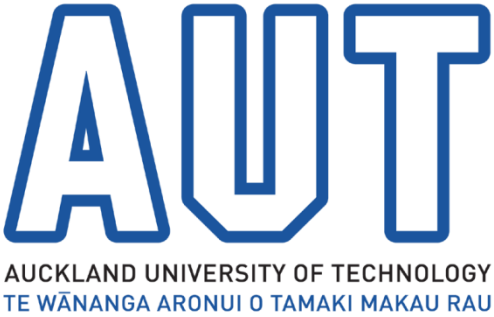


Faculty of Design and Creative Technologies

School of Engineering

Department of Electronic & Electrical Engineering

BACHELOR of ENGINEERING

(HONOURS)

Robot Football

Supervisor: Dr John Collins

Student: Ben Dabin (0946177)

Contents

Introduction

The project is about a robot which can move to a ball and push a ball to a given point on a field. The robot requires the use of a web cam, located directly off the field for providing images for an image processing software program. The image processing software uses an algorithm that calculates the position of the ball and the robot from different colours. The software calculates the position of the ball (X,Y) and robot (X,Y) on the field in millimetres(MM) and it also calculates the heading of the robot in radians. The five readings are then transmitted wirelessly to the robot from the program on the PC using a pair of wireless modules, where one of the modules is on the robot and the other is connected directly to the PC. The robot being used for the project is manufactured by ***Pololu***, and it has an on-board microcontroller module for writing the programming code to control the robot. The robot is also able to perform and carry out its own calculations from the readings being sent from the PC.

1 Theory

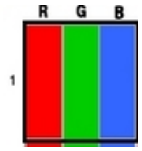
The following section of this report will mention some of the theory and methods that will be used such as image processing on RGB images, and HSV transformations which are used for determining the coordinates of the ball and the robot on the field can be applied in a real life situation like for example how the pixel data was received from the web cam is converted in coordinates in terms of millimetres.

There is also another theory that will be mentioned and used for controlling the speeds of the Left and Right motors on the robot, which enable the robot to move on its own curve to where the ball or target position is located. This algorithm is also designed in a way where the robot's speed will be increased to start off with and then the formulas allow the speed to decrease as it approaches nearer to where the ball is located.

1.1 Image Processing

1.1.1 RGB Image from Camera

The produced image from the Web Cam is in RGB(Red Green and Blue) format. RGB is a type of colour system whereby red, green, and blue light are added together in various ways to reproduce a broad array of colours. The name of the model comes from the initials of the three additive primary colours, red, green, and blue. The image below shows an example of an RGB pixel and as you can see one Pixel consists of the following 3 colours which are for Red, Green and Blue.



An example of a frame that would be processed by the ***software program*** is shown in **Figure 1.** The image shows the type of the colours that would be processed by the program

on the PC. For example the 4 coloured circles, 2 coloured circles on the robot and the orange ball.

Figure 1

Later on an algorithm will be required for calculating and processing an RGB image to determine the pixel coordinates for each of the 7 colours in the image.

1.1.2 Hue Saturation and Value

As the Web Cam produces the readings in RGB the pixels would need to be converted into HSV(Hue Saturation Value) so that a single colour or hue can be determined. The image in, ***Figure 2*** shows the range of Hue that is represented. The Hue

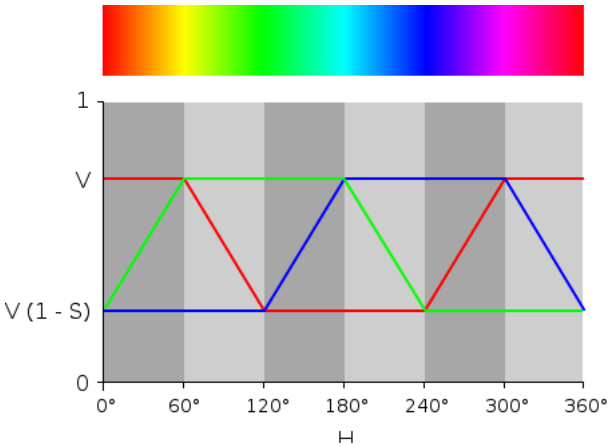


Figure 2

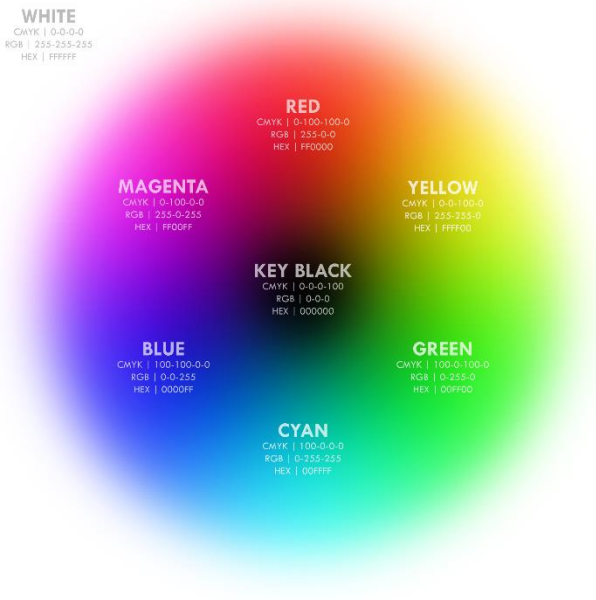


Figure 3

value is expressed as an angle that ranges from 0 to 360°, with colours in the 0 - 60° range being red, oranges and yellows, 60 to 180° is mainly green values, 180 to 300 is blue colours and finally 300 to 360 is purples and reds. Also notice that the red value ranges wraps around itself which can be thought of as a donut or ring shape with subdivided regions for each colour. The Hue of an object is the fixed colour property that never changes whereas the Saturation is the intensity of the colour value which varies depending on the lighting conditions of the environment. The Hue value is can also be thought of as a donut or ring shape as stated in ***Figure 3***. Extracting the RGB components from the image is much simpler for image processing because a particular range of hue can be extracted from the image. For extracting the coloured circles in the football field example in ***Figure 1,*** it makes it possible to extract the desired range of colour by developing an algorithm that converts RGB pixels into HSV and then checking if the angle of where the colour range lies on the HSV colour wheel and therefore making it possible to calculate the X and Y pixel coordinates for each of the 7 colours of interest.

The 7 colours of choice must be chosen in a way that prevents overlapping of Hue values, as indicated by the HSV colour wheel in Figure 3***.*** The colours chosen were Red, Yellow, Green and Blue for the four corners of the field and Cyan and Magenta for the robot and Orange for the ball. After determining the pixel coordinates in terms of pixels, the readings can then be converted into millimetres by applying the theory that was mentioned earlier.

1.1.3 Position Calculations in Millimetres (MM)

The image below shows field below represents the field that the robot and the ball move along. The field is represented as a rectangle as ABCD. The orange ball is represent at a point E which could even be the robot as well. This is the field represented in ideal conditions.

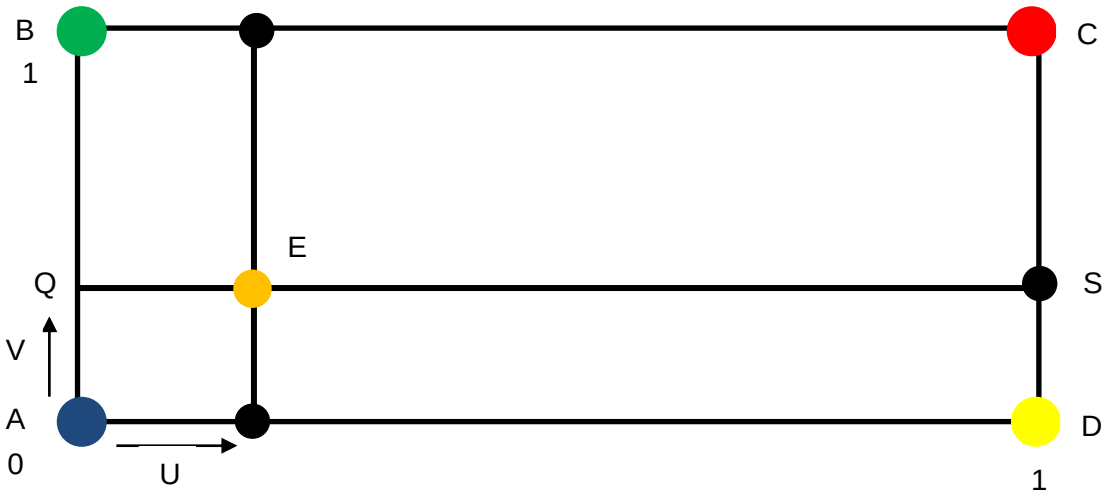


Figure 4

$U = \frac{AP}{AD}$ so $0 \leq U \leq 1$ (U = 0 when P = A, U = 1 when P = D)

$V = \frac{AQ}{AB}$ so $0 \leq V \leq 1$ (V = 0 when Q = A, V = 1 when Q = B)

The camera see the image like following image shown in ***Figure 5*** due to the way the camera was positioned. As you can see that where the red and green circles are in the image that there will be a perspective. There will also be a point where if the distance of the red and green circles were to continue there will be a point where they would cross over and a solution would end up in a random area.

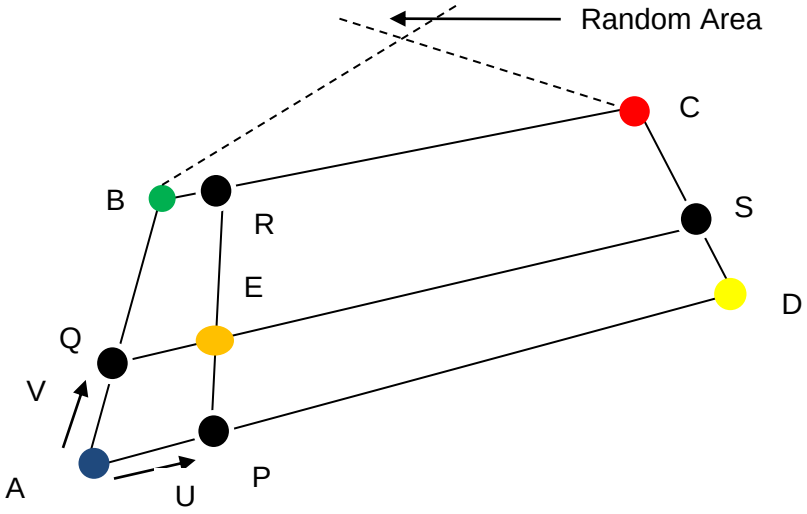


Figure 5

The information that was worked out so far the information that is provided to carry out the calculations are the pixel coordinates of the field, robot, and ball. For now the theory will focus on the ball's positioning on the field which is basically the same principle for the robot.

The field has the pixel positions for A,B,C,D,E to convert these into actual positions in millimetres (mm), to find the position of the ball in the field we need to make the following approximation.

Assume points PQRS are along the same proportion along the sides as shown in ***Figure 5 (the real life situation)***. This will not be exactly correct because of the distortion in the image, especially due to perspective.

Calculation

Calculate U and V for ***Figure 5,*** so that the lines ***PR*** and ***QS*** are calculate to determine the point ***E.***

$$P = A + U(D - A) = (1 - U)A + UD \qquad (1)$$

$$R = B + U(C - B) = (1 - U)B + UC \qquad (2)$$

If E is on the line PR then the following equation is:-

$$E = P + V(R - P) = (1 - V)P + VP \qquad (3)$$

Substitute P and R from 1 and 2 into 3.

$$E = (1-V)\big((1-U)A + UD\big) + V\big((1-U)B + UC\big)$$

$$E = (1-U)(1-V) + (1-U)VB + UVC + U(1-V)D \quad (4)$$

This formula applies separately to the X and Y coordinates or pixels of A,B,C,D,E and these values are all known from the image. Therefore we have 2 equations in U,V that can be solved.

After rearranging (4)

$$E = A - UA - VA + UVA + UB - UVB + UVC + UD - UVD$$
$$E = A + U(-A + D) + V(-A + B) + UV(A - B + C - D)$$

$$E - A = U(D - A) + V(B - A) + UV(A - B + C - D)$$

After calculating the following formulas we can now separate the components for Y and X and then we have the following equations:-

$$Ex - Ax = U(Dx - Ax) + V(Bx - Ax) + UV(Ax - Bx + Cx - Dx)$$
$$Ey - Ay = U(Dy - Ay) + V(By - Ay) + UV(Ay - By + Cy - Dy)$$

Therefore we let:-

$\mathrm{a} = \mathrm{Ex} - \mathrm{Ax}$	$e = Ey - Ay$
$\mathrm{b} = \mathrm{Dx} - \mathrm{Ax}$	$f = Dy - Ay$
$\mathrm{c} = \mathrm{Bx} - \mathrm{Ax}$	$g = By - Ay$
$d = Ax - Bx + Cx - Dx$	$h = Ay - By + Cy - Dy$

Then:-

$$a = Ub + Vc + UVd \quad (5)$$
$$e = Uf + Vg + UVh \quad (6)$$

From (5) $a - Ub = V(c + Ud)$ so $V = \frac{a - Ub}{c + Ud}$ (7)

From (6) $e - Uf = V(g + Uh)$ so $V = \frac{e-Uf}{g+Uh}$ (8)

Therefore $\frac{a-Ub}{c+Ud} = \frac{e-Uf}{g+uh}$

Or $(\mathrm{a} - \mathrm{Ub})(\mathrm{g} + \mathrm{Uh}) = (\mathrm{c} + \mathrm{Ud})(\mathrm{e} - \mathrm{Uf})$

Expanding out the following equation we get the following

$\mathrm{ag} + \mathrm{ahU} - \mathrm{bgU} - \mathrm{bhU}^2 = ce - cfU + deU - dfU^2$

$(\mathrm{ag} - \mathrm{ce}) + (\mathrm{ah} - \mathrm{bg} + \mathrm{cf} - \mathrm{de})\mathrm{U} + (\mathrm{df} - \mathrm{bh})\mathrm{U}^2 = 0$

$m + lU + kU^2 = 0$

Therefore this forms the following equation

$$U = \frac{-l \pm \sqrt{l^2 - 4km}}{2k}$$

This gives quadratic equation will produce 2 solutions which are the following:-

$0 \leq U \leq 1$

Then we can use (7) or (8) to calculate V

Therefore the following equations are generated

$Ex = U(Dx - Ax)\ mm$

$Ey = V(By - Ay)\ mm$

These formulas can later on used by the PC software program for converting X and Y pixel coordinates into actual lengths in terms of millimetres.

1.2 Calculating the arc length of the robot on the field

The theory states that the robot goes round its own circle and uses some curve to head towards where the target point or where the ball is located. The following theory is used for calculating the wheel speeds of the robot. This helps the robot to move towards the ball or target point in straight position.

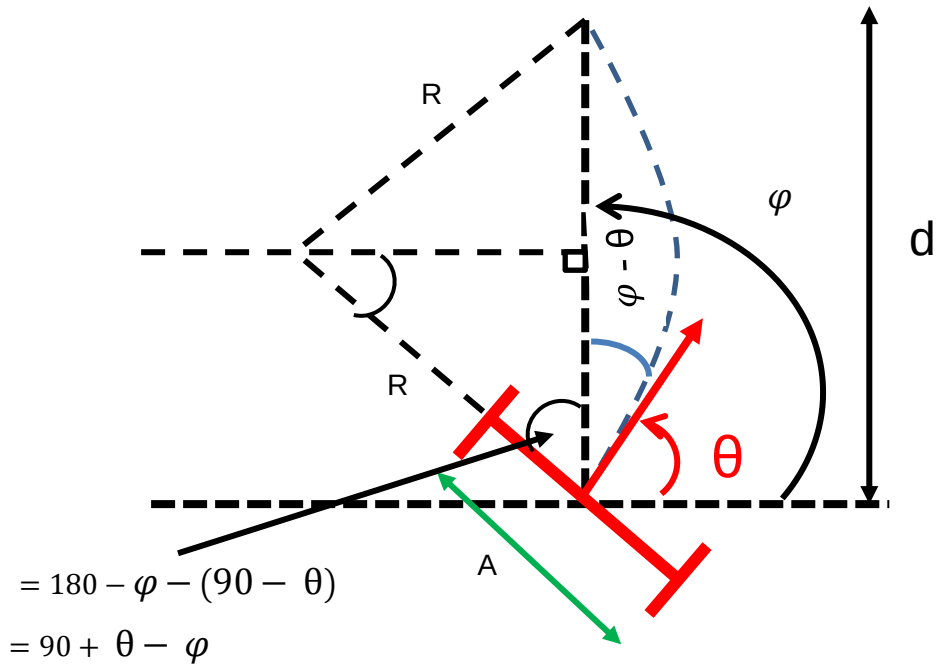


Figure 6

The robot is shown in red with the axle length specified as “A”. The following equations that can be ge

$$\frac{d}{2} = R\cos(90 + \theta - \varphi) = R * \sin(\varphi - \theta)$$

Where the axle length = A.

The formulas for the left wheel gives:-

$R - \frac{A}{2}$ Which gives:- $SL = k\left(R - \frac{A}{2}\right)$ (1)

The formula for the right wheel gives:-

$R + \frac{A}{2}$ Which gives:- $SR = k\left(R + \frac{A}{2}\right)$ (2)

The formula for working out R gives:-

$$R = \frac{\frac{d}{2}}{\sin(\varphi - \theta)} \quad (3)$$

Working out other formulas gives:-

$$\cos(90 + \theta - \varphi) = \frac{\frac{d}{2}}{R}$$

The following formula can be rearranged to give:-

$$R = \frac{\frac{d}{2}}{\cos(90+\theta-\varphi)} \qquad (4)$$ (Pololu 3pi Robot, 2014)

Equations (3) and (4) can be substituted into (1) to solve for the speed value for the left wheel which gives the following.

$$SL = k\left(\frac{\frac{d}{2}}{\sin(\varphi - \theta)} - \frac{A}{2}\right)$$

Which simplifies to the following:-

$$SL = \frac{k}{2\sin(\varphi - \theta)}(d - A\sin(\varphi - \theta))$$

The same situation occurs by substituting equations (3) and (4) into (2) which can solve the speed value for the right wheel.

$$SR = k\left(\frac{\frac{d}{2}}{\sin(\varphi - \theta)} + \frac{A}{2}\right)$$

$$SR = \frac{k}{2\sin(\varphi - \theta)}(d + A\sin(\varphi - \theta)$$

Therefore the following values which calculate the speed of the robot when moving straight are as follows:-

Where k is just some constant. These formulas will later on be used in the robot program.

$$SL = k(d - A\sin(\varphi - \theta)) \qquad (5)$$

$$SR = k(d + A\sin(\varphi - \theta)) \qquad (6)$$

2 Hardware

This section of the report will discuss the hardware requirements of the project and will go into further detail regarding the type of hardware used for the project and the reasons for choosing the hardware. The hardware consists of choosing the right type of microcontroller to use and also choosing a suitable robot to perform the task's needed to do the job.

As well as choosing the required hardware it was also important to design some of the hardware for the robot as not everything was readily available to use. A suitable case or housing for the robot as well as section on the front which helps it to push the ball along the field.

2.1 Hardware Block Diagram

The Block Diagram in Figure 7, shows an overview of all the important hardware involved in the project.

The ***1st block*** shows the Logitech C920 Web Cam, which produces the frames or images of the robot and the football field.

The ***2nd block*** shows the PC, this contains the software program that processes the frames or images into position coordinates and then sends the readings or messages to the Serial Port(Wixel Module). There are two messages that the PC sends to the serial port which are position message(POS) which contains all of the position coordinates and a stop message(STOP). The computer program also has a GUI(Graphical User Interface) that contains all of the features or buttons that the user can use to control the program as well as showing the frames that are received from the Web Cam, this will be mentioned in more detail later on.

The ***3rd Block*** shows the Wixel Module Pair block which is basically a pair of communication modules that the PC and the robot need to communicate with each other. The PC and the robot each have their own module that enables them to exchange information with each other.

The ***4th Block*** shows the Pololu robot used for pushing the ball along the field.

The ***5th Block*** shows the Football Field and Robot Ball.

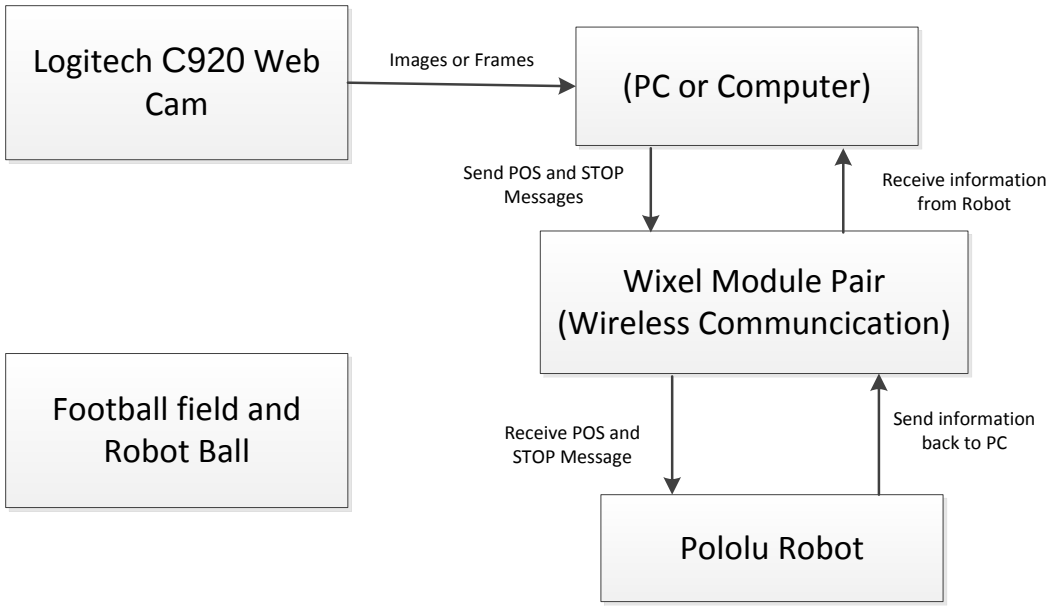


Figure 7

Each of the 5 blocks in the following diagram will be mentioned in further detail.

2.1.1 Logitech C920 Web Cam (First Block)

Figure 8 shows the Logitech C920 Web Cam that was used as the camera for producing the frames and images that the PC receives to perform the calculations needed to determine the location of the robot and the ball on the field.

Figure 8

Figure 9 shows how the camera is used and was mounted above the football field. It was important to have the camera mounted at a suitable position and height to enable it to capture the entire image of the field and the robot and ball on the field. Also shown in the image is the football field which consists of 4 coloured circles, the robot which has 2 coloured circles on top, and the orange ball that the robot pushes along the field.

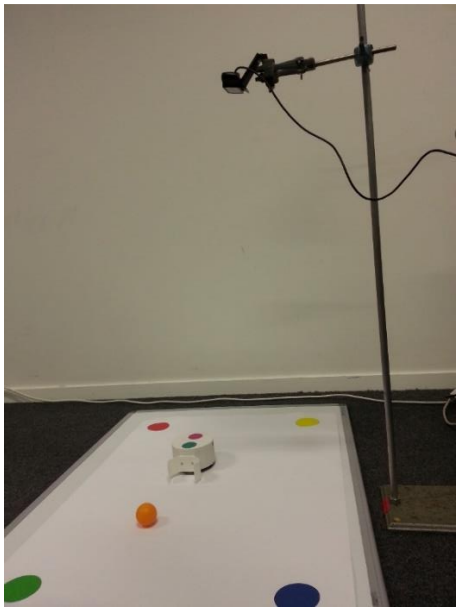


Figure 9

2.1.2 PC or Computer (2nd Block)

The PC is the hardware device that will run the software program that performs the image processing calculations on the frames being received from the Web Cam as well as providing a GUI (Graphical User Interface) for the user. As shown in ***Figure 10*** the computer is running off a laptop.

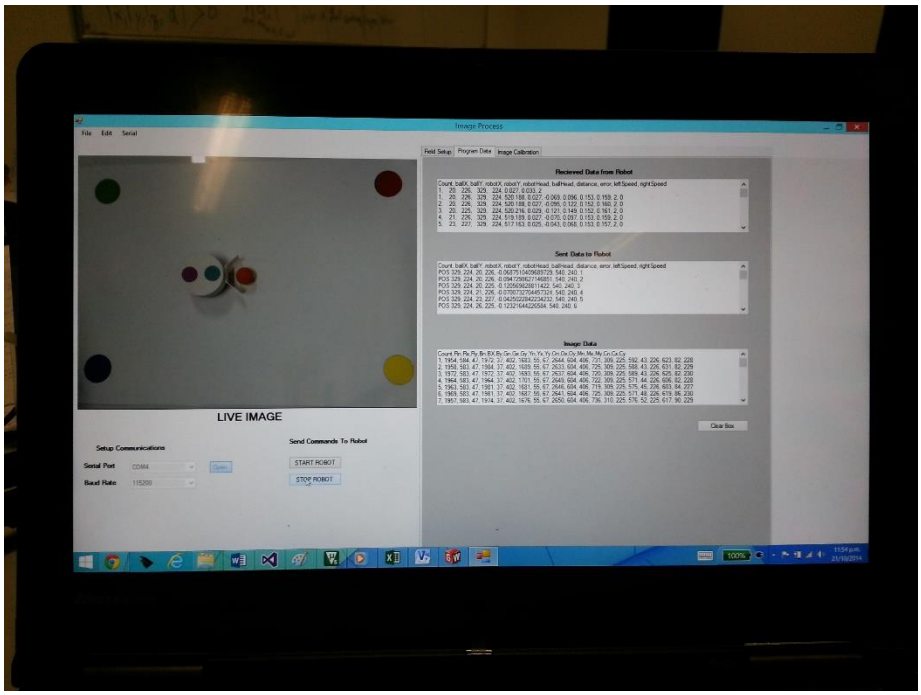


Figure 10

2.1.3 Pololu Robot (Third Block)

The following robot used in this project is the ***M3pi*** robot which is manufactured by a company called ***Pololu*** as shown in **Figure 11.** The reasons for choosing this type of robot is that it was important to choose a robot which had enough suitable on-board hardware features to do the job as well as allowing for additional hardware to be added on if required such as a wireless communication module, colour sensors, and long range sensors for detecting obstacles. The robot also consists of 2 PCB's, the top PCB is the expansion PCB and allows for additional hardware to be added as well as providing a dedicated slot for allowing a microcontroller module to be added. The microcontroller module that will be used and mentioned later on is the LPC1768. This module contains an on-board ARM microprocessor and will be the fundamental device used for controlling all the robot's main control algorithms such as carrying out its own calculations for finding the location of the ball and implementing PID control for moving in a straight line. The microcontroller module on the top PCB communicates with a slave microcontroller on the bottom PCB. Communication between both microcontrollers is established serially through a USART interface. The robot also has a dedicate slot for connecting a wireless communication module. Wireless communication is important because the robot needs to be able to receive position coordinates and data being transmitted wirelessly from the program on the PC.

