gyro1.cscale=CSCALEBLUE;
    }
    else {
        gyro1.ascale=ASCALE;
        gyro1.cscale=CSCALE;
    }

    gyro2.port=GYRO_PORT;   // If using two gyros, this would be
     the port of the second gyro
    gyro2.truedir=0;
    gyro2.last=sensor_gyro.get_value();
    if (autonSelect == REDBACKAUTON || autonSelect ==
     BLUEBACKAUTON) {
        gyro2.ascale=ASCALEBLUE;
        gyro2.cscale=CSCALEBLUE;
    }
    else {
        gyro2.ascale=ASCALE;
        gyro2.cscale=CSCALE;
    }

    gyroDirection=0;
}

while(true) {

    // Check which bot the code is running on, and adjust tuning
     values accordingly
    if (autonSelect == REDBACKAUTON || autonSelect ==
     BLUEBACKAUTON) {
        gyro1.ascale=ASCALEBLUE;
        gyro1.cscale=CSCALEBLUE;
        gyro2.ascale=ASCALEBLUE;
        gyro2.cscale=CSCALEBLUE;
    }
    else {
        gyro1.ascale=ASCALE;
        gyro1.cscale=CSCALE;
        gyro2.ascale=ASCALE;
        gyro2.cscale=CSCALE;
    }

    checkGyro(&gyro1);
    checkGyro(&gyro2);
```

```
        if (gyro1.truedir>gyro2.truedir) {      // Check if gyro1 is
         larger
            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 the gyros
        if (gyroDirection<0) gyroDirection+=3600;
         // Unwrap negative case
        pros::delay(20);
    }
}


// Functions to get average values of drive encoders
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
//
// Stop driving
void driveStop() {
    autoTime = 0;
    autoMode = DRIVEMODE_USER;
    autoSpeed = 0;
    speedOverride = false;
    drivingToPos = false;
}

// Drive for a time
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();
}

// Drive for a distrance
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();
    usingSonarDist = false;

    // Check which direction we are driving & set target accordingly
    if (s > 0) {
        targetDistance = (dist * ticksPerTile) + (recordedDistRight +
         recordedDistLeft)/2;
    }
    else {
        targetDistance = (-dist * ticksPerTile) + (recordedDistRight +
         recordedDistLeft)/2;
    }
}

// Drive to a sonar value
void driveDistSonar(double s, double dir, double dist, double t = 10)
 {
    // speed, direction, distance, timeout
    autoSpeed = s;
    targetDirection = dir;
    autoMode = DRIVEMODE_SONAR;
    autoTimeOut = t*1000;
    recordedTime = pros::millis();
    recordedDistLeft = getLeftEnc();
    recordedDistRight = getRightEnc();
    usingSonarDist = true;
    targetDistance = dist * cmPerTile;
}

// Drive until told to stop
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;
}

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

// Turn relative to current direction
void turnRelative(double a, double t = -1) {
    // angle, timeout
    recordedTime = pros::millis();
    targetDirection = (gyroDirection / 10) + a;
    autoTimeOut = t*1000;
    autoMode = DRIVEMODE_TURN;
    turnMode = TURNMODE_GYRO;
}

// Turn using encoders
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
//

// Set position of robot
void setPosition(double x, double y, double d) {
    xPosition = x;
    yPosition = y;
    trackingDirection = d;
}

// Function to update the estimate position
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;

    trackingDirection += angleChange;               // Find cumulative
     direction

    if (trackWithGyro) {                            // If we are using
     gyro, then ignore encoder direction
        trackingDirection = gyroDirection / 10;
    }

    xPosition += distChange * cos(trackingDirection * M_PI / 180);  //
     Find cumulative xPos
    yPosition -= distChange * sin(trackingDirection * M_PI / 180);  //
     Find cumulative yPoS

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

// Funcion to turn to face a point
void turnToPoint(double x, double y, double t = -1) {
    double dx = x - xPosition;      // Find delta x,y
    double dy = y - yPosition;
    if (dx == 0) dx = 0.0000000001;
    double dir = atan(abs(dy/dx)) * 180 / M_PI; // Calculate direction
     from this
    if (dx > 0 && dy > 0) dir = 360 - dir;      // Find which quadrant
     we need
    if (dx < 0 && dy > 0) dir += 180;
    if (dx < 0 && dy < 0) dir += 90;
    if (dx > 0 && dy < 0) dir = dir;
    turnTo(dir);                                // Turn to that
     direction
}
void driveTo(double s, double x, double y, double t = 10) {
    targetX = x;
    targetY = y;
    targetS = s;
    double dx = x - xPosition;                  // Find delta x,y
    double dy = y - yPosition;
    double dir = atan(abs(dy/dx)) * 180 / M_PI; // Find direction
    if (dx > 0 && dy > 0) dir = 360 - dir;      // Find which quadrant
     we need
    if (dx < 0 && dy > 0) dir += 180;
    if (dx < 0 && dy < 0) dir += 90;
    if (dx > 0 && dy < 0) dir = dir;
    double dist = hypot(x,y);                   // Find distance
    driveDist(s, dir, dist, t);                 // Drive to that
     distance
    drivingToPos = true;                        // We are driving to
     position, so this sould be true
}


