Team FG&B Software Documentation

# Brisbane Boys’ College, Queensland Australia

# Introduction

This documentations purpose is to provide a short and precise explanation and discussion of the software for robots designed by Team FG&B in the year of 2017. At the end of 2017 all code will be open sourced and publically available for anyone to download, modify and use. The robot is programmed in Arduino, a variant of C++ and runs on Teensy 3.2 microcontrollers that can be found at [www.pjrc.com](http://www.pjrc.com).

# The basic design

The 2017 software for FG&B takes advantages of libraries in cpp. This help the readability and neatness of code. Software for our robot design can me separated into two main areas, sensors and movement. The premise of our software design is that the two areas operate completely independent of each other until the point where sensors directly affect the movement of the robot. The controlling of the sensors and movement is done by the primary libraries within the software being; *DirectionController* and *RotationController*. These two libraries each control a primary movement type and take advantage of other sensors libraries to determine the movement. This gives us a hierarchy that allows the robot to make decisions based on different inputs from sensors and other feedback. This is an efficient way of calculating movement as all feedback to the robot is considered in the calculation of its movement.

//Insert Diagram Here (Click charts)

# The libraries

# Buzzer (.h)

The buzzer libraries primary job is to control the buzzer used for feedback on the robot’s condition during match or debugging. The buzzer library consists of one primary class and three primary functions; *playTone*, *errorTone* & *readyTone.* The *playTone* method takes in one input of type integer and plays a buzz sound for the time specified (in milliseconds) in the input. It does this by pulling a digital pin on the Teensy and connected to the buzzer to the high state which then plays a tone. After the delay the pin is pulled low stopping the sound. The *errorTone* method is sounded when the robot encounters an error. The method simply sounds the buzzer and will not stop until the program is stopped. The final method *readyTone* plays when the robot and completed its setup which includes checking the SPI communication between the three teensy controllers on each robot. The tone is simply used to notify the user when setup is complete and the robot is ready to be used.

# Compass (.h, .cpp)

The compass library handles the task of controlling the mpu9250 9-axis compass, gyro and magnetometer chip. The compass communicates with the Teensy using the I2c protocol. The library consists of many public and private methods; *calibrate, read, update, getHeading, setTarget*. The *calibrate* method handles the task of calibrating the compass, specifically the gyroscope. It does this by taking multiple reading over a certain delay to find the drift of the gyroscope while the robot is standing still. This allows us to counter the natural drift that the gyro has. Each reading throughout the game will take or add the drift to ensure that all headings are accurate. The *read* method simply takes a reading from the compass over I2c and gets the Z component (from XYZ) of the gyroscope. It then returns the reading from the gyro. The *update* method takes the reading, counters the drift and mods the value from 0 to 360 degrees. This allows us to have an accurate heading in degrees. We store this heading and use it later. The *setTarget* method takes in one variable being of type double which allows the software to set the desired heading of the robot. This allows us to easily make the robot face towards the goal by changing the target heading that the robot is trying to achieve. The *getHeading* method adds 180 to the heading which means that instead of 0,360 being at the centre of the robot, 180 is. This helps because if 0,360 is at the centre front, the robot ticks between the two values and doesn’t correct as well. There are other ways to fix this but by adding 180 to the heading is the easiest and fastest option software wise.

# Config (Config.h, Defines.h, Pins.h)

The different header files under the name config is where the majority of variables that can be changed live along with constants and pin definitions. Each header file consists of mostly *#define* constants that can be changed by the user to dictate how the robot plays. This is simply a much cleaner way of having a setting file.

# Direction Controller (.h, .cpp)

The direction controllers main purpose is to handle the final movement of the robot. The library itself receives sensor information and sorts it into a final direction of movement for the robot. The methods within the library are; *calcMotors, calcLight and move.* The *calcMotors* method had two separate definitions, one that calculates an exact angle to move and one that calculates a restricted angle based on input in the program. The job of *calcMotors* is to use trigonometry to find a final *pwm* value to write to the motor (set between -255 and 255). The equation used is as follows: . The second definition of *calcMotors* simply limits the output of the above equation to factors of 45, giving the robot 8 directions of orbit. The *calcMotors* method also takes in the rotation determined by the *Rotation Controller* library. The *calcLight* method simply takes an input from the Light Sensor library in the form of a direction and sets the angle to move to said angle. If the Light Sensors aren’t seeing anything of value, the method returns a struct with the values of a Boolean and integer in the format of: *{false, 0} or {true, \_\_angle\_\_}*. The *move* method simply checks the returned values from *calcLight* to determine which angle to follow, the one from the light sensors or the ball. It then calls the *calcMotors* method with the final angle as an input and adds rotation from the *Rotation Controller* library.