My Projects

page	project	date
1	Programming Introduction	Nov 12, 2017
2	Sensor Overview	Nov 12, 2017
3	Main Mechanisms	Nov 12, 2017
4	Controller Config	Nov 13, 2017
5	Base Control: Deadzone	Nov 13, 2017
6	Stalling Motors Problem	Nov 14, 2017
7	Slew rate Code	Nov 14, 2017
8	Autonomous Selection	Nov 14, 2017
9	Autonomous Selection Code	Nov 14, 2017
10	Claw Hold Feature	Nov 15, 2017
11	Auto Drive Forward	Nov 15, 2017
12	Proportional-Integral-Derivative (PID)	Nov 15, 2017
13	Lift Task	Nov 16, 2017
14	~~Autro~~ Auto Gyro Turn	Nov 16, 2017
15	PID Code Commented	Nov 19, 2017
16	PID Explanation Continued	Nov 19, 2017
17	Contined PID Explanation	Nov 19, 2017
18	Drive Lift Assistance	Nov 18, 2017
19	Autonomous Selection Update	Jan 13, 2018
20	New Navigation Tactic	Jan 13, 2014
21	Aligning with 10 point Bar	Jan 14, 2018
22	Programming Skills	Jan 14, 2018
23	Mobile Goal - Auto Control	Jan 17, 2018
24	Programmed Lift Stabilization	Jan 16, 2018
25	Calibration Button	Jan 16, 2018
26	Upper Lift Syncronization	Jan 19, 2018
27	Upper Lift Control	Jan 19, 2018
28	Auto stacking Task	Jan 19, 2018
29-31	Program Defines	Feb 22, 2018
32	Motor Functions	Feb 26, 2018
33	Auto Gyro Turn 2.0	Feb 26 2018
34	Triple Gyro Average	Feb 26 2018
35	Mechanism Calibration 2.0	Feb 26 2018
36	Driver Base Task	Feb 26 2018
37	Driver Simple Controls	Feb 26 2018
38	Auto Mogo Control 2.0	Feb 26 2018
39	Auto 4 Bar Control 1.0	Feb 26 2018
40	Auto Drive Forward	Feb 26 2018
41	Drive For Time	Feb 26 2018

My Projects

page	project	date
42	Secondary Drive Function	Feb 26 2016

Importance of Programing

11/12- Without a strong, reliable program, our robot cannot function, no matter how well its build might be. The team's programmer will tackle tasks such a coding a 15 second autonomous, a 60 second autonomous, and creating reliable functions to assist the drive during a match. The builders and programmers must communicate to ensure the installation of sensors and to identify build weaknesses and vulnerabilities.

While the builders only have 4 hours every Friday to work, the programmers have much more time to program as they can do it from home.

IF possible, the main programmer should have prior knowledge around programming, and possibly a human resource. While I, Jacob, do have past knowledge of programming, I will give the role of head ~~builder~~ programmer to Theo, my new team member.

Theo has done amazing work with programming LEGO robots using a vast range of sensors. Also, programming's something he enjoys to do. -Jw

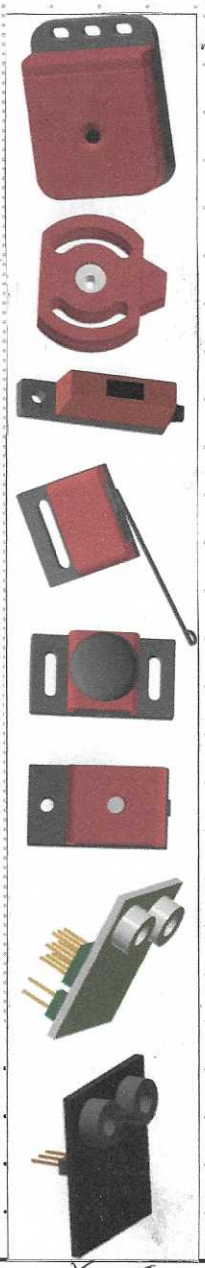
project Programming Intro designed by: Jacob W. witnessed by:

date: Nov 12 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Sensor Overview

11/12 - There are many sensors that VEX has to offer that will allow use to create more intelligent programs. It is important that we know what they are and how they work. -Jr



Optical Shaft encoder
- Used to measure rotation of an axle.
- Measures degrees. 360 degrees = 1 rotation

Potentiometer
- Used to measure rotation, like the optical sheft encoder. Limits to 270° and output: 0-4095

Line Followers
- Measures reflective light. (White-Black)

Limit Switch
- Digital sensor that gives a 1 or a 0 when the button is pressed.

Bumper Switch
- Same as the limit switch, but no arm

Light Sensor
- Measures ambiant light (Hues)

Accelerometer
- Measures directional velocity. XYZ

Gyroscope Sensor
- Measures angular velocity.
- Can measure ~~up to~~ down to one tenth of a degree. -Jr

project Jr. Sensors designed by: Jacob W. witnessed by:

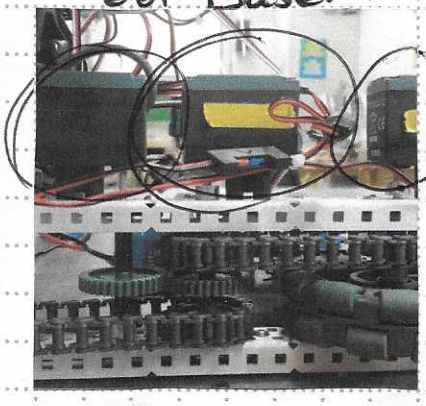
date: May 12 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Main Mechanisms

11/12- Our robot needs programming for many of its mechanisms. The buildings is mostly complete of our robot as of now, so we have the sensors in place.

Our Base:

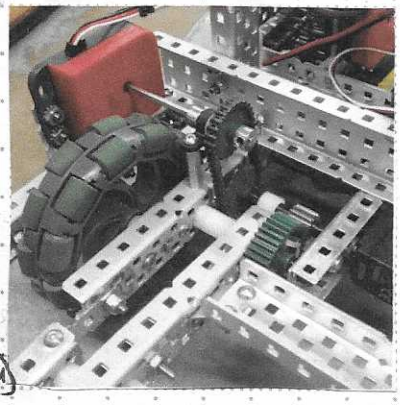


- 3 Motors Per side
- Geared 1:1
- 160 RPM Motors
- One Shaft encoder
- 3" tolerance

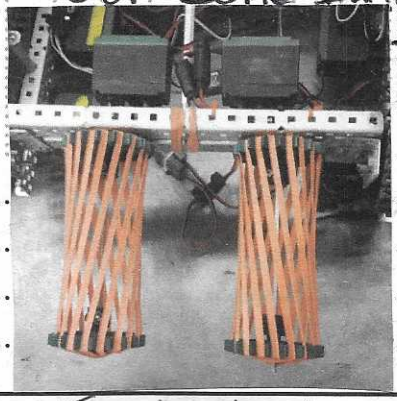
Our Lift: 2 Motors (160 RPM) w/ one Encoder.

Our Mobile Goal Lift:

- 2 Torque motors (100 RPM)
- 1 Shaft encoder
- 1:1 geared, but before the 1:3 step up. 90° = 200° on encoder.



Our Cone Intake:



~~Intake~~: 2 motors (240 RPM)
No Sensors.
- Needs to move inward and outward to intake and place cones.
- Needs a button to apply small power to hold cones.

project ~~Jacob~~ Mechanisms designed by: Jacob L. witnessed by:

date: Nov 12 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Controller Configuration

11/12 - To control our robot, we use a standard VEX Joystick. It has 8 buttons on the front, 4 buttons on the back, and 2 sticks on the front. Below is our desired control layout.

-This will be our standard config for driving because of how easy it is to learn. The placement of our mobile goal lift buttons allow us to drive forward and back while lifting.

This will be updated as we use more of the buttons when we come up w/ new features.

project Controller Config designed by: Jacob W. witnessed by:

date: Nov 13 2012

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Base Control: Dead zone

11/13: One issue we had with driving was our controller giving unintended values to the base motors. Even when we did not touch the controller, the base would still move at a very slow rate because the controller rested at 15 power. -Jr

To counteract this, we ~~em~~implemented a thing called a deadzone, which only applies power if the controller is above a certain value.

```
//Base ////////////////////////////////////////////////////////////
if(abs(vexRT[Ch2]) > 10 || abs(vexRT[Ch4]) > 10 )
{
  motor[right] = vexRT[Ch2] - vexRT[Ch4];
  motor[left] = vexRT[Ch2] + vexRT[Ch4];
}
else
{
  motor[right] = 0;
  motor[left] = 0;
}
//Base ////////////////////////////////////////////////////////////
```

Controller Controller

Otherwise, turn motors off.

Only update the motor power when the controller is at a power of 10 or higher.

-Written By Theo

This also stops the motors from whining we it is not in use. When motors are ran at low power and cannot convert the electricity into motion, it is made into sounds and heat. That is not a good thing because it will make our motors stall faster and drain our batteries faster. -Jr

project Dead Zone designed by: Jacob W. witnessed by:

date: Nov 13 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Problem: Stalling Motors

11/13 A ~~mag~~ major problem our team ~~th~~ was experiencing was stalling. After our motors stall, we need to wait 5-10 seconds before continuing to drive. The reason our motors stall is because the power is converted into heat instead of motion because our base cannot overcome its inertia quick enough.

We have found a way to combat this through programming, though. The problem is created when we go from 0-127 power too quickly, or even worse, ~~e~~ -127-127 power. To stop this from happening, the driver can drive more ~~et a~~ carefully to ensure that the speed is gradually increased over time instead of at once.

What we have insted done is implemented a "Slew rate" function to gradually adjust speeds for us. Even if the driver goes from -127 power to +127 power, the robot will only allow itself to make that ~~adj~~ adjustment in ≈400ms.