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

void run_drive(void* params) {

    double currentTime = 0;     // Var to store current time
    double leftPower = 0;       // How much power to each side
    double rightPower = 0;
    double leftSpeed = 0;       // How fast to run each side
    double rightSpeed = 0;

    double lastAngle = 0;       // Angle we were facing
    double turnPulse = 0;       // Track the state of pulsing

    double slewRate = 2;        // How much dampening on the drive

    int turnGoodCount = 0;      // How long have we been facing the
     correct angle
    double slewPower = 0;       // Slew for BlueBot

    while (true) {

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

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

```
}

// This is where the fun begins

double forward = 0;     // Forward/backward speed of the bot
double turn = 0;        // Turn speed of bot

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

double currentDistLeft = getLeftEnc();  // Find encoder values
 of drive
double currentDistRight = getRightEnc();
currentDist = (currentDistRight + currentDistLeft)/2;   //
 Average these to find current distance

// Check controller
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
        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
        // Controller values will be 0 in auton, but we still
         want translation while aiming
        forward = (controller.get_analog(ANALOG_LEFT_Y) +
         controller.get_analog(ANALOG_RIGHT_Y));
        autoSpeed = 0;
    }

    // If we are driving a distance
    if (autoMode == DRIVEMODE_DIST) {   // If auto move should
     end with a distance
        // We want to slow down when approaching desired
         position to avoid overshoot
```

```
        double slowDown = abs((targetDistance - currentDist) /
         (0.35 * ticksPerTile));

        if (slowDown > 1) slowDown = 1;     // Don't want to
         speed up before then

        forward *= slowDown;                // Apply slow down
         speed

        // Clamp speed above minimum threshold
        if (autoSpeed > 0 && forward < minForward) forward =
         minForward;
        if (autoSpeed < 0 && forward > -minForward) forward =
         -minForward;

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

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

    // If we are driving to a sonar range
    if (autoMode == DRIVEMODE_SONAR) {
        currentDist = sonar.get_value();    // Current dist is
         form sonar, not encoders

        // We want to slow down when approaching desired
         position to avoid overshoot
        double slowDown = abs((targetDistance - currentDist) /
         (0.35 * cmPerTile));

        if (slowDown > 1) slowDown = 1;     // Don't want to
         speed up before then

        forward *= slowDown;                // Apply slow down
         speed

        // Clamp speed above minimum threshold
        if (autoSpeed > 0 && forward < minForward) forward =
         minForward;
        if (autoSpeed < 0 && forward > -minForward) forward =
         -minForward;
```

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

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

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

// Turn code
double driveMag = abs(autoSpeed);       // How strong to
 drive
double seek = targetDirection;          // Direction we
 want to face
double angle = 0;                       // Var to store
 error and then power

if (turnMode == TURNMODE_GYRO) {        // If we are
 turning based on gyro
    angle = seek - direction;           // Angle is error
     (where we want to be) - (where we are)
}
else if (turnMode == TURNMODE_ENCODER) {    // If we are
 turning based on encoder
    // Calculate angle based on difference between left
     and right encoder values
    angle = (recordedDistRight - recordedDistLeft)/2;
    angle -= (currentDistRight - currentDistLeft)/2;
    angle /= ticksPerDegree;
}

// Clamp angle to +/- 180 so bot always turns smaller
 angle
if (angle < 0) angle += 360;
if (angle > 180) angle -= 360;

// Scale by turnRate to allow tuning
angle /= (2 * turnRate);
angle *= 127;

// If we are driving slowly, then turning should be
 stronger to help us stay on course
if (driveMag < minSpeed) {
    angle *= 2;
}

// Cap turn power at maximum threshold
if (angle < -maxTurn) angle = maxTurn;
if (angle > maxTurn) angle = maxTurn;

// If we are driving faster than minSpeed, do some
 clamping when angle is small
// This helps when driving fast in a straight line
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 {                  // If we are below minSpeed
    turn = angle;       // Remember angle in turn
    angle = abs(angle); // Find absolute angle
    if (angle < minSpeed) { // If angle is small
        // If we have crossed from - to + or + to -, then
         stop since we are at destination
        if (((lastAngle > 0) && (turn < 0)) || ((lastAngle
         < 0) && (turn > 0))) {
            angle = 0;
        }
        else {
            if (angle > minSpeed/5) {   // If angle is
             between minSpeed and minSpeed/5, make it
             minSpeed
                angle = minSpeed;
            }
```

```
                else {                      // If angle <
                 minSpeed / 5
                    turnPulse++;                    //
                     Increment turnPulse
                    if (turnPulse < pulseTime) {    // If bot
                     should be pulsing
                        angle = minSpeed;               // Set
                         angle = minSpeed
                    }
                    else {                          //
                     Otherwise
                        angle = 1;                      //
                         Make angle small
                        if (turnPulse > pulsePause) {   //
                         Check if we've waited long enough
                         between pulses
                            turnPulse = 0;
                        }
                    }
                }
            }
        }
        if (turn < 0) angle *= -1;      // Un-abs(angle)
    }

    turn = angle;       // Set turn power to angle

    // If we are turning with no drive
    if (autoSpeed == 0 || (autoMode == DRIVEMODE_TURN &&
     forward != 0)) {
        // If we are within our accepted error
        if (abs(direction - targetDirection) < turnAccepted) {
            turnGoodCount++;            // Count this as a
             good sample
            if (turnGoodCount > 10)     // Check if we've had
             10 good samples
                autonComplete = true;   // If we have, then we
                 are done with the turn
        }
        else {
            turnGoodCount = 0;      // If we are not within
             accepted value, reset count
        }
    }

    lastAngle = angle;      // Remember last angle for next
     loop

}
```

```
// Auto-move is complete, so stop moving
if (autonComplete && (autoFireState == -1 || fireBoth)) {
    // We want to stop moving so set appropriate flags
    autonComplete = false;          // Reset 'stop' flag
    autoMode = DRIVEMODE_USER;      // Give control back to
     user
    forward = 0;                    // Set power back to 0
    turn = 0;
    autoSpeed = 0;
    drivingToPos = false;           // Stop driving to a
     position
    nextCommand = true;             // Let the next auton
     command start
    std::cout << "Drive Move Done: " << currentTime <<
     std::endl;
}

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

    // We use the same code for each bot, so differentiate by
     which autonomous routine the bot is runnning
    if (autonSelect == REDAUTON || autonSelect == BLUEAUTON ||
     autonSelect == SKILLSAUTON) {
        // Tank Controls For Sam
        // Each control mode has a different front/back of the
         robot
        if (controlMode == FLYWHEEL && armSeek !=
         ARM_KNOCK_POS) {
            leftSpeed = controller.get_analog(ANALOG_LEFT_Y);
            rightSpeed =
             controller.get_analog(ANALOG_RIGHT_Y);
        }
        else {
            rightSpeed = -
             controller.get_analog(ANALOG_LEFT_Y);
            leftSpeed = -
             controller.get_analog(ANALOG_RIGHT_Y);
        }
    }
    else {
        // Arcade Controls For RJ/Ramon
        // Each control mode has a different front/back of the
         robot
        if (controlMode == FLYWHEEL && armSeek !=
         ARM_KNOCK_POS) {

