August 13, 2014

Jerry Fan

Impulse

Kristin School

Impulse Programming and Strategies

The “Time” in Impulse

Contents

[Versioning 2](#_Toc395683849)

[Version 1 – Time V1 3](#_Toc395683850)

[Version 2 – Time V2 4](#_Toc395683851)

[Autonomous 5](#_Toc395683852)

[User Control 6](#_Toc395683853)

[Autonomous Strategy 7](#_Toc395683854)

# Version 1 – Time V1

Compiler: RobotC

Sensors Setup:

#pragma config(Motor, port1, LDB, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port2, RDB, tmotorVex393, openLoop)

#pragma config(Motor, port3, LL1, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port4, LL2, tmotorVex393, openLoop)

#pragma config(Motor, port5, RL1, tmotorVex393, openLoop)

#pragma config(Motor, port6, RL2, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port9, RDF, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port10, LDF, tmotorVex393, openLoop)

The first robot we had was quite a simple one without any real sensors, which made autonomous programming hard and imprecise. The build was finished very close to the scrimmage date, so I did not have chance to work on an autonomous routine, given that the user control code was more important to implement first.

User Control:

task usercontrol()

{

int rightLift;

int leftLift;

int leftDF;

int rightDF;

int leftDB;

int rightDB;

while (true)

{

//Lift

//Get Controller Values

leftLift=(vexRT[Btn5U]-vexRT[Btn5D])\*127;

rightLift=(vexRT[Btn5U]-vexRT[Btn5D])\*127;

//Give Lift Values

motor[LL1] = leftLift+10;

motor[LL2] = leftLift+10;

motor[RL1] = rightLift+10;

motor[RL2] = rightLift+10;

//Drive

//Left Stick Control (Strafe and movement)

leftDF=vexRT[Ch3]+vexRT[Ch4];

leftDB=vexRT[Ch3]-vexRT[Ch4];

rightDF=vexRT[Ch3]-vexRT[Ch4];

rightDB=vexRT[Ch3]+vexRT[Ch4];

//Right Stick Control (Turning)

leftDF=leftDF+vexRT[Ch1];

leftDB=leftDB+vexRT[Ch1];

rightDF=rightDF-vexRT[Ch1];

rightDB=rightDB-vexRT[Ch1];

//Give motors values

motor[LDF]=leftDF;

motor[LDB]=leftDB;

motor[RDF]=rightDF;

motor[RDB]=rightDB;

}

}

This was what I put together quickly to make the first robot competition worthy. Integers are declared early on which act as a ‘buffer’ to hold values sent by the controller before they are assigned to the motors

# Version 2 – Time V2

Compiler: RobotC

Sensors Setup:

#pragma config(I2C\_Usage, I2C1, i2cSensors)

#pragma config(Sensor, in1, LL\_pot, sensorPotentiometer)

#pragma config(Sensor, in2, RL\_pot, sensorPotentiometer)

#pragma config(Sensor, I2C\_1, RDB\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_2, RDF\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_3, LDF\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_4, LDB\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Motor, port1, RDF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_2, 1000)

#pragma config(Motor, port2, RDB, tmotorVex393, openLoop, encoder, encoderPort, I2C\_1, 1000)

#pragma config(Motor, port3, LLU, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port4, LLD, tmotorVex393, openLoop)

#pragma config(Motor, port5, RLU, tmotorVex393, openLoop)

#pragma config(Motor, port6, RLD, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port7, RIN, tmotorVex393, openLoop)

#pragma config(Motor, port8, LIN, tmotorVex393, openLoop)

#pragma config(Motor, port9, LDB, tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C\_4, 1000)

#pragma config(Motor, port10, LDF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_3, 1000)

After the first scrimmage, many more sensors were added onto the cortex. This included two potentiometers and the addition of integrated encoders on the motors. On the programming side, this meant that they needed to be added to the config. I focused on getting the autonomous working properly for this version of the program, as it came to the stage where most robots would be built and would have autonomous for the next scrimmage.