This approach has worked great for us so far and has ~~reb~~ reduced stalling, not entirely, but somewhat. Careful driving is still important.

project Stalling Motors designed by: Jacob W. witnessed by:

date: Nov. 14, 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Slew Rate Code

File: C:\Users\Owner\Google Drive\7842F Vex Code\Up To Date Current Code\SlewRateTa

```
//Slew Rate ////////////////////////////////////////////////////////
#define MOTOR_NUM                kNumbOfTotalMotors
#define MOTOR_DEFAULT_SLEW_RATE  15      // Default will cause 375mS from full fwd t
#define BASE_SLEW                8
#define MOTOR_TASK_DELAY         15      // task 1/frequency in mS (about 66Hz)
int motorReq[ MOTOR_NUM ];
int motorSlew[ MOTOR_NUM ];

task MotorSlewRateTask()
{
  int motorIndex;
  int motorTmp;
  // Fill Up Arrays with 0 and Default slew rate for every motor
  for(motorIndex=0;motorIndex<MOTOR_NUM;motorIndex++)
  {
    motorReq[motorIndex] = 0;
    motorSlew[motorIndex] = MOTOR_DEFAULT_SLEW_RATE;
  }
  // Full Task
  while( true )
  {
    // for loop through every motor
    for( motorIndex=0; motorIndex<MOTOR_NUM; motorIndex++)
    {
      // So we don't keep reading current motor speed from robot
      motorTmp = motor[ motorIndex ];
      // Do we need to increment the motor speed ?
      if( motorTmp != motorReq[motorIndex] )
      {
        // increasing motor value
        if( motorReq[motorIndex] > motorTmp )
        {
          motorTmp += motorSlew[motorIndex];
          // limit
          if( motorTmp > motorReq[motorIndex] )
            motorTmp = motorReq[motorIndex];
        }
        // decreasing motor value
        if( motorReq[motorIndex] < motorTmp )
        {
          motorTmp -= motorSlew[motorIndex];
          // limit
          if( motorTmp < motorReq[motorIndex] )
            motorTmp = motorReq[motorIndex];
        }
        // finally set motorTmp to real motor
        motor[motorIndex] = motorTmp;
      }
    }
    // Wait To give structure of time
    wait1Msec( MOTOR_TASK_DELAY );
  }
}
// Slew Rate////////////////////////////////////////////////////////
```

I like the comments keep this up

project Slew Rate designed by: Theo L. witnessed by:

date: Nov. 14 2017

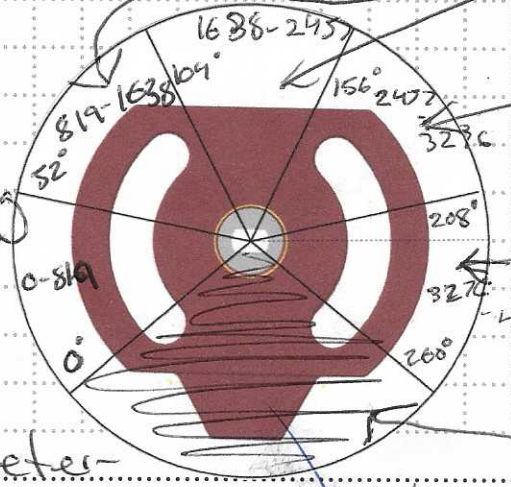
PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Autonomous Selection

11/13 - One of our biggest goals this year is to do well in programming. Since I have an amazing team member named Theo, who is a very talented programmer, we are headed on the right track.

It is important to have more than 1 autonomous program in case ours conflicts with our alliance's. We also need different versions of one program because of the 2 sides of the court (right and left.)

We can use the potentiometer (outputs up to 4095) and the LCD to show our selection. Currently, we are using 5 autonomous programs, so to make that work, we need to break the potentiometer into 5 different parts



As you can see on the right, we can divide the potentiometer's max value of 4095 by 819 to get 5 different sections. By using the wonderful switch/case statement, we can easily refer to the sensor's value to determine which program to run. We also made this its own task so it can constantly update all the time. -Jacob

project Auto Selection designed by: Jacob V. witnessed by:

date: Nov 14 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Autonomous Selection Code

```
task LCD(){

bLCDBacklight = true; //Turns LCD backlighting on
  clearLCDLine(0); //Clears LCD values

  while(true){ //Loop forever
    switch(SensorValue[auto]/819){ //Check value of potentiometer reading
    case 0:                        //dividied by 819 to equal a maximum of 5
    //If potentiometer divided by 819 is 0, complete following:
      //Blue 1
      displayLCDPos(0,0);
      displayNextLCDString("   Blue Right   ");
      displayLCDPos(1,0);
      displayNextLCDString("    20 Point    ");

      break;
    case 1:
    //If potentiometer divided by 819 is 1, complete following:
      //Blue 2
      displayLCDPos(0,0);
      displayNextLCDString("   Blue Left    ");
      displayLCDPos(1,0);
      displayNextLCDString("    20 Point    ");

      break;
    case 2:
    //Etc...
      //Skills
      displayLCDPos(0,0);
      displayNextLCDString("     Skills     ");
      displayLCDPos(1,0);
      displayNextLCDString("");

      break;
    case 3:
      //Red 1
      displayLCDPos(0,0);
      displayNextLCDString("    Red Right   ");
      displayLCDPos(1,0);
      displayNextLCDString("    20 Point    ");

      break;
    case 4:
      //Red 2
      displayLCDPos(0,0);
      displayNextLCDString("    Red Left    ");
      displayLCDPos(1,0);
      displayNextLCDString("    20 Point    ");

      break;
    default:
      displayLCDPos(0,0);
      displayNextLCDString("     7842F      ");
      displayLCDPos(1,0);
      displayNextLCDString("   gr8 bot m8   ");

  }
```

Only if sensor/819 > 0 and < 1

Only if potentiometer/819 > 1 and < 2

Only if 3 > potentiometer/819 > 2

Only if 4 > sensor/819 > 3

Only if 5 > sensor/819 > 4

Only if nothing else

project Auto Select Code designed by: Jacob W. witnessed by:

date: Nov 14 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

"Claw Hold" Feature

11/15 - After picking up a cone, we should continue to apply a small amount of power to grasp the cone as we move around.

There are multiple ways we could have done this through mechanical changes, but we decided to take a programming approach. Jr

We will use a standard "if" statement to check if we press the button to close our cone intake, and if it is not pressed, we will do something other.

```
//Claw ////////////////////////////
if(DriverClawOverride)
{
  if (vexRT[Btn5D])
  {
    motorReq[claw] = 127;
    claw_power = 20;
  }
  else if (vexRT[Btn5U])
  {
    motorReq[claw] = -127;
    claw_power = 0;
  }
  else
  {
    motorReq[claw] = claw_power;
  }
}
//Claw ////////////////////////////
```

1.) When the down button is pressed,

2.) Set the cone intake to full power to outtake cones.

3.) When the up button is pressed:

4.) Set cone intake to full power to intake cones.

5.) Set claw_power to 20.

6.) When neither button is pressed, set intake to claw_power

While super simple, the claw hold feature will be amazingly helpful.

project Claw hold designed by: Jacob W. witnessed by:

date: Nov 15 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto Drive Forward

11/15- for our autonomous programs, we need a way to go to exactly the desired distance ~~without~~ no matter how charged the batteries are.

To ~~to~~ do this, We will use the base encoders that we installed on our base. To get the distance we will go per 360 wheel degrees, we need to:

4" Diameter wheel = $4 \cdot \pi$ = 12.566" per rotation.
360° / 12.566 = ~~6~~ 28.65° per inch

∴ if we want to go 48";

To go 48", the encoder must reach 1375.2 degrees.

We drive toward our goal until we reach it, then, we go back a bit

```
// Distance Drive Function ////////////////////////////////
void drive(int inch) ← input
  //Encoders set to 0
  SensorValue[leftEn] = 0;
  /*Integer 'deg' (degrees) is the quotient of our specified
  | distance and our wheel's circumference multiplied by 360.
  */
  int deg = inch / 12.57 * 360;
  //When the average value of the encoders are less than 'deg', run motors.
  while ( abs(SensorValue[leftEn]) < abs(deg)-100)
  {
    //If the specified distance is positive, run the motors forward...
    if (inch > 0){
    motor[right] = 127;
    motor[right2] = 127;
    motor[left] = 127;
    motor[left2] = 127;
      } else {
      //... otherwise, run them backwards.
      motor[right] = -127;
    motor[right2] = -127;
    motor[left] = -127;
    motor[left2] = -127;
```

project Auto Drive Forward designed by: Jacob W. witnessed by:

date: Nov. 15 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Proportional-Integral-Derivative (PID)

For our robot, we are using many PID control systems. We use them for Autonomous code such as driving based on the Encoder or turning based on the Gyroscopic sensor. We also use it to control the lift to make the lift self-supporting and be able to maintain a position. Here is an explanation of how our PID systems work.

This is our code for the driving forwards: Refer to page 15

Let's start with no PID. For this example, we will use the encoders to drive forwards a distance.

A code for this would be something along the lines of:

```
while ( CurrentDistance != WantedDistance) // Loops until current distance is equal to wanted dista
{
  CurrentDistance = SensorValue[leftEn]; // Stores the current distance

  //If the wanted distance is greater than current distance, run the motors forward...
  if (WantedDistance > CurrentDistance){
    motor[right] = 127;
    motor[right2] = 127;
    motor[left] = 127;
    motor[left2] = 127;
    } else {
    //... otherwise, run them backwards.
    motor[right] = -127;
    motor[right2] = -127;
    motor[left] = -127;
    motor[left2] = -127;
  }
}
// Stop the motors
motor[right] = -127;
motor[right2] = -127;
motor[left] = -127;
motor[left2] = -127;
```

Very simple. If the wanted distance is greater than current distance, run the motors forward, otherwise run them backward.

However, this code will never work. Since the robot can only move at 100% power, it will speed up very fast to the goal and gain much momentum. It will then overshoot the goal, and exit out of the loop because the Encoders were in the right place for a fraction of a second. It will then go forwards for quite a distance more because of the robot's momentum, and there is no brake.

To fix this we could put a wait at the end of the loop. This will let the robot overshoot past the goal without stopping the loop, and apply reverse power. However, the code will still not work. The robot will oscillate forwards and backward at full power and never stop because the robot will never be at 0 power at the correct spot. It will always overshoot its goal because it is applying 100% power when the robot only has a tiny amount of distance left to achieve.