            // Dampen acceleration (driver preference)
            slewPower = slewPower +
             (controller.get_analog(ANALOG_LEFT_Y) -
             slewPower) / 16;
```

```
            leftSpeed = slewPower +
             controller.get_analog(ANALOG_RIGHT_X);
            rightSpeed = slewPower -
             controller.get_analog(ANALOG_RIGHT_X);
        }
        else {
            slewPower = slewPower +
             (controller.get_analog(ANALOG_LEFT_Y) -
             slewPower) / 16;
            leftSpeed = -slewPower +
             controller.get_analog(ANALOG_RIGHT_X);
            rightSpeed = -slewPower -
             controller.get_analog(ANALOG_RIGHT_X);
        }
    }

    // Check if the joystick input is below deadzone, and set
     to zero
    if (abs(leftSpeed) < deadZone) leftSpeed = 0;
    if (abs(rightSpeed) < deadZone) rightSpeed = 0;
}
else {
    // If not user controls, turn the autonomous powers into
     left & right power levels
    leftSpeed = forward - turn;
    rightSpeed = forward + turn;
}

// If we want to override with a constant speed, do so
if (speedOverride) {
    leftSpeed = leftRunSpeed;
    rightSpeed = rightRunSpeed;
}

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


// Send powers 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
//

// Get speed of flywheel
double getFlywheelSpeed() {
    return (flywheel_1.get_actual_velocity() +
     flywheel_2.get_actual_velocity() ) / 2;
}
// Get inner IR value
bool getInnerSensor() {
    return upper_IR.get_value();
}
// Get outer IR value
bool getOuterSensor() {
    return lower_IR.get_value();
}
// Get distance from flags
double getDistance() {
    // Sonar not implemented, so return default distance
    return defaultFlywheelDistance;
}


// Read vision sensor to get angle needed to turn
// Returns angle to desired target
// Or 0 if error
double getRelativeAngle(int location = CENTER, int target = DEFAULT) {

    int lookingFor = BLUE_FLAG;     // default to red-team
    if (autonSelect == BLUEAUTON || autonSelect == BLUEBACKAUTON)
        lookingFor = RED_FLAG;      // but change to blue if needed

    if (target != DEFAULT)
        lookingFor = target;

    // Containers for the things we'll see
    std::vector<vision_object_s_t> blueThings;
    std::vector<vision_object_s_t> redThings;

    // Find number of objects visable
    int noObjs = camera.get_object_count();
```

```
    if (noObjs > 100)       // Camera error, so don't aim
        return 0;

    // Got through all objects seen
    for (int i = 0; i < noObjs; i++) {
        vision_object_s_t thisThing = camera.get_by_size(i);
        // Print their info

        // If object is a colour code
        if (thisThing.type == 1) {
            // Red flags should have angle ~0°
            if (thisThing.signature == RED_CODE_ID &&
             abs(thisThing.angle) < 90) {
                redThings.push_back(thisThing);
            }
            // Blue flags should have angle ~180°
            if (thisThing.signature == BLUE_CODE_ID &&
             abs(thisThing.angle) > 90) {
                blueThings.push_back(thisThing);
            }
        }
    }

    std::vector<vision_object_s_t> *theseThings;

    if (lookingFor == BLUE_FLAG)
        theseThings = &blueThings;
    if (lookingFor == RED_FLAG)
        theseThings = &redThings;

    if (theseThings->size() == 0)
        return 0;

    // Find which object is closest to left/middle/right
    double closestDist;
    if (location == CENTER) {
        closestDist = 10000;
        for (int i = 0; i < (*theseThings).size(); i++) {
            if (abs((*theseThings)[i].x_middle_coord -
             (VISION_FOV_WIDTH/2)) < closestDist) {
                closestDist = (*theseThings)[i].x_middle_coord;
            }
        }
    }
    if (location == LEFT) {
        closestDist = 10000;
        for (int i = 0; i < (*theseThings).size(); i++) {
            if ((*theseThings)[i].x_middle_coord < closestDist) {
                closestDist = (*theseThings)[i].x_middle_coord;
            }
        }
    }
    if (location == RIGHT) {
        closestDist = -10000;
        for (int i = 0; i < (*theseThings).size(); i++) {
            if ((*theseThings)[i].x_middle_coord > closestDist) {
                closestDist = (*theseThings)[i].x_middle_coord;
            }
        }
    }

    // Aim at the edge of the flag for better chance of toggleing
    if (lookingFor == RED_FLAG) closestDist += FLAG_OFFSET;
    if (lookingFor == BLUE_FLAG) closestDist -= FLAG_OFFSET;

    closestDist = closestDist - (VISION_FOV_WIDTH/2);

    if ((-closestDist/VISION_SEEK_RATE) == 0)
        return 0.001;

    return -closestDist/VISION_SEEK_RATE;

}


/////////////////////////////
// Flywheel Task
//
void run_flywheel(void* params) {
    // Declare any local variables
    bool ballIsIn = false;
    bool ballWasIn = false;
    bool justToggledAutoBall = false;
    bool toggledCoast = false;
    bool fireBall = false;
    bool justAskedForFire = false;
    bool doSet = false;
    int lastBlinkTime = millis();
    double lastIntakeSpeed = 0;
    bool justToggledScraper = false;

    while (true) {

        double scraperSpeed = 0;
        scraperPos = scraper.get_position() + scraperOffset;

        // 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) fireBall = false;

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];
            }
        }
}

if (flywheelRunSpeed != -1)
    targetSpeed = flywheelRunSpeed;