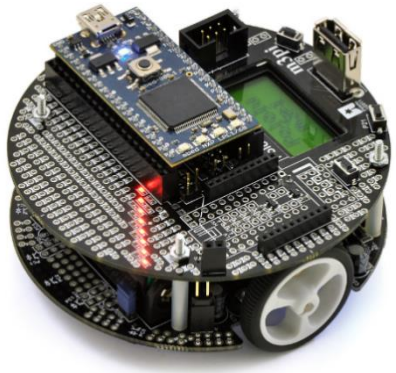


Figure 11

The bottom PCB contains an ATmega 328 microcontroller which controls the on-board hardware such as motors, LCD and line sensors. This PCB also contains the batteries that provide power to the whole robot. **Figure 12** shows the bottom PCB of the robot to be used.

Figure 12

2.1.3.1 Microcontroller module

The following microcontroller is to be used for programming the robot. The diagram in **Figure 13** shows the LPC1768 MCU module and its corresponding pin-outs. The main reasons for choosing this module is that it works with the ***mbed*** tool suite that allows prototyping and hardware programming without having to work with low-level microcontroller details, making programming and experimenting more efficient, most importantly the module is compatible with the m3pi robot.

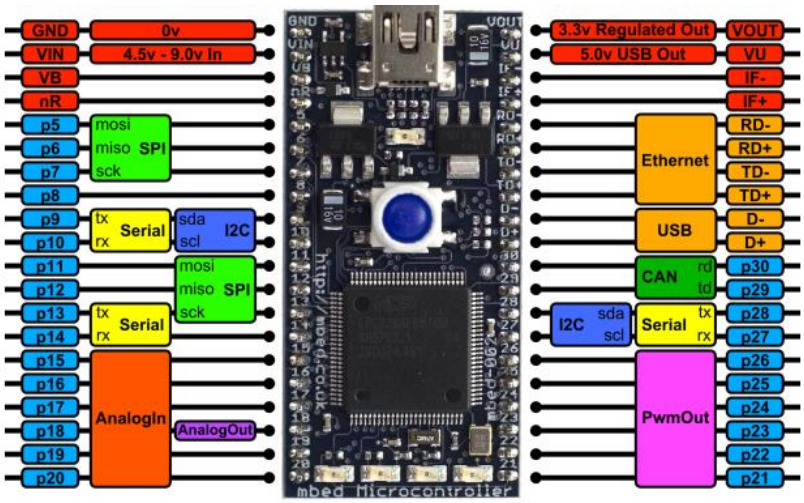


Figure 13

The other advantages with using this module is that it has Cortex-M3 hardware. The Cortex-M3 hardware consists of a 100MHz ARM microcontroller with 64 KB of SRAM, 512 KB of Flash, Ethernet, USB OTG, SPI, I^2C, UART, CAN, GPIO, PWM, ADC and DAC. The most important hardware that will be required for this project is the UART interface. The diagram of the module shows that up to 3 UART's which can be used, one of the UART (p9 and p10) is being used for the slave serial communication between the top and bottom PCB's. The other UART is reserved for a wireless communication module that will be attached to the top PCB where the tracks connect directly to pins 27 and 28. This leaves only one UART for adding additional hardware that requires a UART interface such as colour sensors or long range sensors if required later on. The ARM microcontroller itself comes in a 100-pin LQFP package but the amount of available accessible features is limited to what is being shown in **Figure 13** that the LPC1768 module can provide.

An initial idea for the project was to have a camera mounted onto the robot but this was ruled out because of the amount of available RAM that the microcontroller had which was 64 KB. Therefore the decision was made to have a camera operating independently to the robot, and have a computer program to carry out all of the image processing calculations for determining the coordinates and have the readings transmitted to the robot as the PC has a

plentiful supply of RAM and is a very powerful tool. This would also increase the efficiency and simplify the programming code of the robot as the robot would have software constraints to deal with.

2.1.4 Wireless Communication Pair (3rd Block)

As indicated in the 3rd block of the hardware block diagram in ***Figure 7***, it explains that there is a “Wireless Communciation Pair” which means that there are two wireless communication modules. One of the modules is connected to the PC and the other is on the robot. The purpose of these modules is to exchange information with each other. The wireless communication module of choice for the robot and the PC is shown **Figure 14** which is called ***Wixel*** and is manufactured by ***Pololu.*** It features a 2.4 GHz radio.

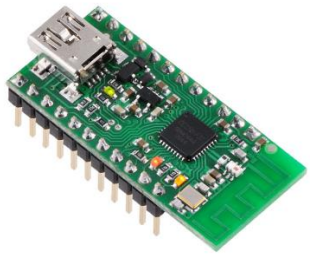


Figure 14

Figure 15 shows the Wixel module connected to the PC(Laptop)

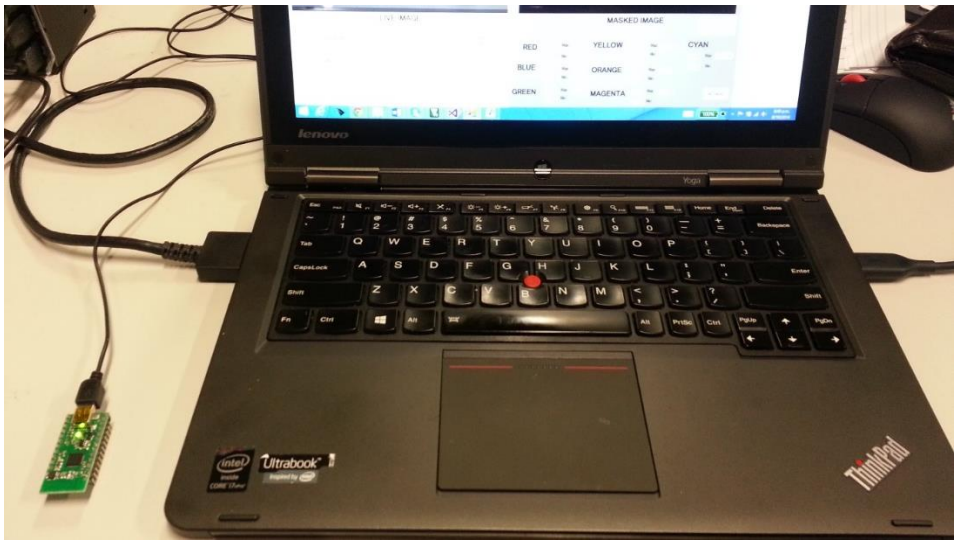


Figure 15

2.1.4.1 *Configuring the communication modules*

The Wixel or module is configured using the Wixel Configuration Utility which allows different profiles to be selected, you can specify the type of user configuration for the wireless module. As well as specifying the type of configuration for the module you can also specify the baud rate types as well. As shown in ***Figure 16*** below.

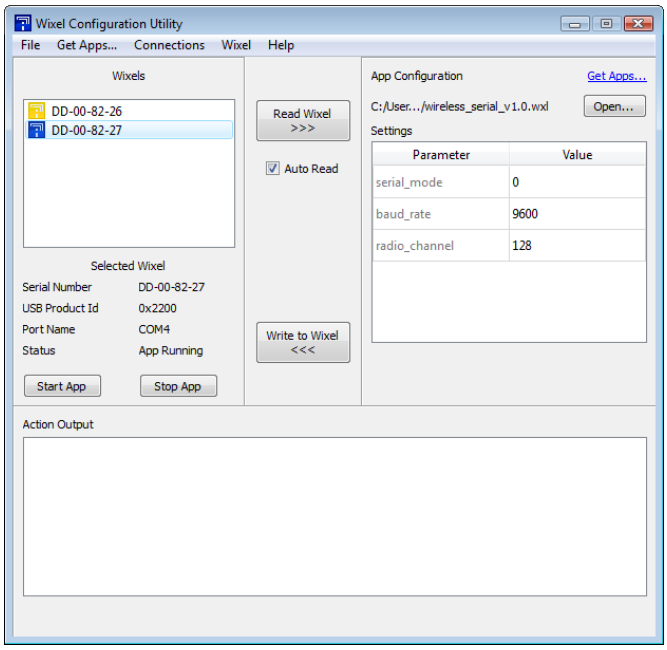
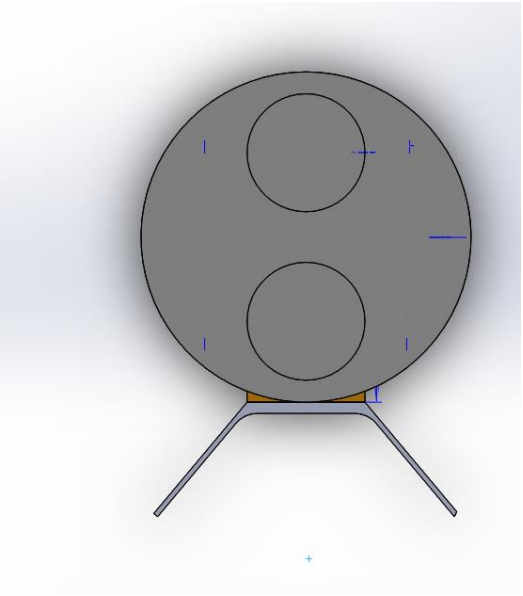


Figure 16

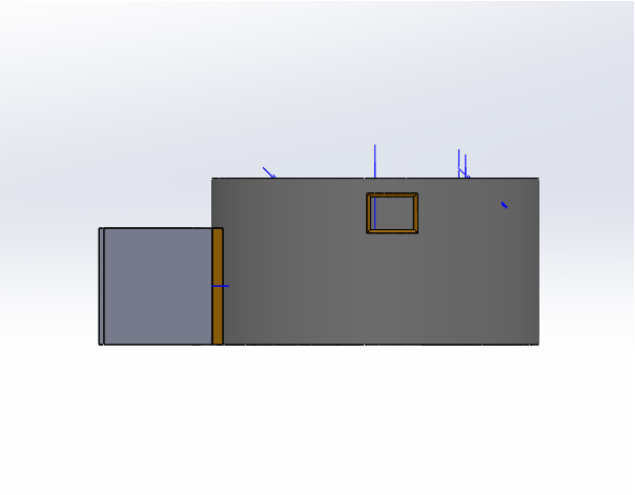
2.2 3D Printed Parts for the Robot

It was necessary to create a 3D printed model case that would fit snuggly on the ***Pololu Robot*** using Solid Works in order to make the robot look more presentable and so that an attachment for moving the ball along the field can be mounted on the front of the robot. Originally a light weight plastic ping pong ball and a cardboard paper model for the cover was used for the robot during the prototyping stages.

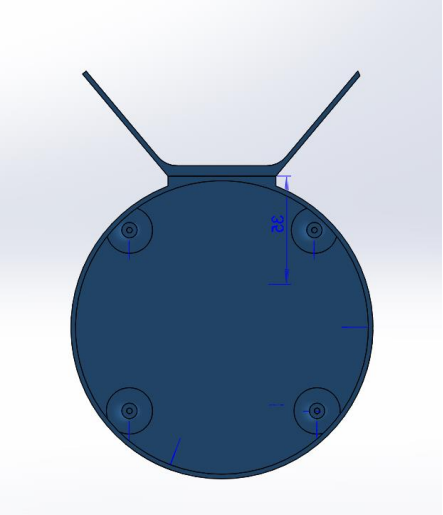
The images below shows the 3D models of the following robot, shown in different angles. The circles on the top of the robot allow the coloured paper circles to be mounted on the top and they have a diameter of 35mm.



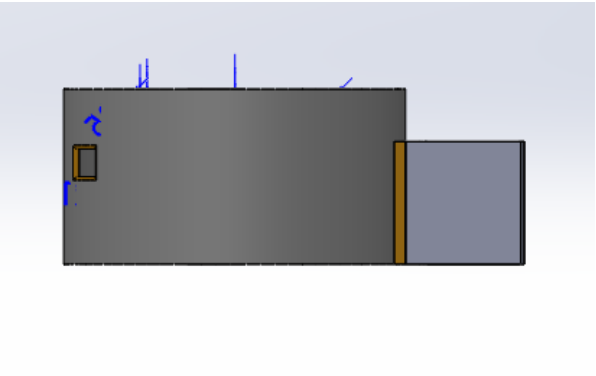
Top View



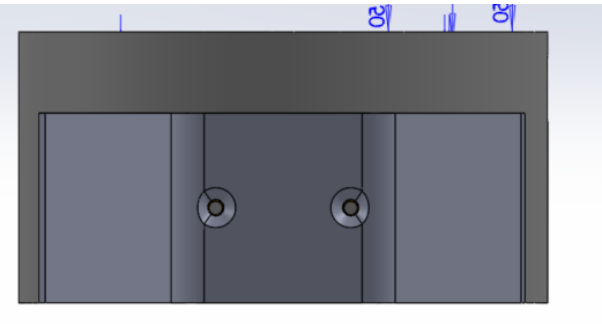
Right View



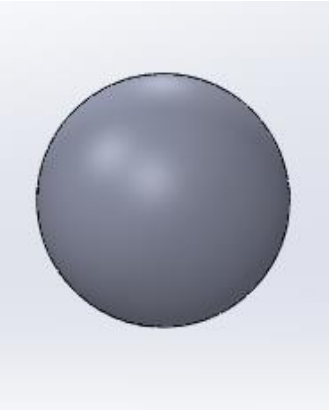
Bottom View



Left View

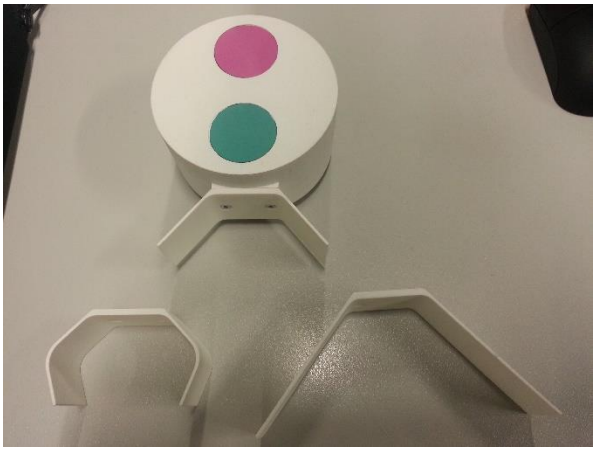


Front View



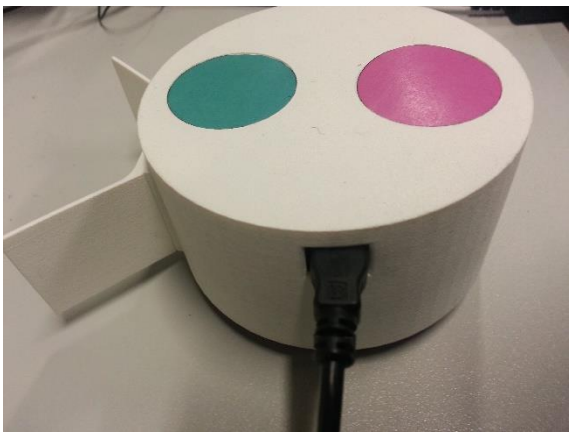
3D printed Ball

The image to the left shows the prototype paper model that was mounted onto the robot previously used and the image to the right shows the 3D printed model that was mounted onto the robot as well as the various types of parts used. The following parts are interchangble and can be screwed onto the front of the plastic housing depending on the required programming application of the robot.



The added difficulty with desgning the case for the robot is that it had to be designed in a way that provided access for the USB programming cable and to provide access for charging the battery on the robot using a charger. The left view shows the programming slot for the robot whereas the right slot shows the charging slot to access the port on the robot that charges the Pololu robot. The following images show that the slots designed for the case had to take into consideration the access to charging the battery of the robot as well as providing acess to programming the Arm microcontroller module of the Pololu robot.

Robot connected to charging Port



Programming in robot slot

As well as creating the 3D printed model and parts for the robot for the robot, it was also necessary to create a 3D printed ball that is modelled on the existing ping pong ball that was

previosuly used.The difficulties that were discovered with using the Ping Pong ball was the weight, like for example the robot would regularly push the ball off the field even as gentle as a small tap. The advantage of having the 3D printed ball is that it is much heavier as it is made from solid plastic and is not hollow unlike the ping pong ball is. After getting the ball 3D printed it was also spray painted to replicate the exact colour of the Ping Pong ball, as indicated in the following image.

3 Software

This section will mention in full detail the software that was designed for the project. It was important to design suitable software that performed the appropriate tasks. The appropriate tasks in this case was designing the software program on the PC, which was needed because it was important to have a software program that was able to calculate the X and Y positions of the robot and the ball on the field from the pixels in the image, received from the web cam and to also have the readings calculated in millimetres(MM) which are done applying the theory that was mentioned in ***Section 1.1***. A suitable GUI(Graphical User Interface) was also a very important aspect in the software design because this allows the user to have access to the control features of the software such as calibration, seeing each web frame being displayed and observing the data being sent and received from the serial port which will be mentioned in further detail later on. A partial object orientated solution for the Web Cam features was created for the final programing design of the PC software, where a separate class file was wrote for an ImageCapture Class containing methods that provide access to the functionality features of the camera. The methods created replaced the need of having a button to turn on the camera so a method was created to incorporate this action whereas previously this was done by pressing an ON/OFF button on the GUI. This section will also mention the class that was created in further detail.

As well as having a software program on the PC, it was equally important to have software on the robot as well. The software involved on the robot is a lower level embedded software that enables the robot to perform its own calculations and various other control features that determine which control its motors to move in the required direction. The robot relies on the positions strings or messages transmitted wireless from the software program on the PC for determining its feedback which it uses to determines its position and heading as well as the current position of the ball on the field.

3.1 Software Program on the PC

3.1.1 Current Architecture

The following block diagram in ***Figure 17*** shows a hierarchal view of the software architecture of the program on the PC including the hardware that the program interacts with. The PC software blocks is shown inside the black dashed rectangle which interfaces to the hardware blocks such as the Web Cam and the Wireless Wixel Module(Serial Port).

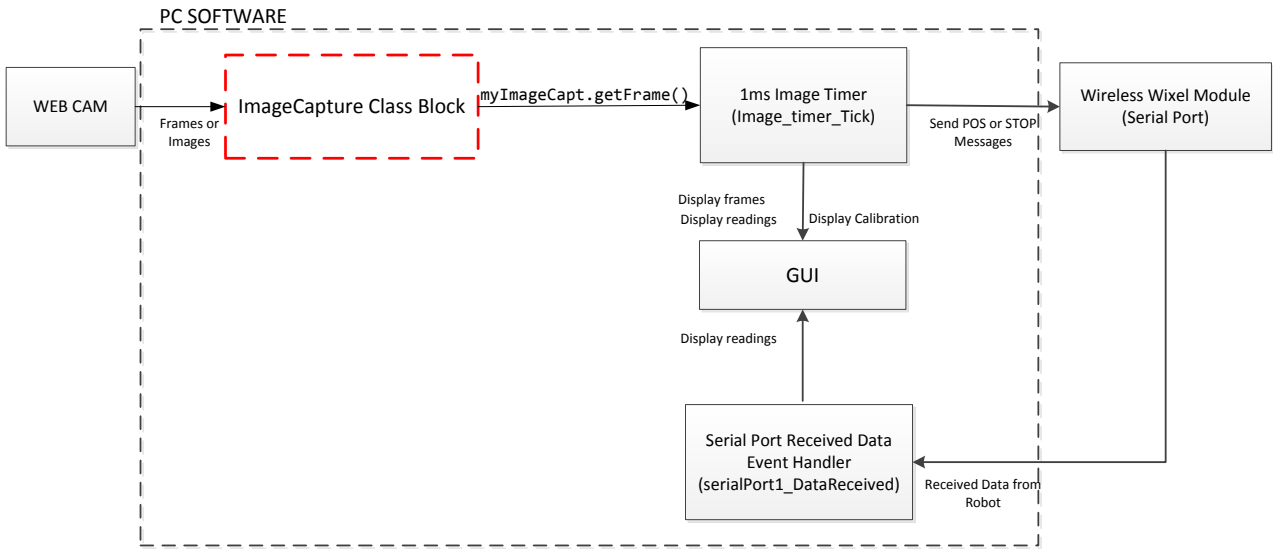


Figure 17

The higher hierarchal level of the software consists of the Event Handlers and various other functions that interface with the hardware. Each event handler inside the software block consists of many other functions which perform the image processing tasks and calculating the X and Y positions of the field in MM(Millimetres) and sending the readings out to the robot via the Wixel Module or serial port which is done further down the hierarchy. Each of these individual blocks, including the lower hierarchy details will be mentioned in this section.

The blocks show that a hardware device such as a Web Cam produces frames at a rate of 33 frames/sec. These frames then get passed to the ***ImageCapture* class** block as indicated by the dashed rectangle which triggers the ***Web Cam Frame Event Handler*** inside the class. Each frame is then captured, copied, and then stores as 24 bit ***Bitmap*** image somewhere in memory or in a global variable.

The ***1ms Image Timer Event*** Handler is a timer interrupt event handler that is executed periodically every one 1ms. The timer can access stored in memory inside the frames that are captured and stored in memory by the event handler are accessed by executing the `myImageCapt.getFrame()` which checks every 1ms to see if the latest Bitmap is not null, if the variable is null then then the program returns or exits the event handler. When the Bitmap is not equal to null then the ***1ms Image Timer Event*** displays the current Bitmap on the GUI and after that it copies the Bitmap and then performs image processing which converts the pixel readings into millimetres and then sends the readings out to the serial port as a message (POS or STOP) or as a string, that the robot receives, more information on these strings will be mentioned later on.

3.1.1.1 Graphical User Interface(GUI)

The Graphical User Interface or GUI provides a user friendly interface for the software program on the PC that performs the imaging processing calculations. Over the course of the project many of the following features of the interface had been changed, altered where unnecessary features were removed such as buttons for turning on the web cam have been simplified to just turning on the web cam when the PC program starts up. ***Figure 18*** shows the final and complete GUI for the project.

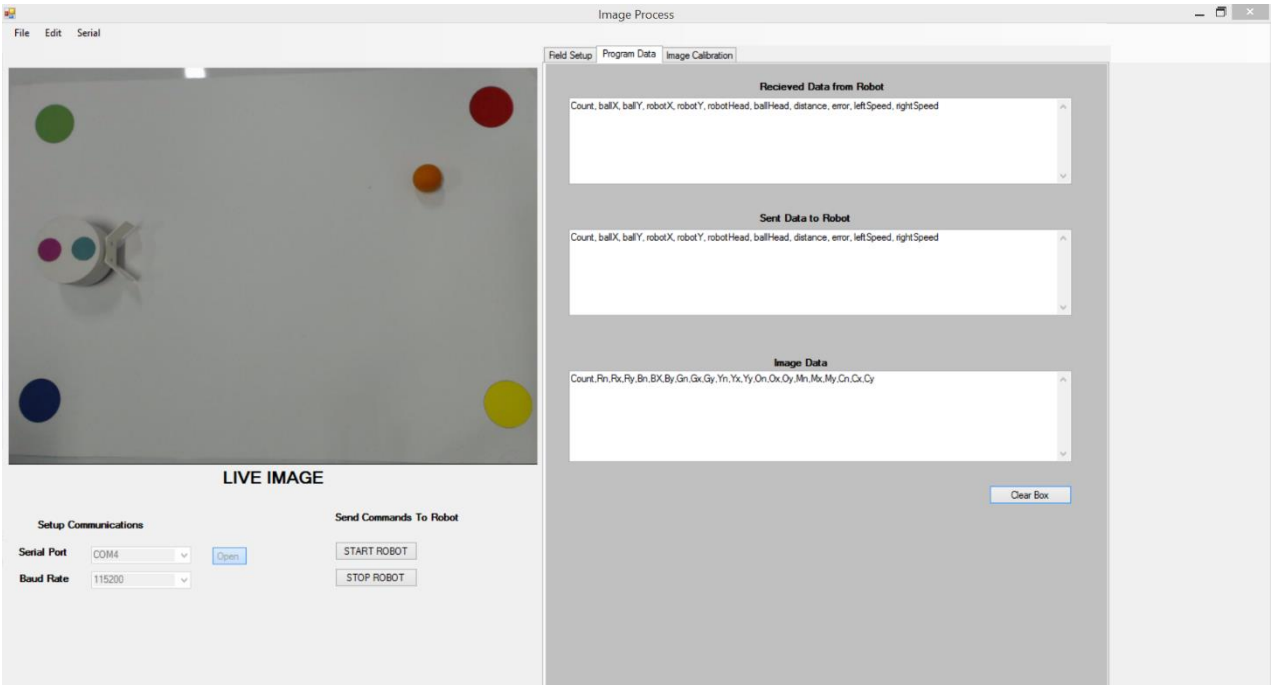
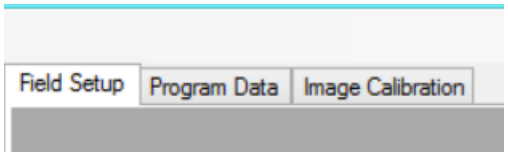


Figure 18

What is basically shown here is that the frames that are received from the Web Cam are outputted to the following picture box indicated by the text “LIVE IMAGE” which is basically showing the live images as they are being received from the web cam, which allows the user to make sure that the setup of the field and the coloured circles are present.



The other features that are shown are the 3 textboxes, in the right of the GUI labelled, “Received Data from Robot”, “Sent Data to Robot” and “Image Data”. Each of these texts boxes allow the user to analyse the data that is being received from the robot or calculated by the software program to allow for error checking and making sure that the readings are consistent.

Also notice in the GUI that there are 3 different tab options which provide access to multiple features and parts of the GUI. The names for these tab features are shown in the left hand side of the following image which are called “Field Setup”, “Program Data” and “Image Calibration”. The current option selected is the “Field Setup” option and what this option basically does is allow the user to specify the size of the field that is being used. In the following example shown the size of the field is 685MM x 344MM. This length was worked out by measuring the length of field from the centre points of each of the circles as state in ***Figure 20,*** the field area is located within the black rectangle.