What most people do to resolve this issue is to have a code that applies 100% power to the robot until it has completed 80% of its goal. Then the robot will slowly move to the final goal at...

Continued on page 16

project PID Intro designed by: Theo L. witnessed by:

date: Nov 19, 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Lift Task

11/16- We use PID on our lift to hold it exactly where we want it. We have a shaft encoder on the lift, geared down 5:1, so we can be accurate to down to 0.2° degrees.

```
//Lift Speed PID Task //////
task driverLiftPid()
{
  float finalpower = 0;
  int lasterror = 0;
  int error = 0;
  int touched = 0;
  float CapError = 256;
  float totalerror = 0;
  float kp = 0.3;
  float ki = 0.000;
  float kd = 1;
  int lift_power;
  SensorValue[liftEn]=0;
```

Above is are variables.

```
  while(true) Contines to loop while running.
  {
    //
    if(touched > 0 && vexRT[Btn6U] + vexRT[Btn6D] < 1)
    {
      DriverLiftDesired=SensorValue[liftEn];
      touched = 0;
    }
```

Above's what causes the desired angle to update.

kp, ki, and kd are all values we use to control our PID.

```
  error = DriverLiftDesired - SensorValue[liftEn];
  finalpower = (kp * error) + (ki * totalerror) + (kd * (error - lasterror));
  if(vexRT[Btn6D]) 0.3 x error + 0 + rate.
  {
    for (lift_power = 20; vexRT[Btn6D]; lift_power++)
    {
      motor[chainbar] = lift_power;
      wait1Msec(5);
    }
    touched = 1;
  }
  else if(vexRT[Btn6U])
  {
    for (lift_power = 20; vexRT[Btn6U]; lift_power++)
    {
      motor[chainbar] = -lift_power;
      wait1Msec(5);
    }
    touched = 1;
  }
  else
  {
    motor[chainbar] = finalpower;
  }
  totalerror += error;//used for I
  if(totalerror > CapError){
    totalerror = CapError;
  }
  lasterror = error;//used for D
```

Code to control the lift manually

When the lift isn't manually controlled, PID turns on

```
    wait1Msec(20);
  }
}
```

At the end, we have a 20 ms delay to save resources.

project Lift Task designed by: Theo Lemay witnessed by:

date: Nov 16 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto Gyro Turn

11/16- We use our gyro sensors and PID to get to the exact same place each time. As we get closer to our goal we slow our speed as to not overshoot.

```
void AutoGyroTurn(int wantedDegrees){
  int motorPower;
  float error;
  int currentDegrees = 0;
  float kp = 0.18;
  float ki = 0.05;
  float kd = 0.015;
  float previousError;
  float rateError;
  float totalError;
  int CapError = 300;
  int MaxTime = 2000;
  int RefreshRate = 20;

  while(wantedDegrees != currentDegrees && MaxTime != 0)
  {
    currentDegrees = SensorValue[in8];

    error = wantedDegrees - currentDegrees;

    if(abs(error) < 250)
    {
      totalError += error;
      if(abs(totalError) > CapError) {
        totalError = CapError;
      }
      else if(abs(totalError) < -CapError) {
        totalError = -CapError;
      }
    }

    rateError = error - previousError;
    motorPower = (error * kp) + (totalError * ki) + (rateError * kd);
    motor[right] = motorPower;
    motor[right2] = motorPower;
    motor[left] = -motorPower;
    motor[left2] = -motorPower;
    previousError = error;
    MaxTime -= RefreshRate;
    wait1Msec(RefreshRate);
  }
  motor[right] = 0;
  motor[right2] = 0;
  motor[left] = 0;
  motor[left2] = 0;
}
```

Function includes a specified angle.

Carefully Tuned

While turn is 25 degrees away from goal

Maximum milliseconds we allow the loop to go for

Syncs program w/ motor rate

Gathers gyro sensor angle

25° away from goal

Stop the robot from going too fast while close to goal.

Motors are assigned power based on how far away from the goal the robot is.

At the end of the function, motors are set to 0 to stop.

project Gyro turn designed by: Theo L & Jacob witnessed by:

date: Nov. 16. 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

PID Code Commented

```
void AutoDriveInch(int inch){

  int wantedDegrees = inch / 12.57 * 360; // Calculates how many degrees are wanted from Inch
  int motorPower; // Variable to store motor power before outputing it to the motors
  float error; // Variable to store difference between wanted degrees and current degrees
  int currentDegrees = 0; // Variable to store current degrees
  //This is so that we only need to read from the cortex once every loop instead of multiple times

  float kp = 0.6; // The Porpotional Weight Constant - Porportional

  float ki = 0.07; // The Integral Weight Constant - Accumilated Error
  /* The Integral will harm me to move forwards because it will reach its max and never lower until I overshoot,
  but then take a while to correct.
  This is solved by only changing the I if the robot is within 40 degrees from its target */

  float kd = 0.5; // The Derivative Weight Constant - Rate of change

  float previousError; // A variable storing the last error used to calculate the rate of change for D
  float rateError; // A variable storing the rate of change used for D

  float totalError; // The accumilated error used for I
  int CapError = 230; // The maximum threshold for I

  SensorValue[leftEn] = 0; // Reset the Encoder to 0

  while(wantedDegrees != currentDegrees)  // Loops until wanted degrees equals current degrees
  {
    currentDegrees = SensorValue[leftEn]; // Stores the current degrees of the Encoder into a variable

    // Calculates the error by subtracting the current degrees from the wanted degrees
    error = wantedDegrees - currentDegrees;

    if(abs(error) < 40) // Only adds to I if the robot is within 40 degrees from its target
    {
      totalError += error; // Adds the current error to the total error variable

      // If the total error is greater than the maximum, the total error becomes the maximum
      if(totalError > CapError) {
        totalError = CapError;
      }
      // If the total error is smaller than the negative of the maximum,
      // the total error becomes the negative maximum
      else if(totalError < -CapError) {
        totalError = -CapError;
      }
    }

    // Calculates the rate of change by subtracting the previous error from the current error
    rateError = error - previousError;

    motorPower = (error * kp) + (totalError * ki) + (rateError * kd);
    // The motorPower variable is assinged:
    // the error times the porportional constant, plus
    // the accumilated error times the integral constant, plus
    // the rate of change times the derivative constant.

    // All drive motors are assinged the calculated power from above
    motor[right] = motorPower;
    motor[right2] = motorPower;
    motor[left] = motorPower;
    motor[left2] = motorPower;


    // Stores the error into a variable for the next loop to use for calculating the rate of change for D
    previousError = error;

    // Waits a specified time to give the rate of change a time reference,
    // and to let the robot overshoot past the goal to apply reverse power.
    wait1Msec(20);
  }

  // After the loop exits, apply 0 to all the drive motors
  motor[right] = 0;
  motor[right2] = 0;
  motor[left] = 0;
  motor[left2] = 0;
```

Continued on next page...

project PID Code Commented designed by: Theo L. witnessed by:

date: Nov 19, 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

PID Explanation Continued...

20% power. This works, but is not ideal in many conditions, and is time-consuming.

The best solution for this is to have the motor power be dynamically calculated. This is what a PID does.

A PID system calculates the power sent to the motors based on three things:

- The difference between the wanted position and the current position (error) times a proportional constant – P for Proportional
- The accumulated error times a proportional constant – I for Integral
- The current error minus the previous error times a proportional constant – D for Derivative (rate of change)

Each of these three sections each output a number. The trick for tuning a PID is figuring out what is the likely range for each of these outputs and what do we want them to proportionally output to the motor.

To explain, let's start with P.

If we make the motor power be proportional to the amount of distance remaining, it could work. We can stick to the example of driving forwards based on the Encoders.

If the wanted distance is 1000 Encoder degrees, and its current degree is 0, and the Proportional Constant (Kp) is 0.1, then the motors will initially have the power of 100 applied to them. As the robot moves forwards, the current encoder value increases, so the remaining distance decreases.

Let's say the robot has now moved half of its wanted distance. Then the wanted degrees minus the current degrees (error) equals 500 degrees. That times 0.1 equals 50, which is applied to the motors.

If we add a wait of 20ms at the end of the loop like the non-PID example, our robot will near the goal at a very low power and overshoot. Then the robot will slowly fine-tune its position until it is right on target.

Success!

However, there can still be issues with P. This is where the I comes in.
In a proportional program like the one above, if the robot settles 1% from the goal and has no momentum, you have a problem. Using the earlier example, the distance remaining will be 100 degrees. 100 multiplied by 0.1 equals 10 power. In many cases, that is not enough power to move the robot. the motors will just whine. Therefore. the robot is stuck.
I is the accumulated error. If every loop the error of 100 degrees is added up every 20ms, then

project PID Explanation designed by: Theo L. witnessed by:

date: Nov. 19, 2017

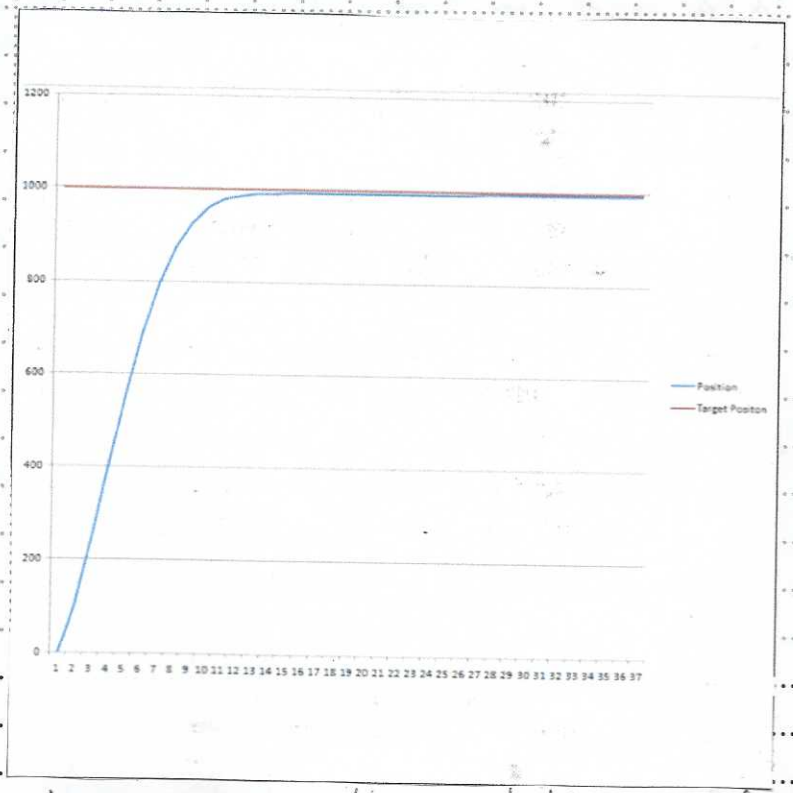
PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

More PID Explanation

quickly the motors will have enough power to start moving. As soon as the robot overshoots, the error becomes negative, so the I decreases. Another example is if the goal is moving at the same speed as the robot and it is not catching up because the P is not changing, the I will accelerate the robot to catch up to the goal. The issue with I is that it is not very responsive, so gets in the way of going forwards. This is because it goes to its maximum right at the start, and takes a while to wind back down to 0 after it overshoots. This can be fixed by only using the I if the robot is 5% from its goal.