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


// 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;
                fireBall = false;
                flywheelRunSpeed = -1;
                driveStop();
                driveDist(127, direction, 0.75, 2);
                autoFireState = 2;
                fireBoth = true;
            }
            else {
                autoFireState = -1;
                fireBoth = false;
                targetSpeed = flyWheelDefaultSpeed;
                fireBall = false;
                flywheelRunSpeed = -1;
                driveStop();
            }
        }


    }   // end of auto-fire

    // Code to blink light if got balls
    if (getInnerSensor() && getOuterSensor()) {
        if (millis() > abs(lastBlinkTime) + 125) {
            if (lastBlinkTime > 0) {
                lastBlinkTime = -millis();
                ballLight.set_value(1);
            }
            else {
                lastBlinkTime = millis();
                ballLight.set_value(0);
            }
        }
    }
    else if (getInnerSensor() || getOuterSensor()) {
        if (millis() > abs(lastBlinkTime) + 500) {
            if (lastBlinkTime > 0) {
                lastBlinkTime = -millis();
                ballLight.set_value(1);
            }
            else {
                lastBlinkTime = millis();
                ballLight.set_value(0);
            }
        }
    }
    else {
        ballLight.set_value(1);
    }


    // Check controller buttons...
    // Set flags for preset flywheel speeds & auto-aim-fire
    // If manual intake buttons pressed, override intake speeds
    if (controlMode == FLYWHEEL) {

        // Toggle button for scraper
        if (controller.get_digital(BTN_TOGGLE_SCRAPER)) {
            if (!justToggledScraper) {
                if (scraperSeek == SCRAPER_UP_POS)
                    scraperSeek = SCRAPER_DOWN_POS;
                else
                    scraperSeek = SCRAPER_UP_POS;
            }
            justToggledScraper = true;
        }
        else {
            justToggledScraper = false;
        }


        if (controller.get_digital(BTN_FIRE_LOW)) { //} &&
         autonSelect != SKILLSAUTON) { // auto fire low
            wristSeek = WRIST_VERTICAL_POS;
            doSet = false;
            if (!justAskedForFire) {
                if (autoFireState <= 0 && autonSelect != REDAUTON
                 && autonSelect != BLUEAUTON && autonSelect !=
                 SKILLSAUTON)
                    autoFireState = 2;
                else
                    autoFireState = 1;

                autoFireTimeout = -1;
                targetFlag = 1;
                fireBoth = false;
                fireBall = false;
                justAskedForFire = true;
                flywheelRunSpeed = -1;
```

```
        flipCapWIntake = false;
    }
}
else if (controller.get_digital(BTN_FIRE_HIGH)) { // auto
 fire high
    wristSeek = WRIST_VERTICAL_POS;
    doSet = false;
    if (!justAskedForFire) {
        if (autoFireState <= 0 && autonSelect !=
         REDBACKAUTON && autonSelect != BLUEBACKAUTON)
            autoFireState = 2;
        else
            autoFireState = 1;
        fireBoth = false;
        targetFlag = 2;
        autoFireTimeout = -1;
        fireBall = false;
        justAskedForFire = true;
        flywheelRunSpeed = -1;
        flipCapWIntake = false;
    }
}
else if (controller.get_digital(BTN_FIRE_BOTH)) { // auto
 fire both
    doSet = true;
    if (!justAskedForFire) {
        // Blue Bot this button is anti-park, Red Bot it
         is double-fire
        if (autonSelect != BLUEBACKAUTON && autonSelect !=
         REDBACKAUTON) {
            //driveDistSonar(127, direction, 1.5, 2);
            fireBoth = true;
            autoFireState = 2;  // Don't aim
            targetFlag = 3;
            autoFireTimeout = -1;
            fireBall = false;
            justAskedForFire = true;
            flywheelRunSpeed = -1;
            flipCapWIntake = false;
        } else {
            justAskedForFire = true;
            if (armSeek == ARM_KNOCK_POS && wristSeek ==
             WRIST_KNOCK_POS && flipperSeek == (FLIP_POS1
             + FLIP_POS2)/2) {
                armSeek = 1;
                wristSeek = WRIST_VERTICAL_POS;
                flipperSeek = FLIP_POS1;
            }
            else {
                armSeek = ARM_KNOCK_POS;
                wristSeek = WRIST_KNOCK_POS;
                flipperSeek = (FLIP_POS1 + FLIP_POS2)/2;
            }
        }
    }
}
else {
    justAskedForFire = false;
}
/*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;
    flipCapWIntake = false;
}
if (controller.get_digital(BTN_INTAKE_OUT)) { // manual
 run intake out
    intakeSpeedInner = -127;
    intakeSpeedOuter = -127;
    runTillBall = 0;
    forceIntake = false;
    flipCapWIntake = false;
}
if (controller.get_digital(BTN_TOGGLE_INTAKE)) { // toggle
 auto ball intake
    flipCapWIntake = false;
    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;
            fireBall = false;
            flywheelRunSpeed = -1;
            flipCapWIntake = false;
            scraperSeek = -1;
        }

        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;
            }
        }

        if (flipCapWIntake) {
            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 (autonSelect == SKILLSAUTON)
        //            targetSpeed = 450;


        if (targetSpeed != 0 && (autonSelect == REDBACKAUTON ||
         autonSelect == BLUEBACKAUTON)) {
            targetSpeed += 20;
            if (targetSpeed > 600)
                targetSpeed = 600;
        }


        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 &&
         autonSelect != SKILLSAUTON) {
        //            flywheelSpeed = flyWheelDefaultSpeed;
        //        }


        ///////////////////////////////
        // flywheelSpeed = 0;
        ///////////////////////////////

        if (scraperSeek != -1) {
            scraperSpeed = (scraperSeek - scraperPos) /
             scraperSeekRate;
            if (scraperSpeed > 127) scraperSpeed = 127;
            if (scraperSpeed < -127) scraperSpeed = -127;
        }


        if (lastIntakeSpeed > 0 && intakeSpeedInner == 0) {
            intakeSpeedInner = -127;
        }

        // 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);

        scraper.move_voltage(scraperSpeed*12000/127);

