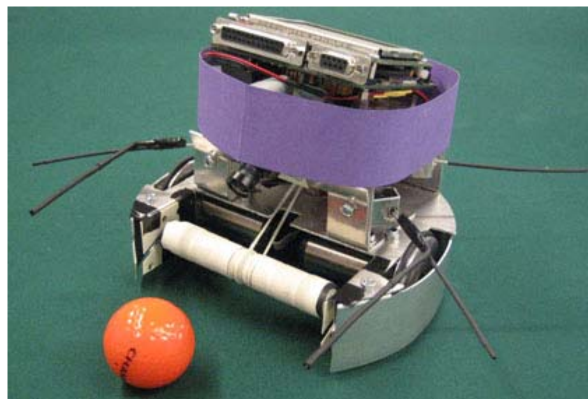
Robotics and Autonomous Systems 2D1426

Group RAS5 – Spring 2007

Project Report for

ZIDANE



Authors:

Salvatore Sannino sannino@kth.se
Fabio Sessa sessa@kth.se
Emil Sigursveinsson emilsi@kth.se
Khurram Yousaf khurramy@kth.se

Index

INDEX 2
ABSTRACT 3
1. INTRODUCTION 4
1.1 Group building activity 4
1.2 Project requirements 4
1.3 Understanding and Ideas 4
1.4 Project Timeline 5
2. ROBOT DESIGN 5
2.1 Mechanical Design 5
2.2 Electrical Design 7
2.2.1 Roller drive 7
2.2.3 Whiskers 8
3. ELECTRICAL COMPONENTS 8
3.1 EyeBot 9
3.2 Camera 10
3.3 Batteries 10
3.4 Motors 10
4. SOFTWARE 11
4.1 Software interconnections 11
4.2 Vision 11
4.3 Robot locomotion 13
4.3.1 Integrating drive 13
4.3.2 Drive scheme 14
4.4 High level strategy and flowchart 14
5. COMPETITION 16
5.1 Seeding 16
5.2 Competition results 16
6. CONCLUSIONS 16
APPENDIX – CODE 17
Image processing 17
Classify.h 17
Classify.c 19
Integrating drive 30
Drive.h 30
Drive.c 30
High level strategy 38
Strategy.h 38
Strategy.c 39
Main function 43
Ras5.c 43
REFERENCES 46

Abstract

This report presents various steps involved in the design and development of a soccer playing robot 'Zidane' which was made as a course project for 2D1426 – Robotics and Autonomous systems. We start by highlighting the background work done to initiate the project. Various aspects of mechanical and electrical design and constructions of the robot are then discussed. This is followed by a brief overview of the electrical components provided for robot constructions such as the controller, camera and motors. Major part of the report is dedicated to the programming of the robot. This comprises of development of different program modules including vision, locomotion and high-level strategy. The performance of our team is specified at the end.

1. Introduction

Robotics is a multi-disciplinary field which makes it very interesting. One of the most important aspects of robotics is that it brings theory into practice. Constructing robots involves mechanical construction, building electric circuits and computer programming. In this report various steps involved in the development of a soccer playing robot of the team RAS5 are given. The robot was named 'Zidane'.

1.1 Group building activity

For a team of students from different backgrounds it is always a good idea to start work with some group building activity. It helps the team members to bind together and work in a friendly fashion. Since all team members of RAS5 were from different origins group building activity in the beginning gave a good start.

1.2 Project requirements

The course Robotics and Autonomous Systems – 2D1426 requires a soccer playing robot to be built keeping in consideration a number of set rules. This robot should be able to score goals in an arena with a golf ball. A layout of the arena is given in Figure 1.1. The arena consists of two goals; Yellow and blue. The aim is to score goals with and without an opponent. There are certain design limitations such as size of the robot, handling of ball, type of controller and motors etc. Keeping in view all the limitations a robot has to be designed.

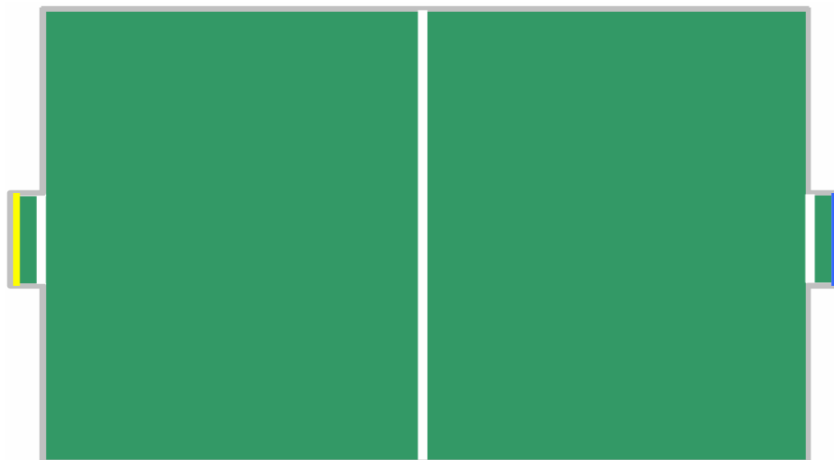


Figure 1.1 - Layout of arena

1.3 Understanding and Ideas

Before construction it is best to go through all the information in order to understand the problem. Table 1.1 lists the major restrictions that were kept in mind before design. The different functionalities that the robot was required to achieve like locomotion, sensing, dribbling, kicking, scoring etc. were also discussed. The detectables like ball, goal, line, opponent, wall etc. in the game field were highlighted. Dribbling involves locomotion with control of ball. Different possible dribbling and kicking ideas along with their possible construction were also discussed.

No.	Restriction
a.	Pass the course
b.	Autonomous robot
c.	Size limitation
d.	Cannot remove DOF of ball
e.	Ball area percentage outside convex hull
f.	2 minutes for goal
g.	No other drive motors
h.	No other main controller
i.	No other camera
j.	Color marker

Table 1.1 - Major Restrictions

1.4 Project Timeline

In order to finish the project in time and keep track of the team performance a timeline was introduced. The project timeline is given in Figure 1.2.

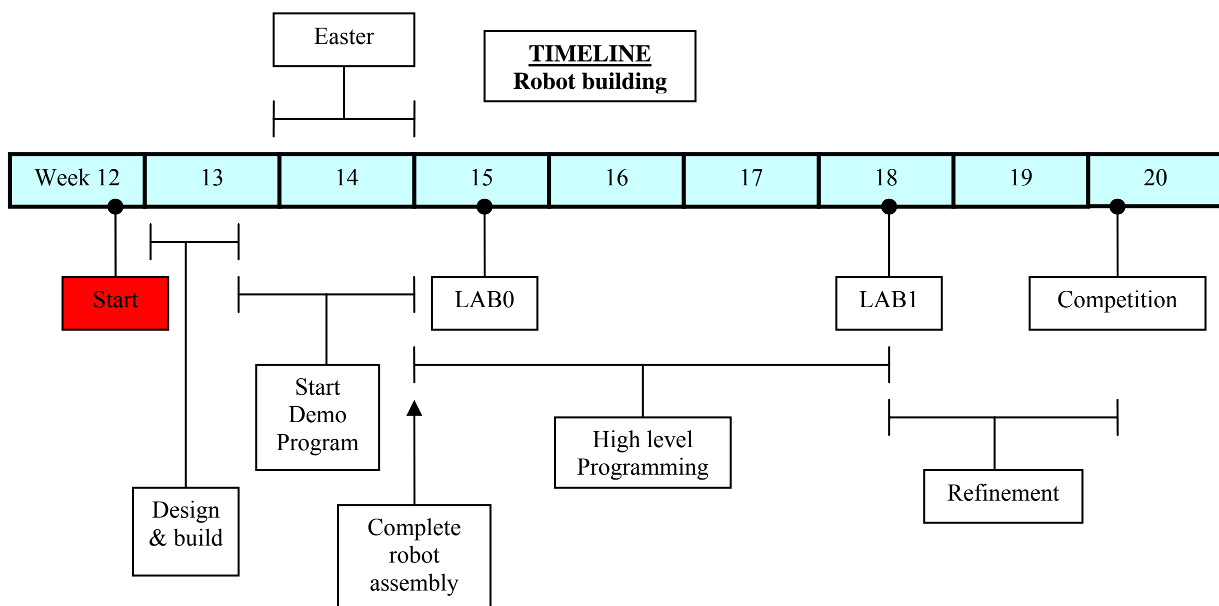


Figure 1.2 - Project Timeline

2. Robot Design

2.1 Mechanical Design

Regarding mechanical design what we wanted to do was to build a mechanical structure that was light and where it was easy to put all the components. The robot was firstly designed using a paperboard to avoid wasting aluminium sheet during the building. The robot is built on a single circular plate. The base of Zidane was built making an aluminium circle with a radius of 180 mm to respect specifications and then cutting some parts of this circle just to give to the robot a more functional shape. Firstly we cut the lateral parts of the circle so that wheels could stay in the allowed area. Another part of the circle was modelled to give space for the roller in the front of the

robot (we derived roller supports in the base without using separate strips). In the base we made some mounting points to fix the third support for locomotion and to mount the motors and the platform we used to fix EyeBot, camera, roller motor and whiskers. A little opening in the front was made to permit the rubber band passing from the motor to the roller without any disturbances. We created also a big hole to permit to the cables coming from the wheels and from the batteries to come up to the base. In fact we chose to put batteries down the base to avoid weight distribution on the robot.