For extra responsiveness, the D is added. D is a way to predict the future, and therefore act on it. D is the rate of change, so if the target moves or the error suddenly increases, for example, if we are using PID for balancing an arm and force is applied to the arm, the D jumps up. This is good for aggressively maintaining a position or sticking to a moving goal.

All of these added together, and weighed with their respective constants, resulting in a smooth and efficient code.



Instead of overshooting the target, PID allows us to slow as we approach our goal.

This saves us a lot of time during autonomous because we don't need to do corrections.

You can see in the graph on the left that approach slows as the robot's position gets closer to the target.

project PID Explanation designed by: Theo Lemay witnessed by:

date: Nov. 19 2017

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Driver Lift Assistance

11/19 - While driving in the court, there is a lot to focus on. We have developed a system to automatically stack cones. This makes it much easier for the driver because the robot takes the load of stacking off.

Continued 2 months later...

We programmed the lift to know where it needed to be in order to properly stack cones.

Each time the cone stack button was pressed the lift would go to a set place and stack the cone. It would also use the same button to add cones to the LED indicator: PAGE 30 BUILD JOURNAL.

While this method was somewhat unreliable at the time, we have since improved it on our new robot.

We also no longer have the cone for this assistance function. One error with our method in this case was that while the sequence of stacking, the driver couldn't control the robot. We have since changed this in newer versions.

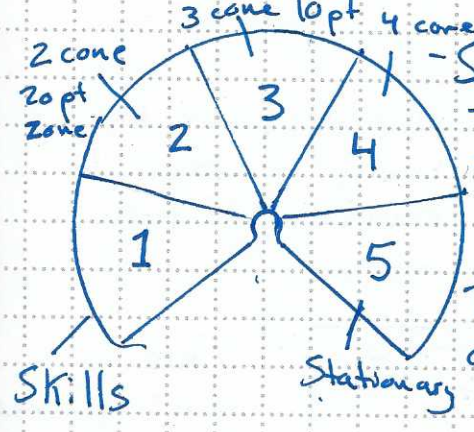
project Driving Assist designed by: Jacob W. witnessed by:

date: Nov. 18, 2012

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

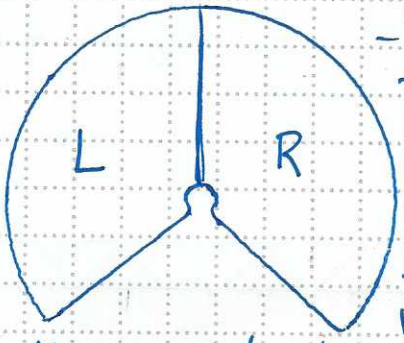
Auto Selection Update

1/13 - We now have 2 dials on our robot for autonomous selection. One is for choosing one of 5 programs and the other is for selecting either the right ~~of~~ or left side.



- Selector 1: Program Selector
- Up to 5 different programs
- Created to attach to the potentiometer with ease.

This lets us change our program on the fly.



- Selector 2: Side Selector
- Allows us to communicate to the robot if we are on left or right side.

We have implemented this by multiplying our turn angles by -1. this inverts the turns, making them work on the opposite side of the field.

Like before, we are using switch/case statements. More on that on page 9.

We also painted them Orange, which is the most important part, obviously.

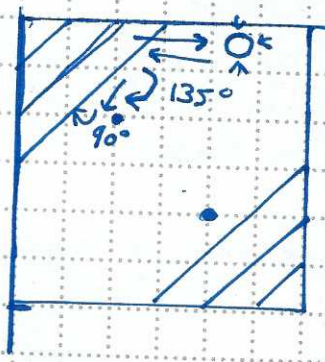
project Auto Selection Update designed by: Jacob W. witnessed by:

date: Jan 13 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

New Auto Navigation Tactic

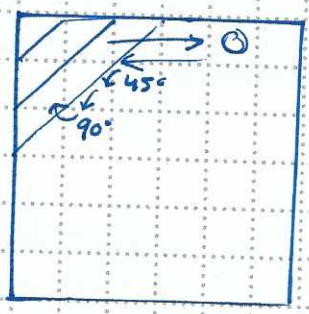
1/13 - In our autonomous programs previously, we would drive up to the mobile goal and pick it up. Then we would reverse and turn right 135° and drive ahead again before another 90° turn to face the 20 pt zone:)



135 + 90 = 225°
45 + 90 = 135°

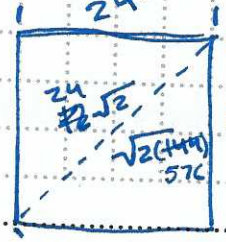
The new ~~method~~ of turns can be done 66% faster. -Jm

We discovered that it was actually faster to do a reverse 45° turn like so:



This is much faster ~~than~~ than the ~~13~~ 135 degree turn.

★ Tile Dimensions and Math ★



Width = 24" Diagonal: $24\sqrt{2}$" ≈ 33"

~~Lenght~~

Length: 24" Diagonal = 33"

We will use both pieces of information on the following pages. -Jm

project Nav Tactic + Math designed by: Jacob W witnessed by:

date: Jan 13 2015

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Aligning with 10 point Bar

1/14 - While we use a gyro sensor for our turns, sometimes if we use it too much it drifts. This causes us to miss our target. In order to fix that issue, we can back into the ten point bar to align with the mobile goals.

We can do this by reverse the base motors slow enough ~~st~~ so they won't overcome the bar, but press against it. We will also use the base encoders to check the rate of update the sensors show so the ~~rod~~ robot knows when it hits the bar. -

Usage: alignWithBar(-60);

```
void alignWithBar(int power){
  firstPos = SensorValue[baseEn]; //Creates 'firstPos' and sets to base encoc
  motor[left] = power; //Runs base at specified speed.
  motor[left2] = power;
  motor[right] = power;
  motor[right2] = power
  wait1Msec(100); //Allows base to move
  secondPos = SensorValue[baseEn]; //Creates 'secondPos' and sets to base enc
  //(Should be higher than firstPos if the robot moved at all
  while(abs(secondPos - firstPos) > 0){ //firstPos (lesser value) is subtract
  //As long as the outcome is greater than 0, continue alligning.
    firstPos = secondPos; //first pos is set to second pos for later use.
    motor[left] = power; //Base continues to move.
    motor[left2] = power;
    motor[right] = power;
    motor[right2] = power;
    wait1Msec(100); //Allows base to continue moving.
    secondPos = SensorValue[baseEn];
    //secondPos is reset to current base value. Loop is run again and differe
  }
  //Loop breaks when secondPos - firstPos = 0, meaning the base has stopped m
    motor[left] = 0; //Motors stop
    motor[left2] = 0;
    motor[right] = 0;
    motor[right2] = 0;
}
```

Initial rate collection.

Loop: Reverse until the robot is forced to stop.

This will allow the robot to detect when it collides with the bar. It will be aware of its surroundings. -

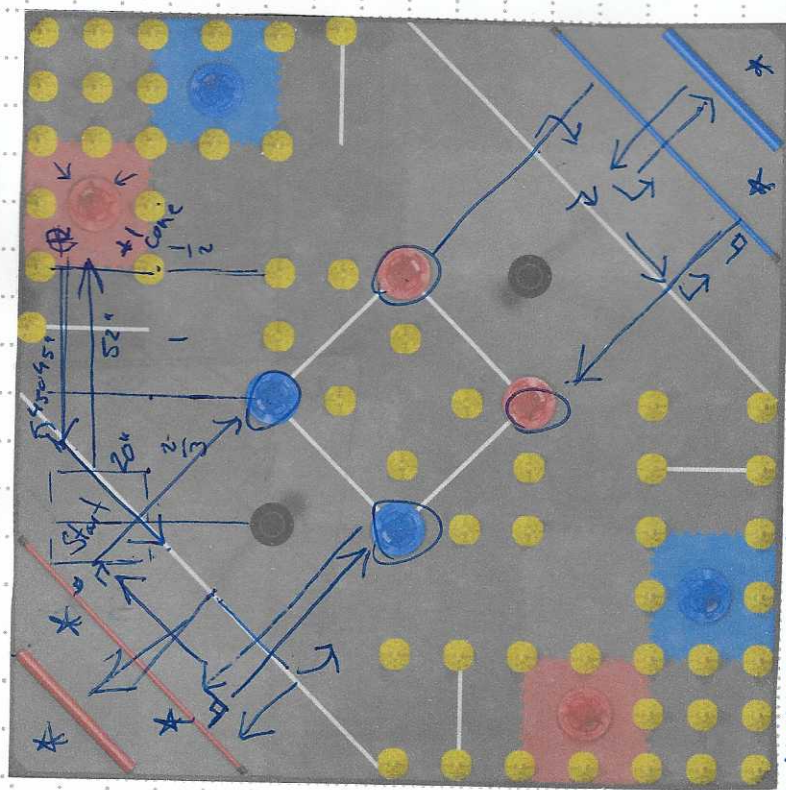
project Bar Alignment designed by: Jacob L. witnessed by:

date: Jan 14, 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Programming Skills (January)

1/14. With our programming goals, we aim to have 70 points in programming skills. This is our plan, using info from ~~that~~ the past 3 pages.