        // Remember ball info for fireing
        ballWasIn = ballIsIn;

        lastIntakeSpeed = intakeSpeedInner;

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


///////////////////////////////
// Arm Task
//
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;
double timeLastStep = 0;

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;

    // If we want to stack something, follow the steps
    switch (stackStep) {
            // High Stacking
        case HIGH_STACK_START:
            if (!controller.get_digital(BTN_ARM_HIGH)) {
                stackStep++;
            }
            break;
        case HIGH_STACK_START + 1:
            armSeek = ARM_POS_HIGH;
            wristSeek = WRIST_VERTICAL_POS;
            if (armPos > ARM_POS_HIGH - 50) {
                stackStep++;
            }
            break;
        case HIGH_STACK_START + 2:
            wristSeek = WRIST_BACKWARD_DROP_POS;
            if (wristPos < WRIST_BACKWARD_DROP_POS + 15 + ( armPos
             * 3 / 5 )) {
                stackStep++;
                timeLastStep = millis();
            }
            break;
        case HIGH_STACK_START + 3:
            if (timeLastStep + 250 < millis()) {
                stackStep++;
            }
            break;
        case HIGH_STACK_START + 4:
            armSeek = 1;
            if (armPos < ARM_POS_HIGH / 2) {
                stackStep++;
            }
            break;
        case HIGH_STACK_START + 5:
            wristSeek = WRIST_VERTICAL_POS;
            if (armPos < armSeek) {
                stackStep++;
                timeLastStep = millis();
            }
            break;
        case HIGH_STACK_START + 6:
            if (timeLastStep + 250 < millis()) {
                armSeek = -1;
                stackStep = -1;
            }
            break;

            // High Knock Off
        case KNOCK_HIGH_START:
            armSeek = ARM_POS_HIGH;
            wristSeek = WRIST_BACKWARD_DROP_POS - 100;
            if (armPos > ARM_POS_HIGH) {
                stackStep++;
            }
            break;
        case KNOCK_HIGH_START + 1:
            wristSeek = WRIST_VERTICAL_POS;
            if (wristPos > WRIST_VERTICAL_POS - 20) {
                stackStep = HIGH_STACK_START + 3;
                timeLastStep = millis();
            }
            break;

            // Low Stacking
        case LOW_STACK_START:
            if (!controller.get_digital(BTN_ARM_LOW)) {
                stackStep++;
            }
            break;
        case LOW_STACK_START + 1:
            armSeek = ARM_POS_LOW;
            wristSeek = WRIST_VERTICAL_POS;
            if (armPos > ARM_POS_LOW - 50) {
                stackStep++;
            }
            break;
        case LOW_STACK_START + 2:
            if (controller.get_digital(BTN_ARM_LOW)) {
                stackStep++;
            }
            break;
```

```
        case LOW_STACK_START + 3:
            slowSeek = true;
            wristSeek = WRIST_FORWARD_DROP_POS;
            if (wristPos > WRIST_FORWARD_DROP_POS - 15 + ( armPos
             * 3 / 5 )) {
                stackStep++;
            }
            break;
        case LOW_STACK_START + 4:
            if (controller.get_digital(BTN_WRIST)) {
                stackStep = LOW_STACK_START + 1;
            }
            if (controller.get_digital(BTN_ARM_LOW)) {
                stackStep++;
            }
            break;
        case LOW_STACK_START + 5:
            armSeek = 1;
            wristSeek = WRIST_VERTICAL_POS;
            if (armPos < armSeek) {
                stackStep++;
                timeLastStep = millis();
            }
            break;
        case LOW_STACK_START + 6:
            if (millis() > timeLastStep + 250) {
                armSeek = -1;
                stackStep = -1;
            }
            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;
                if (autonSelect == SKILLSAUTON) {
                    flipperSeek = FLIP_POS1;
                    wristSeek = WRIST_FORWARD_POS;
                }
            }
            else if (controlMode == ARM) {
                controlMode = FLYWHEEL;
            }
        }
        justToggledMode = true;
    }
    else {
        justToggledMode = false;
    }



    // Flip in either mode

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

    // Wrist in either mode
    if (controller.get_digital(BTN_WRIST)) {
        if (!justWristToggled) {
            if (false) {     //autonSelect == SKILLSAUTON) {
                if (armSeek == 1) {
                    armSeek = ARM_HOLD_POS;
                    runTillBall = 2;
                }
                else {
                    armSeek = 1;
```

```
                runTillBall = 0;
            }
        }
        else {
            if (wristSeek != WRIST_VERTICAL_POS) {
                wristSeek = WRIST_VERTICAL_POS;
                slowSeek = false;
            }
            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;
                    if (autonSelect == REDBACKAUTON ||
                     autonSelect == BLUEBACKAUTON)
                        wristSeek += WRIST_FORWARD_EXTRA;
                }
            }
        }
    }
    justWristToggled = true;
}
else {
    justWristToggled = false;
}