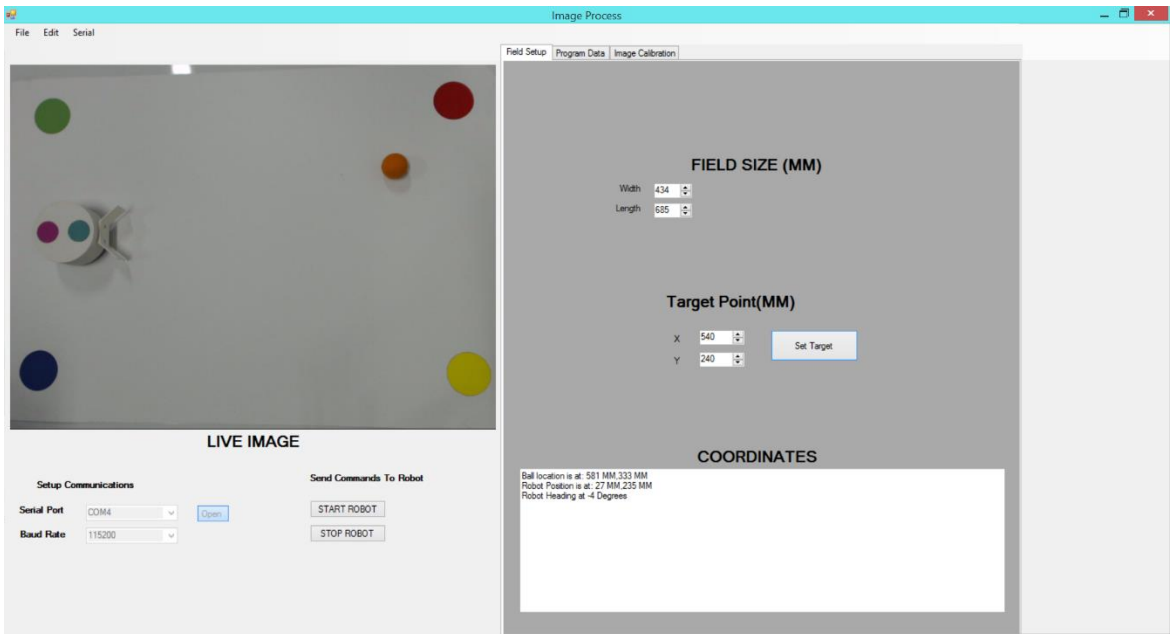


Figure 19

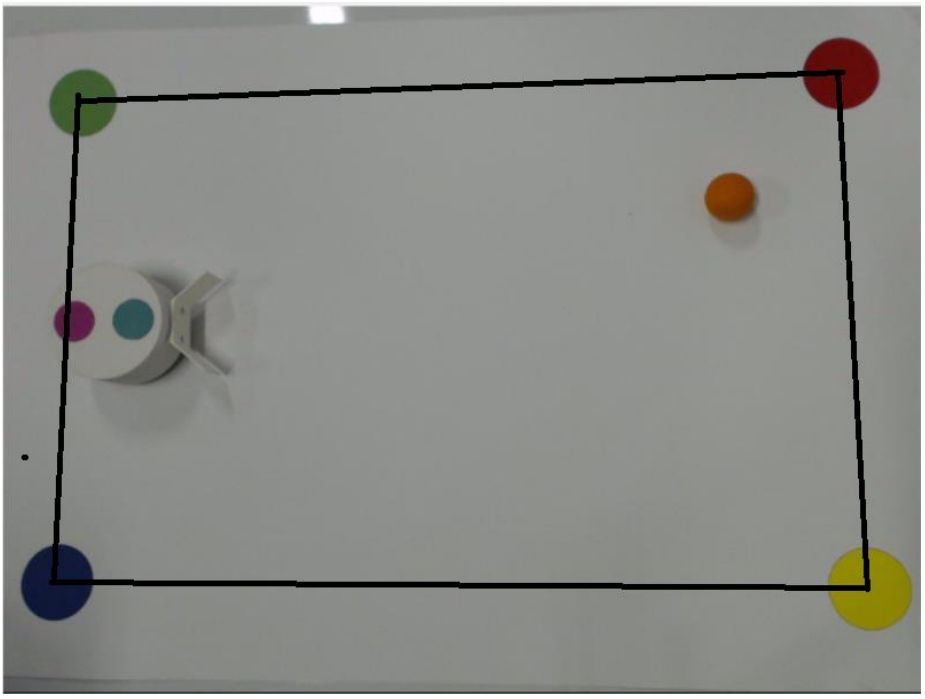


Figure 20

The “Calibration tab options allows the user to calibrate the field by varying the Saturation and Hue values for each of the 7 different colours that are present in the ***“Live Image”.*** The calibration options displays a masked version of the current live image which informs the user if the software needs to be calibrated or not. As the lighting conditions change which are dependent on the time of day or the and the presents of natural light, these can have a profound effect on the number of pixels required to carry out the calculations needed.

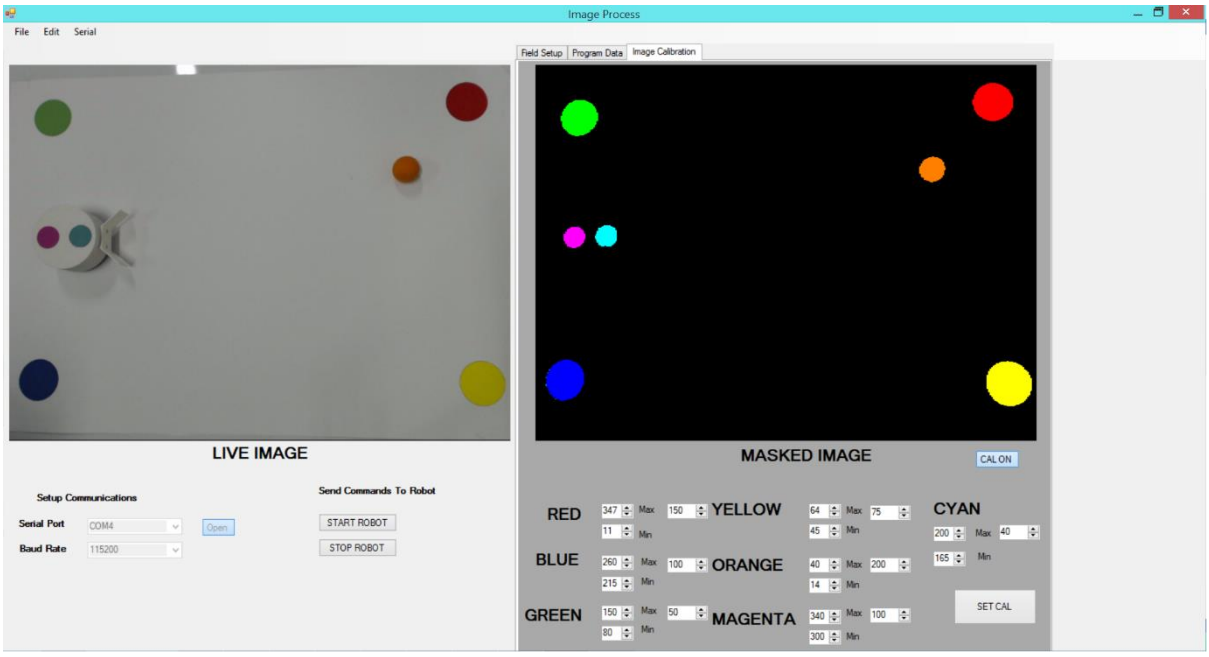
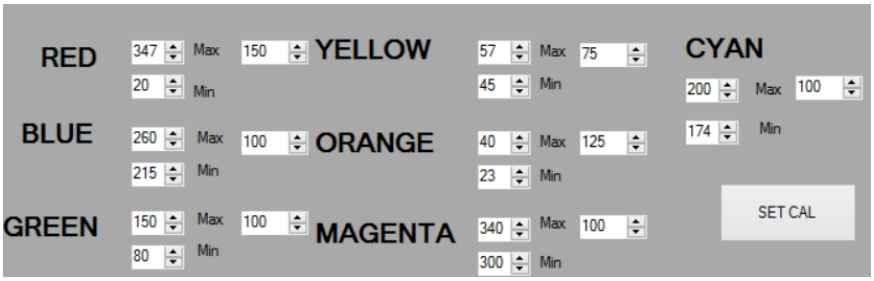
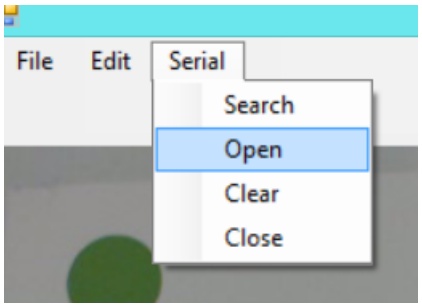


Figure 21

Fortunately the ***"CAL ON"*** featured can be pressed and then the masked version of the image will start to appear as indicated by the image shown on the right hand side of the GUI in the picture box. If one of the colours on the MASKED IMAGE picture box starts to fade away or disappear then the following Saturation and limits of calibration can be changed if

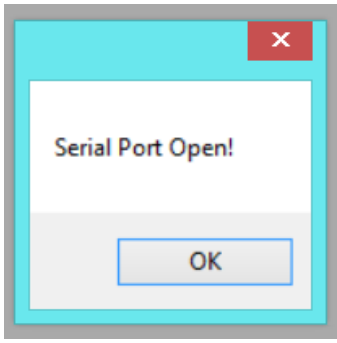


required until the image shows all of the 7 colours like the image in ***Figure 21*** shows. When the user is satisfied with the calibration, the following feature can be turned off by pressing the ***"CAL ON"*** button a second time and the colour of the button should change from blue to grey again.

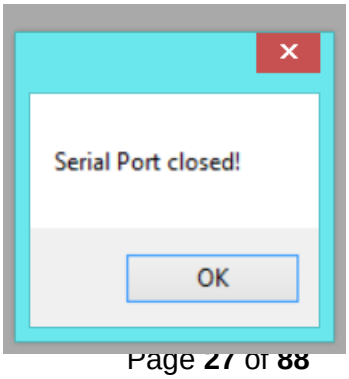


The other features that can be configured is the configuring and setting up of the ***Serial Port***. The serial device that sends out the readings can be configured by selecting the "Serial" menu option in the top of the program bar. What this basically does is open the serial port device that was automatically detected during program start up, example detecting the Wixel module that is connected to the PC which configured as ***"COM4".*** The drop down menu enables the user to select the open option and what basically happens is that a "Message Box" appears informing the user that the following serial port has been opened. The status of the serial port being opened is also shown in the Setup Communications option which will basically changes from being enabled to disabled as indicated. As well as opening the serial port, the closing option of the serial port can also be selected and what basically does is stop the program from transmitting any

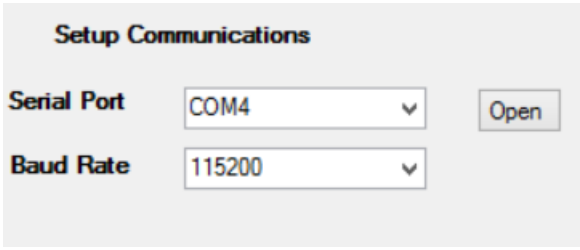
for



is



this

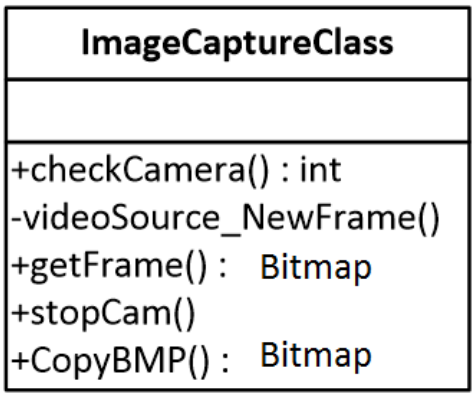


message strings to the robot. The status of the serial port being closed is indicated by the following message and the ***"Setup Communications***" option will change back to the enabled state again, these are stated in the following images.

After mentioning the serial port features of the GUI, the part that actually initiates the sending of the position string messages out to the serial port is done by pressing the ***"START ROBOT"*** and ***"STOP ROBOT"*** buttons, which are indicated below the ***"Send Commands To Robot"*** text. Pressing the "START ROBOT" button initiates the sending of the readings that the PC software actually calculates, this can only happen if the serial port is open, if the serial port is closed then no strings are sent and a Message Box appears informing the user that the serial port is closed is displayed. The "STOP ROBOT" button is used for sending "STOP" commands to the robot, which informs the robot that it needs to stop moving.

3.1.1.2 ImageCapture Class Block

The ***ImageCapture Class*** Block is a separate class block which contains all of the functions or methods that relate to controlling the Web Cam features. The methods shown inside the ***ImageCaptureClass*** is shown in the class diagram in ***Figure 22.***



/

Figure 22

Most of the methods or functions inside this class are accessible from outside the class. The only method that are not accessible outside this class is the ***videoSource_NewFrame()*** event handler which is declared as private. Each of these methods and how they are accessed in the main function of the program will be explained in more detail later on.

3.1.1.2.1 checkCamera() Method

This method is the method of the ImageCapture Class and is responsible for checking if the Logitech Web Cam has been connected to the PC as well as creating the New Frame Event handler which in this case is the Logitech c920 Web Cam. When the software program was first executed the following method operates by running some code which returns a “0” if the webcam has not been found or otherwise it returns a “1”. The structure diagram in

Figure 23 mentions how the following method operates.

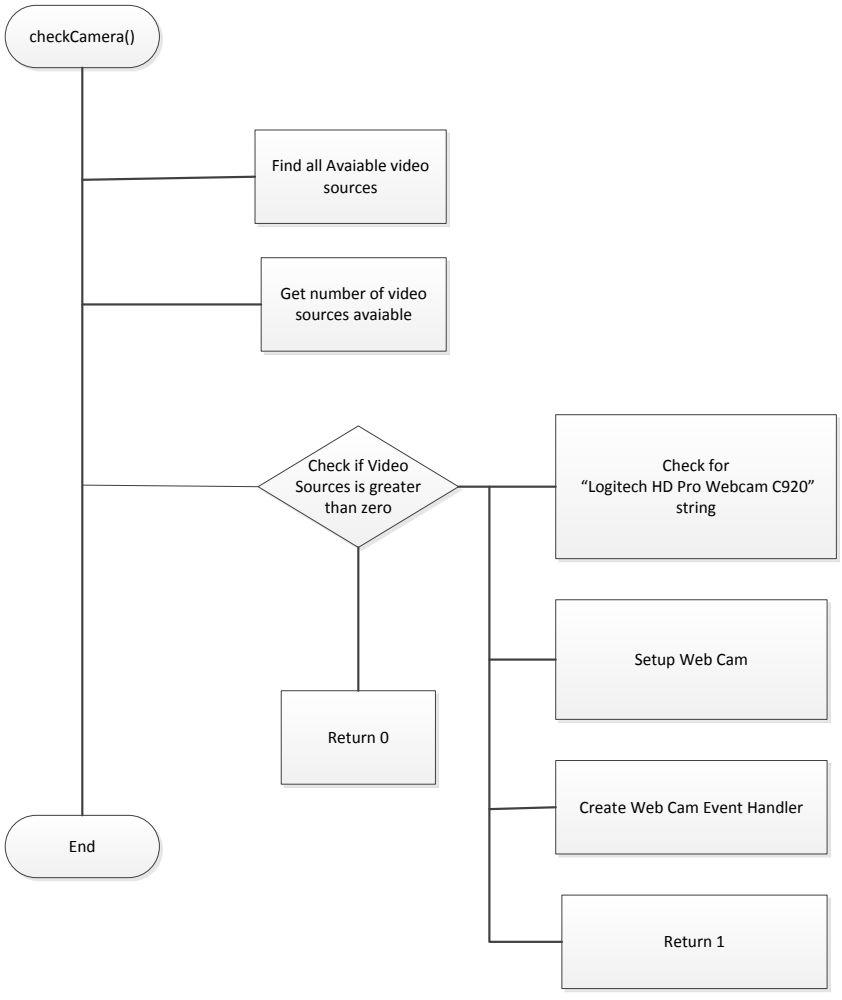


Figure 23

The structure diagram of the ***checkCamera()*** method shows that when the program is first executed it tries to find all of the available video sources where it basically lists all of the

video capture devices it finds connected to the PC as shown in the following coding segment below. The **FilterInforCollection** finds all of the available videos sources that are connected or found on the computer, this could be the Logitech Web Cam or other devices such as an integrated web cam on the laptop.

```
//List all available video sources. (That can be webcams as well as tv cards, etc)
FilterInfoCollection videosources = new FilterInfoCollection(FilterCategory.VideoInputDevice);
```

Once the information of all the video sources was retrieved it was important to obtain the count or number of video devices connected in an effort to try to extract the device of most importance. This is what the following segment of code shows here. It shows that the count method for the ***videoSources*** object can be obtained by placing a dot at the end of the object and this method will return a number which informs the user of the number of video sources available. If the number is greater than 0 then the next block is executed, if the number happened to be zero then the checkCamera() method returns zero.

```
if (videosources.Count > 0)
{
    for (int i = 0; i < videosources.Count; i++)
    {
        //For example use first video device. You may check if this is your webcam.
        videoSource = new VideoCaptureDevice(videosources[i].Name);
        if (videoSource.Source == "Logitech HD Pro Webcam C920")
        {
            sourceCheck = "Logitech";
            videoSource = new VideoCaptureDevice(videosources[i].MonikerString);
        }
    }
```

The checkCamera() method is also checking for a particular string where it would check for the “Logitech HD Pro Webcam C920” string, which can be accessed by accessing the ***Source*** method of the ***videoSource*** object or ***videoSource.Source***. A string variable called sourceCheck will be assigned the string “Logitech” when the “Logitech HD Pro Webcam C920” string has been found. After doing this the ***videoSource*** object will be assigned a ***MonikerString***. What the MonikerString does is that it allows the PC program to have control over the webcam hardware which can be accessed through methods inside the ***VideoCaptureDevice*** Class.

Once the ***videoSource*** object methods or controls can be accessed, the setup of the web cam can commence as indicated by the “Setup Web Cam” block in the structure diagram in

Figure 23. The following section of code below does exactly that. The next section of code checks to see if the ***sourceCheck*** variable contains the string “Logitech” which was assigned when the Web Cam was found successfully. The next thing that happens is that

the resolution of the Web Cam is configured. The resolution of the Web Cam is specified by setting the ***video.VideoCapabilities[0]***, this sets the resolution to ***640x480***. Where the Width of the frames from the Web Cam is equal to 640 and the Height is 480 pixels. Next the frame rate of the web Cam is set to 33, where 33 frames per second is the maximum frame rate of the Web Cam and the last thing that is configured is the focus which is set to manual. The reasons for configuring this as manual was to prevent the web cam from auto focusing, because the web cam will be in a static position all the time so auto focusing is not necessary for this particular application.

```
if (sourceCheck == "Logitech") //check if video source found is the logitech webcam!
{

    try
    {
        //Check if the video device provides a list of supported resolutions
        if (videoSource.VideoCapabilities.Length > 0)
        {
            //Set the highest resolution as active
            videoSource.VideoResolution = videoSource.VideoCapabilities[0];
            //Set Frame rate to highest
            videoSource.DesiredFrameRate = 33;
            videoSource.SetCameraProperty(CameraControlProperty.Focus, 0, CameraControlFlags.Manual); //set focus to auto
        }
    }
    catch
    {

    }
```

Finally the last thing that is configured in the checkCamera() method is the event handler for when a frame is received from the Web Cam. Then the videoSource.Start() switches on the Web Cam and starts producing the frames or images. After this happens the checkCamera method returns a value of 1, to indicate that the camera setup was successful.

```
videoSource.NewFrame += new AForge.Video.NewFrameEventHandler(videoSource_NewFrame);
videoSource.Start(); //turn video on
return 1;
```

3.1.1.2.2 VideoSource_NewFrame event Handler

In order for it to be possible to receive frames from the Web Cam into the software program, an event handler would need to be created. An Event Handler is a method that contains the code that gets executed in response to a specific event that occurs in an application, the specific event in this case is the frames being received from the Web Cam. The event handler is called ***videoSource_NewFrame***. The following coding segment for the event handler is shown below. What basically happens here is that the frames coming from the web cam, trigger the following event handler and are passed to the ***Event Handler*** as an ***eventArgs.Frame*** where every frame that is received is converted into a Bitmap image and then saved in the ***newBmp*** variable. Next a copy of the ***newBmp*** variable is made where it

is passed onto the ***CopyBMP(newBmp)*** function. The reasons for making separate copies of the Bitmaps is so that individual Bitmap's can be displayed on the GUI and the other can be used for image processing. Once the copy of the Bitmap has been made it is saved into memory or placed into the global variable called ***lastestBmp***, and the ***bitmapFlag*** is then set to ***true.*** The setting of the flag is important to signify to the getFrame method that the variable contains the latest Bitmap and can be passed onto the ***getFrame()*** method which will be explained in the next section.

```
private void videoSource_NewFrame(object sender, AForge.Video.NewFrameEventArgs eventArgs)
{
    Bitmap newBmp;
    newBmp = (Bitmap)eventArgs.Frame;   //recieve new frame from webcam and convert into Bitmap
    latestBmp=  CopyBMP(newBmp);
    bitmapFlag = true;
}
```

3.1.1.2.3 getFrame() method

This method checks the global variable or the lastestBmp to see if it is not equal to null, and after checking that it clears the bitmapFlag and then returns the latestBmp to the class or function that called the following method. The following method was made public because this allows the main form function to have access to this method that deals with getting the frames from the Web Cam. The software program uses an image timer event in the main form function which accesses the frames by calling this method.

```
public Bitmap getFrame()
{
    if (bitmapFlag)
    {
        //check if event handler has set latest image to true
        bitmapFlag = false; //clear flag after recieving the latest bitmap
        return latestBmp;
    }
    else
        return null;
}
```

3.1.1.2.4 stopCam() Method

This method is responsible for turning off or stopping the Web Cam from producing frames. This method is important because the user may want to terminate the program's execution when the close button of the form has been selected so it was important to ensure that the

program was shut down properly and to prevent the Web Cam from producing further frames. The following code for this is shown below.

```
public void StopCam()       //turn witch cam off
{
    videoSource.NewFrame -= videoSource_NewFrame;   //remove new Frame event handler
    videoSource.Stop();
    videoSource.SignalToStop();
    videoSource.WaitForStop();
}
```

What basically happens here is that the event handler is removed, then the VideoSource object is stopped by accessing the Stop() method, the SignalToStop() method is also activated to allow the request signal to be made available, then finally the WaitForStop() method is used so it can wait for the Web Cam object to stop sending the frames. These methods allow proper termination of the program in which the Web Cam is used.

3.1.1.2.5 CopyBMP method

A copying function called ***CopyBMP*** was created which uses the Lock bits method for locking the Bitmap image into a rectangle and into system memory. The reasons for using this function is because it was important to make a copy of the Bitmap itself as opposed to copying just the reference.

```
public Bitmap CopyBMP(Bitmap bmp)
{
    Bitmap newBMP = null;
    if (bmp != null)        //if bmp is not null
    {
        //clone the Bitmap
        Rectangle cloneImage = new Rectangle(0, 0, bmp.Width, bmp.Height);
        System.Drawing.Imaging.PixelFormat format = bmp.PixelFormat;
        newBMP = bmp.Clone(cloneImage, format);
    }
    return newBMP;
}
```

This method is accessible outside the ImageCapture Class as well as being used by the event handler for copying Bitmap images inside the class due to the function inside the class being public.

3.1.1.3 1ms_Image_Timer(Image_timer_Tick)

The purpose of using a 1ms Form timer for carrying out the image processing is that it replaces the need for have a button to carry out the processing, this will be explained in more detail later on. The architecture for the image timer event handler is shown in the structure diagram below in ***Figure 24***.

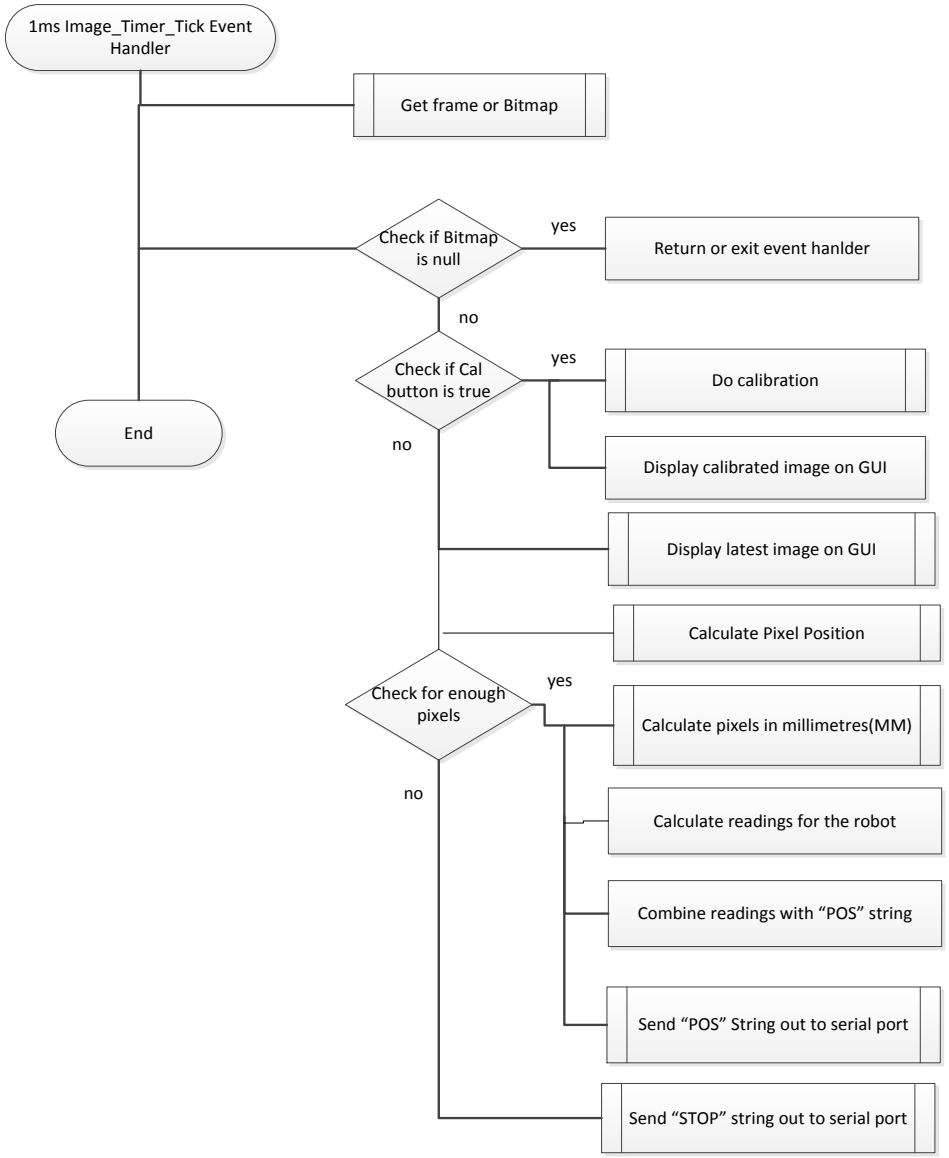


Figure 24

The thing that is happening here is that when the event handler is first executed, as indicated by the ***"Get frame or Bitmap"*** block where it gets the latest image Bitmap image from the Web Cam, it uses the ***.getFrame()*** method inside the ***ImageCapture*** class that was mentioned earlier on to access the bitmap. The following coding snippet for doing this is shown below. The object for the ***imageCapture*** class is declared as ***myImageCapt*** inside the main form. The method returns a Bitmap type which gets placed inside the bmp variable.

```
bmp = myImageCapt.getFrame();    //get frame from webCam
```

The statement in the diagram checks to see if the Bitmap image is equal to null if it's equal to null then the program returns from the event handler otherwise it continues onto the next stage. The reasons why checking this was necessary is because the image timer is being executed at a rate of 1ms but the Web Cam event handler only receives 33 frames a second or 1 frame ever 30.3ms. This would mean that there would be periods where null frames are being received because the event handler has not yet been triggered. The check prevents null bitmaps from corrupting the next stage of the image processing which would otherwise throw ***"null"*** exception errors further down the line.

```
if (bmp == null)        //return if Bitmap is null
    return;
```

The next statement in the diagram checks to see if the calibration button or ***calButt*** has been set or enabled. What this basically means is that on the software user interface or GUI there is a button (CAL ON), which enables the user to perform calibration on the current Bitmap image. ***Figure 25*** shows that the that the status of the ***CAL ON(calButt)*** button is true or Checked so this means that the latest Bitmap stored in the ***bmp*** variable will be calibrated instead of being used for performing calculations that convert pixels into millimetres. The code that describes this is shown below

```
if (calButt.Checked == true)           //check if calibration button has been enabled
{
    bmpCal = myImageCapt.CopyBMP(bmp);   //copy frame
    imageMaskBox.Image = CalTimerImageFunction(bmpCal);      //display masked image
}
```

Figure 25

For the ***"Do Calibration"*** block the bmp variable is passed onto a function that deals with calibrating the Bitmap stored in bmp called ***CalTimerImageFunction()***. Before this happens a copy of the bmp variable is made by passing it onto the ***.CopyBMP*** function method, which basically copies the bitmap variable and stores it in ***bmpCal*** and then the following variable is passed onto the ***CalTimerImageFunction(bmpCal)*** function. The purpose of copying the variable is to keep the bitmap used for calibration and for image processing separate. The ***CalTimerImageFunction(bmpCall)*** function uses an algorithm which performs a HSV transform for extracting certain colours from the image and then returns a Bitmap showing the colours of most importance. More information on HSV and image calibration will be explained later on. The Bitmap that is returned from this function is displayed on the (Masked Image) picture box of the GUI form. The masked image will always be shown all while the ***CAL ON*** button is enabled. ***Figure 26*** shows the final GUI that is being used for the project and the picture box to the right shows the masked image. The masked image is a modified version of the image to left but it shows the colours that are of most importance. This method helps the user to determine if the image is stable and if the user needs to recalibrate the image or not, if the lighting conditions of the surrounding environment were to change suddenly, then some of the colours could disappear from the image mask, so the user would need to recalibrate the GUI accordingly.