Total: 70 pt.

1.) Mobile goal Down
2.) ↑ 57" 3.) mogo ↑
4.) ↓ 45" 5.) ↰ 45°
6.) ↓ 20" 7.) ↰ 90° 8.) ↑ 28"
9.) mogo down 10.) ↓ 28"
11.) ↰ 90° 12.) ↑ 14" 13.) ↰ 90°
14.) align (-60) 15.) ↑ 40"
16.) mogo up 17.) ↓ 20"
18.) ↻ 180° 19.) ↑ 20"
~~20.)~~ 20.) mogo down 21.) ↓ 10"
22.) ↱ 90° 23.) align (60)
24.) ↑ 84 25.) ↱ 90° 26.) ↑ 14"
27.) ↰ 90 28.) ↑ 28"
29.) mogo down 30.) ↓ 28"
31.) ↱ 90° 32.) ↑ 14"
33.) ↱ 90 34.) align (60)
35.) ↑ 40" 36.) mogo up 37.) ↻ 180° 38.) ↑ 40" 39.) mogo down
40.) ↓ 14 41.) ↰ 90 42.) ↑ 28 43.) ↰ 90 44.) align (60)
45.) ↑ 40 46.) mogo up 47.) 180° 48.) ↑ 40 49.) mogo down
50.) ↓ 20"

With the use of AutoGyroTurn (page 14) and AutoDriveInch and alignWithBar, we should be able to correctly navigate through these 50 steps, scoring 70 points.

project Programming Skills designed by: Jacob V. witnessed by:

date: Jan 14, 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Mobile Goal - Auto Control

1/12. To make autonomous programming easier for us, Theo has created a task that excepts "Up", "Down", "Brake", and "Standby". They each control the mogo intake by taking it in, out, by stopping it, and by letting it offload with the top lift (respectively).

```
task autoMogoP()
{
  int motorPower;
  float Error;
  float Kp = 0.1; //Not Tested

  while(true)
  {
    Error = autoMogoDegrees - SensorValue[intakeEn];
    motorPower = (Error * Kp);
    motor[intake] = motorPower;
    wait1Msec(10);
  }
}
```

P loop controls the mogo intake by multiplying the error by a given value before assigning it as a power to the motors.

```
task autoMogoCommand()
{
  while(true)
  {
    if (strcmp(autoMogoPos, "Up") == 0)
    {
      autoMogoDegrees = -17; //Up Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Down") == 0)
    {
      autoMogoDegrees = -144; //Down Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Brake") == 0)
    {
      autoMogoDegrees = SensorValue[intakeEn]; //Keep at current posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Offload") == 0)
    {
      //autoMogoDegrees = 0; A bit of code to ofload the goal with the top 4b
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Standby") == 0)
    {
      wait1Msec(10);
    }
    else /* default: */
    {
      wait1Msec(10);
    }
  }
}
```

In autonomous, all we need to do is set autoMogoPos to "Up" to Bring and hold the lift at the locked and upright position.

Small wait at the end of the loop to save the CPU's life. -L

project - Mobile Goal Control designed by: Theo L. witnessed by:

date: Jan 17, 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Programmed Lift Stabilization

PID control is used to keep our opposite lift sides in sync. This stabilized the lift because both side are working to gether at the same rate.

```
task driverLiftPid()
{
  float leftFinalPower = 0;
  float rightFinalPower = 0;

  int leftLastError = 0;
  int rightLastError = 0;

  int leftError = 0;
  int rightError = 0;

  int touched = 0;
  float kp = 4;
  float kd = 1;

  SensorValue[liftLeftEn] = 0;
  SensorValue[liftRightEn] = 0;

  while(true)
  {
    //If Button Was Pressed recently and neither ones are currently pressed,
    if(touched > 0 && vexRT[Btn6U] + vexRT[Btn6D] < 1)
    {
      leftDriverLiftDesired = SensorValue[liftLeftEn];
      rightDriverLiftDesired = SensorValue[liftRightEn];

      touched = 0;
    }

    leftError = leftDriverLiftDesired - SensorValue[liftLeftEn];
    rightError = rightDriverLiftDesired - SensorValue[liftRightEn];

    leftFinalPower = (kp * leftError) + (kd * (leftError - leftLastError));
    rightFinalPower = (kp * rightError) + (kd * (rightError - rightLastError));

    if(vexRT[Btn6U])
    {
      motor[leftLift] = 127;
      motor[rightLift] = 127;
      touched = 1;
    }
    else if(vexRT[Btn6D])
    {
      motor[leftLift] = -127;
      motor[rightLift] = -127;
      touched = 1;
    }
    else
    {
      motor[leftLift] = leftFinalPower;
      motor[rightLift] = rightFinalPower;
    }
    leftLastError = leftError;//used for D
    rightLastError = leftError;//used for D
```

project Lift Stabilization designed by: Thea L. witnessed by:

date: Jun 16 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Calibration Button

1/16- Theo had a great idea to add a button to our robot that, when pressed, calibrates all our sensors

→ This is mostly because the gyro sensor takes 2-3 seconds to zero. ~~At~~ After powering on the robot, the gyro used to start calibration instantly, which meant that we had to keep the robot completely still

* Now, when we press the calibration button, the gyro calibrates, as well as our other Lifts. This means that we can move the robot while it starts up and then calibrate the sensors again after we set up.

Our lifts calibrate by moving, and checking the rate of our sensors to measure when they reach resistance. After resistance is met, the lifts set to 0 because they know they are at their farthest back point.

* We also placed the button under the base so it can't be accidently ~~pta~~ pressed by a cone during a match.

Its also placed ~~sf~~ so that the 10 points bar cannot touch it.

This method is super easy for ~~use~~ us to use, and I think it is fairly unique. We will get a lot of use ~~of~~ out of this.

project Calibration Butt designed by: Theo L. witnessed by:

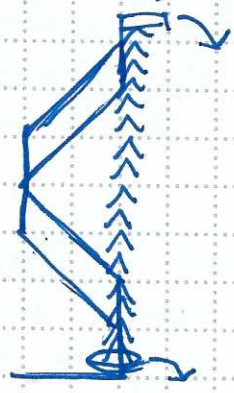
date: Jan 16 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Secondary 4 Bar Sync.

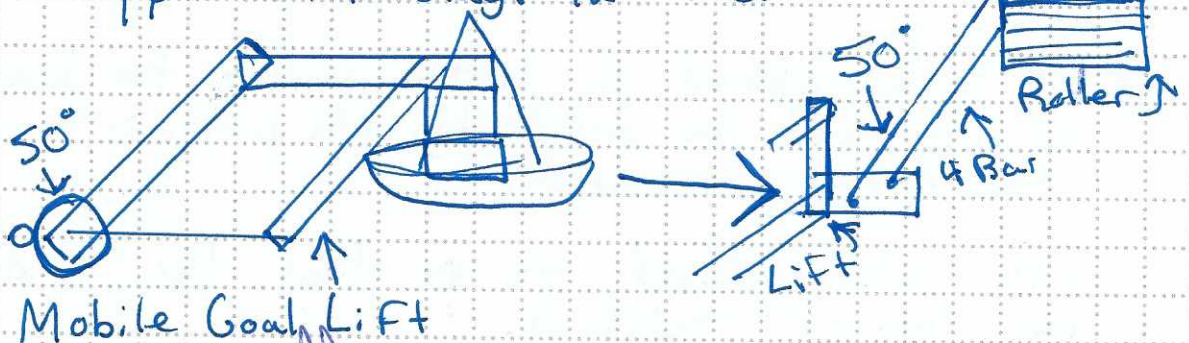
1/17 - When we need to score large, high stacks, we will want to support the stack from the top. This will stop it from tipping over. We run into a problem with scoring though, if the stack is supported by a fixed object because it won't be able to be offloaded.

We have to use our upper 4 Bar to move with the cones to support them.



As the mobile goal offloads a large stack, we can use the top lift to assist the mobile goal by holding onto the cones.

We can do this by matching the mobile goal intake value with the upper lift's sensor values so the upper lift stays in line.



Although, we may use this rarely, it will be great for when we need to offload large stacks.

project Secondary Lift Sync designed by: Jacob W. witnessed by:

date: Jan 19 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Upper Lift Control

1/19- We have ~~ab~~ ~~abt~~ added a potentiometer to the top lift in order to keep track of its position. This allows Theo to program an incredibly useful control loop that moves the upper lift between two set states: scored pos. and collection pos.

P Control Loop for assigning motor power

```
task driverFourBarP()
{
  int motorPower;
  float Error;
  float Kp = 0.1; //Not Tested

  while(true)
  {
    Error = SensorValue[fourBarPot] - autoFourBarDegrees;
    motorPower = (Error * Kp);
    motor[fourBar] = motorPower;
    wait1Msec(10);
  }
}
```

autoFourBarDegrees is set here

```
task driverFourBarControl()
{
  autoFourBarDegrees = 1000;
  while(true)
  {
    if(vexRT[Btn7R])
    {
      autoFourBarDegrees = 3863; //Out Posistion
    }
    else if(vexRT[Btn7L])
    {
      autoFourBarDegrees = 1700; //Scoring Posistion
    }
    else /* default: */
    {
      wait1Msec(10);
    }
  }
}
```

~~4 bar Positi~~ 4bar Position assignment code.

measured positions

Wait to save the CPU's life.

Building Information: Page 4.3 of Building Booklet

Driving is much easier with this feature.