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

    // Manual Overrides
    // Manual arm down
    if (controller.get_digital(BTN_ARM_DOWN)) {
        armSpeed = -100;
        armSeek = -1;
        stackStep = -1;
    }
    // Manual arm up
    if (controller.get_digital(BTN_ARM_UP)) {
        armSpeed = 100;
        armSeek = -1;
        stackStep = -1;
    }
    // Manual wrist down
    if (controller.get_digital(BTN_WRIST_DOWN)) {
        wristSpeed = -100;
        wristSeek = -1;
        stackStep = -1;
    }
    // Manual wrist up
    if (controller.get_digital(BTN_WRIST_UP)) {
        wristSpeed = 100;
        wristSeek = -1;
        stackStep = -1;
    }
    // Manual spin forks left
    if (controller.get_digital(BTN_FLIPPER_LEFT)) {
        flipperSpeed = -25;
        flipperSeek = -1;
        stackStep = -1;
    }
    // Manual spin forks right
    if (controller.get_digital(BTN_FLIPPER_RIGHT)) {
        flipperSpeed = 25;
        flipperSeek = -1;
        stackStep = -1;
    }
    // Auto stack high pole
    if (controller.get_digital(BTN_ARM_HIGH)) {
        if (stackStep == -1 || stackStep > LOW_STACK_START) {
            stackStep = HIGH_STACK_START;
        }
        else {
            if (stackStep > HIGH_STACK_START)
                stackStep = KNOCK_HIGH_START;
        }
        /*slowSeek = false;
         if (!justArmToggled) {
         if (armSeek == ARM_POS_HIGH) armSeek = ARM_POS_DOWN;
         else armSeek = ARM_POS_HIGH;
         }
         justArmToggled = true;*/
    }
    // Auto stack low pole
    if (controller.get_digital(BTN_ARM_LOW)) {       // &&
     autonSelect != SKILLSAUTON) {
        if (stackStep == -1 || stackStep < LOW_STACK_START) {
            stackStep = LOW_STACK_START;
        }
        /*slowSeek = false;
         if (!justArmToggled) {
         if (armSeek == ARM_POS_LOW) armSeek = ARM_POS_DOWN;
         else armSeek = ARM_POS_LOW;
         }
         justArmToggled = true;*/
    }
    /*else {
```

```
     justArmToggled = false;
     }*/

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


// If we need to seek, then tell the arm, wrist, and flipper
 (lerp code)
// All clamping was 100, changed to 127 3/22/19
if (armSeek > 0) {
    armSpeed = (armSeek - armPos) / armSeekRate;
    if (armSpeed > 127) armSpeed = 127;
    if (armSpeed < -127) armSpeed = -127;
    if (armSpeed < 0) armSpeed /= 1;        // slower on the
     way down
}
if (wristSeek != -1) {
    double wristLowerRed = 0;
    // Don't raise forks all the way up on RedBot due to size
     constraints
    if (autonSelect == REDAUTON || autonSelect == BLUEAUTON ||
     autonSelect == SKILLSAUTON) {
        // If we are raising the arm then we don't need to
         worry about forks
        if (armSeek <= 10) {
            if (wristSeek == WRIST_VERTICAL_POS) {
                wristLowerRed = 10;
            }
        }
    }

    // Calculate new seek based on arm position
    double actualWristSeek = wristSeek + ( armPos * 3 / 5 ) +
     wristLowerRed;

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

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

    wristSpeed = -(actualWristSeek - wristPos) /
     (wristSeekRate * wSR);
    if (wristSpeed > 127) wristSpeed = 127;
        if (wristSpeed < -127) wristSpeed = -127;
    }
    if (flipperSeek != -1) {
        flipperSpeed = (flipperSeek - flipperPos) /
         flipperSeekRate;
        if (flipperSpeed > 127) flipperSpeed = 127;
        if (flipperSpeed < -127) flipperSpeed = -127;
    }


    // 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();
    calibrateVision();

    int driveMode = 0;
    double pauseTime = 0;
    // Set pointer to chosen auton routine
    if (autonSelect == REDAUTON) autonCommand = &redAuton[0];
    if (autonSelect == BLUEAUTON) autonCommand = &blueAuton[0];
    if (autonSelect == SKILLSAUTON) autonCommand = &skills[0];
    if (autonSelect == REDBACKAUTON) autonCommand = &redBackAuton[0];
    if (autonSelect == BLUEBACKAUTON) autonCommand =
     &blueBackAuton[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;

    double startTime = millis();

    int aimTarget = 0;
    int aimLocation = 0;
```

```
bool aimPlease = false;

while (true) {

    // Auton table decipherer - switch statement
    // Commands will set flags / call object funtions
    double ds,dd,dt;
    if (nextCommand) {
        std::cout << "Next Command: " << pros::millis() <<
         std::endl;
        nextCommand = false;
        lastAutonTime = (pros::millis() - startTime)/1000;
        bool skipToElse = false;
        int ifLayer = 0;
        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) {
                    if (pauseTime == UNTIL) {
                        pauseTimeOut = (processEntry() * 1000);
                    }
                    else {
                        pauseTimeOut = (processEntry() * 1000) +
                         pros::millis();
                    }
                }
                break;
            case DRIVE:
                ds = processEntry();
                dd = processEntry();
                if (dd == CDIR) {
                    dd = gyroDirection / 10;
                }
                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 == SONAR) {
                        driveMode = dt;
                         driveDistSonar(ds,dd,processEntry(),proce
                         ssEntry());
                        std::cout << "Drive Sonar" << 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 if (dt <= WHITE_E && dt >= BLACK_R) {
                        driveMode = dt;
                        driveCustom(ds,dd,processEntry());  //
                         custom drive with timeout
                        std::cout << "Drive White Line" <<
                         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 DRIVE_TO:
                driveTo(processEntry(),
                 processEntry(),processEntry(), processEntry());
                std::cout << "Drive To" << std::endl;
                break;
            case TURN_TO:

                 turnToPoint(processEntry(),processEntry(),process
                 Entry());
                std::cout << "Turn To Point" << 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_AIM:
    aimTarget = processEntry();
    aimLocation = processEntry();
    pauseTime = (processEntry() * 1000) +
     pros::millis();
    aimPlease = true;
    std::cout << "Turn Aim" << 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;
    nextCommand = true;
    break;
case FIRE_AIM:
    autoFireState = 1;
    targetFlag = processEntry();
    autoFireTimeout = -1;
    std::cout << "Fire Aim" << std::endl;
    nextCommand = true;
    flywheelRunSpeed = -1;
    break;
case FIRE_AIM_BOTH:
    autoFireState = 1;
    targetFlag = 3;
    autoFireTimeout = -1;
    std::cout << "Fire Aim" << std::endl;
    nextCommand = true;
    flywheelRunSpeed = -1;
    break;
case FIRE:
    autoFireState = 3;
    targetFlag = processEntry();
    //flywheelRunSpeed = processEntry();
    autoFireTimeout = -1;
    std::cout << "Fire" << std::endl;
    nextCommand = true;
    break;
case STOP_FIRE:
    autoFireState = -1;
    std::cout << "Fire Aim" << std::endl;
    nextCommand = true;
    break;
case INTAKE_ON:
    runTillBall = 2;
    flipCapWIntake = false;
    std::cout << "Intake On" << std::endl;
    nextCommand = true;
    break;
case INTAKE_OFF:
    runTillBall = 0;
    flipCapWIntake = false;
    std::cout << "Intake Off" << std::endl;
    nextCommand = true;
    break;
case ARMSEEK:
    armSeek = processEntry();
    std::cout << "Arm Seek" << std::endl;
    nextCommand = true;
    break;
case SCRAPER:
    scraperSeek = processEntry();
    std::cout << "Scraper Seek" << std::endl;
    nextCommand = true;
    break;
case WRISTSEEK:
    wristSeek = processEntry();
    std::cout << "Wrist Seek" << std::endl;
    nextCommand = true;
    break;
case FLIPSEEK:
    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:
    stackStep = LOW_STACK_START;
    std::cout << "Low Stack" << std::endl;
    nextCommand = true;
    break;
case STACK_HIGH:
    stackStep = HIGH_STACK_START;
    stackStep = 1;
    std::cout << "High Stack" << std::endl;
    nextCommand = true;
    break;
```