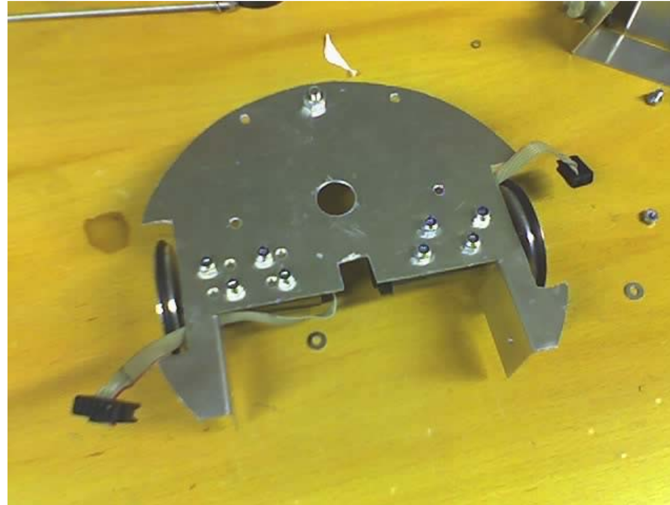


Figure 2.1 - The base of the robot

To protect wheels from collisions with walls and other robots we designed a “skirt”. This consists of two aluminium pieces bended in circular shape wrapping the robot base. To fix the skirt to the base we used the two mounting points in the back and we made also other two holes in the front of the robot. As said, we created a platform consisting of two aluminium strips to fix the other components. Firstly we used this platform to hold the EyeBot on top, in this way it was protected by collision and it was really simple to operate with the EyeBot during the tests and the competition.
To detect collisions we put four whiskers on our robot, they are fixed using a bent support that gives to them a good inclination just to permit to detect collisions coming from everywhere. Inclinated supports for whiskers are mounted in the middle of supports forming the platform. On one of the platform stirrups we fixed also the support for the roller motor.

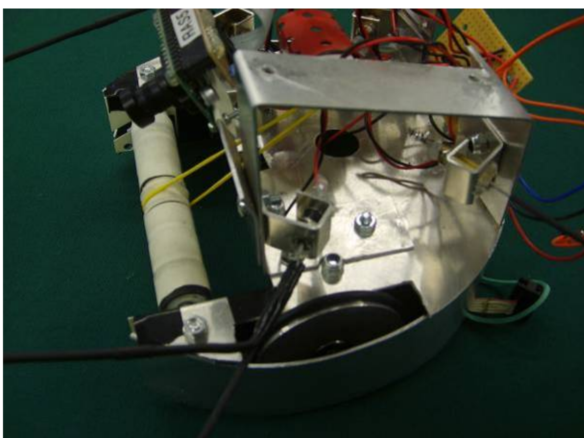


Figure 2.2 - Whiskers supports

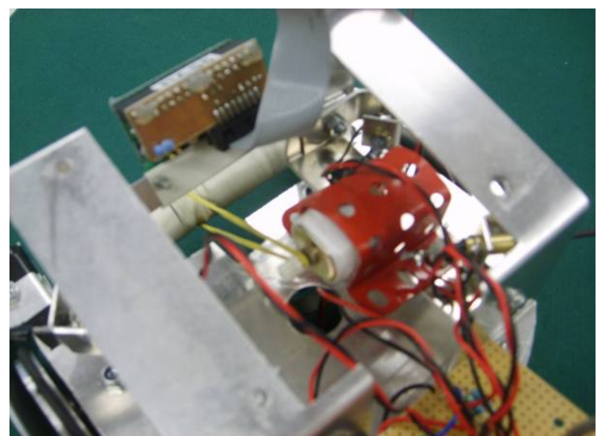


Figure 2.3 - DC Motor – Roller Coupling

A roller is used to be able to keep the ball in possession while driving, in fact it causes the ball to back-spin towards the robot. To make this we changed a lot of rollers because it was really difficult to find a material with a good friction. We tried a lot of different materials: the first we used was too

soft and the second one had too friction and the ball didn't stay on the roller at all. Finally our roller is made from the center plastic from two paint rollers, covered with couple of rounds of tape for larger diameter. To get more friction on the surface a small balloon was used, the contact between the roller and the ball is in the upper part of the ball. The roller is driven by a rubber band hanged from a DC motor. As roller supports are cut in the middle, this same rubber band is used to hold the roller in place, and this makes it really easy to remove the roller if maybe we needed to change the rubber band during the competition, as example, or also to test other rollers. We added also two little plates to achieve that the ball will leave the roller while driving.

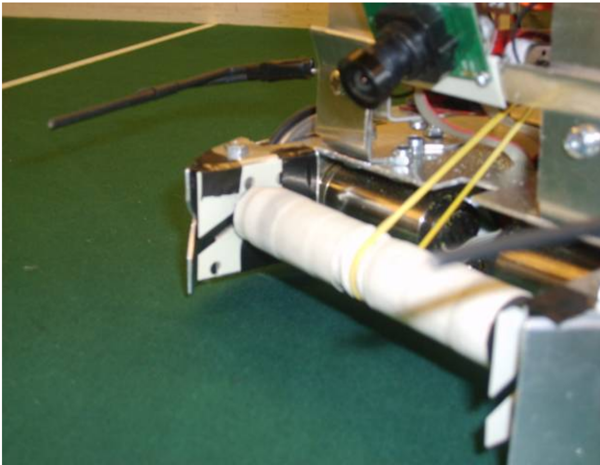


Figure 2.4 - Ruller Supports

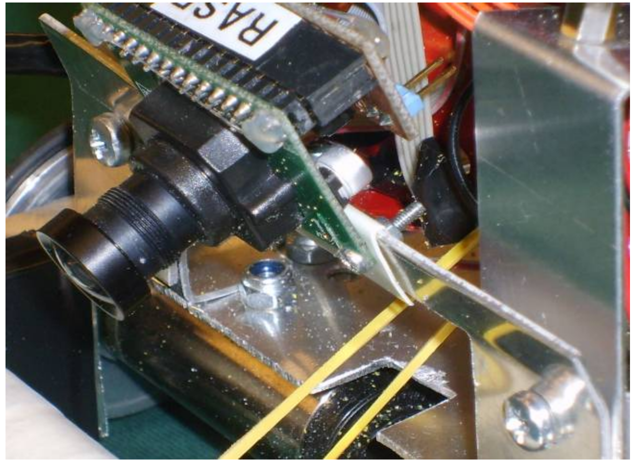


Figure 2.5 - Camera inclinated support

Regarding the camera, we fixed it building a new support that was mounted in front of the platform holding the EyeBot and other components. This support is inclinated so that the camera can see the full field quite good and it can see the ball even if it is on the roller.

2.2 Electrical Design

2.2.1 Roller drive

A transistor circuit was made to power the roller motor. A digital output pin from the Eyebot was used to turn the transistor on and off. The digital output pin gives either 0 or 5 volt output but cannot give enough current to drive the motor. A Darlington power transistor with high gain was used (hfe ≈ 750). This means that if the motor draws 1A the output pin only has to provide 1.33mA to turn the transistor fully on. The circuit can be seen in the figure 2.6.

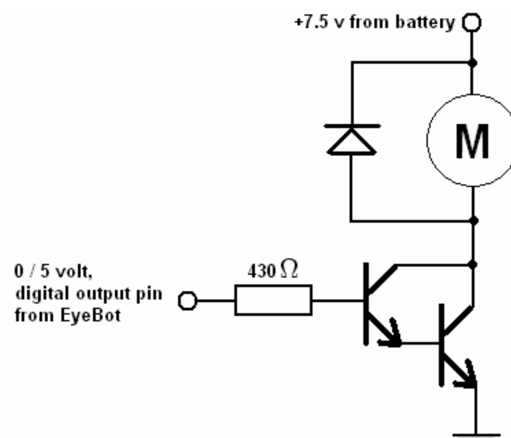


Figure 2.6 - Roller motor circuit

2.2.3 Whiskers

The whiskers work as switches which are normally open but when they hit something these switches close. Each whisker was connected to a digital input on the EyeBot as can be seen in figure 2.7. A 10kΩ pull-up resistor was used so a high value was at the inputs in normal operation.