This is another example of code organization.

project Upper Lift Control designed by: Theo Le witnessed by:

date: Jan 19 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto Stacking Task

-1/19- To make driving easier for the drivers, Theo created ~~an automatic sta~~ a semi-automatic driving feature. It's ~~semi-~~automatic because, as the driver, it is important to feel like you are in control in case a problem arises.

```
task driverAutoStack()
{
  int topLiftPos;
  int firstPos, secondPos; //Used for rate of change

  while(true)
  {

    while(!vexRT[Btn8U]) // Waits for driver to press 8U
    {
      wait1Msec(10);
    }
                          PID disabled to allow the lift auto control.
    stopTask(driverLiftPID); //Stops PID running in the background

    motor[leftLift] = 127;
    motor[rightLift] = 127;
    waitUntil(!vexRT[Btn8U]); //Lifts lift until 8U is let go

    topLiftPos = (SensorValue[liftLeftEn]); // Measures top posisiton to return

    autoFourBarDegrees = 1600; //Scoring Posistion
    wait1Msec(250);

    firstPos = SensorValue[liftLeftEn]; // Lowers the lift until rate of change
    motor[leftLift] = -127;
    motor[rightLift] = -127;
    wait1Msec(500);
    secondPos = SensorValue[liftLeftEn];
    while(abs(secondPos - firstPos) > 10) // Not tested
    {
      firstPos = secondPos;
      motor[leftLift] = -127;
      motor[rightLift] = -127;
      wait1Msec(200);
      secondPos = SensorValue[liftLeftEn];
    }

    motor[leftLift] = 0;
    motor[rightLift] = 0;

    driverRollerPower = -127; //Unloads cone

    leftDriverLiftDesired = topLiftPos; //Sets PID wanted position to the top lc
    //rightDriverLiftDesired = topLiftPos;
    startTask(driverLiftPID); //Resumes lift PID
    leftDriverLiftDesired = topLiftPos; //Just in case something overrides the F
    //rightDriverLiftDesired = topLiftPos;
    wait1Msec(100);
    leftDriverLiftDesired = topLiftPos; //Just in case something overrides the F
    wait1Msec(250);
    waitUntil(SensorValue[liftLeftEn] >= leftDriverLiftDesired - 10); // Waits u


    autoFourBarDegrees = 3863; //Out Posistion
    wait1Msec(250);
    driverRollerPower = 40; //Spins roller in preperation for new cone
  }
}
```

While the 8U button is pressed, the lift goes up util 8U is released. Then it brings back the top 4 bar and lowers the lift until the main lift gets a rate of 0.

After, it resets to where the driver let go.

project Auto Stacking designed by: Theo L. witnessed by:

date: Jan 19 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Program Defines - Driver Values

2/22 - With all the ~~var~~ constant variables we were using, Theo placed all of them in their seperate file. This is a lot cleaner now, and easier to make changes.

```
/*
 _____                 _____ _____ _____ _____ _____ _____
|  _  |               |  _  |  _  |  ___|_   _|_   _/  ___|
| | | |_ __ ___ _ __ | | | | | | | |__   | |   | | \ `--.
| | | | '__/ _ \ '__|| | | | | | |  __|  | |   | |  `--. \
\ \_/ / | |  __/ |   \ \_/ / \_/ / |___  | |  _| |_/\__/ /
 \___/|_|  \___|_|    \___/ \___/\____/  \_/  \___/\____/

*/

  // Potentiomiter Posisitons
  #define FOURBARBACK 1140 //The potentiomiter position for the 4bar to be back
  #define FOURBAROUT 3230 //The potentiomiter position for the 4bar to be out
  #define FOURBARTHRESHOLD 300 //Amount of tolerance the 4BAR semi-p loop has be

  // Driver Power Definitions
  #define CLAWPOWER 90 //The power to use for manual claw control
  #define DRIVERCLAWHOLDPOWER 30 //The power to use for manual claw control to h
  #define MOGOHOLDPOWER 20 //The power to use to hold mogo down after up has bee

  // Joystick threshold for the base
  #define DRIVERBASETHRESHOLD 15

  // Refresh wait of most loops
  #define DEFAULTREFRESHRATE 20
  #define FASTREFRESHRATE 10

  #define MAXMOTORPOWER 100
```

FourBarBack - Potentiometer Value of 4 Back when it's Back.
FOURBAROUT - Potentiometer Value of 4 Bar when it's Out
FOURBARTHRESHOLD - Tolerance of P-loop on 4 Bar
CLAWPOWER - SET MAX power for Claw
DRIVERCLAWHOLDPOWER - Manual claw hold power
MOGOHOLDPOWER - Manual mogo hold power
DRIVEBASETHRESHOLD - Deadzone value for Base (See Page 5)
DEFAULTREFRESHRATE - Normal refresh rate for motors
FASTREFRESHRATE - Faster refresh rate for p loops.
MAXMOTORPOWER - Max motor power Base is allowed.
(See Page 57 of Build Book)

project Driver Defines designed by: Theo L. witnessed by:

date: Feb 25 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Program Defines - Auto Stacking

2/25 - Continuing with Defines... These are the ones we use for auto stacking:

```
/*
 _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____ _____
|  _  |  |  |_   _|     |  _  |  |  |  _  |  |  |_   _|  _  |  |  |  _  |
|     |  |  | | | |  |  |     |  |  |     |  |  | | | |     |  |  |     |
|__|__|_____| |_| |_____|__|__|_____|__|__|_____| |_| |__|__|_____|__|__|
*/

//Auto Stacking Posisitions
#define MOGOHIGHT 70 // Used to define the minimum stack hight
#define FLOORCONEHIGHT 75 // The lift posistion to grab the cone off the floor
#define DRIVERLOADPOS 500 // The lift position for the driverload station

#define DROPHIGHT 170 // Used to define the hight the lift should rise above t
#define CONEHIGHT 40 // Used to define the hight increment the cone gives

#define FOURBARBACKIME 220// Amount to time to bring the 4Bar back before proc
#define FOURBAROUTIME 250// Amount to time to bring the 4Bar out before procee

#define LIFTRATETHRESHOLD 40 // Minimum amount the lift must move every POLLRA
#define LIFTRATEPOLLRATE 100 // Interval to poll the rate of change
```

MOGOHEIGHT - Minimum stack height (First cone on mogo)
FLOORCONEHEIGHT - Lift Pos to get floor cone
DRIVERLOADPOS - Lift Pos to get Driver Load Cones
DROPHEIGHT - Lift up this much more before scoring done
CONEHEIGHT - Cone height increment per cone
FOURBARBACKTIME - Ms Break for 4 Bar to move
FOURBAROUTTIME - " " " " "
LIFTRATETHRESHOLD - Minimum the lift must move to not stop
LIFTRATEPOLLRATE - Rate of which the lift is checked.

Because the auto stacking is still a fairly new feature, we need to change these values often and on the fly. The allow us to do this with ease.

project Auto Stack Defines designed by: Theo L. witnessed by:

date: Feb 25 2015

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Program Defines - Auto Functions

2/25 - More Defines for our autonomous functions:

```
Auto Functions
*/

//Wheel Circumference
#define WHEELCIRCUMFERENCE 12.57

// AutoGyroTurn
#define GYROTURN_FASTKP 0.2// The kp for a fast gyro turn
#define GYROTURN_PTHRESHOLD 50// How close to the goal to do P before braking
#define GYROTURN_BRAKEPOWER 20 // The power to brake in AutoGyroTurn
#define GYROTURN_BRAKETIME 100// The time to brake for in AutoGyroTurn

#define GYROTURN_MAXPOWER 80
#define GYROTURN_MINPOWER 20

// AutoDriveForwards
#define DRIVEFORWARDS_FASTKP 3// The kp for fast movement
#define DRIVEFORWARDS_PTHRESHOLD 50// How close to the goal to do P before bra
#define DRIVEFORWARDS_BRAKEPOWER 20 // The power to brake in AutoDriveForwards
#define DRIVEFORWARDS_BRAKETIME 100// The time to brake for in AutoDriveForwar
#define DRIVEFORWARDS_SLOWCORRECT 35// The minimum power for boolean fine-tuni
#define DRIVEFORWARDS_SLOWTHRESHOLD 20// How close to the goal to do boolean f
```

4" Diameter Wheels

Most of these ones are pretty self explanitory and are used later in the book. Look ahead for examples.

These defines allow us to make fast tweaks to code referanced in many different places in the code. We save ~~some~~ so much time using these defines for our autonomous programs.

project Auto Defines designed by: Theo L. witnessed by:

date: Feb 25 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Motor Functions

2/26- This portion of the program is used to assign motor power to the base and ~~mobile~~ lift. It used defines to regulate the power assignment for the motors.

```
void baseLeftMotors(int value)
{
  if(value > MAXMOTORPOWER) //If the value of power attempted to be assigned
                            //base exceeds MAXMOTORPOWER, it is set to MAXMOT
  {
    motor[left] = MAXMOTORPOWER;
    motor[left2] = MAXMOTORPOWER;
  }
  else if(value < -MAXMOTORPOWER)
  {
    motor[left] = -MAXMOTORPOWER;
    motor[left2] = -MAXMOTORPOWER;
  }
  else
  {
    motor[left] = value;
    motor[left2] = value;
  }
}

void baseRightMotors(int value)
{
  if(value > MAXMOTORPOWER)
  {
    motor[right] = MAXMOTORPOWER;
    motor[right2] = MAXMOTORPOWER;
  }
  else if(value < -MAXMOTORPOWER)
  {
   motor[right] = -MAXMOTORPOWER;
   motor[right2] = -MAXMOTORPOWER;
 }
 else
 {
  motor[right] = value;
  motor[right2] = value;
}
}

void liftPower(int value)
{
  motor[leftLift] = value; // + vexRT[Ch4Xmtr2] / 2;
  motor[rightLift] = value; // - vexRT[Ch4Xmtr2] / 2;
}
```

project Motor Functions designed by: Theo L witnessed by:

date: Feb 26 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto Gyro Turn 2.0

2/26- After more struggle with our turns, we have them very consistantly turning below or above where we want. We've adjusted the code some more and added motor power limits so they never exceed 80 power or go below 20 power.

```
void AutoGyroTurn(int wantedDegrees)
{
  int previousGyro = gyroValue();

  if(SensorValue[sideSelector] < 2047) //Inverts turning angles if left/right se
  {
    wantedDegrees = -wantedDegrees;
  }

  int gyroError = gyroValue() - wantedDegrees - previousGyro; //Figure out error
  int firstError = gyroError; //Used to know the direction of reverse power

  int brakePower = GYROTURN_BRAKEPOWER; //Power to brake with
  int brakeTime = GYROTURN_BRAKETIME; //Time to break for

  int speed;

  float fastKP = GYROTURN_FASTKP; //Kp for moving fast towards goal
  //int slowSpeed = 40; //Speed for correction

  while (abs(gyroError) > 5){

    gyroError = gyroValue() - wantedDegrees - previousGyro;

    speed = (gyroError * fastKP);

    if (abs(speed) > GYROTURN_MAXPOWER){

      if (wantedDegrees > 0){
      speed = GYROTURN_MAXPOWER;

       } else {
      speed = -GYROTURN_MAXPOWER;
       }

    } else if (abs(speed) < GYROTURN_MINPOWER) {
      if (wantedDegrees > 0){
      speed = GYROTURN_MINPOWER;

       } else {
      speed = -GYROTURN_MINPOWER;
       }
    }

    motor[left] = speed;
    motor[left2] = speed;
    motor[right] = -speed;
    motor[right2] = -speed;
  }


    if(firstError > 0) //Brake in opposite direction
  {
    motor[left] = -brakePower;
    motor[left2] = -brakePower;
    motor[right] = brakePower;
    motor[right2] = brakePower;
  }
  else
  {
    motor[left] = brakePower;
    motor[left2] = brakePower;
    motor[right] = -brakePower;
    motor[right2] = -brakePower;
  }

  wait1Msec(brakeTime); //Brake for Time

  motor[left] = 0;
  motor[left2] = 0;
  motor[right] = 0;
  motor[right2] = 0;

}
```

Stage 1: Stage 1 is a modified P loop. The motor is given power based on how far from the goal it is. If above 80, though, it's set to 80. Same for below 20.