```
case STACK_LOW_FROM:
    stackStep = processEntry() + LOW_STACK_START;
    std::cout << "Stack Low From..." << std::endl;
    nextCommand = true;
    break;
case FINISH_LOW_STACK:
    stackStep = LOW_STACK_START + 5;
    std::cout << "Finish Low Stack..." << std::endl;
    nextCommand = true;
    break;
case STACK_HIGH_FROM:
    stackTarget = HIGH;
    std::cout << "Stack high from..." << std::endl;
    nextCommand = true;
    stackStep = processEntry() + HIGH_STACK_START;
    break;
case END:
    std::cout << "Auton Finished: " << pros::millis()
     << std::endl;
    lastAutonTime = (pros::millis() - startTime)/1000;
    std::cout << "Auton Took: " << lastAutonTime << "
     Seconds" << std::endl;
    break;
case STOP_FLYWHEEL:
    autoFireState = -1;
    std::cout << "Stop Flywheel" << std::endl;
    nextCommand = true;
    break;
case STOP_COAST:
    coast = false;
    nextCommand = true;
    break;
case START_COAST:
    coast = true;
    nextCommand = true;
    break;
case INTAKE_FLIP:
    flipCapWIntake = true;
    nextCommand = true;
    break;

case IF:    // Check condition and continue, or skip
 past ELSE/ENDIF
    switch ((int)processEntry()) {
        case GOTBALL:
            if (getInnerSensor() || getOuterSensor())
                skipToElse = false;
            else
                skipToElse = true;
            break;
        case GOTBALLS:
            if (getInnerSensor() && getOuterSensor())
                skipToElse = false;
            else
                skipToElse = true;
            break;
        case AFTER:
            if (millis() - startTime >
             processEntry()*1000)
                skipToElse = false;
            else
                skipToElse = true;
            break;
        case BEFORE:
            if (millis() - startTime <
             processEntry()*1000)
                skipToElse = false;
            else
                skipToElse = true;
            break;
    }
    if (skipToElse) {
        ifLayer = 0;
        while (true) {
            int thisCommand = (int)processEntry();
            if (thisCommand == IF)      // Nested if
             is found
                ifLayer++;              // Count it
            if (ifLayer <= 0)           // If we're on
             the base level
                if (thisCommand == ELSE || thisCommand
                 == ENDIF)
                    break;              // Then break
                     when ELSE or ENDIF
            if (thisCommand == ENDIF)   // End of
             nested if
                ifLayer--;              // Discount it
        }
    }
    nextCommand = true;
    break;
case ELSE:  // Just skip to ENDIF
    ifLayer = 0;
    while (true) {
        int thisCommand = (int)processEntry();
        if (thisCommand == IF)      // Nested if is
         found
            ifLayer++;              // Count it
        if (ifLayer <= 0)           // If we're on the
         base level
            if (thisCommand == ELSE || thisCommand ==
             ENDIF)
```

```
                    break;              // Then break when
                     ELSE or ENDIF
                if (thisCommand == ENDIF)   // End of nested
                 if
                    ifLayer--;              // Discount it
            }
            nextCommand = true;
            break;
        case ENDIF: // Just continue to next step
            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;

// Check if we're withing the lidar dist
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;
}

// Check if we've seen a white line
if (driveMode <= WHITE_E && driveMode >= BLACK_R) {     //
 White line sensors
    switch (driveMode) {
        case WHITE_E:
            if (left_IR.get_value() || right_IR.get_value())
                terminateDrive = true;
            break;
        case WHITE_B:
            if (left_IR.get_value() && right_IR.get_value())
                terminateDrive = true;
            break;
        case WHITE_L:
            if (left_IR.get_value())
                terminateDrive = true;
            break;
        case WHITE_R:
            if (right_IR.get_value())
                terminateDrive = true;
            break;

        case BLACK_E:
            if (!left_IR.get_value() || !right_IR.get_value())
                terminateDrive = true;
            break;
        case BLACK_B:
            if (!left_IR.get_value() && !right_IR.get_value())
                terminateDrive = true;
            break;
        case BLACK_L:
            if (!left_IR.get_value())
                terminateDrive = true;
            break;
        case BLACK_R:
            if (!right_IR.get_value())
                terminateDrive = true;
            break;

        default:
            break;
    }
}

// If we want to aim, make sure we do!
if (aimPlease) {
    double relAngle = getRelativeAngle(aimTarget,
     aimLocation);
    if (abs(relAngle) < AIM_ACCEPT && relAngle != 0)  {
        aimPlease = false;
        nextCommand = true;
        pauseTime = 0;
        std::cout << "Aim Finished - " << pros::millis() <<
         std::endl;
    }
    turnRelative(relAngle, -1);
}