I also added experimental functions to this version of the code. Experimental functions are ideas that I deem to be worth trying out, but have unstable or unverified code. The two I have in this program is the PID control for robot drive and also the logarithmic drive code to increase controller sensitivity at lower values at the cost of sensitivity at higher values. I believe the log drive will help driving, since at high speeds it’s unlikely that a driver would be concentrating on precision, so the loss of sensitivity will not matter. The more precise control could assist in moving small distances, such as when trying to place a skyrise into the slot. The PID however, I could not get to function properly so it was not activated in this program. Both these experimental functions have a switch, which is a constant Boolean variable that can be true or false, either allowing it to run or disabling it respectively.

## Autonomous

The first thing that I worked on for autonomous was code that would let the robot know how far it travels using encoders, and stop itself at a specified value for autonomous.

void drive\_move(int encoder\_distance,int speed)

{

clear\_previous\_error();

clear\_encoders();

bool LDB\_goal\_reached = false;

bool RDB\_goal\_reached = false;

bool LDF\_goal\_reached = false;

bool RDF\_goal\_reached = false;

//Data validation

speed = abs(speed);

while ((LDB\_goal\_reached == false)||(RDB\_goal\_reached == false)||(LDF\_goal\_reached == false)||(RDF\_goal\_reached == false))

{

if (abs(nMotorEncoder[LDB]) < abs(encoder\_distance))

{

motor[LDB] = speedstep\_drive(encoder\_distance,speed,nMotorEncoder[LDB],3);

}

else

{

motor[LDB] = 0;

LDB\_goal\_reached=true;

}

if (abs(nMotorEncoder[RDB]) < abs(encoder\_distance))

{

motor[RDB] = speedstep\_drive(encoder\_distance,speed,nMotorEncoder[RDB],0);

}

else

{

motor[RDB] = 0;

RDB\_goal\_reached=true;

}

if (abs(nMotorEncoder[LDF]) < abs(encoder\_distance))

{

motor[LDF] = speedstep\_drive(encoder\_distance,speed,nMotorEncoder[LDF],2);

}

else

{

motor[LDF] = 0;

LDF\_goal\_reached=true;

}

if (abs(nMotorEncoder[RDF]) < abs(encoder\_distance))

{

motor[RDF] = speedstep\_drive(encoder\_distance,speed,nMotorEncoder[RDF],1);

}

else

{

motor[RDF] = 0;

RDF\_goal\_reached=true;

}

}

}

The main feature in autonomous is a while loop that checks each motor’s encoder individually to see if they have reached the target distance as set when the code is written. If it has, then it only switches off that individual motor, whilst the rest will keep going until they reach their respective targets. I find this is more accurate, especially with turning, which uses a variation of this code with two motors reversed.

The lift code autonomous function works in a simplified manner, as it was decided to only use one potentiometer due to them both having different scaling values. The value of the left and right potentiometers did not match, and the rate of change of those values were different as well. Basically the lift function brought the lift up or down towards a certain potentiometer value. It stopped the lift motors when the target is reached.

The intake was easy to program, as it only involved activating the two intake motors by setting them to a value, and then back to 0 when I wanted them to stop.

## User Control

The user control section of this program was harder to program due to the fact that values on the controller needs to be constantly updated. This means that the main usercontrol task’s while loop cannot be clogged up with anything.

As previously, to assign values to motors, the program checks for the controller values and then stores them in a variable, before the motor is set to equal that variable. Trigger (5U&D, 6U&D) values are multiplied by 127, which corresponds to the maximum motor speed or -127, for the minimum speed. Left triggers control lift, right triggers control intake.

Because the lift kept falling down when a load was applied, a trim was added that constantly kept the lift motors running at 30, to prevent it from sliding back down when not being controlled. However, there was another problem with rubber bands not letting the lift be at its lowest location. To solve this problem, I thought of a trim that would switch to keeping the lift down when the lift was below a certain height. I found this ‘threshold’ using RobotC and the potentiometers on the lift. As a result, the lift did stay at the height it was intended to be at.