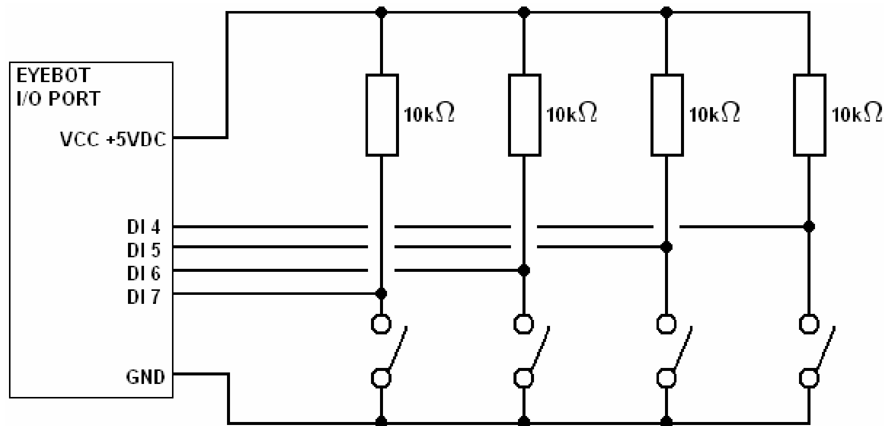


Figure 2.7 - Electrical circuit for the whiskers

The front whiskers are split into Y, this was done to be able to detect a high probability front collision without losing the ability to detect side collisions. The working mode is well commented in the drive section in the Appendix.

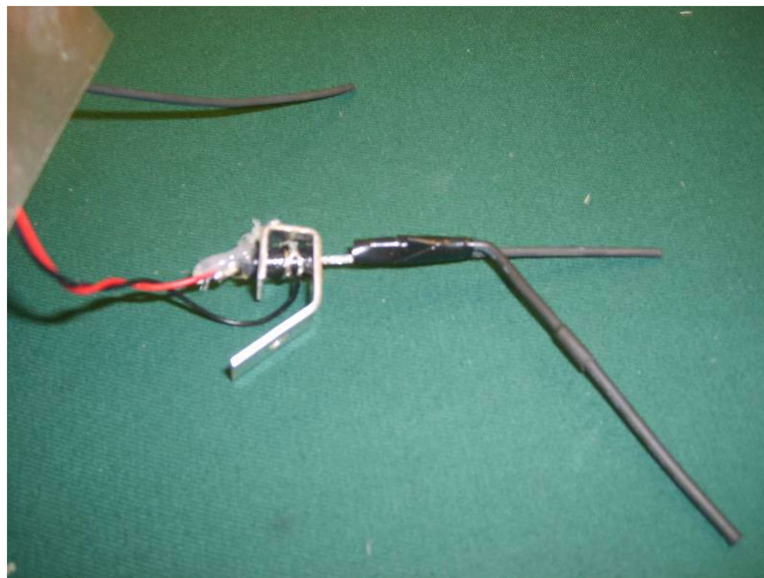


Figure 2.8 – Front Whisker

3. Electrical components

Components used for the project are: an EyeBot controller board, a camera, batteries and a charger, two high quality motors with encoders, gearboxes and wheels. Other components were available, such a servo motor, but we were leaning towards using less components as possible having easy circuitry, less cables and minimum size and weight.

3.1 EyeBot

EyeBot is an advanced controller for mobile robots. It can be used for mobile robots with wheels, walking robots or flying robots. It consists of a powerful 32-Bit microcontroller board with a graphics display and a digital color camera allowing it to do on-board image processing. Its operating system RoBios enables abstraction of the connected hardware making the board easy to program. Programs, written in C or in assembly language, can be downloaded via serial cable and a large LCD monitor enables on-screen info and batteries status. For further details see [4] and [5].

Technical Data:

- 35MHz 32bit Controller (Motorola 68332)
- 2MB RAM
- 512KB ROM (for system + user programs)
- 1 parallel port
- 1 serial port
- 8 digital inputs
- 8 digital outputs
- 8 analog inputs
- 2 motor drivers
- interface for color camera
- large graphics LCD
- 4 input buttons
- reset button, power switch
- speaker
- microphone
- battery level indicator

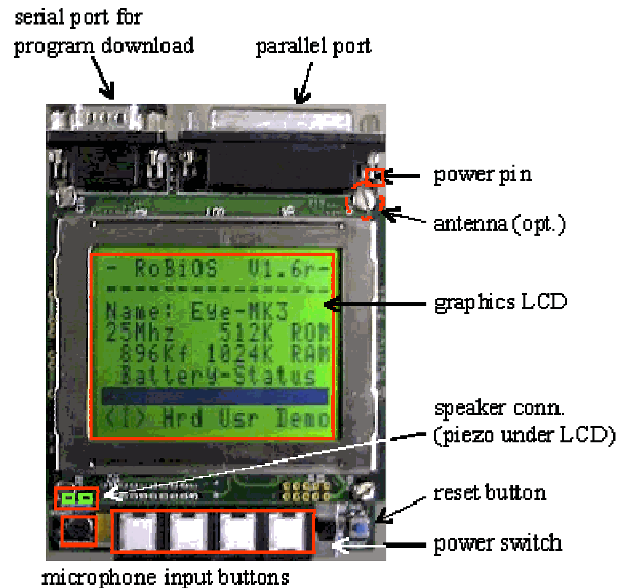


Figure 3.1 – EyeBot front view

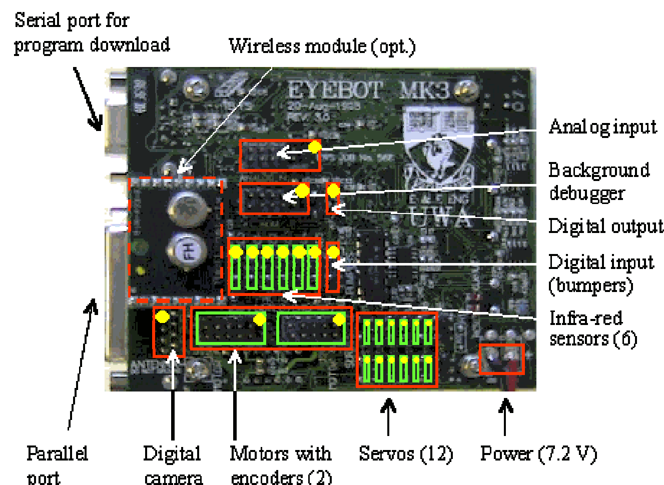


Figure 3.2 – EyeBot bottom view

3.2 Camera

The EyeCam is a 24 bit color digital camera to be used with the EyeBot controller. Its low resolution is sufficient for most robotics tasks and allow fast image processing. The camera has auto-brightness function that autonomously adjusts the brightness of the image to the light level. This caused some problems while classifying various objects like white lines and yellow-goal. The supplied camera had a wide angle lens which gave a vision angle of about ± 35°. This was the main reason why the camera was not put on a servo like many other groups did. It would only cause extra unnecessary complexity.

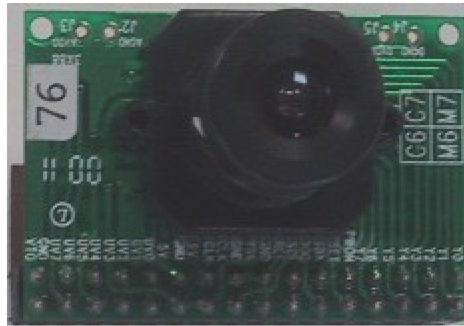


Figure 3.3 - Camera

3.3 Batteries

The EyeBot can use power supply ranging from 6 to 9 Volts. We used our battery package that consists of a series of six 1,2 Volts batteries. Package was located under the robot base to get more stability and was fixed with rubber bands to easy change the batteries. A transistor circuit was used to power the roller motor directly from the batteries.

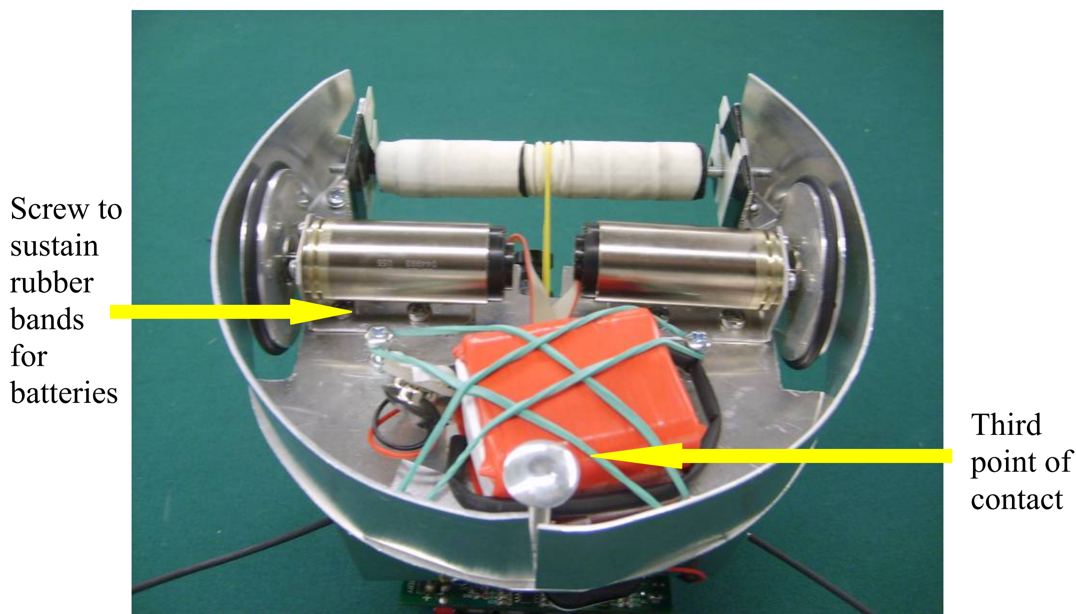


Figure 3.4 – Zidane bottom view

3.4 Motors

Two DC motors with encoders, gearbox and wheels were used in a differential drive configuration to drive the robot. Wheels were aligned on the diameter and a third point of contact was created on the back of the chassis. Motors were connected to the EyeBot through the two motor drivers, enabling us to retrieve information from the encoders. A third DC motor was used to drive the roller.