// Check if we should stop pausing
if (pauseTime == UNTIL) {
    if (millis() - startTime > pauseTimeOut) {
        pauseTime = 0;
        nextCommand = true;
        pauseTimeOut = 0;
        std::cout << "Pause Finished Wait Till - " <<
         pros::millis() << std::endl;
    }
} else 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;
            aimPlease = false;
            driveStop();
            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() ||
         getOuterSensor())) {
            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 (pauseTime == STACKED && stackStep == -1) {
            nextCommand = true;
            pauseTime = 0;
            std::cout << "Pause Finished - " << pros::millis() <<
             std::endl;
        }
        if (pauseTime == SCRAPER_UP && scraperPos < 10) {
            nextCommand = true;
            pauseTime = 0;
            std::cout << "Pause Finished - " << pros::millis() <<
             std::endl;
        }
        }

        // If we should stop driving, then do!
        if (terminateDrive) {
            std::cout << "Stop Drive" << std::endl;
            driveMode = 0;
            driveStop();
            nextCommand = true;
        }


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


// Print the auton mode to the controller screen
void run_screen(void* params) {
    while (true) {
        if (autonSelect == REDAUTON)
            controller.print(0,0, "RED FRONT ");
        else if (autonSelect == BLUEAUTON)
            controller.print(0,0, "BLUE FRONT");
        else if (autonSelect == SKILLSAUTON)
            controller.print(0,0, "SKILLS    ");
        else if (autonSelect == REDBACKAUTON)
            controller.print(0,0, "RED BACK  ");
        else if (autonSelect == BLUEBACKAUTON)
            controller.print(0,0, "BLUE BACK ");

        delay(200);
    }
}


/**
 * 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

    calibrateVision();

    int lastBlinkTime = millis();
    bool justToggledAuto = false;
    int startTime = millis();
    int vibDone = 0;

    if (autonSelect == SKILLSAUTON) {    // Auto-deploy at start of
     driver skills
        wristSeek = WRIST_VERTICAL_POS;
        scraperSeek = SCRAPER_DOWN_POS;
        runTillBall = 2;
        coast = true;
    }


    while (true) {

        // Button to reset the gyro for testing
        if (controller.get_digital(BTN_CHOOSE_AUTON)) {
            setGyro(0);
        }

        // Print some info the the terminal
        std::cout << "Sensor: " << sensor_gyro.get_value() << " Gyro:
         " << gyroDirection << " Direction: " << direction <<
         std::endl;
        //std::cout << " Arm Pos: " << armPos << " Wrist Pos: " <<
         wristPos << " Flip Pos: " << flipperPos << " Stack Step " <<
         stackStep << std::endl;
        std::cout << "X: " << xPosition << ", Y: " << yPosition << ",
         D: " << trackingDirection << std::endl;
        std::cout << "Last Auton Took: " << lastAutonTime << "
         Seconds" << std::endl;
        int  count_B = 0;
        int  count_R = 0;
        int  count_G = 0;
        int noObjs = camera.get_object_count();
        std::cout << "N: " << noObjs << std::endl;

        // Flash the light fast if the camera is not working
        if (noObjs > 1000) {
            if (millis() > abs(lastBlinkTime) + 62) {
                if (lastBlinkTime > 0) {
                    lastBlinkTime = -millis();
                    ballLight.set_value(1);
                }
                else {
                    lastBlinkTime = millis();
                    ballLight.set_value(0);
                }
            }
        }

        // Code to count number of each type of flag
        if (noObjs > 27) noObjs = 27;
        for (int i = 0; i < noObjs; i++) {
            vision_object_s_t thisThing = camera.get_by_size(i);
            if (thisThing.signature == BLUE_FLAG)
                count_B++;
            if (thisThing.signature == RED_FLAG)
                count_R++;
            if (thisThing.signature == GREEN_FLAG)
                count_G++;
        }
        if (count_B > 0 || count_R > 0 || count_G > 0)
            std::cout << "B: " << count_B << " R: " << count_R << " G:
             " << count_G << std::endl;

        // Print auton mode to brain screen
        if (autonSelect == REDAUTON)
            pros::lcd::print(0, "FRONT RED FRONT RED FRONT RED FRONT
             RED FRONT RED FRONT RED FRONT RED");
        else if (autonSelect == BLUEAUTON)
            pros::lcd::print(0, "FRONT BLUE FRONT BLUE FRONT BLUE
             FRONT BLUE FRONT BLUE FRONT BLUE");
        else if (autonSelect == SKILLSAUTON)
            pros::lcd::print(0, "SKILLS SKILLS SKILLS SKILLS SKILLS
             SKILLS SKILLS SKILLS SKILLS");
        else if (autonSelect == REDBACKAUTON)
            pros::lcd::print(0, "BACK RED BACK RED BACK RED BACK RED
             BACK RED BACK RED BACK RED");
        else if (autonSelect == BLUEBACKAUTON)
            pros::lcd::print(0, "BACK BLUE BACK BLUE BACK BLUE BACK
             BLUE BACK BLUE BACK BLUE");

        // Print some more info to screen
        pros::lcd::print(2, "Direction: %f", direction);
        pros::lcd::print(3, "Arm: %.0f Wrist: %.0f Flipper: %.0f",
         armPos, wristPos, flipperPos);
        pros::lcd::print(4, "Stack Step: %f", stackStep);
        pros::lcd::print(5, "(%.3f, %.3f,  %.3f)", xPosition,
         yPosition, trackingDirection);
        pros::lcd::print(1, "Auton Time: %f", lastAutonTime);
        pros::lcd::print(6, "Sonar Dist: %i", sonar.get_value());
        pros::lcd::print(6, "Arm Diff: %f", (armSeek - armPos));
```

```
        // If we press this button combo, switch auton mode
        if ( controller.get_digital(BTN_ABORT) &&
         controller.get_digital(BTN_CHOOSE_AUTON) ) {
            if (!justToggledAuto) {
                autonSelect++;
                if (autonSelect > NUMBER_AUTONS - 1) {
                    autonSelect = 0;
                }
                calibrateVision();
            }
            justToggledAuto = true;
        }
        else {
            justToggledAuto = false;
        }

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