Something new I added on to version two of the program was a preset lift height system. When a button is pressed, the lift would automatically move towards a set height, by checking the current potentiometer value, deciding whether to go up or down, and moving there. These would be activated by controller buttons, with potentially 8 presets. However, only 6 were programmed in, with two being reserved for a possible skyrise intake.

The skyrise intake however did pose a major problem. Moving the lift with it on meant that the side that it was attached on tilted due to its weight. The motors applied a certain amount of force, but because of the uneven weight distribution, it would be unable to lift the side with the claw as fast as the side without, creating this tilt. In trying to solve this, the side that the claw was not attached on had its lift motors underpowered, to compensate the slowness of the other side of the lift. In going down, the reverse was true for the lift. The side without the claw was going down faster, and so is underpowered. To detect which side the claw was on, a potentiometer was added to the robot so that we could manually indicate which side the claw was attached, so that the program would know which motors to underpower when lifting. When the potentiometer was disconnected, I found it had a value of 240-250, and that is the range I set for the indication of no skyrise intake.

## Autonomous Strategy

Because the skyrise claw that was designed was hard to use in autonomous, it was decided that autonomous would consist of cube scoring only. Two different autonomous strategies were devised, so four total routines were written in code, two for each side. One was a floor goal scorer, dragging one cube onto the mat. The other attempted to score on a post. It was decided to use the floor goal scoring as the default as it was reliable.

|  |  |
| --- | --- |
| C:\Users\Aeon\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Skyrise 920p.png  **1**  Figure 1 Outer floor autonomous | C:\Users\Aeon\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Skyrise 920p.png  **1**  Figure 2 Inner floor goal autonomous |
| C:\Users\Aeon\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Skyrise 920p.png  **1**  **2**  Figure 3 Outer post scorer | C:\Users\Aeon\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Skyrise 920p.png  **2**  **1**  Figure 4 Inner post scorer |

On the red side, both of these would be reversed as the field is mirrored. Essentially the inner scorer becomes the outer and vice versa.

# Version 3 – Time V3

Compiler: RobotC

Sensors Setup:

#pragma config(I2C\_Usage, I2C1, i2cSensors)

#pragma config(Sensor, in1, LL\_pot, sensorPotentiometer)

#pragma config(Sensor, in2, RL\_pot, sensorPotentiometer)

#pragma config(Sensor, in3, One, sensorPotentiometer)

#pragma config(Sensor, I2C\_1, RDB\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_2, RDF\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_3, LDF\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_4, LDB\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Motor, port1, RDF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_2, 1000)

#pragma config(Motor, port2, RDB, tmotorVex393, openLoop, encoder, encoderPort, I2C\_1, 1000)

#pragma config(Motor, port3, LLU, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port4, LLD, tmotorVex393, openLoop)

#pragma config(Motor, port5, RLU, tmotorVex393, openLoop)

#pragma config(Motor, port6, RLD, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port7, RIN, tmotorVex393, openLoop)

#pragma config(Motor, port8, LIN, tmotorVex393, openLoop)

#pragma config(Motor, port9, LDB, tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C\_4, 1000)

#pragma config(Motor, port10, LDF, stmotorVex393, openLoop, encoder, encoderPort, I2C\_3, 1000)

This program was a cleaned up version of version 2. I got another programmer – Kristian to look through my code as he is already familiar with java, a similar language to C. I am introducing him to robotics so I thought this would be a good exercise. However due to the shortage of time, this program was never implemented as it was untested when the next scrimmage came.

A new structure was implemented by Kristian that used switches instead of if based switches.

void presetAssign()

{

switch (liftPreset)

{

case 2:

if (SensorValue[LL\_pot] < liftVal[1])

{

liftAutonVal[0] = liftVal[1];

liftPresetMonitor();

}

else

{

liftPreset = 0;

}

break;

case 1:

if (SensorValue[LL\_pot] > liftVal[0])

{

liftAutonVal[0] = liftVal[0];

liftPresetMonitor();

}

else

{

liftPreset = 0;

}

break;

……………

And so on.