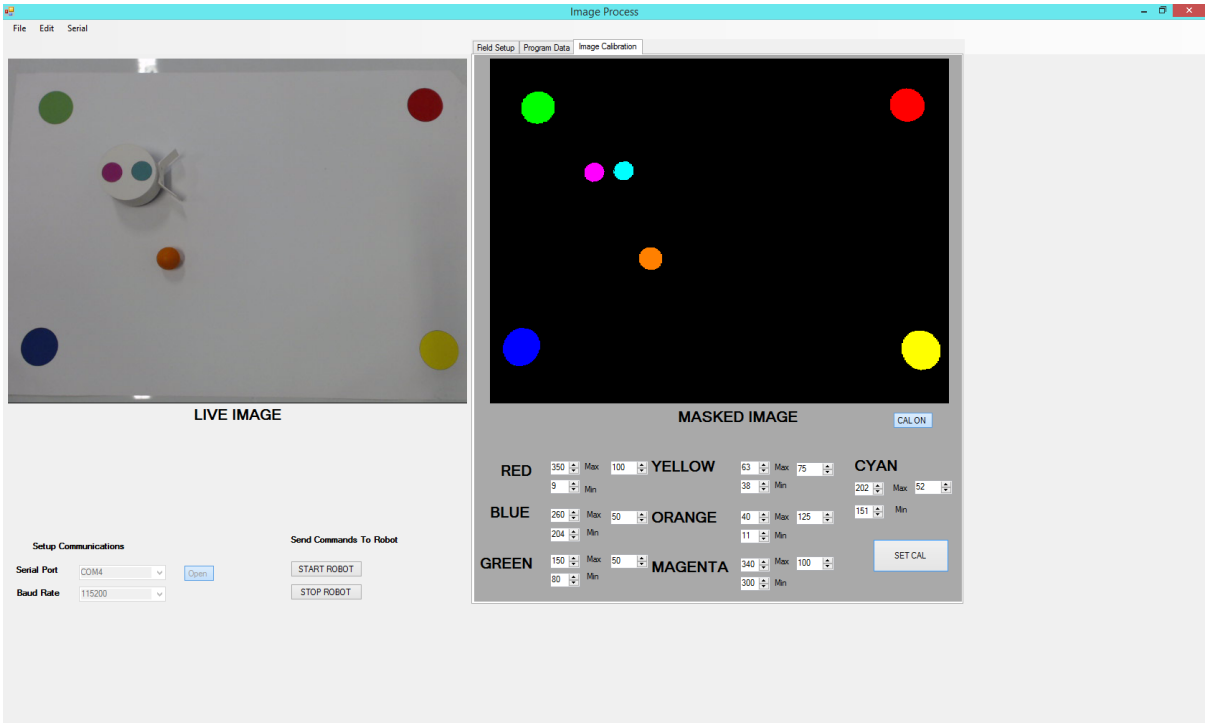


Figure 26

The situation for when the ***CAL ON*** button is not enabled, the block diagram in ***Figure 24***, performs the other steps. The next step in the diagram is indicated as ***"Display latest***

Image on the GUI", which will basically call a function that would display the frames as they are being received from the web cam and not modify them. The masked image that was previously shown on the GUI form would disappear because the following segment of code makes the imageMaskBox.Image equal to null. The next thing that occurs is that the bmp variable is passed onto the ***SetPicture(bmp)*** function. The only thing that this function does is display the bitmap on the form window as indicated on the GUI shown in ***Figure 26*** above the "LIVE IMAGE" text.

```
else
{
    imageMaskBox.Image = null;
    imageProcess = bmp;
    SetPicture(bmp);                    //make frame equal to bmp
    calcPixelPosition(imageProcess);    //calculate the pixel value coordinates and pass frame to function
    Image_timer.Enabled = false;
```

Finally the ***calcPixelPosition(imageProcess)*** function is called which passes the current bitmap onto a function that performs the general image processing of the bitmap image. The following function will be explained in more detail later on, this stage is indicated by the ***"Calculate Pixel Position"*** block in ***Figure 24.*** What the function basically does is apply a HSV transform on the entire image, similar to what had been applied on the image for the image mask calibration, and then it calculates the number of pixels it sees in the image and produces ***"X"*** and ***"Y"*** pixel positions for each of the following 7 colours. This information is important because it is possible to know exactly where each colour is located on the field and later on these readings can be converted into positions in terms of millimetres. Notice that the function is not specified as a return type. The reasons for this is because it was not a practical solution as the function would need to return 21 different numbers in total, where 3 variables would be required for each of the 7 colours which are the ***"X"*** and ***"Y"*** values for the pixels and ***"n"*** which represents the number of pixel counts for each colour. Instead the function places the variables inside position structures. The other advantage of using structures is that these can be accessed globally by other functions that need to have access to these variables such as the calcPosMM() that will later on convert the pixels into millimetres on the field. The following coding snippet below shows the position structures that were used for placing the 3 readings, which is done for the red pixel, located inside and at the end of the ***calcPixelPosition(imageProcess)*** function

```
//load variables into the structures
redPosPixel.n = red_pixel;
redPosPixel.x = xposRed;
redPosPixel.y = yposRed;
```

3.1.1.3.1 CalcPixelPosition() Funtion

This is the function that deals with the image processing of the bitmaps which calculates the pixel positions of the colours and places the readings into structures. The structure diagram that explains how this function works is shown below.

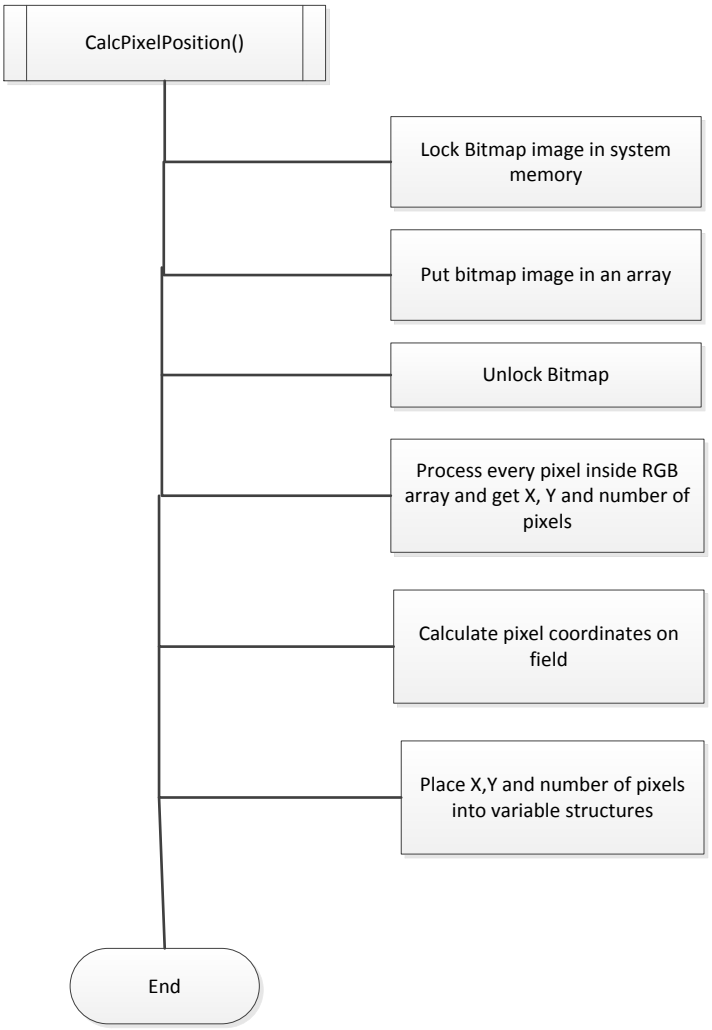


Figure 27

For the image processing software, it was necessary to use a method which is capable of extracting the RGB components of the Bitmap as fast as possible for carrying out the imaging processing calculations.

The part on the diagram in ***Figure 27*** that ***"Lock Bitmap in System Memory"*** is where the ***bData*** variable is declared as a ***BitmapData*** which specifies the attributes of the Bitmap, such as size, pixel format, the starting address of the pixel data in memory, and the length of each scan line or stride. A rectangle is specified that structures the specified portion of the Bitmap to lock onto, and other parameters are specified. The format that is specified is 24bit

RGB(3 Bytes per pixel and 1 Byte per colour), which means that each colour is 8 bits. The ***latestBmp*** Bitmap data variable contains the latest image that was received from the webcam event handler and loaded into this variable. Also the width(640) and height(480) of time Bitmap are specified. The code for doing this is shown below

```
//calculates the colours of the webcam image into pixel coordinates
private void calcPixelPosition(Bitmap latestBmp)
{
    byte[] rgbValues;       //this value will store the current Bitmap for processing the image
    IntPtr ptr;
    int bytes;
    BitmapData bData;

    bData = latestBmp.LockBits(new Rectangle(0, 0, latestBmp.Width, latestBmp.Height), ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);
```

The next block on the diagram is ***"Put Bitmap Image in array"***, where the code inside the ***unsafe*** region gets the ***pointer(ptr)*** of the first line of the address in memory where the Bitmap is located. After that an array to hold the Bitmap needs to be declared and the ***bytes*** variable gets the number of bytes that the array needs to hold the Bitmap. The array that will store the Bitmap is declared as ***rgbValues***. A marshalling technique was used for copying the Bitmap stored in memory into the ***rgbValues*** array. Having the Bitmap serialised in an array increases the speed of the image processing software because the pixel data can now be accessed directly from the array. An Unsafe region is required when pointers in C# programming code are used. As indicated by the following code below.

```
unsafe
{
    //get address of the first line
    ptr = bData.Scan0;
    // Declare an array to hold the bytes of the bitmap.
    bytes = Math.Abs(bData.Stride) * latestBmp.Height;
    rgbValues = new byte[bytes];
    // Copy the RGB values into the array.
    System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes);
} // unsafe
```

Once the ***latestBmp*** bitmap has been locked it must be unlocked as indicated by ***"Unlock Bitmap"*** in the diagram so the variable can be reused for another frame, for the event handler, which will place variables inside this data type. The RGB byte array (rgbValues) stores the entire image or Bitmap in a serialised format with a total size of ***9216000 bytes***. This was the size that was required to store the entire image because the resolution of the image from the web cam was ***640 x 480 pixels***. The colour format that the image uses is ***RGB24*** which means that each pixel consists of 3 bytes in the RGB array for ***Red***, ***Green*** and ***Blue***. Each RGB component is an 8 bit value and the intensity of the colour can be varied from 0 to 255.

```
// Unlock the bits.
latestBmp.UnlockBits(bData);            //unlock the image
latestBmp = null;
```

After obtaining the RGB components of the image, the next step was to perform a HSV to RGB conversion on every pixel to determine the pixel's actual colour in terms of its Hue and Saturation. This was a crucial part of the program because extracting the desired pixels made it possible to filter out and extract an area of an image that is of most importance such as the coloured circles and then uses this information to calculate the pixel coordinates of the different colours on the field as well as the ball and the robot. The diagram in **Figure 28** shows the pixel arrangement of where the image of the pixels are stored inside the RGB array.

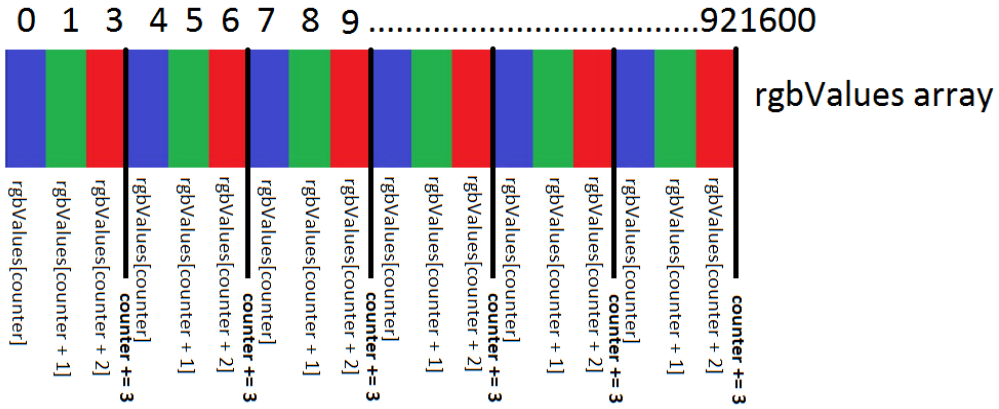


Figure 28

Referring back to the diagram in ***Figure 27,*** where the following block mentions ***"Process every pixel inside RGB array and get X, Y and number of pixels",*** is where the pixels inside the RGB array need to be extracted and the hue and saturation values for every pixel are determined. This was done using a *"For Loop"* which loops round about 307200 times, for every pixel as indicated in the following coding snippet below. The code shows that every stored pixel inside the array can be accessed by incrementing the *"For Loop"* counter in increments of 3 and the RBG components can be accessed individually by modifying the array index by either 0, 1 or 2. The array shows that the RGB components which are arranged as BGR instead of RGB.

```
for (int counter = 0; counter < rgbModValues.Length; counter += 3)
{
    CalcHueSat(rgbModValues[counter + 2], rgbModValues[counter + 1], rgbModValues[counter], ref hue, ref sat);
```

At the same time as extracting the pixels from the array, the Hue and Saturation values are obtained for each pixel where the ***CalcHueSat()*** performs the action that converts them into the corresponding values. The breakdown of the ***CalcHueSat()*** function which performs the

HSV transformation is shown in the following code snippet below. The way that this function works is that the RGB components are passed to the function as parameters the following formulas determine where on the HSV colour wheel or the chart shown in the values of the Hue are positioned on. The Hue values can lie anywhere from 0 to 360°. For the saturation value this is now expressed as an 8 bit value between 0 and 255, instead of a number lying between 0 and 1, in fractional increments. The saturation component determines the level or colour density of a particular colour as described.

```
private void CalcHueSat(int red, int green, int blue, ref int hueVal, ref int satVal)
{
    hueVal = 0;
    satVal = 0;
    if (blue >= green)
    {
        if (red >= blue)
        {
            // maxVal = red; // 300-360 R>B>G
            // minVal = green;
            if (red != green)
            {
                hueVal = 360 - 60 * (blue - green) / (red - green);
                satVal = 255 * (red - green) / red;
            }
        }
        else // blue > red
        {
            // maxVal = blue; // 180-300
            if (red >= green)
            {
                // minVal = green; // 240-300 B>R>G
                if (blue != green)
                {
                    hueVal = 240 + 60 * (red - green) / (blue - green);
                    satVal = 255 * (blue - green) / blue;
                }
            }
            else
            {
                // minVal = red; // 180-240 B>G>R
                if (blue != red)
                {
                    hueVal = 240 - 60 * (green - red) / (blue - red);
                    satVal = 255 * (blue - red) / blue;
                }
            }
        }
```

For example applying the formulas into practice for the following conditions for the RGB are as described below:-

$$Red = 255, Blue = 128, green = 150$$

$$hueVal = 360 + 60 * \frac{(blue - green)}{(red - green)} = 360 + 60 * \frac{(128 - 150)}{(255 - 150)} = 360 - 12.6 = 347$$

$$satVal = 255 * \frac{(128 - 150)}{(128)} = 44$$

As calculated for the following formulas the hue is 347° and the saturation is 44, which suggests that the colour lies somewhere in the red range.

Referring back to the structure diagram in ***Figure 27*** in the block that says ***"Calculate pixel coordinates on the field"*** this allows the pixel count readings that were recorded previously to be converted into useful information that can be used to determine the location of the four corners of the field as well as the robot and the position of the ball in the image. In order to do this and referring back to the coding snippet mentioned previously, it shows that the variable name ***red_pixel*** stores the number of red pixels that were detected and as well as recording the total number of X and Y pixels which are stored in the corresponding variables called *redXtotal* and *redYtotal*.

```
if ((hue > redHueValMax || hue < redHueValMin) && (sat > redSatValueA))      //red red pixel range
{
    red_pixel++;
    redXtotal += pixel_x;
    redYtotal += pixel_y;
}
else if ((hue > blueHueValMin && hue < blueHueValMax) && (sat > blueSatValueA))  // dark blue pixel range
{
    blue_pixel++;
    blueXtotal += pixel_x;
    blueYtotal += pixel_y;
}
```

The code snippet below which is also located inside the "*For Loop"* records the number of times an X pixel is detected after a particular colour has been determined. The number of X values will accumulate until it exceeds 640 pixels, because this is the maximum number of Y pixels that the Bitmap image can hold. After exceeding 640 pixels the number of X pixels is cleared and then then the Y value of the image is incremented.

```
        pixel_x++;
        if (pixel_x >= 640)
        {
            pixel_x = 0;
            pixel_y++;
        }
    }
```

In a nutshell the code is basically sweeping through all the X pixels it finds on each line of the image and then when it exceeds 640 it increments over to the next line and repeats the same procedure as stated in the image in **Figure 29.** As the height of the Bitmap is 480, this means that the Y value can only reach this amount or once the image has been processed

on all 480 lines of Y for every 640 X pixel numbers detected.

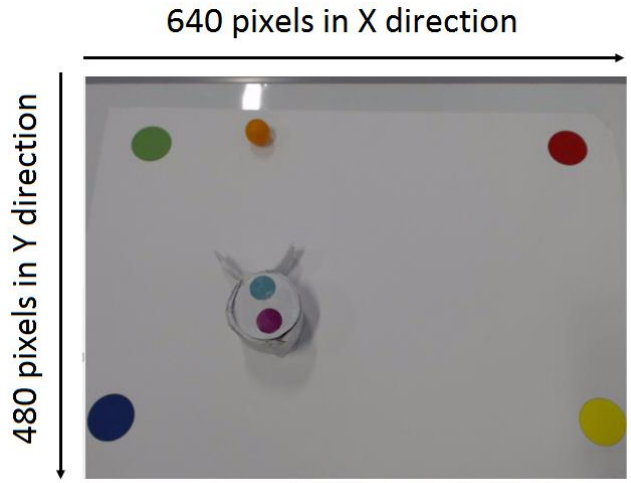


Figure 29

Once the X and Y pixel totals as well as the pixel counts for each colour were determined these readings could be used and converted into (X,Y) pixel coordinates. To convert them into X and Y coordinates, the following coding snippet below describes this.

```
//calculate pixel coordinates
int xposRed = -1, yposRed = -1;
if (red_pixel > 0)
{
    xposRed = redXtotal / red_pixel;
    yposRed = redYtotal / red_pixel;
}
int xposBlue = -1, yposBlue = -1;
if (blue_pixel > 0)
{
    xposBlue = blueXtotal / blue_pixel;
    yposBlue = blueYtotal / blue_pixel;
}
int xposGreen = -1, yposGreen = -1;
if (green_pixel > 0)
{
    xposGreen = greenXtotal / green_pixel;
    yposGreen = greenYtotal / green_pixel;
}
int xposYellow = -1, yposYellow = -1;
if (yellow_pixel > 0)
{
    xposYellow = yellowXtotal / yellow_pixel;
    yposYellow = yellowYtotal / yellow_pixel;
}
int xposOrange = -1, yposOrange = -1;           //for the orange ball
if (orange_pixel > 0)
{
    xposOrange = orangeXtotal / orange_pixel;
    yposOrange = orangeYtotal / orange_pixel;
}
int xposMagenta = -1, yposMagenta = -1;        //for the magenta circle on robot
if (magenta_pixel > 0)
{
    xposMagenta = magentaXtotal / magenta_pixel;
    yposMagenta = magentaYtotal / magenta_pixel;
}
```

Once the X and Y pixel coordinates have been determined they are placed inside the position structures this is also indicated in the structure diagram in ***Figure 27*** as ***"Place X,Y and number of pixels into variable structures"***. For which these will be used for determining the pixel coordinates in MM(Millimetres).

```
//load variables into the structures
redPosPixel.n = red_pixel;
redPosPixel.x = xposRed;
redPosPixel.y = yposRed;

bluePosPixel.n = blue_pixel;
bluePosPixel.x = xposBlue;
bluePosPixel.y = yposBlue;

greenPosPixel.n = green_pixel;
greenPosPixel.x = xposGreen;
greenPosPixel.y = yposGreen;

yellowPosPixel.n = yellow_pixel;
yellowPosPixel.x = xposYellow;
yellowPosPixel.y = yposYellow;

orangePosPixel.n = orange_pixel;
orangePosPixel.x = xposOrange;
orangePosPixel.y = yposOrange;

magentaPosPixel.n = magenta_pixel;
magentaPosPixel.x = xposMagenta;
magentaPosPixel.y = yposMagenta;

cyanPosPixel.n = cyan_pixel;
cyanPosPixel.x = xposCyan;
cyanPosPixel.y = yposCyan;
```

3.1.1.3.2 Calculate pixels in Millimetres function()

After knowing the pixel coordinates in terms of X and Y values, these can then be converted into Millimetres which will be used for determining the location of the robot. Applying the theory that was mentioned previously, the following segment of code explains the millimetres conversion function.

```
//this function calculates the X,Y coordinates of the colours on the field in terms of Millimetres(MM)
//private void calcPosMM(int xposRed, int yposRed, int xposBlue, int yposBlue, int xposGreen, int yposGreen, int xposYellow, int yposYellow, int xposCalc, int yposCalc)
private void calcPosMM()
{
    //validPos = true;
    int L = lengthValue;                                //length and width of the field set by global variables in calibration mode
    int W = widthValue;
    int[] a = new int[4];
    int[] b = new int[4];

    a[0] = bluePosPixel.x;                                      //calculate formulas for the four colours
    a[1] = yellowPosPixel.x - bluePosPixel.x;
    a[2] = greenPosPixel.x - bluePosPixel.x;
    a[3] = redPosPixel.x + bluePosPixel.x - greenPosPixel.x - yellowPosPixel.x;
    b[0] = bluePosPixel.y;
    b[1] = yellowPosPixel.y - bluePosPixel.y;
    b[2] = greenPosPixel.y - bluePosPixel.y;
    b[3] = redPosPixel.y + bluePosPixel.y - greenPosPixel.y - yellowPosPixel.y;
    long[] q = new long[3];                //calculate the ball's position in the field
    q[0] = a[1] * b[3] - a[3] * b[1];       //this value will be the same for the other colours

    q[1] = a[1] * b[2] - a[2] * b[1] + a[0] * b[3] - a[3] * b[0] + orangePosPixel.y * a[3] - orangePosPixel.x * b[3];
    q[2] = a[0] * b[2] - a[2] * b[0] + orangePosPixel.y * a[2] - orangePosPixel.x * b[2];


    double ballx = -1000; // in mm
    double bally = -1000;
    long D;
    if (q[0] != 0)
    {
        D = q[1] * q[1] - 4 * q[0] * q[2];
        if (D >= 0)
        {
            if (q[1] >= 0)
            {
                ballx = (-q[1] + Math.Sqrt(D)) / (2 * q[0]);
            }
            else
            {
                ballx = (-q[1] - Math.Sqrt(D)) / (2 * q[0]);
            }
        }
    }
```

The results of the millimetres code and applying the theory that was mentioned in 1.1.3 which states that the proportion of the image from the top to the bottom moves at a proportional from 0 to 1 Height(Y direction) and the same happens for the proportion of the field in the (X direction) from 0 to 1. The L and the W values are the sizes of the field that were specified by the GUI form, so instead of the proportion in millimetre being represented as a fractional number on a scale from 0 to 1, the proportion is now represented as a number from (0 to 685) Height and from (0 to 434) Width.

```
if (ballx != -1000 && a[2] + a[3] * ballx != 0)
{
    bally = (orangePosPixel.x - a[0] - a[1] * ballx) / (a[2] + a[3] * ballx);
    ballx = L * ballx;
    bally = W * bally;
    //load the structures with the calculate coordinates in MM
    orangePosMM.x = ballx;
    orangePosMM.y = bally;
}
```

3.1.1.3.3 Sending "POS" messages out to the serial port

After receiving the pixel positions from the following image the next step was to convert the X and Y coordinates into millimetres. Before converting the pixels into millimetres, a pixel check was necessary to determine if there are enough pixels in the image to carry out the calculations, where the "***.n***" or pixel count number at the end of each structure contains the number that stores the amount of pixels in the structure. If as sensible number of pixels have been found i.e. more than 700 pixels for red, blue, green and yellow, 300 for orange and 200 for cyan and magenta, then the ***calcPosMM()*** is called in the next stage.

```
//check for a sensible amount of pixels to do the calculations
if (redPosPixel.n > 700 && bluePosPixel.n > 700 && greenPosPixel.n > 700 && yellowPosPixel.n > 700 && orangePosPixel.n > 300 && magentaPosPixel.n > 200 && cyanPosPixel.n > 200)
{
    //calculate X,Y position of ball
    calcPosMM();    //calculate the pixel values into MM which are loaded into structures
```

The ***calcPosMM()*** uses an algorithm which converts the X,Y pixel values into actual coordinates in terms of millimetres, the result of the function places the millimetre readings into structures for the cyan and magenta for the robot and orange for the ball.

```
//load the structures with the calculate coordinates in MM
orangePosMM.x = ballx;
orangePosMM.y = bally;
//load the structures with the calculate coordinates in MM
cyanPosMM.x = cyanx;
cyanPosMM.y = cyany;
//load the structures with the calculate coordinates in MM
magentaPosMM.x = magentax;
magentaPosMM.y = magentay;
```

Soon after the millimetre readings are placed in the structures a series of calculations are performed. The following section of code below shows the X and Y value for the orange ball being converted into a string that will later on be used for sending out to the serial port.

```
//Convert BallX and BallY into a string
ballX = Convert.ToString(Convert.ToInt64(orangePosMM.x));
ballY = Convert.ToString(Convert.ToInt64(orangePosMM.y));
```

The next step is to calculate the robot X and Y positions as shown in the following section of code below. This is done by calculating the average values of the two colours mounted on the top of the robot (Cyan and Magenta) and this will determine the centre point position as indicated in the sketch in ***Figure 30***.

```
//Calculate robot position
double robotAvrX = (cyanPosMM.x + magentaPosMM.x) / 2;
double robotAvrY = (cyanPosMM.y + magentaPosMM.y) / 2;
robotX = Convert.ToString(Convert.ToInt64(robotAvrX));
robotY = Convert.ToString(Convert.ToInt64(robotAvrY));
```

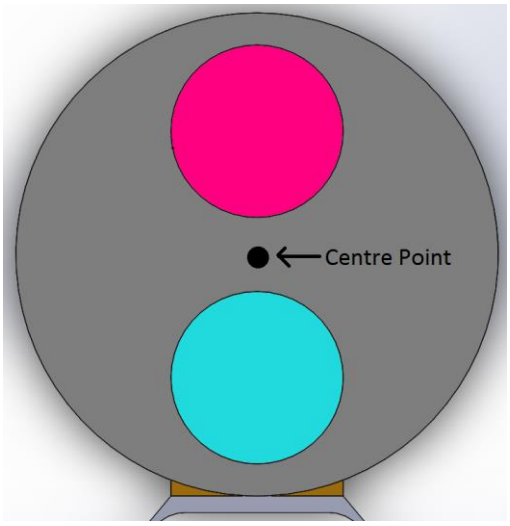


Figure 30

The main reasons why the 2 colours were mounted on the top of the robot is because it was important to know which direction the robot was pointing in so a heading angle could be determined. The sketch of the robot in ***Figure 31*** shows that the 2 colours on the robot form a vector which produces an angle as the robot turns and therefore this makes it possible to determine the heading angle of the robot, and hence the direction the robot is pointing in. Also noticed that cyan was selected as the front position of the robot.