4. Software

4.1 Software interconnections

The code implemented for the robot was divided into four source files as follows:

- *classify.c* and *classify.h* for all camera and image processing functions.
- *drive.c* and *drive.h* for all functions used for putting the correct power to the wheels and driving the robot.
- *stategy.c* and *strategy.h* contain functions that know how to find objects, such as the goals and the ball and how to go to them.
- *ras5.c* contains the main function and the main state machine that tries to find the ball and move it into the opponent goal.

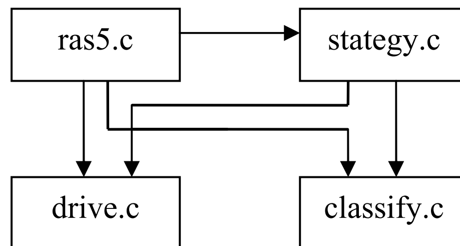


Figure 4.1 - Dependency of source files

4.2 Vision

The main sensor of the robot is a camera mounted at the front of the robot. The purpose of the camera is to classify and locate all the various objects on the field, such as the goals and the ball. All code needed for handling images from the camera, classifying objects in the image, giving their location and other camera-related functions were put in *classify.c* and *classify.h*.

The full frame image from the camera is 176 x 144 pixels and by calling a *RoBios* function an image from the camera could be copied into a memory block of size 176 x 144 x 3, i.e. each pixel is represented by three values. These values represent the red, green and blue (RGB) values of that pixel and can take values from 0 to 255. To make the task of classifying objects on the field easier, each item had its own colour. Thus we needed some way to map or classify the RGB values of each pixel into a known object. Our first solution was to transform the values into hue values using the *RGBtoHue* function found in the sample code *findball*.c. This function turned out to be quite unreliable. It has to go through four *if* statements for each pixel which takes to much processing power for the entire image and through experiments we found that the hue value of objects were overlapping and moving around due to light conditions and an auto brightness function of the camera.

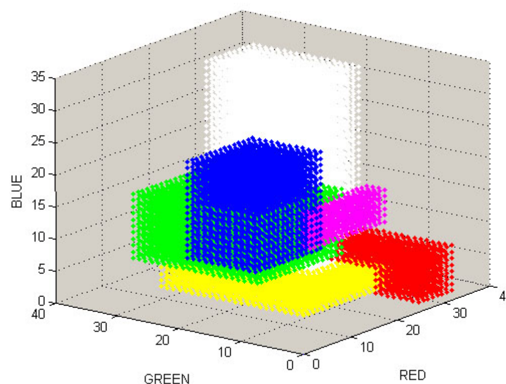


Figure 4.2 - 3 dimensional representation of the classification lookup-table.
Each colour represents one object on the field

The solution we used was to use a three dimensional classification lookup table where each dimension represents each of the RGB colours. Each colour is represented by one byte, but a 255 x 255 x 255 lookup table would take too much memory space. A 5 bit representation proved to be enough so our lookup table became of size 32 x 32 x 32. Each of the RGB values needed to be divided by eight to get the input values down to the 5 bit range. This was done in a fast way by shifting the bits three times to the right. Minimum and maximum RGB values for each object were defined in a header file and used to populate the classification table. This was done as a startup routine. Several pictures were taken with the camera and uploaded to the computer. There an image processing program called *The Gimp* was used to determine the minimum and maximum RGB values for each object.

The colour range of each object became quite large. This was necessary mostly due to an auto brightness function of the camera that we could not turn off. This problem did for example make the white lines on the field look like the yellow goal while the robot is in the yellow goal, but these same lines look white otherwise. A white balance function was implemented early on but proved to make things even worse due to the auto brightness and was thus discarded.

A simple approach was used to evaluate the centre of gravity of each recognizable object on the field. Every pixel of an image was checked through the classification table starting with the bottom pixels and moving up. If a pixel belonged to certain object the count of such pixels was increased and the x and y locations of such pixels were added together. When all pixels had been processed the summed up x and y values were divided by the number of such pixels to find the center of gravity in x and y coordinates of the image.

Some simple measures were made to decrease the risk of misclassifications. The reason why images were scanned from bottom to top was to make it easier to stop searching for the ball. We know that there is only supposed to be one ball on the field and the scanning program stops counting ball pixels if it does not find a ball pixel in the line after a line where such pixel was found. This reduces the risk of for example a person standing by the field in an orange t-shirt being classified as the ball. It was also known that the camera could never see the goals below a certain point due to its mounting and thus was no point in searching for goal pixels below this point. The final measure was to determine a minimum amount of pixels for each object. Goals had to have at least 150 pixels to be recognized as goals and the ball had to be at least 3 pixels, but that was the amount of pixels recognized as a ball when the robot was on one end of the field and the ball at the other. It turned out that more measures were needed to avoid wrong classifications. A good example is when the robot is close to the yellow goal, looking into it. Due to the image processing in the camera driver a mixture between the white goal line and the green field becomes orange like the ball. This is the reason why our robot sometimes thought the ball was in the yellow goal when it was not. Another problem occurred when competing against another robot. The shiny aluminium colour of other robots appeared as a colourful mixture of colours on images from the camera. These colours were in to many coactions misclassified as the ball.

Tests were made to transform x and y center of locations into angle and distance from the robot. A small test program was run on the robot showing the center of gravity of the ball. The ball was put in various locations on the field and the distance to the robot and angle from its center was measured. It turned out that a linear relation was between the angle and the horizontal center of gravity of the ball:

$$angle = (x/2) - 41$$

where x is the horizontal center of gravity. There was on the other hand an exponential relation between the vertical center of gravity and the distance to the ball. Instead of implementing power calculations on the EyeBot a distance lookup table was created. The angle measurements turned out to be quite accurate and the distance measurements were accurate close to the robot, up to 20cm.

When the ball was located further away a difference of one pixel in the vertical gravity center could mean a 40cm difference in distance.
A PC test code was made in order to test the classification code properly. This code was put in the same source file as the code which was to be run on the EyeBot. Compiler based if statements were used to determine if the code was to be compiled for the EyeBot or for the PC. This way we could be sure that the classification code running on the EyeBot was exactly the same as used for tests on the PC. The PC test code was used to test images uploaded from the EyeBot. It read each image into memory and stored them there exactly the same way they are stored in memory in the EyeBot. It then classified the images and wrote another image file showing how the original image was classified. This gave a visual representation of the classification which turned out to be quite helpful. The PC test code also reported the center of gravity and distances/angle to each of the object recognized form the given image. A sample image from the test code can be seen in figure 4.3.

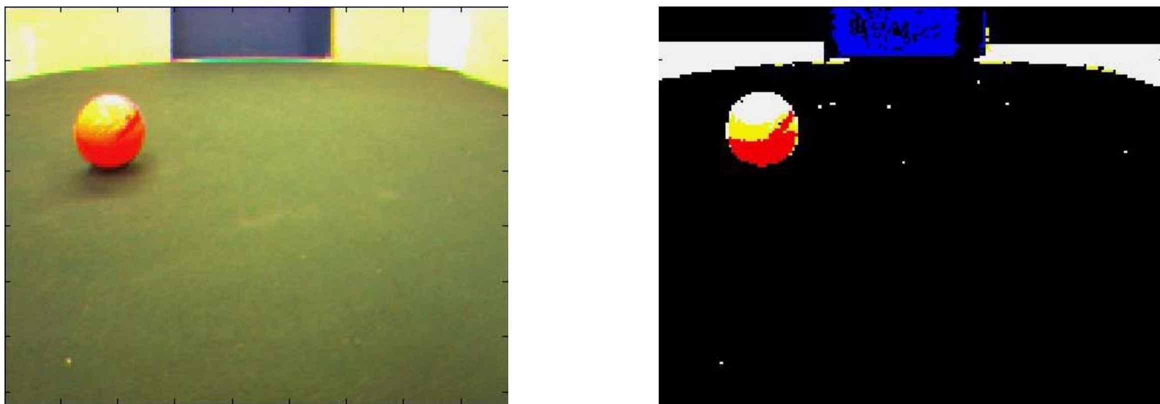


Figure 4.3 - A sample picture and the resulting classification

4.3 Robot locomotion

Customized code for robot locomotion was written instead of using the VW drive. Originally the idea of using customized locomotion functions was introduced to save computation time used up by the VW drive. It turned out that making a perfect locomotion scheme was not so simple and it had bugs. In later stages of the project there was not much time left to switch back to VW drive. However we observed that most of the other groups were using VW drive quite efficiently so probably it is a better idea to use it.