Date:

Sensors Setup:

#pragma config(I2C\_Usage, I2C1, i2cSensors)

#pragma config(Sensor, in1, PotentiometerLL, sensorPotentiometer)

#pragma config(Sensor, in2, PotentiometerRL, sensorPotentiometer)

#pragma config(Sensor, in3, , sensorGyro)

#pragma config(Sensor, in4, LineL, sensorLineFollower)

#pragma config(Sensor, in5, LineM, sensorLineFollower)

#pragma config(Sensor, in6, LineR, sensorLineFollower)

#pragma config(Sensor, dgtl1, UltrasonicL, sensorSONAR\_mm)

#pragma config(Sensor, dgtl3, UltrasonicR, sensorSONAR\_mm)

#pragma config(Sensor, dgtl12, PistonClaw, sensorDigitalOut)

#pragma config(Sensor, dgtl6, EIN, sensorQuadEncoder)

#pragma config(Sensor, I2C\_1, RDB\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_2, RDF\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_3, LDF\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_4, LDB\_encoder, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Motor, port1, RDF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_2, 1000)

#pragma config(Motor, port2, RDB, tmotorVex393, openLoop, encoder, encoderPort, I2C\_1, 1000)

#pragma config(Motor, port3, LLU, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port4, LLD, tmotorVex393, openLoop)

#pragma config(Motor, port5, RLU, tmotorVex393, openLoop)

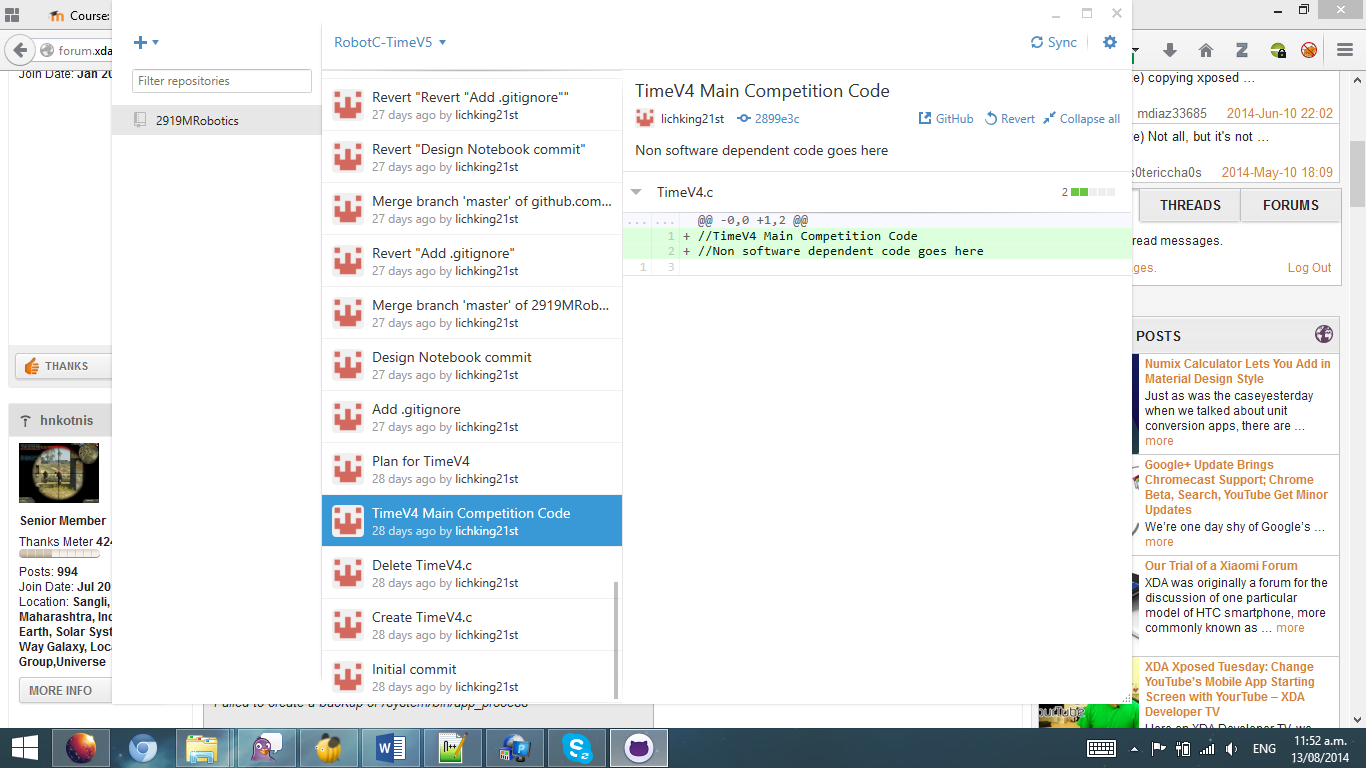
#pragma config(Motor, port6, RLD, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port7, RIN, tmotorVex393, openLoop)