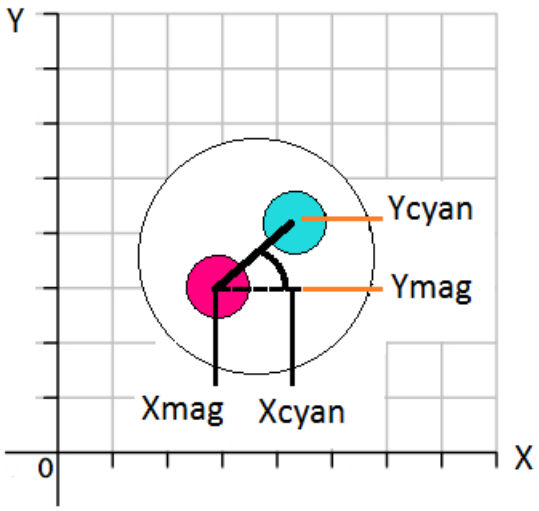


Figure 31

The code that calculates the robot heading is shown below. The ***Atan2*** function was used because it returns an angle by supplying it with the difference between the upper and lower readings for both the X and Y values. After determining the heading angle the robot heading is converted into a string and placed inside the ***robotHead*** variable.

```
//Calculate the heading of the robot
double robotHeadX = cyanPosMM.x - magentaPosMM.x;
double robotHeadY = cyanPosMM.y - magentaPosMM.y;
double robotHeadRad = Math.Atan2(robotHeadY, robotHeadX);  //calculate robot heading in radians
string robotHead = Convert.ToString(Convert.ToDouble(robotHeadRad));     //send readings in radians
```

The final step that is performed is that the target position coordinates for the X and Y values are converted into a string, and these will be later on combined into a single string which will form part of the “POS” message. The following coding snippet below shows the target readings from the GUI being converted into a string.

```
string tgtXVal = Convert.ToString(Convert.ToInt64(targetX));
string tgtYVal = Convert.ToString(Convert.ToInt64(targetY));
```

The ***targetX*** and ***targetY*** variables that get assigned with the values that are typed into the number box and this occurs after the ***“Set Target”*** button is pressed as indicated in the GUI

in ***Figure 32***. This will later on specify the target set point or destination that the robot on the field will push the ball to.

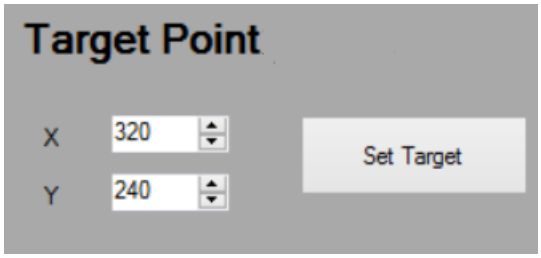


Figure 32

Now the final step is to ***"Combine readings with "POS" string"*** as indicated in the diagram in ***Figure 24.*** The following segment of code which describes this is shown below.

```
robotData = "POS" + " " + ballX + ", " + ballY + ", " + robotX + ", " + robotY + ", " + robotHead + ", " + tgtXVal + ", " + tgtYVal + ", ";
```

The final stage of the block diagram, sends the POS string out to the robot. The rate at which strings are sent to the robot is limited by how fast the Web Cam event handler receives images. As an image happens approximately 30ms and this means that a position string is being sent to the robot every 30ms. Before sending the ***robotData*** string, a variable called ***msgCount*** whose value increments as every string is being sent out to the serial port, records the number of times that the data is being sent out to the serial port. The variable was used for debugging purposes to enable the user to fault find and debug inconsistencies with the data being calculated and sent from the PC and compared it with the data that has been received from the robot. The count values can be checked and compared with each other. The strings are displayed in the textbox of the GUI before being transmitted out to the robot as well as the image data that was recorded from when the pixel values were called are also displayed. The code that displays the readings sent to the robot from the GUI is done by writing the string to robotSentDataBox.text. The imageDataBox.Text prints or displays the pixel position or count values of the readings.

```
if (openToolStripMenuItem2.Enabled == false && serialPort1.IsOpen && startRobotButt.Enabled == false)    //check if serial port has been oper
{
    msgCount++;
    string msgStr = Convert.ToString(msgCount);
    string robotDataStr = robotData + msgStr + "\r\n";
    robotSentDataBox.Text += robotDataStr;

    imageDataBox.Text += msgCount + ", " + redPosPixel.n + ", " + redPosPixel.x + ", " + redPosPixel.y
                + ", " + bluePosPixel.n + ", " + bluePosPixel.x + ", " + bluePosPixel.y
                + ", " + greenPosPixel.n + ", " + greenPosPixel.x + ", " + greenPosPixel.y
                + ", " + yellowPosPixel.n + ", " + yellowPosPixel.x + ", " + yellowPosPixel.y
                + ", " + orangePosPixel.n + ", " + orangePosPixel.x + ", " + orangePosPixel.y
                + ", " + magentaPosPixel.n + ", " + magentaPosPixel.x + ", " + magentaPosPixel.y
                + ", " + cyanPosPixel.n + ", " + cyanPosPixel.x + ", " + cyanPosPixel.y + "\r\n";

    sendToSerial(robotDataStr);   //send data to robot
}
```

Once this has been done the final part is to call the ***sendToSerial()*** function that sends the POS string out to the serial port.

3.1.1.3.4 Sending "STOP" messages out to the serial port

There is also another case when the pixels in the image are insufficient, for example when sudden changes in lighting conditions occur or if the ball or robot are no longer on the field, then a ***"STOP"*** message would need to be sent to the robot to make sure that the motors on the robot are turned off and the robot does not start to spinning round or moving off in a random direction. Instead of the performing the calculations that calculate the positon coordinates which form part of calculating the ***"POS"*** string, the following coding segment below is performed instead. The same procured for sending the POS string out the serial port happens with a "STOP" message, where the ***msgCount*** is added on to the end of the string and then sent out to the serial port.

```
else
{
    textBox1.Text = "CALIBRATION NEEDED!";
    robotData = null;
    robotData = "STOP "; //construct a stop string to send to the robot!
}
```

3.1.2 Calibration

The Hue and Saturation values now have a useful application for extracting a particular colour of interest in an image or frame where "*thres-holding"* can be applied. The theory that was mentioned in section 1.2.2 can be applied to this problem. The following code below shows that an upper and lower limit for a Hue value and a saturation value can be set and changed when required. Each time a colour is detected and falls within a certain Hue range for a particular saturation value the number of the pixel count changes until the entire image

has been processed.

```
for (int counter = 0; counter < rgbValues.Length; counter += 3)
{
    CalcHueSat(rgbValues[counter + 2], rgbValues[counter + 1], rgbValues[counter], ref hue, ref sat);

    if ((hue > redHueValMax || hue < redHueValMin) && (sat > redSatValueA))      //red red pixel range
    {
        red_pixel++;
        redXtotal += pixel_x;
        redYtotal += pixel_y;
    }
    else if ((hue > blueHueValMin && hue < blueHueValMax) && (sat > blueSatValueA))  // dark blue pixel range
    {
        blue_pixel++;
        blueXtotal += pixel_x;
        blueYtotal += pixel_y;
    }
    else if ((hue > greenHueValmin && hue < greenHueValMax) && (sat > greenSatValueA))   //dark green pixel range
    {
        green_pixel++;
        greenXtotal += pixel_x;
        greenYtotal += pixel_y;
    }
    else if ((hue > yellowHueValMin && hue < yellowHueValMax) && (sat > yellowSatValueA))   //yellow pixel range
    {
        yellow_pixel++;
        yellowXtotal += pixel_x;
        yellowYtotal += pixel_y;
    }
    else if ((hue > orangeHueValMin && hue < orangeHueValMax) && (sat > orangeSatValueA)) //orange pixel range
    {
        orange_pixel++;
        orangeXtotal += pixel_x;
        orangeYtotal += pixel_y;
    }
```

The limits of calibration can be changed, by entering the following information in the software ***GUI*** on the PC as shown ***Figure 33.*** The numbers for the limits are changed by the changing the numbers in the numeric up/down boxes and these get stored as global variables inside the software program. The values entered enable a certain area or colour in the image to be filtered or sampled. For example referring back to the code snippet above the **blueHueValMax** stores the upper limit for detecting blue and **blueHueValMin** store the lower limit. The sampling limits for the **redHueValMin** and **redHueValMin** is slightly more complicated because part of the red hue range includes values from 0 to 20 as well as the values from 300 to 360, where values past 360 are wrapped around to 0. Referring back to the image in ***Figure 33***, the picture box to the left above the “LIVE IMAGE” text shows the current live image being received and displayed directly from the webcam and the “MASKED IMAGE” picture box to the right shows the image after thresholding. An ideal image would have all the area and colours of interest shown such as the *“Orange Ball”,* the four colours for the corners of the field and the two colours on the robot. The colours outside the Hue and Saturation checking range are ignored and are shown in black instead.

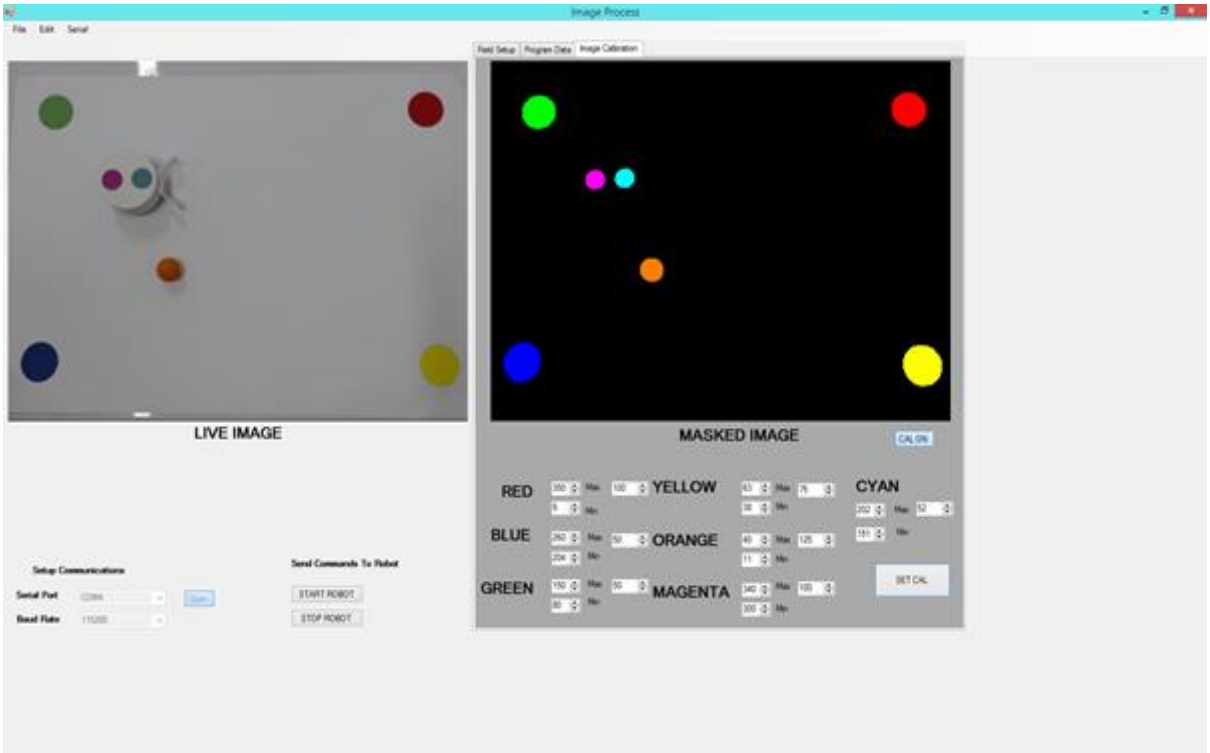


Figure 33

The colours were chosen in a way that ensures that the range of Hue values do not overlap so the following values were worked out based on their positioning on the HSV colour wheel and the values are taken as shown below:-

Red Min: 347	Blue Max: 260	Green Max: 150	Yellow Max: 59
Red Max: 6	Blue Min: 210	Green Min: 80	Yellow Min: 42
Orange Max: 40	Cyan Max: 207	Magenta Max: 340	
Orange Min: 16	Cyan Min: 173	Magenta Min: 300	

The values gives an ideal threshold starting point for calibrating or thresholding the image and the values can be changed by modifying the Up/Down boxes in the GUI form shown in ***Figure 33*** depending on the lighting conditions. It was discovered that the lighting conditions of the surrounding environment would throughout the day and therefore it was necessary to have the following calibration and threshold image built into the user interface.

3.1.3 MASKED IMAGE IN GUI FORM

The way the way that the ***"MASKED IMAGE"*** was displayed in the GUI form window uses a very similar method and algorithm for extracting the desired pixel colours from the bitmap

image provided. It uses the same limits of values for the Hue and Saturation values which are entered into the GUI boxes, that the ***CalcPixelPosition()*** function uses to extract the pixels for determining the pixel coordinates of the colours. If we refer back to the block diagram in ***Figure 24*** which mentions the ***"Do calibration"*** function block, it shows that (AForge.NET, 2013)Cam event handler are sent to "Do Calibration" function instead of being sent to the ***"CalcPixelPosition()"*** function. The technique for processing and storing the Bitmaps in an array are identical except for the fact the RGB array that stores the pixels is called rgbModValues instead of rgbValues. Also as each pixel is processed the RGB components of the pixels are modified as shown in the following code below which colours the pixels that fall within the red hue and sat range into red pixels with a single discrete colour. The same thing happens for the other 6 colours.

```
for (int counter = 0; counter < rgbModValues.Length; counter += 3)
{
    CalcHueSat(rgbModValues[counter + 2], rgbModValues[counter + 1], rgbModValues[counter], ref hue, ref sat);

    if ((hue > redHueValMax || hue < redHueValMin) && (sat > redSatValueA))      //red red pixel range
    {
        red_pixel++;
        rgbModValues[counter + 2] = 255;
        rgbModValues[counter + 1] = 0;
        rgbModValues[counter] = 0;
    }
```

The colours that are not specified within the Hue and saturation limits which are stated in the numeric up and down boxes are turned to black. The following coding segment that turns the pixels black is shown below.

```
else
{
    rgbModValues[counter + 2] = 0;
    rgbModValues[counter + 1] = 0;
    rgbModValues[counter] = 0;
}
```

After processing the image data stored in memory inside the the rgbModValues array, it is then converted back into a Bitmap and unlocked and then the modified Bitmap is returned to where the function was last called from be from the inside of the 1ms_Image_Timer().

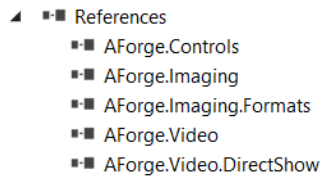
The code that displays the modified Bitmap that was returned from the "Do Calibration" function or as stated in the code "CalTimerImageFunction()" will now display the returned Bitmap masked image in the right side of the picture box.

```
unsafe
{
    //get address of the first line
    ptr2 = bData2.Scan0;
    System.Runtime.InteropServices.Marshal.Copy(rgbModValues, 0, ptr2, bytes);
}
calBMP.UnlockBits(bData2);            //unlock the image
return calBMP;
```

```
if (calButt.Checked == true)            //check if calibration button has been enabled
{
    bmpCal = myImageCapt.CopyBMP(bmp);   //copy frame
    imageMaskBox.Image = CalTimerImageFunction(bmpCal);      //display masked image
}
```

3.1.4 3rd Party Software Libraries(AForge.Net framework)

The following ImageCapture Class Block incorporates methods or functions which control the Web Cam from classes within the ***AForge.NET*** framework. The ***AForge.NET*** are 3rd party libraries which make it possible to have access and control to many features of the Logitech c920 Web Cam. When the software program was first developed these had to be imported into C# Visual Studio as a references as shown below.



The lines of code that allows access to the directives of the AForge library methods is shown below. The most import library feature is the AForge.Video, which contains methods that have access to the hardware features of the Web Cam device.

```
//create using directives for easier access of AForge library's methods
using AForge.Video;
using AForge.Video.DirectShow;
using AForge.Imaging;
```

The following coding snippet below declares the Web Cam class object, ***videoSource*** as a VideoCaptureDevice. The videoSource object represents the Logitech Web Cam, and the ***VideoCaptureDevice*** makes it possible to setup up the camera, such as turning it on and specifying the format and resolution of the frames to be produced by the web Cam.

```
VideoCaptureDevice videoSource;
```

3.2 Software on the Robot

As well as having the PC software that processes the frames that it receives from a web cam and then sends them to the robot, it was also important for the robot to have its own software that enables it to perform calculations such as calculating the ball heading relative to the robot heading, calculating the distance relative to the ball, etc and then locating the ball on the field and then pushing it to the required target point. The robot received the readings from the PC software at regular invervals.

The diagram in ***Figure 34*** shows a higher hierarchal level of how the following robot software operates.

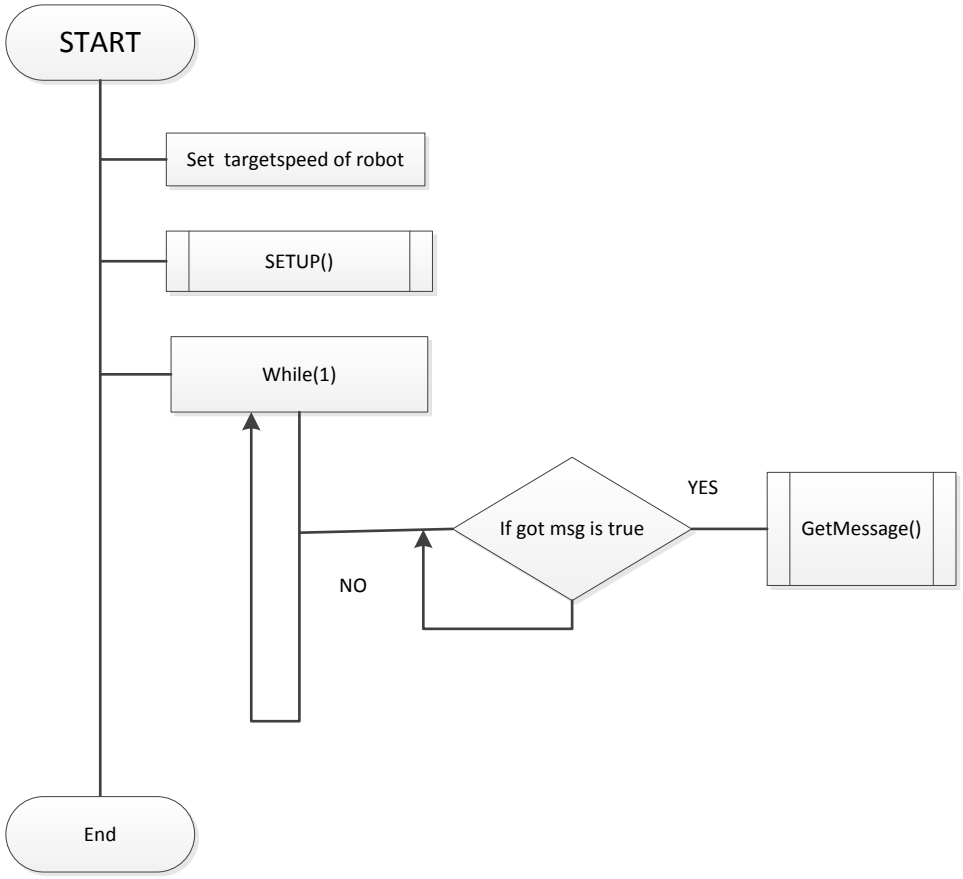


Figure 34

Each of these blocks will be explained in further detail in the following section.

3.2.1 GetMessage() Function

The following function consists of the following components as shown in ***Figure 35***.

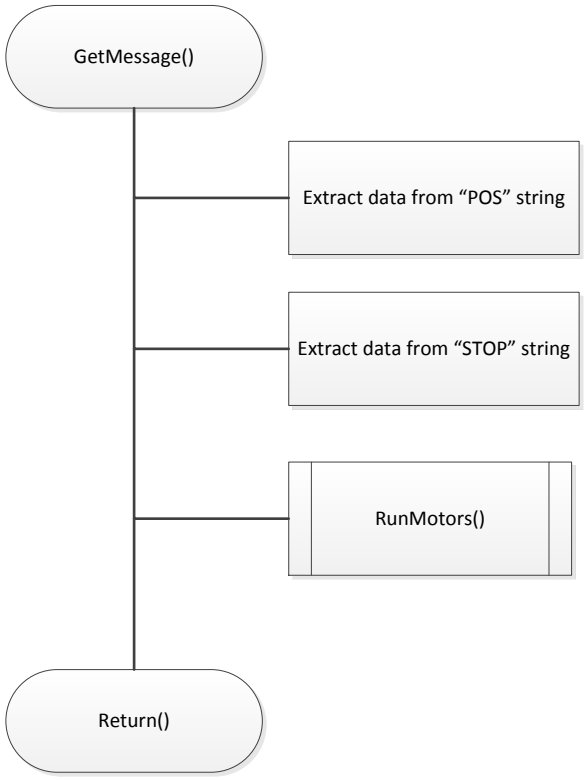
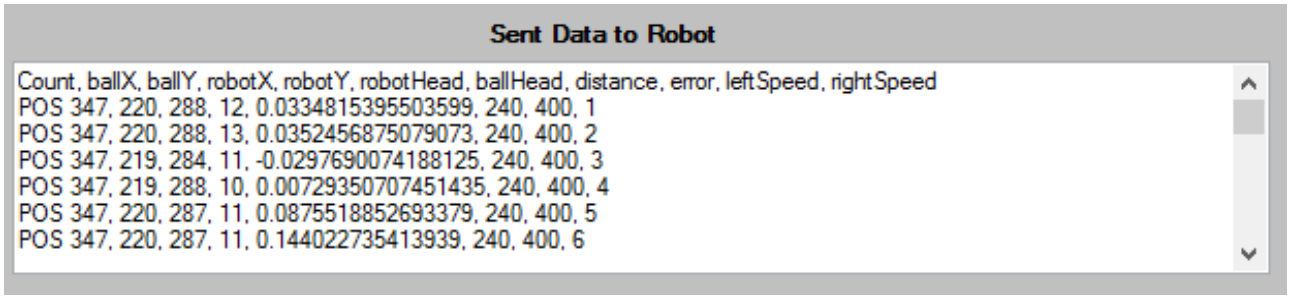


Figure 35

The block in the diagram that mentions ***“Extract data from “POS”*** string is responsible for extracting the data stored inside the POS string that the software program on the PC sends to the robot, the following code that describes this is shown below. The data or strings that the PC sends to the robot gets placed it inside the global ***arrayBuff*** variable and the strncmp() function checks to see if the string message is a ***“POS”*** message.

```
if(strncmp((char*)arrayBuff, "POS",3)== 0)
{
  //extract the position values
  n= sscanf((char*)arrayBuff + 3, "%f,%f,%f,%f,%f,%f,%f,%i",&f1, &f2, &f3, &f4, &f5, &f6, &f7, &i1);
  if (n == 8)    //got all the numbers
  {
    pos->ballX = f1;    //load variables into structures
    pos->ballY = f2;
    pos->robX = f3;
    pos->robY = f4;
    pos->robHdg = f5;
    tgt->X = f6;
    tgt->Y = f7;
    pos->count = i1;

    //*****************************************************************************
    //*       perform calculations on the numbers and place into position structure
    //*****************************************************************************
    //set position flag and string
    pos->newMsg= POS_MSG; //this will change the flag to position message
    pos->ballHdg = atan2((pos->ballY - pos->robY),(pos->ballX - pos->robX)); //calculate ball heading relative to the robot heading

  }
}
```

If the message is a string message then the ***n= scanf()*** checks to see if the string contains the following numbers in the specified string format. In the following example it is checking to see if the string contains 8 numbers. An example of what the “POS” string would look like sent from the PC is shown in the GUI form in the following image. The software Program on the PC always shows the status or numbers of each calculated POS message that it sends out to the robot. The reason for doing this is for enabling consistency checking between the data sent to the robot from the PC and the data sent back to the PC from the robot. The count value which is also shown in the textbox is incrementing as each string is sent, this is also for enabling error checking of the readings.

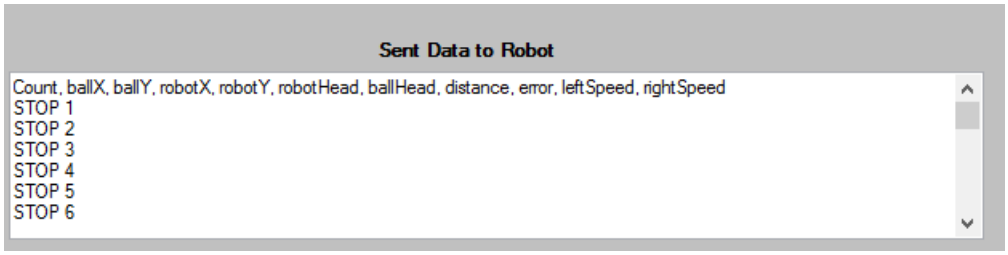


We can also see that the following box above contains the 8 numbers that the sscanf() function is checking for. Once this is successful the numbers are then extracted from the string as the code suggests and are then placed in their corresponding structures. Also notice in the ***“Sent Data to Robot”*** textbox that the “POS” string message also contains the target values that were defined by setting the target coordinates in the boxes which are denoted as (240, 400). The other important thing that the code does is set the pos->newMsg to ***POS_MSG*** which basically specifies the type of message that has been received from the serial port. The last thing that happens is that the ball heading (ballHdg) which is the angle relative to the ball (X,Y) position and the robot (X,Y) position is calculated.