4.3.1 Integrating drive

The customized drive of the robot can achieve three different maneuvers. These are:

a. Go straight forward or backward
b. Turn in clockwise or counterclockwise direction
c. Turn in clockwise or counterclockwise direction with ball

The difference between b. and c. is that when the robot is turn(s) with a ball only one wheel is actuated as compared to when it turns without a ball, in which case both wheels are actuated. The above-mentioned maneuvers are implemented as separate functions. The inputs to these functions are the distance or angle required to be moved, the speed and collision detection enable/disable. In case of straight motion the units of distance are cm and the units of speed are mm/s. In case of both turn maneuvers the units of angle are degrees and the units of speed are degrees/s. Constant values cannot be allocated for different speeds due to change in battery power. So a look-up table will not work here. This is why a drive with an integrating effect has been introduced. What actually happens is that the speed of the robot is gradually increased until the required speed is achieved. This actually slows down the locomotion but the resulting displacement is quite accurate.

4.3.2 Drive scheme

The scheme of all three manoeuvres are similar in nature except the calculations for displacement and speed and the actuation values for the motors. Figure 4.4 gives program flowchart of the robot locomotion. The detailed code (drive.h and drive.c) are attached as Appendix. If a collision with an object is detected motion is stopped and the designated motion sequence for the collision is performed. A timeout is also introduced in the program to cope with stalling issues.

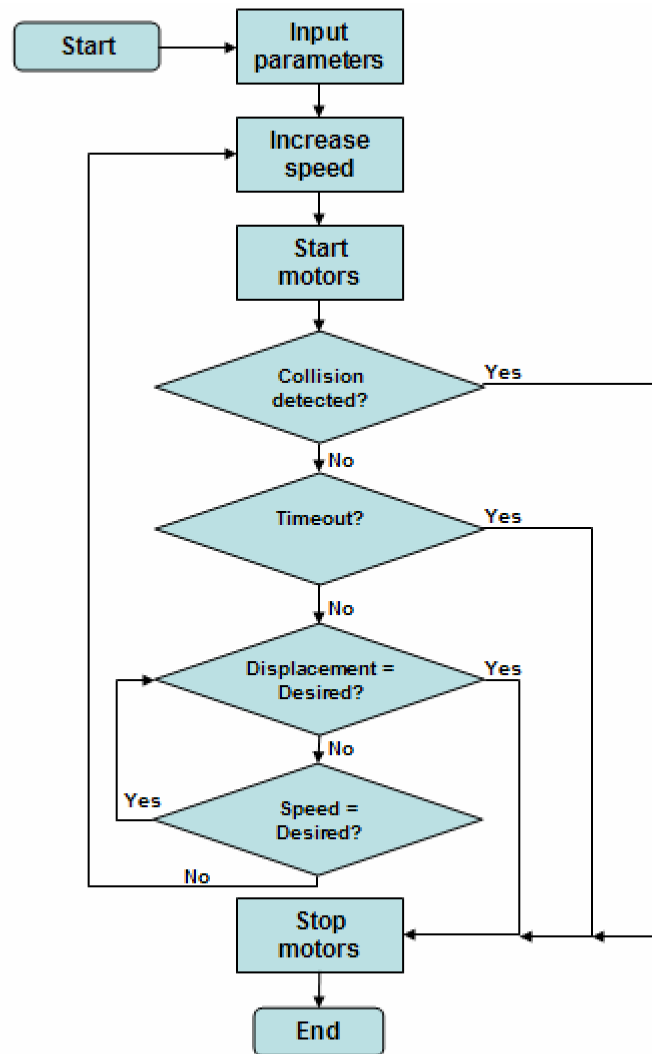


Figure 4.4 - Drive flowchart

4.4 High level strategy and flowchart

In figure 4.5 the flowchart of our strategy is illustred. The scoring goal strategy is as follows: first we find the ball and drive to it. Because we cannot take pictures while driving we drive some distance to the ball, stop and update our location, getting a new image, and then do another motion. When the robot has the ball in reach the roller is turned on and the ball is approached slowly so we don't kick it away. If we see our goal we rotate 180º. If we don't see any goal we rotate direction*30º until a goal is found. SelectRotationAngle is a image processing function that establishes the rotation direction counting the number of white pixels on both sides of the image. We decided to turn in the opposite direction of white pixels to avoid wall collisions. If we see the opponent goal we try to align the ball in a straight line between the goal and the robot. When this has been accomplished the roller is turned off an the robot drives at maximum speed to the ball to

kick it into the goal. After this we just assume that we have scored and drive a little bit backwards and start looking for the ball again.
This assumption caused a problem during the competition: Zidane had ball in posession and it was going to score, the opponent robot was blocking it, after the drive timeout Zidane stopped its motion and the ball did not have enough kinetic energy to roll to the goal and it stopped about 20 cm from it. We knew that assumption could give some problem, but our great problem was to configure drive, so we had less time to work on image processing to identify a goal-scored, or extract this information through whiskers and it also was late to use an IR-sensor to detect goal line, so we decided to use no feedback from the sensors and drive straight for a fixed distance.
Function details could be found in the section strategy in the Appendix where the code is well commented.

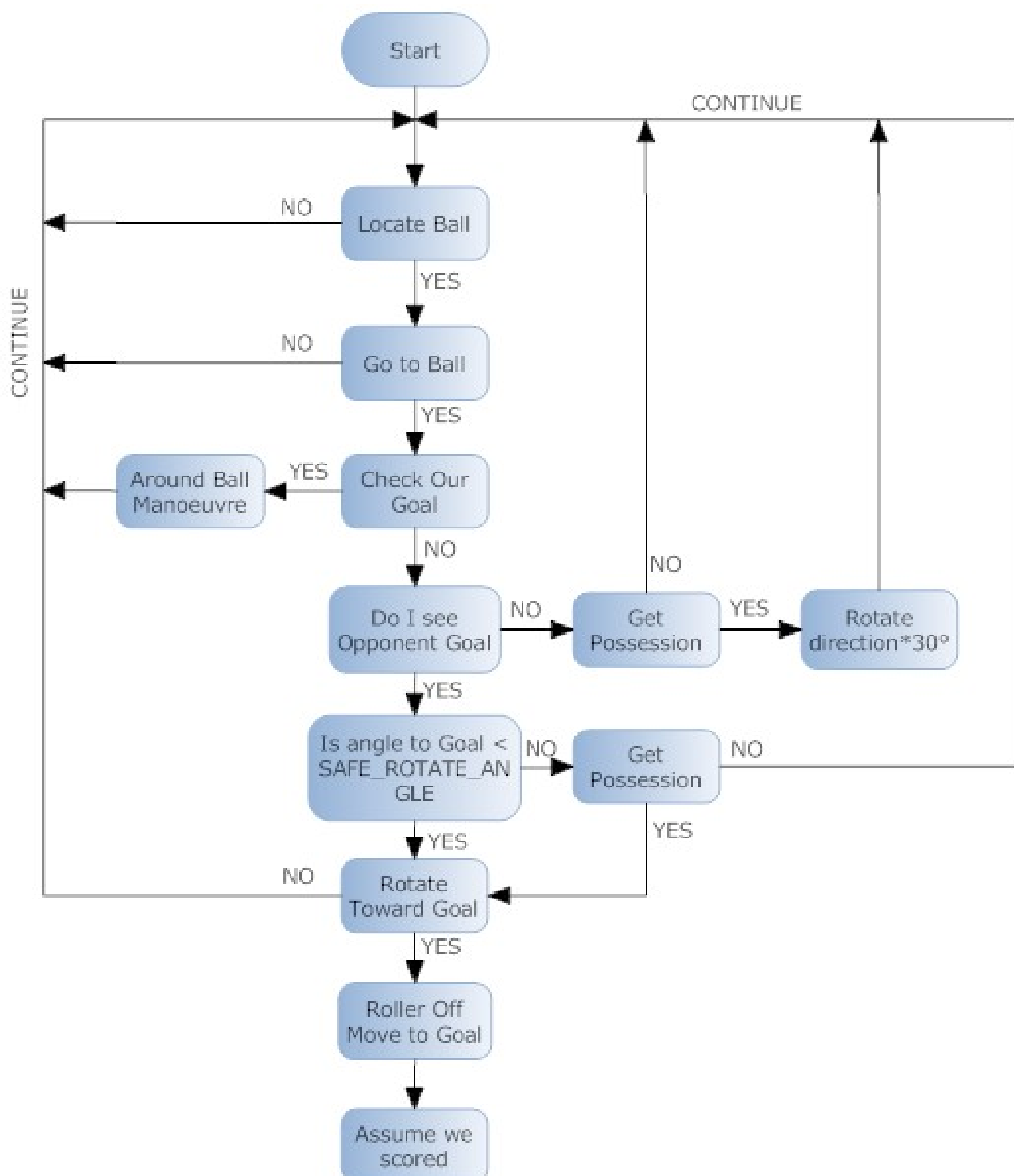


Figure 4.5 – Strategy stateflow

5. Competition

A competition was held at the end of the project. This involved a seeding round followed by a contest between teams based on the seeding.