↖ use of defines ↙

Stage 2: The second stage reverses the motors briefly to kill momentum.

project Auto Gyro Turn 2.0 designed by: Jacob W. witnessed by:

date: Feb 26 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Triple Gyro Control Average

2/26 - To make our turns more accurate, we've used 3 gyro sensors to get an average. (See Page 53 of Build Book). Half the battle was finding a way to mount them all, but to other half is controlling them with the program. We have to get the average of the sensors that are the most similar to eachother

```
/*
 ____                    _                                                  
|  _ \ _   _ _ __ ___   / \__   _____ _ __ __ _  __ _  ___                  
| |_) | | | | '__/ _ \ / _ \ \ / / _ \ '__/ _` |/ _` |/ _ \                 
|  _ <| |_| | | | (_) / ___ \ V /  __/ | | (_| | (_| |  __/                 
|_| \_\\__, |_|  \___/_/   \_\_/ \___|_|  \__,_|\__, |\___|                 
       |___/                                     |___/                      
*/
float gyroValue()
{

  int gyroAverage;

  //Checks if Gyro1 is different from Gyro2 by 10 degrees.
  if (abs(SensorValue[in8]) > abs(SensorValue[in7] + 50)
    ||abs(SensorValue[in8]) < abs(SensorValue[in7]) - 50)
  {
    //If it is, Gyro1 is compared with Gyro3 to see if it is also different.
    if (abs(SensorValue[in8]) < abs(SensorValue[in6] + 50)
      || abs(SensorValue[in8]) > abs(SensorValue[in6] - 50))
    {
      gyroAverage = (SensorValue[in8] + SensorValue[in6])/2; //If Gyro1
      //is similar to Gyro3, Gyro2's reading is discarded.
    }
    //If Gyro1 and Gyro3 are different too, then Gyro1 is discarded.
    else
    {
      gyroAverage = (SensorValue[in7] + SensorValue[in6])/2;
    }
  }
  //Checks if Gyro1 is different from Gyro3 by 10 degrees.
  else if (abs(SensorValue[in8]) > abs(SensorValue[in6] + 50)
    || abs(SensorValue[in8]) < abs(SensorValue[in6]) - 50)
  {
    gyroAverage = (SensorValue[in8] + SensorValue[in7])/2; //If so,
    //then Gyro3 is discarded because Gyro1 and Gyro2 have already
    //been confirmed to be alike if the program gets this far.
  }
  //If none of the above is true and all three sensors are similar
  //to eachother, an average is taken from all three.
  else
  {
    gyroAverage = (SensorValue[in8] + SensorValue[in7] + SensorValue[in6])/3;
  }
  return gyroAverage; //the float average is returned when gyroValue() is used.
```

project Triple Gyro Control designed by: Jacob W. witnessed by:

date: Feb 26 2015

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Mechanism Calibration 2.0

2/26 - We've change how the robot is ~~calid~~ calibrated. We now use the LCD buttons to start calibration because it was getting bumped accidentally during our matches. We have also improved the format.

```
/*
 Calibrate Lift
*/
void calibrateLift()
{


  int firstPos, secondPos;
  firstPos = SensorValue[liftLeftEn];
  liftPower(-MAXMOTORPOWER);
  wait1Msec(1000);
  secondPos = SensorValue[liftLeftEn];
  while(abs(secondPos - firstPos) > LIFTRATETHRESHOLD - 5)
  {
    firstPos = secondPos;
    liftPower(-100);
    wait1Msec(LIFTRATEPOLLRATE);
    secondPos = SensorValue[liftLeftEn];
  }
  SensorValue[liftLeftEn] = 0;
  SensorValue[liftRightEn] = 0;
  liftPower(MAXMOTORPOWER);
  wait1Msec(700);
  liftPower(0);


}

/*
 Calibrate Mogo

*/

void calibrateMogo()
{

  mogoHaltForCalibrate = true;
  wait1Msec(DEFAULTREFRESHRATE);

  int firstPos, secondPos;
  firstPos = SensorValue[intakeEn];
  motor[intake] = MAXMOTORPOWER;
  wait1Msec(500);
  secondPos = SensorValue[intakeEn];
  while(abs(secondPos - firstPos) > 0)
  {
    firstPos = secondPos;
    motor[intake] = MAXMOTORPOWER;
    wait1Msec(100);
    secondPos = SensorValue[intakeEn];
  }
  SensorValue[intakeEn] = 0;
  motor[intake] = 0;
  mogoHaltForCalibrate = false;
}
```

```
    waitUntil(nLCDButtons == 0); //Waits for button release.

    overrideLCD = true; // Pauses LCD task

    displayLCDPos(0,0);
    displayNextLCDString("  Calibrating   ");
    displayLCDPos(1,0);
    displayNextLCDString("      Mogo      ");
    calibrateMogo();

    displayLCDPos(1,0);
    displayNextLCDString("      Lift      ");
    calibrateLift();

    displayLCDPos(1,0);
    displayNextLCDString("      Gyro      ");
    wait1Msec(2000);
    calibrateGyro();

    displayLCDPos(0,0);
    displayNextLCDString("  Calibrating   ");
    displayLCDPos(1,0);
    displayNextLCDString("      Done      ");
    wait1Msec(1000);


    overrideLCD = false;
  }

}
```

```
 Calibrate Gyro

*/
void calibrateGyro()
{
  SensorType[in8] = sensorNone;
  SensorType[in7] = sensorNone;
  SensorType[in6] = sensorNone;
  wait1Msec(1000);
  //Reconfigure Analog Port 8 as a Gyro sensor and allow time for ROBOTC to cali
  SensorType[in8] = sensorGyro;
  SensorType[in7] = sensorGyro;
  SensorType[in6] = sensorGyro;
  wait1Msec(2500);
  SensorScale[in8] = 136; //Scaled sensors - Gyro 1
  SensorScale[in7] = 136; //Gyro 2
  SensorScale[in6] = 140; //Gyro 3
  wait1Msec(200);
  SensorFullCount[in8] = 36000;  // Set the rollback to be 10 rotations
  SensorFullCount[in7] = 36000;  // Set the rollback to be 10 rotations
  SensorFullCount[in6] = 36000;  // Set the rollback to be 10 rotations
  SensorValue[in8] = 0;
  SensorValue[in7] = 0;
  SensorValue[in6] = 0;
  wait1Msec(500);
}
```

Page 4 of 4

project: Calibration 2.0 designed by: Theo L. witnessed by:

date: Feb. 26. 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Driver Base Task

2/26- For better organization, we've put the base driving functions in its own task. (See page 5 for Deadzone info) This lets us be better at controlling the program because we can specify when the base is controllable.

```
//Base Speed Task ////////////////////////////////////////////////////////////
task driverBaseControl()
{
  while(true)
  {

    if(abs(vexRT[Ch2]) > DRIVERBASETHRESHOLD || abs(vexRT[Ch4]) > DRIVERBASETHRE
    {
      baseRightMotors(vexRT[Ch2] - vexRT[Ch4]);
      baseLeftMotors(vexRT[Ch2] + vexRT[Ch4]);
    }
    else
    {
      baseLeftMotors(0);
      baseRightMotors(0);
    }
    wait1Msec(DEFAULTREFRESHRATE);
  }

}
//Base Speed Task ////////////////////////////////////////////////////////////
```

We have the ability to easily and quickly change DEFAULTREFRESHRATE and DRIVEBASETHRESHOLD. These values are explained on page 29. To understand these values, you can look there

This is a super simple program but it becomes more ~~abo~~ advanced with the defines because its limited to '100' currently. The 100 is defined as driverBaseThreshold.