The block that mentions ***"Extract data from "STOP"*** string a very similar thing occurs, except that the string only contains the count number which is used for error checking. This string is sent to the robot if the colours in the image are missing or there are insufficient pixels available for doing the calculations. Also what happens is that the pos->newMsg is set to "***STOP_MSG"*** to signify a stop message.

```
else if(strncmp((char*)arrayBuff, "STOP",4)==0)
{
  n= sscanf((char*)arrayBuff + 4, "%i", &i1);
  if(n == 1)
  {
    pos->count = i1;
    pos->newMsg= STOP_MSG;
  }

}
```

As you can see from the following textbox below a ***"STOP"*** string message along with the count value was sent to the robot, which gets extracted by the coding segment above. The cause of this message was due to the fact that the orange ball had rolled off the field.



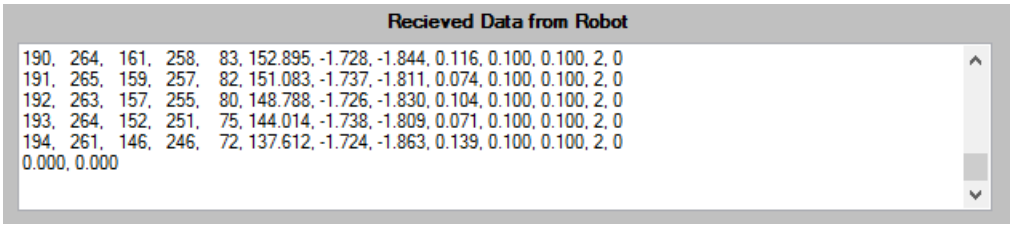
3.2.2 RunMotors() function

The ***RunMotors()*** function contains the state machine which decides the type of state for the robot to move to. The following function is contained inside the **GetMessage()** function, which previously extracted the numbers from the ***arrayBuff*** and then it placed enums for each of the following types of messages for STOP and POS which correspond to STOP_MSG and POS_MSG respectively into the following structures. ***Figure 36*** shows the state diagram mentioned. The state diagram only performs the following states when a POS_MSG is received.

In the case of receiving a STOP_MSG, then the following code below is executed, where both the motor speeds are turned off and the ***robotState*** variable is set to the STOP state, this was to ensure that when a POS_MSG is next received the robot initially goes to the stop state again.

```
case STOP_MSG :
    SL = 0;
    SR = 0;
    LeftMotor(SL);      //send speed values to the robot motors
    RightMotor(SR);
    sprintf(msg, "%5.3f, %5.3f\r\n", SL, SR);
    len = strlen(msg);  //get length of message
    OutputString(msg, len);     //send message to PC
    robotState = STOP;
break;
```

The string that the robot sends back to the PC after receiving a STOP_MSG, which sends the status of the motor speeds to indicate that the robot has stopped. The STOP_MSG can be sent to the robot by either pressing the “STOP ROBOT” button on the GUI or sent automatically by the software program on the PC when there is insufficient image data available for the robot to do perform any calculations.



By default the robot state will be placed inside the STOP state upon entering the STOP state for the RunMotors() function for the first time on start-up.

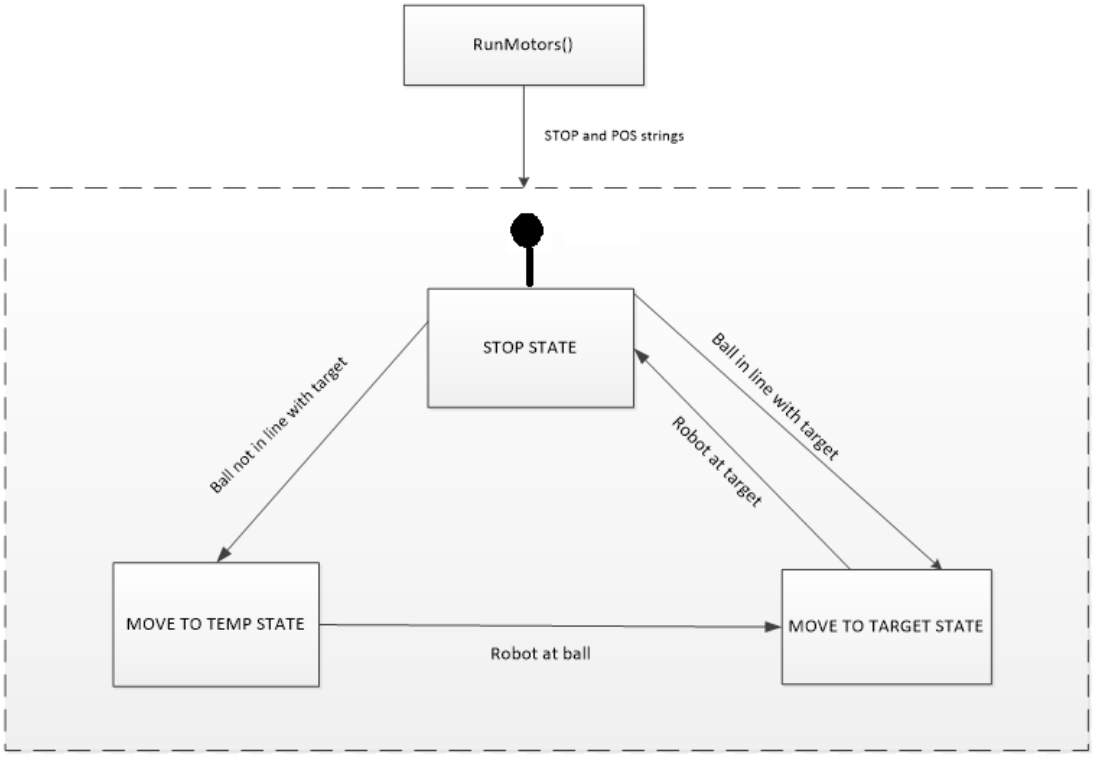
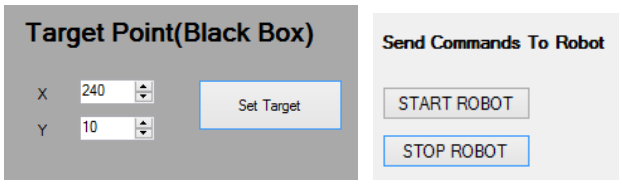


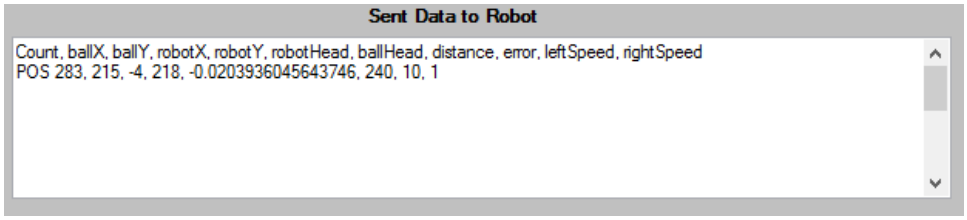
Figure 36

3.2.2.1 STOP STATE

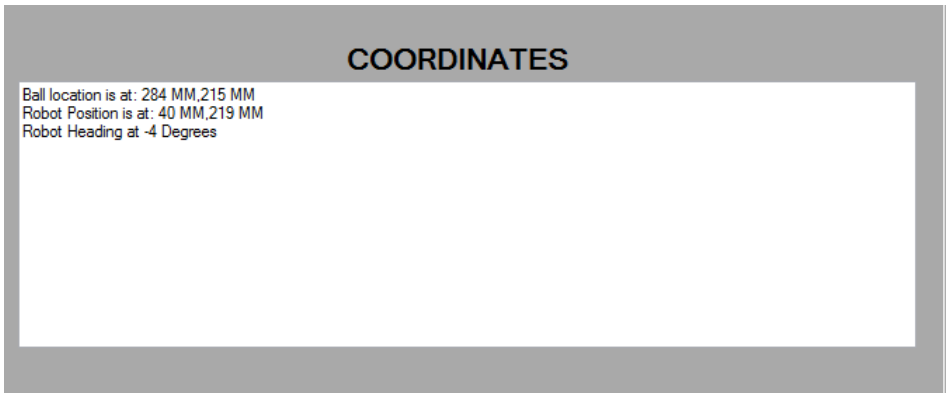
In this state the robot is not moving and it needs to decide whether to go for the target or go to a temporary point that is in line with both the target and the ball. The photo in ***Figure 37*****Error! Reference source not found.** shows the robot and the orange ball on the field. The letter ***"X"*** represents the target that the robot needs to push the ball to. The target position of the robot is defined by the Software GUI where the following values can be entered into the following coordinate's box and then pressing the ***"Set Target"*** button to confirm the changes. Here is an example with the robot target set to (240, 10). Pressing the ***"START ROBOT"*** button initiates the sending and transmission of the calculated readings from the PC to the robot. When there is sufficient image data available to do the calculations the software program on the PC will send Position strings



("POS") as shown in the ***"Sent Data to Robot"*** textbox.



The coordinates Box below on the PC software GUI shows that the calculated heading of the robot tested for the case in ***Figure 37***, where the robot is shown to be pointing in the -4 degrees direction. It also shows the ball position at (284, 215).



The robot is required to push the ball towards the target point which is indicated by the "X" in the image in ***Figure 37.*** It shows that the angle relative to the robot and the target position are not in line so the robot will be required to move to the ***"MOVE TO TEMP STATE"*** as indicated in the following diagram in ***Figure 36.*** There is an angle of tolerance that is acceptable in the robot software where the angle($\boldsymbol{\theta}$) has to fall within the following range $\mathbf{0.1} < \boldsymbol{\theta} > -\mathbf{0.1}$ rads.

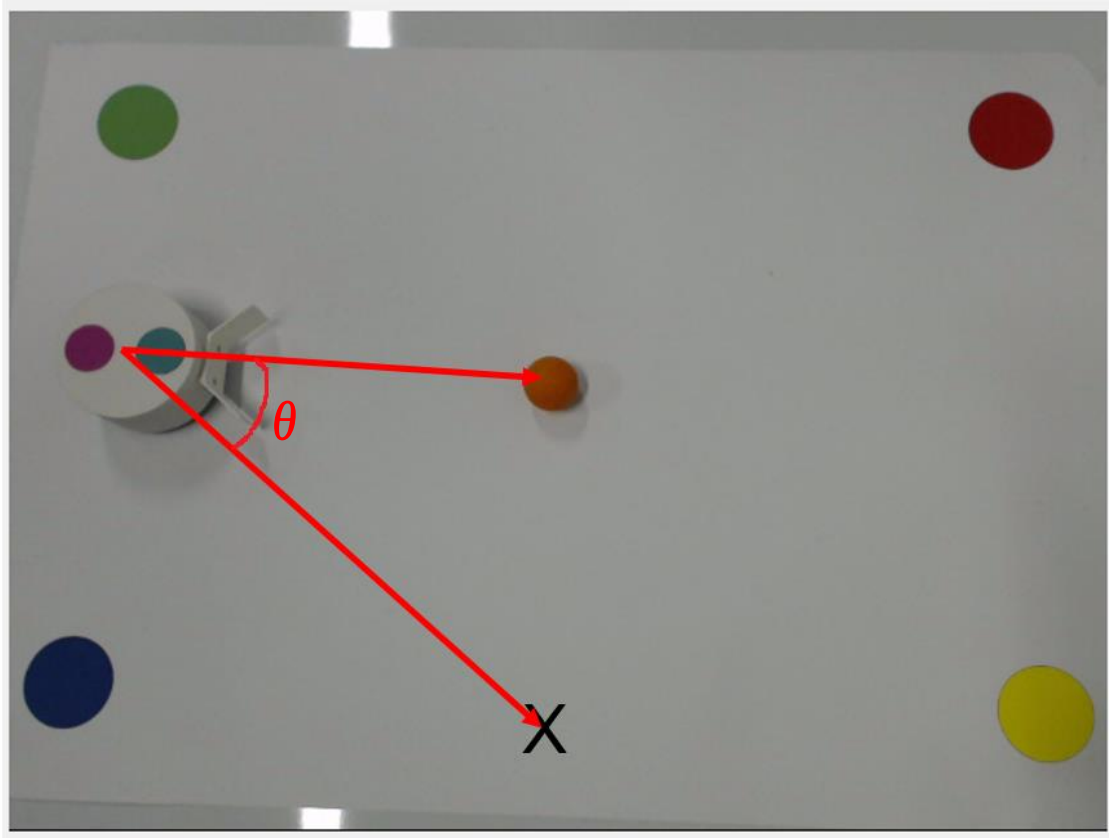


Figure 37

The target heading is calculated as follows:-

$$\text{targetHeading} = \tan^{-1}\left(\frac{targetY - robY}{targetX - robX}\right)$$

The following angle (θ) is calculated as follows.

$$\theta = \text{targetHeading} - \text{ballHeading}$$

After knowing the following information the robot now needs to check the distance between the ball and the target. This is to ensure that the ball is not already at the target point and this is calculated as follows:-

$$\text{ballTgtDistance} = \sqrt{((targetY - ballY)^2 + (targetX - ballX)^2)}$$

After calculating this information the following state transitions inside the STOP state occur. As indicated by the following coding segment.

```
if(ballTgtDist >100)      //check if the ball is within 100mm of the target position point
{
    if(ballTgtError > 0.1 || ballTgtError <-0.1)          //check if angle between TGT and Ball, relative to robot is
    {
      robotState = MOVE_TO_TEMP;              //move robot to temp point as ball is not in line with the target positi
      repeat = 1;
    }
    else //else move to target
    {
      robotState = MOVE_TO_TGT;             //move robot straight to the target
      repeat = 1;
    }

}
```

The ***robotState*** variable by default is set equal to ***STOP*** as shown in the following coding fragment.

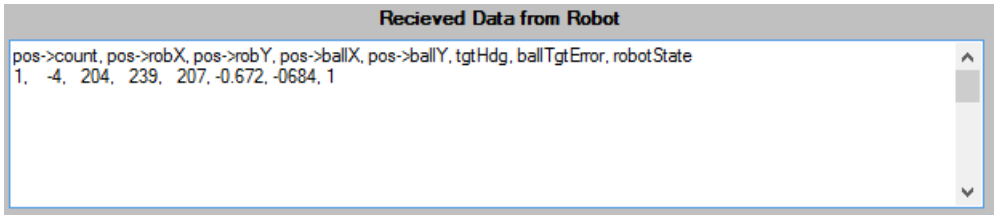
```
static int robotState = STOP;
```

The situation that is presented here will change the ***robotState*** to transition to the ***MOVE_TO_TEMP (“MOVE TO TEMP STATE”*** in the diagram) because the angle is outside the $\mathbf{0.1} < \boldsymbol{\theta} > -\mathbf{0.1}$ rads range. Refferging to the code segment notice that on every transition a variable or flag called ***“repeat”*** is set equal to 1. The reasons for having the repeat variable is that it was important for the robot to change to the next state immediately instead of waiting for the next message or string from the serial port after leaving the “***STOP***” state because the new readings sent to the robot may not be up to date or valid. As the following state machine is contained inside a ***“DO WHILE”*** loop it will ensure that after the repeat has been set to 1 that it will stay inside the loop, then clear the repeat variable and then enter that state after that.

The coding fragment below allows the robot to send status information or stings back to the software program on the PC. The information is basically sending readings that are important to the STOP state which are combined into a string and are then sent out to the serial port using the ***OutputString*** function. As well as sending the string it also sends the length of that string to the ***OutputString*** function.

```
sprintf(msg, "%1i, %5.0f, %5.0f, %5.0f, %5.0f, %5.3f, %5.3f, %1i\r\n", pos->count, pos->robX, pos->robY
len = strlen(msg);          //get length of message
OutputString(msg, len);     //send message to P
```

The numbers in the following box consists of the following readings. It also shows that the ***ballTgtError*** or angle ($\boldsymbol{\theta}$) is -0.684 radians, which falls outside the $\mathbf{0.1} < \boldsymbol{\theta} > -\mathbf{0.1}$ range which suggest that the ball is not in line with target, as stated in ***Figure 37***. Also notice that the ***robotState*** is equal to 1, which is the enum value for the “MOVE_TO_TEMP” state.



3.2.2.2 MOVE TO TEMP STATE

Figure 38 shows a basic indication of how the ***MOVE TO TEMP STATE*** operates.

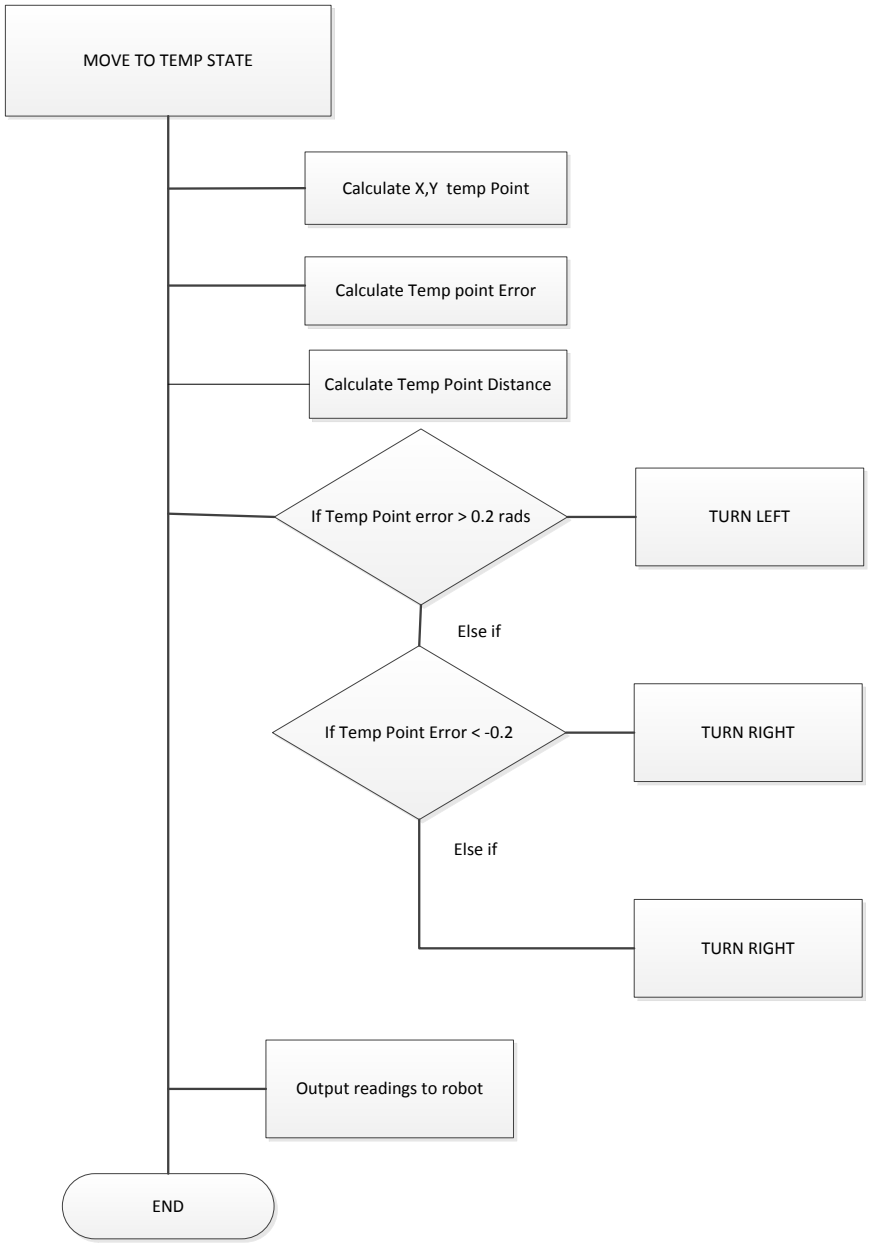


Figure 38

The first thing that happens here is that a temporary point or temporary target on the field is required. The temporary point or target is a point that the robot calculates using trigonometry methods to ensure that it moves to a point that enables it to push the ball directly to the target point if the current position of the robot, the ball and the target are not in line. Referring to ***Figure 39,*** it shows that the angle (φ) is represented as an angle relative to the

temporary point position and the ball position. This angle is also the same when it was taken from the ball position relative to the target position.

The angle φ can be calculated as follows.

$$\tan\varphi = \frac{TempY - BallY}{TempX - BallX}$$

As we are interested in finding out what the solutions are for the temporary points for TempY and TempX , we can come up with the following equations.

$$\sin\varphi = \frac{BallY - TempY}{dRB} \qquad (1)$$

$$\cos\varphi = \frac{BallX - TempX}{dRB} \qquad (2)$$

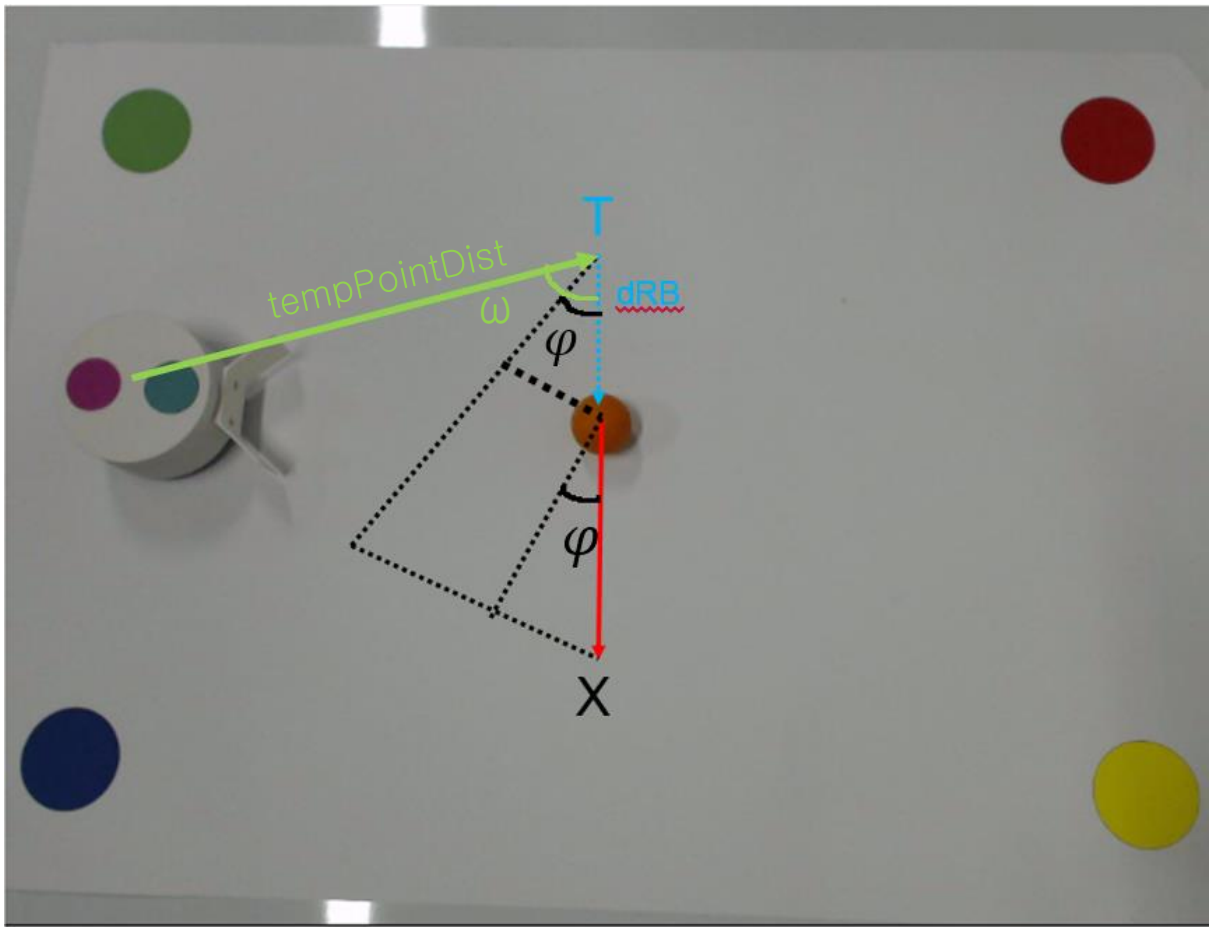


Figure 39

Rearranging Equations (1) and (2) gives the following solutions for calculating the temp points.

$$TempX = BallX - dRB * \cos\varphi$$

$$TempY = BallY - dRB * \sin\varphi$$

Now the robot needs to calculate the distance relative to its position and the angle relative to its heading. The distance relative to the robot's position or Temp Point distance is also calculated as follows as this allows the robot to know how far it needs to travel to the temp point:-

$$tempPointDist = \sqrt{((TempY - RobY)^2 + (TempX - RobX)^2)}$$

The error difference of the angle between the robot heading and the tempPoint heading (ω) needs to be calculated to insure that the robot is facing the point in the correct direction as it moves towards it. The equation to workout out the tempPoint heading (ω) is as follows.

$$\omega = \tan^{-1}\left(\frac{TempY - RobY}{TempX - RobX}\right)$$

Now the error value can be determined by calculating the following equation.

$$error = \omega - robHead$$

The value of the angle ***robHead*** was sent from the PC to the robot. The formulas for calculating the temporary target position are shown in the following code and are applied as follows. Notice that the 150 mm value was chosen which represents the distance between the ball and the temporary target point(***dRB***), this value was chosen to ensure that the robot does not crash into the ball as it moves towards the point and ensures that there is sufficient enough distance away from the ball before it reaches the temporary target point for the robot to turn and move towards it without hitting it.

```
//calculating the 1st temp1
theta = atan2((tgt->Y - pos->ballY),(tgt->X - pos->ballX));   //calculate angle between ball point and target point
yPoint = pos->ballY - 150*sin(theta);
xPoint = pos->ballX - 150*cos(theta);
```

The code for calculating the ***temptPtError*** error is as follows.

```
tempHead = atan2((yPoint - pos->robY),(xPoint - pos->robX));  //calculate the heading angle relative to the temporary point
tempPtError = ErrorCalc(pos->robHdg, tempHead, pos,tgt);      //calculate angle between robot heading and temp point
tempPtDistance = sqrt(pow(yPoint - pos->robY,2) + pow(xPoint  - pos->robX,2));      //calculate temp point distance between pt and robot
```

The robot uses the ***temptPtError*** to check if it is line with the temporary target point that it is heading to. If the robot is not in line with the temporary target then the robot adjusts the

motor speeds accordingly. The following coding segment here checks that the error is greater than 0.2 radians for which the robot needs to turn and correct its angle by adjusting its wheel speeds to move the robot angle to the left. To ensure a smooth turn, the left speed (SL) is set to +0.1 and the right wheel(SR) is set to -0.1. Previously this was done with one wheel set to 0 and the other wheel set to +0.1 but the results of doing this showed that the robot would overshoot or overcorrect the error.

```
if(tempPtError > 0.2)          //TURN LEFT if error greater than 0.2 radians
{

    if(tempPtDistance <= 40 && (ballTgtError < 0.1 && ballTgtError > -0.1))   //check if robot is within 40mm and in line with ball and target
    {
      robotState = MOVE_TO_TGT;       //change state to target point
      repeat = 1;
    }
    else
    {
      SL = -0.1;
      SR = 0.1;
      LeftMotor(SL);    //send speed values to the robot motors
      RightMotor(SR);

    }
  }
```

The same instance happens for if the ***temptPtError <-0.2***, where in this case the robot will need to move over to the right to correct the follow angle. Moving forward is where the robot is required to have both of its wheels set to +0.1. The values of the motor speeds are sent to the following wrapper functions for LeftMotor(SL) and RightMotor(SR). These functions are explained in more detail in the ***(Section 3 Difficulties and Challenges see Page 70)***.