#pragma config(Motor, port8, LIN, tmotorVex393, openLoop)

#pragma config(Motor, port9, LDB, tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C\_4, 1000)

#pragma config(Motor, port10, LDF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_3, 1000)



Version four of the program was a huge step forward in terms of program structure. I started to use header files and includes to clear up clutter on the main file. The main RobotC file is now a list of imports. The advantage I have in doing this is that I can make changes to individual parts of the program such as usercontrol, without having to look through the whole entire program.

I also added this program to a github repository that I set up, this means that other VEX programmers are able to see my work if they search for it, and also so that the other programmer on my team can easily contribute towards parts of the program. Git uses a revision feature, keeping track of changes in code files. This means with each update, I have to submit what is called a ‘commit’, allowing me to record the development of the program, which helps in the design notebook.

The directory structure of TimeV4 goes as follows

* TimeV4.c – Main file
  + Main.h – main header file for code
  + Definitions.h – contains custom macros and definitions
  + Preauton.c – contains preautonomous tasks
  + Usercontrol.c – contains user control tasks
  + Autonomous.c – contains autonomous tasks

The features I implemented was no different to Time V2, however I implemented the features in differing ways. Within this program, I used tasks more than one control loop, especially with autonomous where I have dedicated tasks for setting motor speeds and controlling the lift. I had to manage the different tasks with wait commands so that one task did not take up all the CPU time and essentially jam up the program.

Within the sensor declarations, I added many placeholder sensors for anticipated sensors that would be attached to the robot. These include the three line sensors and the two ultrasonic ones.

This program received less testing than V2 however, and as a result the final results were worse than the previously built program, even though the structure of the code in this program is significantly better.

Date:

Sensors Setup:

#pragma config(I2C\_Usage, I2C1, i2cSensors)

#pragma config(Sensor, in1, potLL, sensorPotentiometer)

#pragma config(Sensor, in2, potRL, sensorPotentiometer)

#pragma config(Sensor, in3, turningGyro, sensorGyro)

#pragma config(Sensor, in4, lineL, sensorLineFollower)

#pragma config(Sensor, in5, lineM, sensorLineFollower)

#pragma config(Sensor, in6, lineR, sensorLineFollower)

#pragma config(Sensor, dgtl1, ultraL, sensorSONAR\_mm)

#pragma config(Sensor, dgtl3, ultraR, sensorSONAR\_mm)

#pragma config(Sensor, dgtl12, piston, sensorDigitalOut)

#pragma config(Sensor, dgtl6, encoderIN, sensorQuadEncoder)

#pragma config(Sensor, I2C\_1, encoderRDB, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_2, encoderRDF, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_3, encoderLDF, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Sensor, I2C\_4, encoderLDB, sensorQuadEncoderOnI2CPort, , AutoAssign)