5.1 Seeding

Seeding involved scoring unopposed. 3 minutes were given to each robot to score as many goals a possible. Once a goal is scored the ball is moved back to the centre line and the robot has to try to score again. Our robot ‘Zidane’ was able to score 1 goal in time and scored another goal 3 seconds after time over.

5.2 Competition results

We were not able to make it to the semifinals. Our robot was not able to score any goals against the opponents and both our opponents scored against us. So the score in all our matches was 0-1.

6. Conclusions

The performance of our robot in the competition was mediocre. Our image processing algorithm was very efficient but our locomotion algorithm had bugs. This ate up a lot of time. We think that if we had used VW drive our performance could have improved. Our robot was very slow to react because of slow locomotion algorithm. Another problem was that our vision system was non-functional during locomotion. This resulted in some actions that were not very smart. Overall we think that the best part of our program was the vision module.

Appendix – Code

Image processing

Classify.h

```
#ifndef CLASSIFY_H
#define CLASSIFY_H

#define EYEBOT
#define TRUE 1
#define FALSE 0
/* Define minimum and maximum values of R,G,B for each object */

#define BRL 210          //Ball Red Low CLASS 1
#define BRH 250          //Ball Red High
#define BGL 10           //Ball Green Low
#define BGH 154          //Ball Green High
#define BBL 10           //Ball Blue Low
#define BBH 60           //BAll Blue High

#define YGRL 102  //Yellow Goal CLASS 2
#define YGRH 250
#define YGGL 74
#define YGGH 240
#define YGBL 13
#define YGBH 19

#define BGRL 30          //Blue Goal CLASS 3
#define BGRH 125
#define BGGL 80
#define BGGH 155
#define BGBL 83
#define BGBH 184

#define FRL 54           //Green Field CLASS 4
#define FRH 181
#define FGL 96
#define FGH 240
#define FBL 61
#define FBH 123

#define ORL 133          //Opponent Purple stripe CLASS 5
#define ORH 240
#define OGL 88
#define OGH 103
#define OBL 81
#define OBH 105

#define WRL 190          //White lines CLASS 6
#define WRH 250
#define WGL 133
#define WGH 250
#define WBL 29
#define WBH 250

#define NUMBER_OF_CLASSES 6
#define NUMBER_OF_COLOURS 6
```

```c
#define LOWEST_PIXEL_FOR_GOAL 72  //Goals must be above this pixel
#define BALL_MIN_AREA 3
#define BALL_MIN_SEE_GOAL_DISTANCE 2
#define BALL_MAX_SEE_GOAL_DISTANCE 20
#define YELLOW_GOAL_MIN_AREA 170
#define BLUE_GOAL_MIN_AREA 3
#define MIN_GOAL_LINE_FOUND 176
#define MIN_GOAL_Y 100
#define MIN_OPPONENT_Y 80
#define Y_GOAL_INSIDE 100
#define B_GOAL_INSIDE 80

#define UNDEFINED 0
#define BALL 1
#define BLUEGOAL 2
#define YELLOWGOAL 3
#define FIELD 4
#define OPPONENT 5
#define WHITE_LINE 6

#define RED 0
#define GRE 1
#define BLU 2
#define FULL_IMAGE_WIDTH 176
#define HALF_FULL_IMAGE_WIDTH 88
#define FULL_IMAGE_HEIGHT 144
#define NOT_FOUND  -1

#define CLASSSIZE 32
#define DIVIDEBY8 3

#define LEFT -1
#define RIGHT 1

#if !defined(_BYTE)
#define _BYTE
typedef unsigned char BYTE;
#endif

typedef struct location
{
      int x;
      int y;
      int a;
      int distance;
      int angle;
}location;

//Function Declarations and public variables
void fillClassificationMatrix();
void fillLengthArray();
void findLocations();
void printLocations();
int getLength(int y);
int getAngle(int x);
unsigned char didWeScore(char goal);
int selectRotationalAngle(void);

location ball,ygoal,bgoal,white_line;

BYTE classifications[CLASSSIZE][CLASSSIZE][CLASSSIZE];
BYTE theimage[FULL_IMAGE_HEIGHT][FULL_IMAGE_WIDTH][3];
BYTE lengths[36];
```

```
int ygoalmaxy; //define the line where we think goal starts.
int bgoalmaxy;
int whiteLeftArea; //used for the selectRotationalAngle function.
int whiteRightArea;

//** Function Declaration and public variables FOR PC ONLY **
#ifndef EYEBOT
int writePPMImage(char * file);
char *ppm_get_token(FILE *fp, char *tok, int len);
int readPPMImage(char * file);
void Check(int x, int y);
void classifyTheImage();

BYTE classifiedImage[FULL_IMAGE_HEIGHT][FULL_IMAGE_WIDTH];


#else
//** Function Declaration and public variables FOR EYEBOT ONLY **
void error(char *str);
int initCamera();
void getImage();
void drawLocations();

#endif

#endif
```