Once the robot has successfully reached the temporary target point as the image in ***Figure 40*** suggests, it is now safe for the robot to head straight for the ball and push it to the target point labelled as ***"X".*** The following state transition to the ***"MOVE TO TARGET"*** state occurs as indicated in the block diagram in ***Figure 36.*** For the state transisition to occur the following conditions must be satisfied where the robot has to be less than or equal to 40mm from the temp point and the ***angle(θ)*** or ***ballTgtError*** has to be within the following range ±0.1 radians, this is stated in the following code.

```
else  //MOVE FORWARD
{
  if(tempPtDistance <= 40 && (ballTgtError < 0.1 && ballTgtError > -0.1))
  {
    robotState = MOVE_TO_TGT;
    repeat = 1;
  }
```

The following coding segment below outputs the readings that are present in the "***MOVE_TO_TEMP***" state. The string that it sends back to the PC is shown below in the ***"Received Data from Robot"*** textbox.

```
sprintf(msg, "%1i, %5.0f, %5.0f, %5.0f, %5.0f, %5.0f, %5.0f, %5.1f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %1i
                                        tempPtDistance, theta, tempPtError, ballTgtError, pos->ballHdg, pos->robHdg
len = strlen(msg);  //get length of message
OutputString(msg, len);     //send message to PC
```

Recieved Data from Robot

```
pos->count, pos->robX, pos->robY, pos->ballX, pos->ballY, xPoint, yPoint,tempPtDistance, theta, tempPtError, ballTgtError,
1,   -4,   204,   239,   207, -0.672, -0684, 1
1,   -4,   204,   239,   207,   238,   357, 286.5, -1.566, 0.555, -0.684, 0.012, 0.008, -0.672, -0.100, 0.100, 1, 0
```

A clearer view of what the following readings for the numbers mean are shown in the excel spreadsheet fragment below which shows that the current state(robotState) is equal to 1 (***MOVE_TO_TEMP***).

pos->count	pos->robX	pos->robY	pos->ballX	pos->ball\	xPoint	yPoint	tempPtDistance	theta	tempPtErr	ballTgtError	pos->ballH	pos->robHdg	tgtHdg	SL	SR	robotState	pos->newMsg
1	-4	204	239	207	-0.672	-684	1										
1	-4	204	239	207	238	357	286.5	-1.566	0.555	-0.684	0.012	0.008	-0.672	-0.1	0.1	1	0

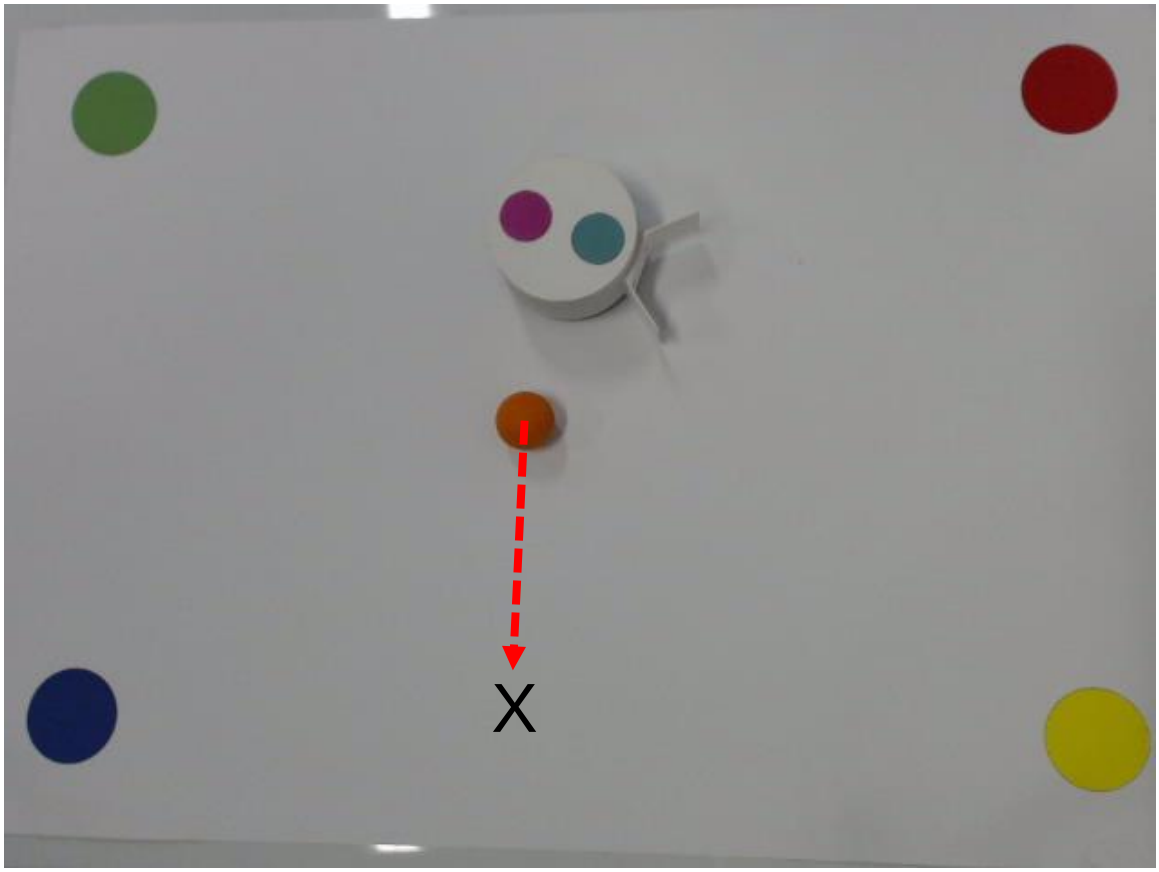


Figure 40

Here is another screenshot of the excel spreadsheet, which shows the ***robotState*** variable has changed to 2 (***"MOVE_TO_TGT"***), just after the robot had reached the temporary target point. The value was outputted just before the robot left the ***"MOVE_TO_TEMP"*** state.

pos->count	pos->robX	pos->robY	pos->ballX	pos->ball\	xPoint	yPoint	tempPtDistance	theta	tempPtErr	ballTgtError	pos->ballH	pos->robHdg	tgtHdg	SL	SR	robotState	pos->newMsg
96	219	339	243	212	245	362	34.9	-1.586	0.101	-0.123	-1.384	0.619	-1.507	0.1	0.1	1	0
97	221	341	243	212	245	362	32.1	-1.586	0.07	-0.112	-1.402	0.644	-1.513	0.1	0.1	1	0
98	227	348	244	212	247	362	24.4	-1.591	-0.055	-0.086	-1.446	0.666	-1.532	0	0	2	0

Also notice in ***Figure 40*** that the robot has successfully reached the target point and the excel spreadsheet shows that the status for the left and right wheel motor speeds for SL and SR are 0, which means the robot stopped when it reached that point. The ***ballTgtError*** is also within the $\mathbf{0.1} < \boldsymbol{\theta} > -\mathbf{0.1}$ range where it is equal to -0.086 radians and the robot position is at (219, 339) and the temporary target position is at (245, 362) the robot. These readings are consistent because the ***tempPtDistance*** shows 34.9 mm which is less than 40mm and this indicates that the robot has stopped in the correct position.

3.2.2.3 MOVE TO TARGET STATE

The purpose of the "MOVE TO TARGET STATE" which was indicated on the state diagram is to push the ball to the target heading which is indicated in ***Figure 40.*** As the state operates in more or less the same way that the ***"MOVE TO TEMP STATE"*** operates there was no need to create a structure diagram for this state, which should be identical to the structure diagram shown in ***Figure 38,*** the only difference being that it doesn't calculate the Temporary target as the target is already available for when the software program on the PC sent it. The other thing that is different it calculates the Target distance relative to the robot and the target error which is the difference between the robot heading and the target heading.

Once in the ***"MOVE TO TARGET STATE"*** the robot can head straight for the ball and push it to the required target point. The following coding segment below shows that the following pieces of information are calculated. Also notice that the angle of error is now checking to see if the robot heading is in line with the target position that the robot will need for pushing the ball to.

```
tgtHdg = atan2((tgt->Y - pos->robY),(tgt->X - pos->robX)); //calculate angle relative to target position and the
tgtError = ErrorCalc(pos->robHdg,tgtHdg,pos,tgt);   //calculate error between robot heading and target heading
tgtDistance = sqrt(pow(tgt->Y - pos->robY,2) + pow(tgt->X  - pos->robX,2));     //calculate the target distance
```

Figure 41 shows that the robot has successfully placed the ball at the required target position.

Figure 41

Notice that when the robot has reached the target point that the ***robotState*** value has changed from 2 to 0, which indicates that the robot has pushed the ball to the target point and the robot state has changed from ***"MOVE TO TARGET STATE"*** to the ***"STOP"*** state.

```
sprintf(msg, "%1i, %5.0f, %5.0f, %5.0f, %5.0f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %1i, %1i\r\n", pos->count, pos->robX, pos->robY, pos->ballX, pos->ballY,
                                        tgtDistance, tgtHdg, pos->robHdg, tgtError, SL, SR, robotState, pos->newMsg);
len = strlen(msg);  //get length of message
OutputString(msg, len);     //send message to PC
```

pos->count	pos->robX	pos->robY	pos->ballX	pos->ballY	tgtDistanc	tgtHdg	pos->robHdg	tgtError	SL	SR	robotState	pos->newMsg
231	262	110	239	42	102.391	-1.787	-2.065	0.278	-0.07	0.07	2	0
232	263	108	235	38	100.663	-1.801	-2.021	0.22	-0.07	0.07	2	0
233	262	105	235	35	97.514	-1.798	-2.002	0.203	0	0	0	0

The formulas that the robot uses for moving to the ball when the robot is moving straight is shown below. It applies the Theory that was mentioned in 1.2 which calculates the speeds of the left and right Wheels (SL and SR) of the robot on the field, the code that produces this is shown below.

```
else
{

  SL= tgt->spd*0.001*(tgtDistance - 100*sin(tgtError));   //Adjust the motor speed using the following algorithm
  SR= tgt->spd*0.001*(tgtDistance + 100*sin(tgtError));

  if(SL <= 0.1 || SR <= 0.1)      //check if speed values drop below 0.1
  {
    SL = 0.1;
    SR = 0.1;
  }
  LeftMotor(SL);
  RightMotor(SR);

}
```

This enables the robot to move in a curve towards where the target point is, which helps the robot motors move straight.

4 Difficulties and challenges

This section will detail the challenges and obstacles that were encountered throughout the duration of the project. This section explains the software issues that were previously tried and tested for when the software program on the PC and the software program on the robot was first developed. It was found that many obstacles were discovered because there were major issues which affected the performance of the software such as trying to fix the memory leak issue with the software program on the PC and the serial port communications issue with the software on the robot.

4.1 Software on the PC

4.1.1 Copying Bitmap Images

As well as receiving the Bitmap's, a copy must be made so that the images can be stored and then used for data manipulation such as storing the image on the software GUI form and/or used for carrying out calculations. Previously the Bitmap's were copied using the ***.Clone()*** method. The main problem behind using this method is that it only performs a shallow copy of the Bitmap because this method creates a new Bitmap object but the pixel data is shared with the original bitmap object. The Bitmap(Image) constructor also creates a new Bitmap object but one that has its own copy of the pixel data. After some research it was discovered that the ***.Clone()*** method can cause problems because it can lock on the file from which it was loaded. The following code segment below shows how the clone method is used. The ***pictureBoxVideo.Image = clone()*** is the code that displays an image on the C# GUI form which is displayed using a Picture Box object.

```
Bitmap frame_copy= (Bitmap)eventArgs.Frame;      //convert webcam frame into a bitmap image
Bitmap clone = (Bitmap)frame_copy.Clone();
pictureBoxVideo.Image = clone;
```

It was also discovered that using the ***.Clone()*** method for copying Bitmap's would throw an exception error ***"A generic error occurred in GDI+",*** which would occur intermittently. This error usually occurs because the Bitmap's implementation only remembers the path of the Bitmap image source as opposed to accessing it directly from memory. This means that when a copy of the Bitmap is made it only copies the reference to the original Bitmap source instead of copying the actual Bitmap itself. For example if another part of the program is trying to access the same Bitmap source at the same time, this could cause the following error to occur, the example could be displaying the Bitmap on the GUI and using the same

Bitmap for data manipulation such as carrying out calculations. This problem might not occur all the time but there could be a point at which this could occur and hence throw the following exception.

To solve this problem a function which performs as deep copy on a Bitmap would need to be made. The issue with using the .clone() was that research shows it only copies the reference or path to the Bitmap location instead of the actual Bitmap itself.

Conclusion

In conclusion to the project the robot is able to track the position of the ball placed anywhere on the field. In addition to this it is also able to move to a temporary point and then push the ball towards where the target is located.

There were many obstacles that were encountered throughout the project duration such as exception errors with the image processing software. Overall after the issues and faults were rectified the outcome of the project was successful.

4.1.2 Image processing

The previous method used a ***GetPixel*** function as part of the ***C#*** class that retrieves the RGB colour values of a Bitmap depending on the ***(X,Y)*** coordinates that are selected. The first algorithm that was developed used this method for extracting each pixel value individually from the Bitmap which would uses a *"For loop"* to loop through all the X and Y values of each pixel and at the same time get the colour value of each pixel. This was shown in the code snippet below. The purpose of writing the code in this way is that it was done for testing purposes because it was important to see how efficient and suitable the following code was for processing a Bitmap image.

```
for (int x = 0; x < image.Width; x++)
{
    for (int y = 0; y < image.Height; y++)
    {
        Color pixel = image.GetPixel(x, y);

    }
}
```

Unfortunately after running the code, the performance of the program would slow down considerably. After extensive research it was found that the problem was because access to the pixel is not a simple reference to a memory location. For example when the ***GetPixel(X,Y)*** function for determining a colour is called it is associated with the invocations of a .NET Framework method, that acts as a wrapper for a native function called ***gdiplus.dll***. This call is through the mechanism of P/Invoke (Platform Invocation), which is used to communicate from managed code to unmanaged API (an API outside of the .NET Framework). For example the bitmap that is required for the project is 640 x 480 pixels, this means that the will be 307200 calls to the ***GetPixel*** method that besides the validations of parameters uses the native ***GdipBitmapGetPixel*** function. Before the colour information can be returned the GDI+ function has to perform an operation that requires calculating the position of the bytes responsible for the description of the desired pixel.

Fortunately there was a much faster and quicker method for extracting the desired coloured pixels from the Bitmap image, which uses a method for locking the Bitmap into system memory and eliminates the need for calling many functions like the ***GetPixe***l method does. This was solved using the code that is currently used in the current program.

```
bData = latestBmp.LockBits(new Rectangle(0, 0, latestBmp.Width, latestBmp.Height), ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);
```

```
unsafe
{
    //get address of the first line
    ptr = bData.Scan0;
    // Declare an array to hold the bytes of the bitmap.
    bytes = Math.Abs(bData.Stride) * latestBmp.Height;
    rgbValues = new byte[bytes];
    // Copy the RGB values into the array.
    System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes);
} // unsafe
```

4.1.2.1 Hue Saturation and Value

After locking the Bitmap into system memory the next step was to extract the pixel colours from inside the Bitmap. The previously used method for extracting the Hue and Saturation values from the bitmap inside the ***rgbValues*** array used C#'s built in ***Color*** class from the .NET Framework for determining the Hue and Saturation which used the corresponding methods ***.GetHue()*** and ***.GetSaturation()***, as shown in the code snippet.

```
// check every pixel colour
for (int counter = 0; counter < rgbValues.Length; counter += 3)
{
    float hue = Color.FromArgb(rgbValues[counter + 2], rgbValues[counter + 1], rgbValues[counter]).GetHue();
    float sat = Color.FromArgb(rgbValues[counter + 2], rgbValues[counter + 1], rgbValues[counter]).GetSaturation();
```

The ***.GetHue()*** method returns a value that is between 0 and 360°, and the .***GetSaturation()*** method returns a value between 0 and 1. The issue that was discovered later on was the fact that these methods worked with ***float*** data types for carrying out the Hue and Saturation calculations and this had a noticeable effect on the performance of the software GUI program. The noticeable effect was a lagging issue when the software GUI form was displaying live images or frames being received from the web cam. The root cause of the problem was because the ***.GetHue()*** and .***GetSaturation()*** are inside a *"For loop"* which loops round and calls these functions every 307200 times and uses data types that work with floats.

The solution to the problem was to develop ***Hue*** and ***Saturation*** functions that worked with whole numbers such as integers instead of floats. The code snippet below shows where the new function ***CalcHueSat()*** that was created for the Hue and Saturation values is called from and the Hue and Saturation values, are passed back as references, this function allows the RGB to HSV transformation to take place.

```
for (int counter = 0; counter < rgbModValues.Length; counter += 3)
{
    CalcHueSat(rgbModValues[counter + 2], rgbModValues[counter + 1], rgbModValues[counter], ref hue, ref sat);
```

4.1.2.2 Image Processing using a Button on the GUI form

Previously during the prototyping stages of the imaging software, a Button was used to initiate the image processing algorithms, the GUI is shown in ***Figure 42***. What is basically shown here is that as the Web Cam event handler is receiving images at a rate of 33 frames a second and pressing the ***"Update Image"*** button below allows the Bitmap image to be processed and the hue and saturation values will be calculated. The image in the right hand side of the form is the current image that shows the latest image that was processed. The textbox below the ***"PROCESSED IMAGE"*** text displays the pixel counts and coordinates of the pixels shown in the image in terms of pixels.

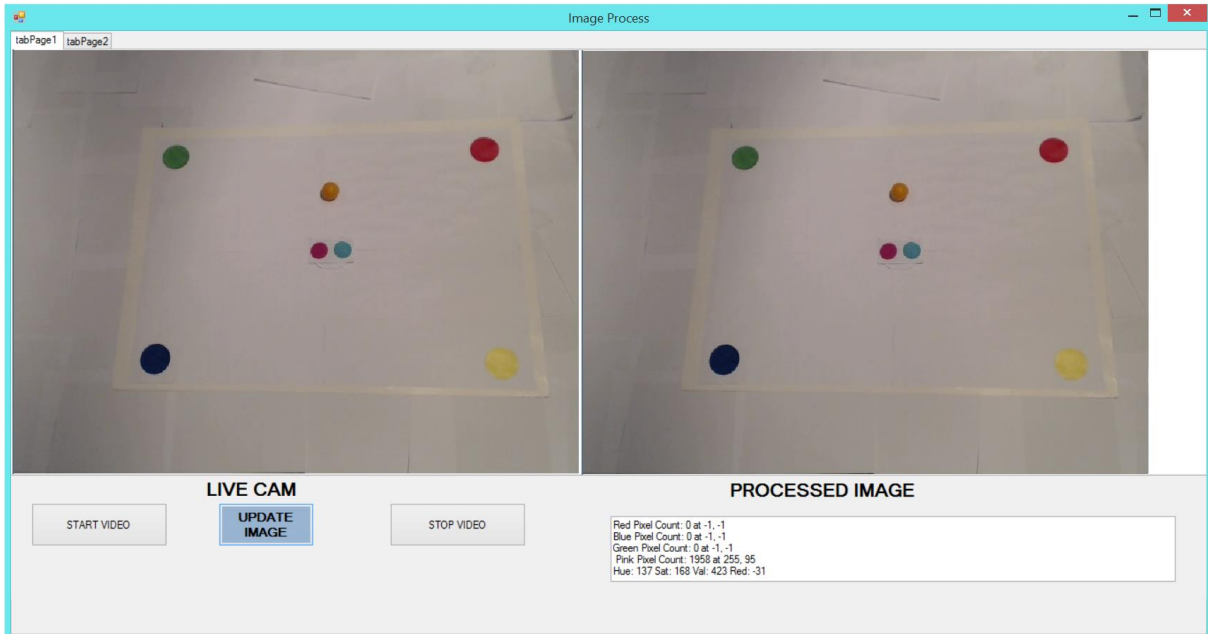


Figure 42

The code for the update button event handler is shown below. This was the code that was previously used for carrying out the image processing. Previously the Bitmap images were

copied directly from the images being displayed in the picture box to the left or by copying the ***pictureBoxVideo.Image*** variable. The downside to using a button event for carrying out the processing is that it is not automated and the latest coordinates will only appear when the update image button is pressed. It was necessary to use the button previously because this had the advantage of deciding when to capture the image of interest.

```
private void Update_Butt_Click(object sender, EventArgs e)
{


    // When you call this, make sure to check that there is some image data first

    System.Windows.Forms.MouseEventArgs mevent = (System.Windows.Forms.MouseEventArgs)e;
    if (mevent.Clicks <= 1)
    {

        try
        {
            Bitmap bmp = (Bitmap)pictureBoxVideo.Image;
            Bitmap newBMP = CopyBMP(bmp);

              pictureEdgeVideo.Image = EdgeDetectDifference(image_process, 0);          //do the edge detection of the picture image
            pictureBoxMod.Image=newBMP;


            BitmapData bData = newBMP.LockBits(new Rectangle(0, 0, newBMP.Width, newBMP.Height), ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb);
            unsafe
            {
```

The other features on the GUI was the "***START VIDEO***" and "***STOP VIDEO***" button, these were previously used to turn on and off the web cam.

The issue was solved using a Form Timer event handler because the Form Timer event will run periodically, depending on the time setting used. As shown in the following image below this was called ***Image_timer.***

The speed setting for the ***image_timer*** function was initially set to 100ms, but this speed was considered to be too slow and the live image picture box would display frames from the web cam at a rate slower than the web cam event handler was displaying frames. A speed setting of 1ms was considered too fast because this would mean that the timer event handler would be running at a faster rate than the web cam event handler receives frames. So the decision was to have the setting of the Form1 timer set to 10ms, because this would enable a rate that wasn't too fast for the imaging software to handle the frames and also it was slow enough to allow the web cam event handler to receive and display the frames that are coming from the web cam.

As the speed of the image_timer tick event handler function was 10ms, which was still much faster than the web cam event handler so a statement for checking if the global variable ***(latestBmp)*** contains the latest bitmap from the web cam event handler was necessary. If the variable does not contain the latest bitmap then the program exits or returns from the function. The following coding segment that describes this is shown below. This section of

code is located inside the image_timer tick function.

```
// When you call this, make sure to check that there is some image data first
if (latestBmp == null)
    return;
timerRunning = true;
```

At the same time as doing this the ***timerRunning*** flag is set to true, the reasons for setting the the flag to true was to prevent the ***videoSource_NewFrame*** event handler from corrupting or putting new Bitmaps in the ***latestBmp*** bitmap global variable when the image_timer function is currently processing the image. The ***timerRunning*** flag was mainly added in to prevent the "***Object is currently in use elsewhere***" exception error, which kept occurring intermittently. The cause of the exception error was probably due to the result of the ***latestBmp*** variable being used by the videoSource_NewFrame when new frames are placed inside this variable at the same time as the image_timer processes the bitmaps.

The previously used new frame event handler code is shown below. The coding segment that is of most importance checks to see if the ***timerRunning*** bitmap is currently being processed, if the ***!timerRunning*** flag is equal to true this means that the current image is still being processed and no new frames will be placed inside this variable so therefore the latest frame received from the web cam will not update this variable, until the flag inside the image_timer event handler function makes the flag equal to false. In other words until the ***timerRunning*** flag is equal to false.

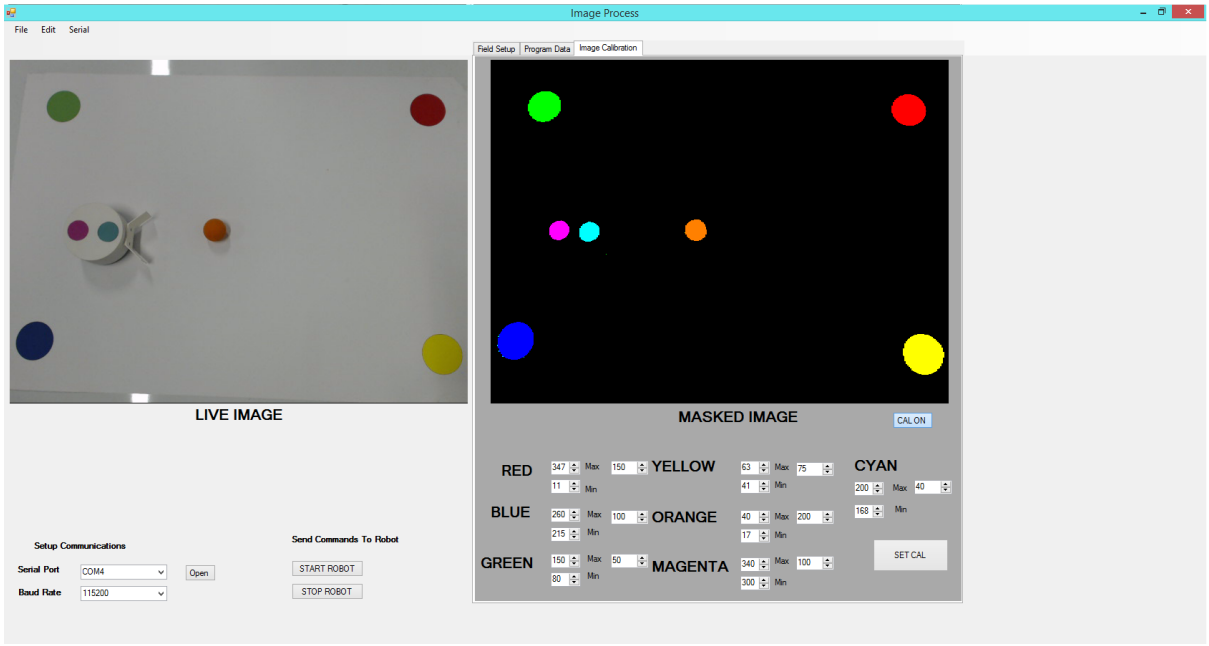
```
//call this event everytime image data is recieved from camera
void videoSource_NewFrame(object sender, AForge.Video.NewFrameEventArgs eventArgs)
{
    Bitmap newBmp, newnewBmp;

    if (stopping)
        return;
    lock (lockObject)
    {
        newBmp = (Bitmap)eventArgs.Frame;
        frameBmp = CopyBMP(newBmp);
        if (frameBmp != null)        //if webcame frames not null
        {
            if (stopping)
                return;
            SetPicture(frameBmp); //Object is currently in use elsewhere exception
        }
        newnewBmp = CopyBMP(newBmp);
        if (!timerRunning)
            latestBmp = newnewBmp;
    }
```

The coding segment below here shows that after the image_timer functions has finished processing the bitmap it unlocks it, makes ***latestBmp*** equal to null, and then the ***timerRunning*** flag is equal to false.

```
unsafe
{
    //get address of the first line
    ptr = bData.Scan0;
    // Declare an array to hold the bytes of the bitmap.
    bytes = Math.Abs(bData.Stride) * latestBmp.Height;
    rgbValues = new byte[bytes];
    // Copy the RGB values into the array.
    System.Runtime.InteropServices.Marshal.Copy(ptr, rgbValues, 0, bytes);
} // unsafe
// Unlock the bits.
latestBmp.UnlockBits(bData);            //unlock the image    (A generic error occurred in
latestBmp = null;
timerRunning = false;
```

4.1.2.3 Calibration software

The issues with the image processing software were that the previous architecture had the Image processing calculations and calibration happen at the same time. So eventually the program was decoupled or broken down and the 2 different parts were separated. Originally there was an architecture that was modified many times which previously 2 different individual Form Timers where one was for calibration and the other was for the image processing calculations.