project Driver Base Task designed by: Theo L. witnessed by:

date: Feb. 26 2016

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

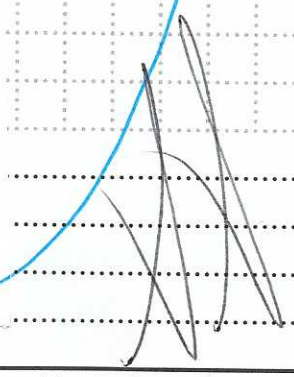
Driver Simple Controls

2/26- Weive also organized our program by making our simple driving controls in their own file to be referenced when needed. These are just basic controls for the claw, 4 Bar, and lift, and mogo intake.

```
int IntakeHold;
task driverSimpleControls()
{
  while(true)
  {
    waitUntil(!mogoHaltForCalibrate); // Checks

    // Claw ------------------------------------
      if(vexRT[Btn5D] || vexRT[Btn5DXmtr2]) // C
      {
        motor[claw] = -CLAWPOWER;
        driverClawPower = -DRIVERCLAWHOLDPOWER;
      }
      else if(vexRT[Btn5U] || vexRT[Btn5UXmtr2])
      {
        motor[claw] = CLAWPOWER;
        driverClawPower = DRIVERCLAWHOLDPOWER;
      }
      else
      {
        motor[claw] = driverClawPower;
      }

    // 4Bar ------------------------------------
      if(vexRT[Btn7R] || vexRT[Btn7RXmtr2])
      {
        driverFourBarDegrees = FOURBAROUT; //Out Posistion
      }
      else if(vexRT[Btn7L] || vexRT[Btn7LXmtr2])
      {
        driverFourBarDegrees = FOURBARBACK; //Scoring Posistion
      }

    // Mogo Intake ----------------------------------------------------
      if(vexRT[Btn6D])
      {
        motor[intake] = -127;
        IntakeHold = -20;
      }
      else if(vexRT[Btn6U])
      {
        motor[intake] = 127;
        IntakeHold = MOGOHOLDPOWER;
      }
      else
      {
        motor[intake] = IntakeHold;
      }


    wait1Msec(FASTREFRESHRATE);
  }
}
```

```
task driverFourBarP()
{
  int motorPower;
  float Error;
  driverFourBarDegrees = SensorValue[fourBarPot];

  while(true)
  {
    Error = SensorValue[fourBarPot] - driverFourBarDegrees;
    if(abs(Error) > FOURBARTHRESHOLD) // If error is outside
    {
      if (Error > 0)
      {
        motorPower = 127;
      }
      else
      {
        motorPower = -127;
      }
    }
    else
    {
    motorPower = 0;
    }
    motor[fourBar] = motorPower;
    wait1Msec(FASTREFRESHRATE);
  }
}
```

Separate Task For 4 Bar



project Driver Simple Controls designed by: Theo L. witnessed by:

date: Feb 26 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto Mogo Control 2.0

2/26 - We've revamped the auto mogo control system and added more settings. See page 23 for old version. We now have a P loop threshold to limit when the mogo lift gets held up.

```
task autoMogoP()
{
  int motorPower;
  float Error;
  while(true)
  {
    Error = autoMogoDegrees - SensorValue[intakeEn]; // Calculates error
    if(abs(Error) > 10) // If error is outside of range, fix
    {
      if (Error > 0)
      {
        motorPower = MAXMOTORPOWER;
      }
      else
      {
        motorPower = -MAXMOTORPOWER;
      }
    }
    else
    {
      motorPower = 0;
    }
    motor[intake] = motorPower;
    wait1Msec(FASTREFRESHRATE);
  }
}


task autoMogoCommand()
{
 while(true)
 {
  if (strcmp(autoMogoPos, "Up") == 0)
  {
      autoMogoDegrees = -17; //Up Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Down") == 0)
    {
      autoMogoDegrees = -160; //Down Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Twenty") == 0)
    {
      autoMogoDegrees = -80; //Down Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Raise") == 0)
    {
      autoMogoDegrees = -120; //Down Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Ten") == 0)
    {
      autoMogoDegrees = -70; //Down Posistion
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Brake") == 0)
    {
      autoMogoDegrees = SensorValue[intakeEn]; //Keep at curre
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Offload") == 0)
    {
      //autoMogoDegrees = 0; A bit of code to ofload the goal
      strcpy(autoMogoPos, "Standby");
    }
    else if (strcmp(autoMogoPos, "Standby") == 0)
    {
      wait1Msec(FASTREFRESHRATE);
    }
    else /* default: */
    {
      wait1Msec(FASTREFRESHRATE);
    }
  }
}
```

The program recognises words and reacts based on the given word.

It's much easier to program with the use of english.

project Auto mogo 2.0 designed by: Theo L witnessed by:

date: Feb 26 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto 4 Bar Control 1.0

2/26- We used that same control schema for the mogo intake for the 4 Bar. See page 27 for driver control. This section is for control over the 4 Bar in autonomous.

```
task autoFourBarP()
{
  int motorPower;
  float Error;
  float Kp = 0.1; //Not Tested
  autoFourBarDegrees = FOURBARBACK;

  while(true)
  {
    Error = SensorValue[fourBarPot] - autoFourBarDegrees;
    motorPower = (Error * Kp);
    motor[fourBar] = motorPower;
    wait1Msec(FASTREFRESHRATE);
  }
}

task autoFourBarCommand()
{
  while(true)
  {
    if (strcmp(autoFourBarPos, "Out") == 0)
    {
      autoFourBarDegrees = FOURBAROUT; //Out Posistion
      strcpy(autoFourBarPos, "Standby");
    }
    else if (strcmp(autoFourBarPos, "In") == 0)
    {
      autoFourBarDegrees = FOURBARBACK; //In Posistion
      strcpy(autoFourBarPos, "Standby");
    }
    else if (strcmp(autoFourBarPos, "Score") == 0)
    {
      autoFourBarDegrees = FOURBARBACK; //Scoring Posistion
      strcpy(autoFourBarPos, "Standby");
    }
    else if (strcmp(autoFourBarPos, "Brake") == 0)
    {
      autoFourBarDegrees = SensorValue[fourBarPot]; //Keep at current posistion
      strcpy(autoFourBarPos, "Standby");
    }
    else if (strcmp(autoFourBarPos, "Offload") == 0)
    {
      //autoFourBarDegrees = 0; A bit of code to ofload the goal with the top 4b
      strcpy(autoFourBarPos, "Standby");
    }
    else if (strcmp(autoFourBarPos, "Standby") == 0)
    {
      wait1Msec(FASTREFRESHRATE);
    }
    else /* default: */
    {
      wait1Msec(FASTREFRESHRATE);
    }
  }
}
```

The code is very similar to the auto mogo code as it also takes words as inputs and completes actions based on the input.

We use this code to control the 4 bar's position with ease.

Just by giving a word, it will go back to the same spot each time without calibration.

project Auto 4 Bar Control designed by: Theo L. witnessed by:

date: Feb 26 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Auto Drive Faward

2/26 - We have also revamped our auto drive forward program to make it more reliable and intelligent. See page 11 for previous version.

```
int brakePower = DRIVEFORWARDS_BRAKEPOWER; //Power to brake with
  int brakeTime = DRIVEFORWARDS_BRAKETIME; //Time to break for

  float fastKP = DRIVEFORWARDS_FASTKP; //Kp for moving fast towards goal - befo
  int slowSpeed = DRIVEFORWARDS_SLOWCORRECT; //Speed for correction - before 40

  void AutoDriveInch(int wantedInch)
  {
  SensorValue[baseEn] = 0; //Reset base encoder

  int wantedDegrees = (wantedInch / WHEELCIRCUMFERENCE) * 360; // Calculates Enc
  int degreesError = wantedDegrees - SensorValue[baseEn]; //Figure out error and
  int firstError = degreesError; //Used to know the direction of reverse power


  while(abs(degreesError) > DRIVEFORWARDS_PTHRESHOLD) //Fast Towards Goal - Thre
  {
    degreesError = wantedDegrees - SensorValue[baseEn];

    motor[left] = degreesError * fastKP;
    motor[left2] = degreesError * fastKP;
    motor[right] = degreesError * fastKP;
    motor[right2] = degreesError * fastKP;
  }

  if(firstError > 0) //Brake in opposite direction
  {
    motor[left] = -brakePower;
    motor[left2] = -brakePower;
    motor[right] = -brakePower;
    motor[right2] = -brakePower;
  }
  else
  {
    motor[left] = brakePower;
    motor[left2] = brakePower;
    motor[right] = brakePower;
    motor[right2] = brakePower;
  }
  wait1Msec(brakeTime); //Brake for Time

  degreesError = wantedDegrees - SensorValue[baseEn];

  motor[left] = 0;
  motor[left2] = 0;
  motor[right] = 0;
  motor[right2] = 0;
}
```

project Auto Drive Forward designed by: Theo L. witnessed by:

date: Feb 26 2018

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Drive for Time

2/26- Along with the advanced P loop Drive Forward, we have created a drive fuactoem that goes based on time. We can assign power and time. This is just a very basic control system to fall back on when we need slow, brute fource approach. Jr

```
void driveForTime(int power, int time){

    motor[right] = power;
    motor[right2] = power;
    motor[left] = power;
    motor[left2] = power;

    wait1Msec(time);

    motor[right] = 0;
    motor[right2] = 0;
    motor[left] = 0;
    motor[left2] = 0;
}
```

This is a very basic example of how we use "user inputs" to manipulate the code.

While it's super simple, we use it often where we some extra movement forward without the use of sensors.

Sensors are great, but sometimes they don't get a read and the program gets stuck. We use this in places where sensor failure is probable. J

project Drive For Time designed by: Jacob W. witnessed by:

date: Feb 28 2015

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.

Secondary Drive Function

2/26- We have also a secondary drive forward function that we can use for slower movements, but still calculated. We use this system to score our preload cone on the stationary goal slowly, but the same way all the time. JW

```
void drive(int inch){
//Encoders set to 0
  SensorValue[baseEn] = 0;

  int deg = inch / WHEELCIRCUMFERENCE * 360;
//When the average value of the encoders are less than 'deg', run motors.
  while ( abs(SensorValue[baseEn]) < abs(deg)-100)
  {
//If the specified distance is positive, run the motors forward...
    if (inch > 0){
      motor[right] = 40;
      motor[right2] = 40;
      motor[left] = 40;
      motor[left2] = 40;
    } else {
//... otherwise, run them backwards.
      motor[right] = -40;
      motor[right2] = -40;
      motor[left] = -40;
      motor[left2] = -40;
    }
  }
//Once the encoders have reached the desired distance, stop.
  if (inch > 0) {
    motor[right] = -10;
    motor[right2] = -10;
    motor[left] = -10;
    motor[left2] = -10;
  } else {
    motor[right] = 10;
    motor[right2] = 10;
    motor[left] = 10;
    motor[left2] = 10;
  }
  wait1Msec(100);
  motor[right] = 0;
  motor[right2] = 0;
  motor[left] = 0;
  motor[left2] = 0;

//Finally, turn off the motors.
}
```

project Drive Function2 designed by: Jacob W. witnessed by:

date: Feb 26 2016

PROPRIETARY INFORMATION all information is the property of, and solely owned by the Designer.