#pragma config(Motor, port1, driveRF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_2, 1000)

#pragma config(Motor, port2, driveRB, tmotorVex393, openLoop, encoder, encoderPort, I2C\_1, 1000)

#pragma config(Motor, port3, liftLU, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port4, liftLD, tmotorVex393, openLoop)

#pragma config(Motor, port5, liftRU, tmotorVex393, openLoop)

#pragma config(Motor, port6, liftRD, tmotorVex393, openLoop, reversed)

#pragma config(Motor, port7, intakeR, tmotorVex393, openLoop)

#pragma config(Motor, port8, intakeL, tmotorVex393, openLoop)

#pragma config(Motor, port9, driveLB, tmotorVex393, openLoop, reversed, encoder, encoderPort, I2C\_4, 1000)

#pragma config(Motor, port10, driveLF, tmotorVex393, openLoop, encoder, encoderPort, I2C\_3, 1000)

Given that Time V3 and V4 was not actually used in any scrimmages due to instability, I wanted to make a program that was stable, and that was the aim for V5. V5’s current user control disables diagonal movement due to a fault we found in the last scrimmage where motors would fail after moving diagonally for an extended amount of time. If the controller is using both the x and y axis on the left stick, then it takes the largest of the two, and uses that instead, resulting in either strafe or straight movement.

This program was built on V4, after I spent one day looking through and trying to eliminate all the bugs I could see. As such it still retains the task structure of multiple tasks controlling separate parts of the robot, rather than a single task running everything. The user control is mainly working, but presets are not for some reason. If they are not easily fixable, I will revert back to the old V2 style of presets that used the main user control task to activate them instead of the newer structure.

Date: 24/07/14

During the short session on this day, I worked on the autonomous part of the robot, making sure all the required functions were in place in V5 of the program. The focus was mainly on driving, with the robot now able to go back and forth with macros to call the required functions in the code quickly.

Date: 28/07/14

Attempted to put in an independent task that could hook on to other tasks during the autonomous section of matches. This task would have the ability to stop the robot if it detects a jam, so if the motor does not move, it would stop powering it to keep it from being damaged. I’m not sure how well this will work however, given that the motor itself has no sensors, so that I’m relying on things like potentiometers and encoders to give me accurate information.

Date: 29/07/14

Maximum speed of the robot was reduced on request of our driver Marco. The intakes have now been programmed on and mapped on to the controller, so that’s working well so far. I also touched up on the antijam function as it wasn’t interacting with the other tasks, hopefully it should work now. A trim switch has been added so that the robot won’t loosen up when it is in compact form, by making sure the lift moves down constantly to oppose the tension of the rubber bands. However the trim will switch to a proper one, holding the lift up when it goes above a certain point. Skyrise intake which is separate from the main intake has also been programmed and mapped.

Date: 6/08/14

Massive code revamp done on the current program version, a lot of bug fixes with the code was done, but the robot is not with me so no testing can be done yet. Theoretically the new code should run better and smarter than before.

Date: 12/08/14

Disabled movement in diagonal directions. Robot is able to move diagonally due to the mek wheels installed but will hang for some reason. Our driver does not need this function so I have disabled it, the robot now moves only forwards and back and left to right for strafing.

Date: 19/08/14

Day of autonomous tests, trying to score points in autonomous whilst the test field is set up. Skyrise auton has been abandoned as it is too hard with the robot to position it correctly. Scoring with cubes however is easier so that will be done.

Date: 30/09/14

New gyro has arrived so I have started to work out how to implement this in my code. I have got the sensor connected and working so far, I just need to tweak the sensitivity so that one turn on robot registers as one turn on the sensor.

Date: 8/10/14

Gyro has been completely integrated into the robot, and now is accurate. It provides a precise indication of where the robot is pointing relative to the starting position of the robot, which is pretty useful compared to the encoders that are less precise.