Classify.c

```
/*****************************************************************************
classify.c
contains functions needed to comunicate with the camera and image processing
functions.  If the simbol EYEBOT is defined in classify.h the code can be
compiled
for the EyeBot.  If it is not defined then the code can be compiled for the PC
to test the classifing functions.
*****************************************************************************/

#include <stdio.h>
#include "classify.h"
#ifdef EYEBOT
#include <eyebot.h>
#endif


//Global variables
BYTE classifications[CLASSSIZE][CLASSSIZE][CLASSSIZE];
BYTE theimage[FULL_IMAGE_HEIGHT][FULL_IMAGE_WIDTH][3];
BYTE lengths[36];
location ball,ygoal,bgoal,opponent, white_line;
int ygoalmaxy; //define the line where we think goal starts.
int bgoalmaxy;
int whiteLeftArea; //used for the selectRotationalAngle function.
int whiteRightArea;

#ifndef EYEBOT
BYTE classifiedImage[FULL_IMAGE_HEIGHT][FULL_IMAGE_WIDTH];
#endif


/*****************************************************************************
```

```
void fillClassificationMatrx
Fills the classification matrix.  The minimum and maximum RGB values for eaach
object are predefined in the header file.
******************************************************************************/
void fillClassificationMatrix()
{
      int x,y,i,j,k;

      /*put known values into the classes matrix*/
      int classes[NUMBER_OF_COLOURS][NUMBER_OF_CLASSES] =
             //Ball  YGoal    BGoal    Field     Oppon WhiteLine
            {{BRL,      YGRL, BGRL, FRL,  ORL,  WRL}, //Red min
             {BRH,      YGRH, BGRH, FRH,  ORH,  WRH}, //Red max
             {BGL,      YGGL, BGGL, FGL,  OGL,  WGL}, //Green min
             {BGH,      YGGH, BGGH, FGH,  OGH,  WGH}, //Green max
             {BBL,      YGBL, BGBL, FBL,  OBL,  WBL}, //Blue min
             {BBH,      YGBH, BGBH, FBH,  OBH,  WBH}};     //Blue max

      int values[NUMBER_OF_CLASSES] = {BALL, YELLOWGOAL, BLUEGOAL, FIELD,
OPPONENT, WHITE_LINE};

      #ifndef EYEBOT
            printf("\n Filling Classification Matrix...");
      #endif


      /*Move range of values from 0-255 to 0-32*/
      for(x=0;x<NUMBER_OF_COLOURS;x+=2){
            for(y=0;y<NUMBER_OF_CLASSES;y++)
            {
                  classes[x][y]=(classes[x][y] >> DIVIDEBY8);
            }
      }
      for(x=1;x<NUMBER_OF_COLOURS;x+=2){
            for(y=0;y<NUMBER_OF_CLASSES;y++)
            {
                  classes[x][y]=(classes[x][y] >> DIVIDEBY8)+1;
            }
      }

      /*PRINT THE MATRIX, REMOVED FOR EYEBOT*/
      #ifndef EYEBOT
      for(x=0;x<NUMBER_OF_COLOURS;x++)
      {
            printf("\n");
            for(y=0;y<NUMBER_OF_CLASSES;y++)
            {
                  printf("%d\t",classes[x][y]);
            }
      }
      #endif

      /*Set all values in the matrix to zero*/
      for(i=0;i<CLASSSIZE;i++)
            for(j=0;j<CLASSSIZE;j++)
                  for(k=0;k<CLASSSIZE;k++)
                        classifications[i][j][k]=UNDEFINED;

      /*Update the classification matrix for known types*/
      for(x=0;x<NUMBER_OF_CLASSES;x++)  //once for each object( ball, goals,
                                        //     etc..)  i = 1:objects
          for(i=classes[0][x];i<classes[1][x];i++) //RED
              for (j=classes[2][x];j<classes[3][x];j++) //GREEN
```

```
                for (k=classes[4][x];k<classes[5][x];k++)  //BLUE
                    classifications[i][j][k]=values[x];

        #ifndef EYEBOT
        printf("\nTEST FROM classification matrix:%d",classifications[16][16][2]);
        #endif
}

/*********************************************************************************
void findLocations()
Scans through an full image found in the global matrix "theimage[][][]" It
counts
pixels belonging to objects in the image (ball, goals, ..) and calculates there
center of gravity.  It also fills in estimated lengths and angles to these
objects.
The newly found locations are then stored in global variables of type "location"
*********************************************************************************/
void findLocations()
{
        int i,j;
        int r,g,b;
        BYTE BallInLastLine = 0;
        BYTE BallPixelFound = 0;
        BYTE BallFound = 0;

        ball.x=0; ball.y=0; ball.a=0;
        ygoal.x=0; ygoal.y=0; ygoal.a=0;
        bgoal.x=0; bgoal.y=0; bgoal.a=0;
        opponent.x=0; opponent.y=0; opponent.a=0;
        white_line.x=0; white_line.y=0; white_line.a=0;

        whiteLeftArea = 0; //used for the selectRotationalAngle function.
        whiteRightArea = 0;

        bgoalmaxy = NOT_FOUND;
        ygoalmaxy = NOT_FOUND;

        /*Start searching at the bottom of the image.*/
        for(i=FULL_IMAGE_HEIGHT-1;i>=0;i--)
        {
                BallInLastLine = BallPixelFound;
                BallPixelFound = 0;

                for(j=0;j<FULL_IMAGE_WIDTH;j++)
                {
                        r = (theimage[i][j][RED]) >> DIVIDEBY8;
                        g = (theimage[i][j][GRE]) >> DIVIDEBY8;
                        b = (theimage[i][j][BLU]) >> DIVIDEBY8;
                        switch(classifications[r][g][b])
                        {
                                case BALL:
                                        if(!BallFound)//only one ball on the field.
                                        {                   //other stuff must be noise.
                                                            //and we are more likely to
                                                            //find the ball on the lower
                                                            //part of the image.
                                                ball.x += j;
                                                ball.y += i;
                                                ball.a++;
                                                BallPixelFound = 1;
                                                #ifndef EYEBOT
                                                classifiedImage[i][j] = BALL;
                                                #endif
```

```
                }
                break;

            case YELLOWGOAL:
                if(i < MIN_GOAL_Y){
                    ygoal.x +=j;
                    ygoal.y +=i;
                    ygoal.a++;
                    #ifndef EYEBOT
                    classifiedImage[i][j] = YELLOWGOAL;
                    #endif
                    if(ygoalmaxy==NOT_FOUND)
                        if(ygoal.a > MIN_GOAL_LINE_FOUND )
                            ygoalmaxy = i;

                }
                break;

            case BLUEGOAL:
              if(i < MIN_GOAL_Y){
                    bgoal.x +=j;
                    bgoal.y +=i;
                    bgoal.a++;
                    #ifndef EYEBOT
                    classifiedImage[i][j] = BLUEGOAL;
                    #endif
                    if(bgoalmaxy==NOT_FOUND)
                        if(bgoal.a > MIN_GOAL_LINE_FOUND )
                            bgoalmaxy = i;

              }
                break;
            case OPPONENT:
              if(i < MIN_OPPONENT_Y){
                    opponent.x +=j;
                    opponent.y +=i;
                    opponent.a++;
                    #ifndef EYEBOT
                    classifiedImage[i][j] = OPPONENT;
                    #endif
              }
                break;


            case WHITE_LINE:
                if(bgoalmaxy==NOT_FOUND && ygoalmaxy==NOT_FOUND){
                    white_line.x +=j;
                    white_line.y +=i;
                    white_line.a++;
                    if(j < HALF_FULL_IMAGE_WIDTH){
                        whiteLeftArea++;
                    }
                    else{
                        whiteRightArea++;
                    }
                    #ifndef EYEBOT
                    classifiedImage[i][j] = WHITE_LINE;
                #endif
                }
            break;

        }
    }
```

```
            if(!BallFound && !BallPixelFound && BallInLastLine &&
ball.a>BALL_MIN_AREA)
                  BallFound = 1;
      } // Finiched scanning the image.

      /*** Finished scanning do end functions ***/

      // Regarding the ball
      if(BallFound){
            ball.x = ball.x/ball.a;
            ball.y = ball.y/ball.a;
            ball.distance = getLength(ball.y);
            ball.angle = getAngle(ball.x);
      }
      else{
            ball.x = NOT_FOUND; ball.y = NOT_FOUND;
      }

      //Regarding the goals
      if(bgoal.a > BLUE_GOAL_MIN_AREA){
            bgoal.x = bgoal.x/bgoal.a;
                bgoal.y = bgoal.y/bgoal.a;
            bgoal.distance = getLength(bgoal.y);
            bgoal.angle = getAngle(bgoal.x);
                ygoal.x = NOT_FOUND;
                ygoal.y = NOT_FOUND;
      }
      else {
            bgoal.x = NOT_FOUND;
            bgoal.y = NOT_FOUND;
            if(ygoal.a > YELLOW_GOAL_MIN_AREA){
                  ygoal.x = ygoal.x/ygoal.a;
                  ygoal.y = ygoal.y/ygoal.a;
                  ygoal.distance = getLength(ygoal.y);
                  ygoal.angle = getAngle(ygoal.x);
            }
            else{
                  ygoal.x = NOT_FOUND;
                  ygoal.y = NOT_FOUND;
            }
      }

      #ifndef EYEBOT
            printf("\nBall found at: x:%d y:%d with area:%d",
ball.x,ball.y,ball.a);
            printf("\nYellow Goal: x:%d y:%d area:%d ygoalmaxy: %d",
ygoal.x,ygoal.y,ygoal.a,ygoalmaxy);
            printf("\nBlue goal: x:%d y:%d with area:%d, bgoalmaxy: %d",
bgoal.x,bgoal.y,bgoal.a,bgoalmaxy);
            printf("\nWhite line: x:%d y:%d with area:%d",
white_line.x,white_line.y,white_line.a);
      #endif

}

/*******************************************************************************
fillLengthArray()  Fills the public "lengths[]" array with predefined values.

The lengts array is a lookup table for distances of the ball.  The index of each
value corresponds to which y-pixel of the camera the center of gravity for the
ball corresponds to.
*******************************************************************************/
```

```
void fillLengthArray()
{
   int i;
        BYTE leng[] =
{180,180,180,171,133,102,77,58,44,33,26,20,16,14,12,11,11,10,10,9,9,8,7,6,5,4,3,
2,2,1,1,1,0,0,0,0};
   for(i=0;i<36;i++)
        lengths[i]=leng[i];
}

/*******************************************************************************
Small functions to get length and angle for given x and y values of the center
of gravity.
*******************************************************************************/
int getLength(int y){return lengths[y>>2];}
int getAngle(int x){return (x>>1)-41;}


/*******************************************************************************
unsigned char didWeScore(char pgoal).  Used to check if the ball is within the
goal line, or if we can see alot of the goal and the ball at the same time.
Used to determine if a goal has been scored.  (This function was never used)
*******************************************************************************/
unsigned char didWeScore(char pgoal)
{
      location goal;
      int lowestGoalYPixel;
      int lowestGoalYcomparePixel = 50;