The current architecture of the program uses only 1 form timer which is called the 1ms_Image_Timer and this still allowed for the calibration to be separate to the calculations. The “CAL BUTTON” on the GUI prevents the calibration or “MASKED IMAGE” from appearing in the picture box, while still only requiring one timer for doing the image processing, was mentioned in more detail in 3.1.1.3.

4.1.2.4 (MM) Position reading errors

The formulas which carry out the Pixel to Millimetre conversions for the ballX and robotX values would output zeros, into the corresponding position structures and for ballY and robY these readings shown inconsistencies of 7132 and 6676, etc. These readings were outputted to the GUI in the corresponding text boxes.

Table 1

ballX	ballY	robotX	robotY	robotHead	ballHead	distance	difference	leftSpeed			Count	Rn	Rx	Ry	Bn
635	316	545	202	29	51.71	145.245	22.71	-0.1	-0.1		POS 635	316	545	202	29
634	314	555	205	35	54.067	134.618	19.067	-0.1	0.1		POS 634	314	555	205	35
634	316	561	210	46	55.446	128.705	9.446	-0.1	-0.1		POS 634	316	561	210	46
634	316	572	217	45	57.943	116.812	12.943	-0.1	-0.1		POS 634	316	572	217	45
0	7132	0	6676	90	90	456	0	-0.1	-0.1		POS 0	7132	0	6676	90
635	314	592	230	46	62.892	94.366	16.892	0	0		POS 635	314	592	230	46
635	314	593	230	44	63.435	93.915	19.435	0	0		POS 635	314	593	230	44
635	314	593	230	44	63.435	93.915	19.435	0	0		POS 635	314	593	230	44
635	314	592	230	46	62.892	94.366	16.892	0	0		POS 635	314	592	230	46
635	314	593	230	45	63.435	93.915	18.435	0	0		POS 635	314	593	230	45
635	314	593	230	46	63.435	93.915	17.435	0	0		POS 635	314	593	230	46
635	314	593	230	45	63.435	93.915	18.435	0	0		POS 635	314	593	230	45
0	7147	0	6749	90	90	398	0	0	0		POS 0	7147	0	6749	90
635	314	593	230	45	63.435	93.915	18.435	0	0		POS 635	314	593	230	45
635	314	591	229	47	62.632	95.713	15.632	0	0		POS 635	314	591	229	47
635	314	593	230	44	63.435	93.915	19.435	0	0		POS 635	314	593	230	44
635	314	594	230	45	63.983	93.472	18.983	0	0		POS 635	314	594	230	45

This issue was later on rectified.

4.1.3 Serial Transmission issues

Previously there was an issues with the serial port on the PC. The issue was when the PC was sending data to the robot but the robot was not turned on. This would cause the program on the PC to crash because the serial port is being hanged because the data buffer where the PC program writes the data to is not being cleared or sent out to the robot. The solution to the following problem was to have a timeout exception of 500ms on the serialPort1.Write. The length of the timeout period can be changed in the properties for the serial port class. The code that describes the sendToSerial function is shown below.

```
private void sendToSerial(string robotData)
{

    //is the serial port open
    if (serialPort1.IsOpen)
    {
        try
        {
            serialPort1.Write(robotData);   //send data to serial port
        }
        catch(TimeoutException)
        {
            //Serial Port communications disabled!
            SerialPortCheck.Checked = false;
            SerialPortCheck.Enabled = true;
            cboPorts.Enabled = true;
            cboBaudRate.Enabled = true;
            openToolStripMenuItem2.Enabled = true;

            if (serialPort1.IsOpen)
            {
                serialPort1.Close();
                this.BeginInvoke((Action)(() => MessageBox.Show("Serial Port closed!")));
            }

        }
    }

}
```

What basically happens now is that if an issue with writing to the serial port buffer on the PC occurs due to the power on the robot receiving the data not being on, then a timeout of 500ms will occur. After that the program will catch the exception and the image processing program on the PC will stop sending information to the robot, and the program will continue running without crashing.

As well as having a timeout exception for transmitting the data it was also necessary to have a timeout exception for when the data on the PC was being received from the robot. The following code below shows the receive data event handler function. What basically happens here is that when the robot transmits data back to the PC it calls a ***SetText(DataBuffer)*** function that writes the incoming data being received from the robot onto the textbox of the software GUI. If a timeout was to occur then a similar scenario to that to the writing to the serial port buffer as the reading operation would occur where the program would resume and normal operation would occur.

```
//Data being recieved from the serial port
private void serialPort1_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
    try
    {
        DataBuffer = serialPort1.ReadExisting();        //read incoming data from the serial port
        if (DataBuffer != String.Empty)
        {
            SetText(DataBuffer); //Put time at front of string
        }
    }
    catch(TimeoutException)
    {
        int y = 0;
        //Serial Port communications disabled!
        SerialPortCheck.Checked = false;
        SerialPortCheck.Enabled = true;
        cboPorts.Enabled = true;
        cboBaudRate.Enabled = true;
        openToolStripMenuItem2.Enabled = true;

        if (serialPort1.IsOpen)
        {
            serialPort1.Close();
            this.BeginInvoke((Action)(() => MessageBox.Show("Serial Port closed!")));

        }
    }

}
```

4.1.4 Memory Leaks in software program

There were many issues with regards to memory leaks in the following software program on the PC. The frames and Bitmaps that was processed by the image processing software needed to be cleared after they were used. This was found to be the root cause of the memory leak issue on the PC program. A sign of the memory leak would be indicated in the ***"Task Manager"*** of the PC which would show the memory increasing is size beyond 1 GB.

4.1.4.1 Copying Bitmap images

The memory leaks arose when an alternative method for copying the Bitmap images stored in memory was used. The following code segment shows that ***Marshal.AllocHGlobal*** method which is thought to be the main issue for the memory leaks when this method is used.

```
BitmapData bData = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height), ImageLockMode.ReadOnly, PixelFormat.Format24bppRgb); //C

    //stride can be negative
    int byteCount = Math.Abs(bData.Stride * bmp.Height);

   IntPtr lscan0 = Marshal.AllocHGlobal(byteCount);
                           //

   // Marshal.FreeHGlobal(lscan0);

    if (bData.Stride < 0)
    {
        //Scan0 points to the beginning of the first scan line, not the beginning of the first byte of data. In a bottom-up bitmap,
        IntPtr srcData = (IntPtr)(bData.Scan0.ToInt64() - ((bmp.Height - 1) * -bData.Stride));


        CopyMemory(lscan0, srcData, byteCount);
        lscan0 = (IntPtr)(lscan0.ToInt64() + ((frame_copy.Height - 1) * -bData.Stride));
    }
    else
    {
        CopyMemory(lscan0, bData.Scan0, byteCount);
    }

    bmp.UnlockBits(bData);
```

The ***Marshal.AllocHGlobal*** allocates an area of unmanaged memory for storing the Bitmap images where the number of bytes of the memory are passed to this method. The issue with using this is method is that the memory is not be freed or put back after it has been used. This was also observed in the ***Task Manage*****r**(***see Figure 43***).

The program that is running is indicated by the ***vshost32.exe(32 bit)*** it shows that the memory had increased to about 1,238.7MB !!!. After leaving this running for a while eventually an ***"Out Of Memory"*** exception error would be thrown, in Visual studio.

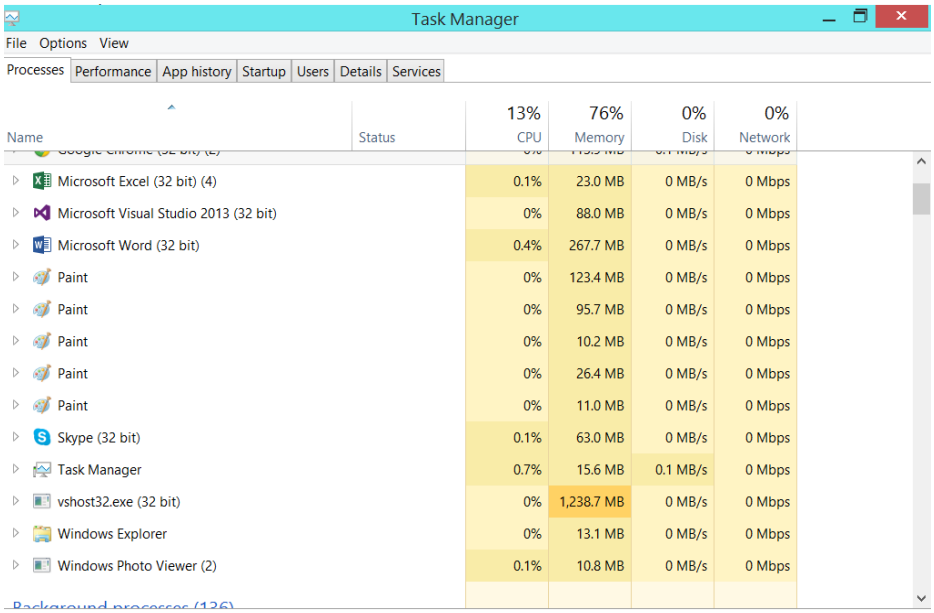


Figure 43

The following code was eventually replaced with the code which uses the ***memcpy*** method for copying the Bitmap images into memory. This code was located inside the previously used ***CopyBMP()*** function and is called from the ***videoSource_NewFrame*** event handler whenever a copy of the Bitmap image is required.

```
unsafe
{
    int width = bmp.Width;
    int height = bmp.Height;

    PixelFormat pixelFormat = bmp.PixelFormat;
    // lock source bitmap data
    BitmapData srcData1 = bmp.LockBits(new Rectangle(0, 0, width, height), ImageLockMode.ReadWrite, pixelFormat);
    // create new image
    newBMP = new Bitmap(width, height, pixelFormat); // throws a parameter not available exception now and again
    // lock destination bitmap data
    BitmapData dstData = newBMP.LockBits(new Rectangle(0, 0, width, height), ImageLockMode.ReadWrite, pixelFormat);
    memcpy(dstData.Scan0, srcData1.Scan0, new UIntPtr((uint)height * (uint)srcData1.Stride));
    // unlock both images
    newBMP.UnlockBits(dstData);
    bmp.UnlockBits(srcData1);
}
```

The coding snippet below here shows the videoSource_NewFrame event handler for the program.

```
//call this event everytime image data is recieved from camera
void videoSource_NewFrame(object sender, AForge.Video.NewFrameEventArgs eventArgs)
{
    Bitmap newBmp, newnewBmp;

    if (stopping)
        return;
    lock (lockObject)
    {
        newBmp = (Bitmap)eventArgs.Frame;
        frameBmp = CopyBMP(newBmp);
        if (frameBmp != null)        //if webcame frames not null
        {
            if (stopping)
                return;
            SetPicture(frameBmp); //Object is currently in use elsewhere exception
        }
        newnewBmp = CopyBMP(newBmp);
        if (!timerRunning)
            latestBmp = newnewBmp;
    }

}
```

There was still a memory leak issue when this code was used, where the Garbage Collector was not being cleared quick enough so the following statement called “System.GC.Collect() manually calls the Garbage Collector.

```
void videoSource_NewFrame(object sender, AForge.Video.NewFrameEventArgs eventArgs)
{
    Bitmap newBmp, newnewBmp;

    if (stopping)
        return;
    lock (lockObject)
    {
        newBmp = (Bitmap)eventArgs.Frame;   //recieve new frame from webcam and convert into Bitmap
        frameBmp = CopyBMP(newBmp);     //copy frame.

        if (frameBmp != null)        //check to see if webcame frames are not null
        {
            if (stopping)
                return;
            SetPicture(frameBmp); //Object is currently in use elsewhere exception
        }
        if ((Image_timer.Enabled == true || calTimer.Enabled == true) && !timerRunning)
        {
            latestBmp = CopyBMP(newBmp);    //call funtion to return Bitmap
            calBMP = CopyBMP(newBmp);    //call funtion to return Bitmap
        }
        else
           System.GC.Collect();     //free the memory and clean garbage collector
    }
}
```

Eventually the whole program that deals with the receiving frames from the Web Cam was re-written and the functions that deal with the web cam and copying Bitmaps were placed inside a separate class which was made into a more object orientated like solution. The new program did not cause any further issues.

4.2 Software on the Robot

4.2.1 Over flowing arrays

There were issues in regards to the software on the robot as far as overflowing arrays were concerned. Data string being transmitted from the PC and to the robot, had to be large enough for storing the entire string. The code segments below output information unique to each of the robot states after data has been sent to the robot.

STOP STATE string

```
sprintf(msg, "%1i, %5.0f, %5.0f, %5.0f, %5.0f, %5.3f, %5.3f, %1i\r\n", pos->count, pos->robX, pos->robY, pos->ballX, pos->ballY, tgtHdg, ballTgtError, robotState);
len = strlen(msg);  //get length of message
OutputString(msg, len);     //send message to P
```

TEMP state string

```
sprintf(msg, "%1i, %5.0f, %5.0f, %5.0f, %5.0f, %5.0f, %5.0f, %5.1f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %5.3f, %1i, %1i\r\n", pos->count,pos->robX, pos->robY,
    pos->ballX, pos->ballY, xPoint, yPoint, tempPtDistance, theta, tempPtError, ballTgtError, pos->ballHdg, pos->robHdg, tgtHdg, SL, SR, robotState, pos->newMsg);

len = strlen(msg);          //get length of message
OutputString(msg, len);     //send message to PC
```

What basically happens here is that the robot would transmit a series of readings and calculations back to the PC GUI textbox after the PC had sent a string to it. This was to signify that the robot had received a response from the PC and communication between the robot and the PC are consistent. An example of a consistent situation would be that if a string from the software program on the PC was sent to the robot and the robot transmitted the data back moments after. Another example of a consistent situation is where the strings sent from the PC matches the data that the robot receives and transmits back to the PC. However ***Table 2*** shows a case where the readings are not consistent. The following situation was due to the size of the arrays not being large enough to hold the string it needs to send back to the PC.

These readings were copied directly from the textbox on the software GUI of the PC, only the readings from the robot are shown. The information that the robot sends back to the PC is information such as its current position, angle of error, which depends on the state it is currently in.

Table 2

count	robX	robY	ballX	ballY	tgtHdg	ballTgtError	robotState						
1	150	41	347	266	0.493	-0.358	1						
count	robX	robY	ballX	ballY	tempPtDis	tempPtError	ballTgtErro	pos->ballHdg	pos->robHd	tgtHdg	SL	SR	robotState
2	151	41	347	266	251.845	1.355	-0.36	0.854	0.026	0.495	-0.1	0.1	1
3	151	41	346	266	251.531	1.166	-0.362	0.857	0.219	0.495	-0.1	0.1	1
4	151	41	347	266	251.845	0.813	-0.36	0.854	0.567	0.495	-0.1	0.1	1
5	150	41	346	266	251.718	0.447	-0.361	0.854	0.933	0.493	-0.1	0.1	1
5	150	41	346	266	251.718	0.447	-0.361	0.854	0.933	0.493	-0.1	0.1	1
7	150	42	347	266	251.055	-0.213	-0.358	0.849	1.589	0.491	0.1	-0.1	1
8	150	42	347	266	251.055	-0.499	-0.358	0.849	1.874	0.491	0.1	-0.1	1
9	149	44	346	266	248.967	-0.214	-0.359	0.845	1.588	0.486	0.1	-0.1	1
10	149	45	346	266	247.986	0.112	-0.359	0.843	1.262	0.484	0.1	0.1	1
11	153	48	347	266	244.594	0.448	-0.362	0.844	0.935	0.482	-0.1	0.1	1
12	159	55	347	266	236.641	0.357	-0.369	0.843	1.045	0.474	-0.1	0.1	1
13	159	56	347	266	235.655	0.014	-0.369	0.841	1.387	0.471	0.1	0.1	1
14	158	62	346	266	229.616	-0.21	-0.369	0.826	1.607	0.457	0.1	-0.1	1
15	159	81	347	266	211.054	-0.203	-0.362	0.777	1.585	0.415	0.1	-0.1	1
16	159	92	347	266	200.261	-0.058	-0.358	0.747	1.429	0.389	0.1	0.1	1
17	160	94	346	266	197.784	0.299	-0.361	0.746	1.081	0.385	-0.1	0.1	1
18	162	95	347	266	196.74	0.454	-0.361	0.746	0.93	0.385	-0.1	0.1	1
19	162	95	346	266	196.429	0.102	-0.364	0.749	1.286	0.385	0.1	0.1	1
20	161	97	346	266	194.649	-0.202	-0.361	0.74	1.584	0.379	0.1	-0.1	1
20	161	97	346	266	194.649	-0.202	-0.361	0.74	1.584	0.379	0.1	-0.1	1
22	164	131	346	266	160.729	-0.181	-0.341	0.638	1.541	0.297	0.1	0.1	1
22	164	131	346	266	160.729	-0.181	-0.341	0.638	1.541	0.297	POS 34	0.1	164
24	164	144	347	266	148.399	0.292	-0.325	0.588	1.043	0.263	-0.1	0.1	1
25	167	147	347	266	144.798	0.355	-0.327	0.584	0.996	0.258	-0.1	0.1	1
26	167	148	347	266	143.822	0.03	-0.325	0.58	1.319	0.255	0.1	0.1	1
27	168	154	347	266	137.75	-0.163	-0.319	0.559	1.51	0.24	0.1	0.1	1
28	170	171	347	266	120.745	-0.178	-0.298	0.493	1.509	0.195	POS 34	266	173
31	174	215	347	266	77.332	-0.091	-0.215	0.287	1.337	0.072	0.1	0.1	1
32	175	215	346	266	76.592	0.256	-0.218	0.29	1	93	43	346	266
2	93	42	347	266	268.003	1.237	-0.289	0.723	-0.071	0.434	-0.1	0.1	1
3	93	43	347	266	267.084	1.023	-0.288	0.72	0.141	0.432	-0.1	0.1	1
4	93	42	347	266	268.003	0.628	-0.289	0.723	0.538	0.434	-0.1	0.1	1
5	94	42	347	266	267.61	0.28	-0.29	0.725	0.889	0.435	-0.1	0.1	1

We can see that by looking at when the count value is 22, we start to see inconsistencies with the readings. The ***POS 34*** string in the ***SL(left wheel speed of robot)*** columns show that some of the data that the PC sends to the robot is being corrupted with the data that the robot transmits back to the PC. It was apparent that there were times when the readings had more decimal points due to the result of the calculations that the robot performs.

The following issue was solved by increasing the size of the array from 100 bytes to 200, as indicated in the following coding segment below.

```
static char msg[200];
```

4.2.2 Motor Speed function issues

There was an issue that was noticed when trying to send the speed values to the following motor speeds. The issue that was noticed was stated in the ***mbed*** documentation for using m3pi.left_motor() and m3pi.right_motor() library methods and was discovered to be incorrect. The documentation stated that the motor speeds can vary from -1 to +1 with positive being forward and negative being the reverse direction. However when these values were tested by setting both motor speeds to forwards or to +1, the exact opposite behaviour occurred, the robot would start moving backwards with a +1 value. Likewise for setting the speed to -1 the robot would move in the forward direction.

To avoid confusion for knowing which way is forwards and backwards a resolution or way round the problem was to create wrapper functions for both the left and right motor speeds which operate by reversing the sign of the value being sent to the functions. For example sending a value of +1 to the wrapper function for moving the robot forward will change the sign into a -1 value and the following method inside the wrapper function will send the number out to the motors and then the motors will move forward. The coding segment for the wrapper functions is shown below.

```
// This is to put a wrapper on the motor speed ].
void LeftMotor(float speed)
{
  m3pi.left_motor(-speed);     //this function is because we running motors backwards
}                              //we want to reverse the sign of the motors to make it move in direction we want

void RightMotor(float speed)
{
  m3pi.right_motor(-speed);    //this function is because we running motors backwards
}
```

4.2.3 Serial Transmit/Receive issues

There was strange behaviour that was noticed with the robot. The behaviour that was noticed was the fact that the robot would intermittently lose communication with the PC software that is sending the position messages to the robot.

Count	ballX	ballY	robotX	robotY	robotHea	ballHead	distance	difference	leftSpeed	rightSpeed		robotHea	ballX	ballY	robotX	robotY	Count
1	145	50	207	321	87	-102.886	278.002	170.114	0	0		POS 145	50	207	321	87	1
2	143	50	206	322	92	-103.041	279.201	164.959	-0.1	0.1		POS 143	50	206	322	92	2
3	144	49	207	328	97	-102.724	286.024	160.276	-0.1	0.1		POS 144	49	207	328	97	3
4	144	49	207	335	108	-102.423	292.857	149.577	-0.1	0.1		POS 144	49	207	335	108	4
5	144	49	208	345	111	-102.2	302.84	146.8	-0.1	0.1		POS 144	49	208	345	111	5
6	145	49	204	348	125	-101.162	304.765	133.838	-0.1	0.1		POS 145	49	204	348	125	6
7	149	54	203	355	132	-100.171	305.805	127.829	-0.1	0.1		POS 149	54	203	355	132	7
8	151	56	201	360	137	-99.34	308.084	123.66	-0.1	0.1		POS 151	56	201	360	137	8
9	151	56	196	362	147	-98.366	309.291	114.634	-0.1	0.1		POS 151	56	196	362	147	9
10	151	56	191	365	156	-97.376	311.578	106.624	-0.1	0.1		POS 151	56	191	365	156	10
11	151	56	188	366	160	-96.806	312.2	103.194	-0.1	0.1		POS 151	56	188	366	160	11
12	151	56	183	373	162	-95.764	318.611	102.236	-0.1	0.1		POS 151	56	183	373	162	12
13	151	56	177	373	172	-94.689	318.064	93.311	-0.1	0.1		POS 151	56	177	373	172	13
14	151	56	170	373	-178							POS 151	56	170	373	-178	14
												POS 151	56	164	378	-178	15
												POS 151	56	157	376	-169	16
												POS 151	56	150	374	-160	17

Table 3

Table 3 shows the readings that were displayed on the software GUI's text boxes, for "Received Data from robot" and "Sent Data from Robot" and were manually copied into an Excel spreadsheet. The readings that were sent from the robot are shown on the left hand side of the table whereas the readings from the PC are shown on the right hand side of the table starting with ***POS 145***. The column of interest is the count value because this enables error checking of the data that was sent from the PC compared with the data that the robot sends back to the PC. In other words the count value that was sent from the PC should match the count value with a response sent back from the robot. We can see that the robot stops sending data when the count value reaches 14, but the PC still keeps sending data. This was the point where the observation of the robot spinning randomly in a circle was noticed.

This presented a major obstacle because it was preventing the robot from operating reliably. In order to debug and fix the problem various methods and attempts were made to try to tackle the root cause of the issue. The problem was originally thought to have something to do with the way the robot was transmitting data back to the PC, where the Wixel.printf() for sending the string out the serial port was used and Wixel.writeable() method checks to see if the buffer is empty which returns a value or true or false.

```
if (Wixel.writeable())
{
   Wixel.printf("%7d, %7.3f, %7.3f, %7.3f, %7.3f, %7.3f\r\n", msgCount, leftSpeed, rightSpeed, ballHead, robot
}
```

The locking issue was tested by checking to see if the robot was would stop transmitting its readings back to the GUI when the orange ball was suddenly removed from the field. Removing the ball from field means that the robot does not transmit any of the numbers back, and this was done several times until the locking issue occurred. The robot was connected to the programming cable off the field and tested using the ***Keil Complier Debugger,*** in an attempt to try to pin point the exact location or line in the programming code where the locking issue occurs***.*** There was also 2 colours placed on the field, which replicated the robot being on the field. The ball was then suddenly removed and placed back on the field various amount of times until the locking issue occurred again. When the locking issue occurred again the debugger's operation was halted and the pointer of the program would go off to some random place and not return, this issue happened after the Wixel.writeable() method had been executed. After hours spent trying to resolve the problem and looking through the mbed documentation the problem was finally resolved. This was due to the type of serial port library class that was being used for creating the Wixel object. Previously the following was being used.

```
Serial Wixel(p28, p27); // tx, rx
```

The documentation mentioned that the RawSerial class which contains the same methods as the Serial class could also be used. So therefore the following class was replaced with the class as follows.

```
RawSerial Wixel(p28, p27); // tx, rx
```

Now the robot was retested using this method for sending or transmitting data strings back to the PC and the locking issue no longer occurred.

To further optimise the performance of the robot instead of using the Wixel.printf() function for sending the data out to the PC, a purpose built function which basically sends the strings out one character at a time. It basically works by passing the length of the following string to the function, then pointer to where the data of the string in memory is located is also passed to the ***OutputString*** function. It uses the Wixel.putc(str[ptr]) for extracting the data from str[ptr].

```
// transmit serial string NOT USING INTERRUPTS
void OutputString(char* str, int length)
{
    //int length = strlen(str);
    int ptr = 0;

    while(ptr < length)
    {
      if(Wixel.writeable())
      {
        Wixel.putc(str[ptr]);
        ptr++;
      }
    }
}
```

References

AForge.NET. (2013). Retrieved from AForge.NET: http://www.aforgenet.com/framework/

ARM mbed NXP LPC1768 Development Board. (2014). Retrieved from Pololu Robotics & Electronics: http://www.pololu.com/product/2150

BitmapData.Scan0 Property. (2012). Retrieved from Microsoft: http://msdn.microsoft.com/en-us/library/system.drawing.imaging.bitmapdata.scan0(v=vs.110).aspx

Configuring Your Wixels. (2014). Retrieved from Pololu Robotics and Electronics : http://www.pololu.com/docs/0J46/4

Copy data from from IntPtr to IntPtr. (2013, April 12). Retrieved from Stackoverflow: http://stackoverflow.com/questions/15975972/copy-data-from-from-intptr-to-intptr

HSL and HSV. (2014). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/HSL_and_HSV

m3pi Robot + mbed NXP LPC1768 Development Board Combo. (2014). Retrieved from Pololu Robotics & Electronics: http://www.pololu.com/product/2153

Memory leak if Bitmap get's raw data from Marshal.AllocHGlobal? (2010). Retrieved from Stack Overflow: http://stackoverflow.com/questions/4531922/memory-leak-if-bitmap-gets-raw-data-from-marshal-allochglobal

Pololu. (2014). *Wixel Programmable USB Wireless Module (Fully Assembled)*. Retrieved from Pololu Robotics & Electronics: http://www.pololu.com/product/1336

Pololu 3pi Robot. (2014). Retrieved from Pololu Robotics and Electronics: http://www.pololu.com/product/975

WixelTest. (2012, March 11). Retrieved from ARMmbed: http://developer.mbed.org/users/aworsley/code/WixelTest/docs/fdee2dea0648/main_8cpp_source.html