      int x,y;
      int i;
      int r,g,b;

      switch(pgoal){
      case YELLOWGOAL:
                  goal = ygoal;
                  lowestGoalYPixel = ygoalmaxy;
                  lowestGoalYcomparePixel = 50;
                  #ifndef EYEBOT
                  printf("\nChecking yellow goal for score...");
                  #endif
      break;
            case BLUEGOAL:
                  goal = bgoal;
                  lowestGoalYPixel = bgoalmaxy;
                  lowestGoalYcomparePixel = 50;
                  #ifndef EYEBOT
                  printf("\nChecking blue goal for score...");
                  #endif
        break;
        default:
            return 0;
    }//switch

    x = ball.x;
    y = ball.y;
    if(x != NOT_FOUND && lowestGoalYPixel > lowestGoalYcomparePixel){
      if(white_line.a > 100){ //we can see white
            for(i=0;i<10;i++){ // search down for white line
                        r = (theimage[y+i][x][RED]) >> DIVIDEBY8;
                        g = (theimage[y+i][x][GRE]) >> DIVIDEBY8;
                        b = (theimage[y+i][x][BLU]) >> DIVIDEBY8;
                        if(classifications[r][g][b] == WHITE_LINE){
                              return 1;
```

```
                        }//if
            }//for
            }//if
      else if(white_line.a < 10){ //we assume that we are inside the goal
            return 1;
            }
    }//big if
      return 0;
}

int selectRotationalAngle(void)
{
      if(whiteLeftArea < whiteRightArea)
            return RIGHT;
      else
            return LEFT;
}


/*
################################################################################
################################################################################
################################################################################

THE FUCTIONS BELOW THIS POINT ARE NOT TO BE RUN ON THE PC.
THEY ARE USED ON THE EYEBOT ONLY.

################################################################################
################################################################################
################################################################################
*/
#ifdef EYEBOT

/*******************************************************************************
If an Error is found, run this code, it will stop the program
*******************************************************************************/
void error(char *str)
{
  LCDPrintf("ERROR: %s\n",str);
  OSWait(200);
  exit(0);
}

/*******************************************************************************
Initializes the camera and checks for errors
*******************************************************************************/
int initCamera()
{
  int camera;
  camera=CAMInit(NORMAL);
  if (camera==NOCAM) error("No camera!\n");
  else if (camera==INITERROR) error("CAMInit!\n");
  else if (camera < COLCAM) error("No colour camera!");
  //printf("camera code is %d pressed \n",camera);
  return camera;
}

/*******************************************************************************
Gets the big picture and puts it in "theimage" global matrix
*******************************************************************************/
void getImage()
{
      BYTE *buf = (BYTE*) &theimage;
```

```
      CAMGetFrameRGB(buf);
}

/****************************************************************************
Prints the location of known objects defined in the global variables
****************************************************************************/
void printLocations()
{
      //Ball
      if(ball.y != -1){
            LCDSetPrintf(1,0,"BA:x%3dy%3da%3d",ball.x,ball.y,ball.a);
            LCDSetPrintf(2,0,"dis:%3d ang:%3d",ball.distance,ball.angle);
      }
      else{
            LCDSetPrintf(1,0,"Ball not found");
      }

    //Yellow Goal
    if(ygoal.y != -1){
        LCDSetPrintf(3,0,"YG:area:%d",ygoal.a);

      LCDSetPrintf(4,0,"d:%3da:%3dy:%3d",ygoal.distance,ygoal.angle,ygoalmaxy);
    }
    else{
        LCDSetPrintf(3,0,"Ygoal not found");
      }

      //Blue Goal
      if(bgoal.y != -1){
            LCDSetPrintf(5,0,"BG:area:%d",bgoal.a);

      LCDSetPrintf(6,0,"d:%3da:%3dy:%3d",bgoal.distance,bgoal.angle,bgoalmaxy);
      }
      else{
            LCDSetPrintf(5,0,"Bgoal not found");
      }

      if(white_line.y != -1){
                  LCDSetPrintf(7,0,"WL:area:%d",white_line.a);
            }
            else{
                  LCDSetPrintf(5,0,"Bgoal not found");
      }


      if(didWeScore(YELLOWGOAL)){
            LCDSetPrintf(5,0,"...yellowGOAL...");
                  OSWait(10);
      }
      if(didWeScore(BLUEGOAL)){
           LCDSetPrintf(3,0,"...BLUEGOAL...");
              OSWait(10);
      }



}

/****************************************************************************
Draws the locations of known objects defined in the global variables
on the eyebot lcd.
****************************************************************************/
void drawLocations()
```

```
{
	int x,y;

	LCDClear();

	if(ball.a != 0)
	{
		x = ball.x*132/182;
		y = ball.y*255/461;
		LCDLine(0,y,127,y,1);
		LCDLine(x,0,x,63,1);
	}
}


#endif

/*
##########################################################################
##########################################################################
##########################################################################

THE FUCTIONS BELOW THIS POINT ARE NOT TO BE RUN ON THE EYEBOT.
THEY ARE USED ON THE PC.

##########################################################################
##########################################################################
##########################################################################
*/

#ifndef EYEBOT


/**************************************************************************
void classifyTheImage()
*************************************************************************/
void classifyTheImage()
{
	int i,j;
	//int r,g,b;

		//printf("\nClassifying image");

	for(i=0;i<FULL_IMAGE_HEIGHT;i++){
			for(j=0;j<FULL_IMAGE_WIDTH;j++)
			{
				/*r = (theimage[i][j][RED]) >> DIVIDEBY8;
				g = (theimage[i][j][GRE]) >> DIVIDEBY8;
				b = (theimage[i][j][BLU]) >> DIVIDEBY8;
				classifiedImage[i][j] = classifications[r][g][b];*/
				classifiedImage[i][j] = UNDEFINED;
			}
	}
}


/**************************************************************************
/* ppm_get_token: read token from PPM file in stream fp into "char tok[len]" */
/*************************************************************************/
char *ppm_get_token(FILE *fp, char *tok, int len) {
    char *t;
    int c;
```

```
    for (;;) {                  /* skip over whitespace and comments */
      while (isspace(c = getc(fp)));
      if (c!='#') break;
      do c = getc(fp); while (c!='\n' && c!=EOF);     /* gobble comment */
      if (c==EOF) break;
    }

    t = tok;
    if (c!=EOF) {
      do {
          *t++ = c;
          c = getc(fp);
      } while (!isspace(c) && c!='#' && c!=EOF && t-tok<len-1);
      if (c=='#') ungetc(c, fp);    /* put '#' back for next time */
    }
    *t = 0;
    return tok;
}

/*******************************************************************************
int readPPMImage(char * file) reads an PPM image into memory.  Arranged in the
same way as images in the EyeBot memory from the camera.
*******************************************************************************/
int readPPMImage(char * file)
{
      FILE *fp;
      char tok[20];
      int width,height;
      BYTE r,g,b;
      int x,y;
      int j;

      /* open PPM file */
      if ((fp = fopen(file, "r")) == NULL) {
      fprintf(stderr, "can't read PPM file %s\n", file);
      return 0;
      }

      /* read PPM header*/
      ppm_get_token(fp, tok, sizeof tok);
      sscanf(ppm_get_token(fp, tok, sizeof tok), "%d",&width);
      //printf("\nWidth: %d",width);
      sscanf(ppm_get_token(fp, tok, sizeof tok), "%d",&height);
      //printf("\nHeight: %d",height);
      ppm_get_token(fp, tok, sizeof tok);

      /* read RGB values */
      j=width;
      //printf("\n\nTEST: %d",j);
      y=0;
      while(y < FULL_IMAGE_HEIGHT)
      {
            x=0;
            while(x < FULL_IMAGE_WIDTH)
            {
                  sscanf(ppm_get_token(fp, tok, sizeof tok), "%d",&r);
                  theimage[y][x][RED] = (BYTE)(r);
                  sscanf(ppm_get_token(fp, tok, sizeof tok), "%d",&g);
                  theimage[y][x][GRE] = (BYTE)(g);
                  sscanf(ppm_get_token(fp, tok, sizeof tok), "%d",&b);
                  theimage[y][x][BLU] = (BYTE)(b);
                  x++;
            }
```

```
            y++;
        }
        fclose(fp);
    return 1;
}

/*****************************************************************************
void Check(int x, int y) Checks the value of a given pixel.
*****************************************************************************/
void Check(int x, int y)
{
        printf("\n r:%f, g:%f, b%f",
(float)theimage[y][x][RED]/255,(float)theimage[y][x][GRE]/255,
(float)theimage[y][x][BLU]/255);
}

/*****************************************************************************
int writePPMImage(char * file) Writes the classified image into a new ppm image.
*****************************************************************************/
int writePPMImage(char * file)
{
        FILE *ppm;
        int x,y;

        char * undef = "0 0 0";
        char * ball = "240 0 0";
        char * ygoal = "240 240 0";
        char * bgoal = "0 0 240";
        char * field = "0 240 0";
        char * opponent = "240 0 240";
        char * whiteline = "240 240 240";

        char *coding[7];
        coding[0] = undef;
        coding[1] = ball;
        coding[2] = bgoal;
        coding[3] = ygoal;
        coding[4] = field;
        coding[5] = opponent;
        coding[6] = whiteline;


        //printf("\n Writing to file: %s ",file);


        /* Open the file for output */
        ppm = fopen(file, "w");
        if( !ppm )
        return 0;

        /* Always write a raw PPM file */
        fprintf(ppm, "P3\n %d %d\n 255\n",FULL_IMAGE_WIDTH , FULL_IMAGE_HEIGHT);

        y=0;
        while(y < FULL_IMAGE_HEIGHT)
        {
                x=0;
                while(x < FULL_IMAGE_WIDTH)
                {
                        fprintf(ppm, "%s\n",coding[classifiedImage[y][x]]);
                        x++;
                }
                y++;
```

```
      }

      fclose(ppm);
    return 1;
}

#endif
```

Integrating drive

Drive.h

```
#ifndef DRIVE_H
#define DRIVE_H

#include "eyebot.h"
#include "strategy.h"

#define TIME_OUT_1      500

//definitions for goStraight()
#define CLICKS_PER_CM 60
#define CLICKS_PER_MM 6
#define ST_SPEED_FACTOR 7
#define ST_SPEED_DELAY 20

//definition for turn() and turnWithBall()
#define CLICKS_PER_DEGREE 8
#define CLICKS_PER_TRWB_DEGREE 16
#define TR_SPEED_FACTOR 7
#define TR_SPEED_DELAY 20

//motor and encoder handles
QuadHandle quadL, quadR;
MotorHandle motorL, motorR;
BOOL whiskers_flag, collisionMotion_flag;
BYTE latch;

//locomotion functions
void initMotion(void);
void releaseMotion(void);
BOOL goStraight(int distance, char speed, BOOL check_collission);
BOOL actual_goStraight(int distance, char speed, BOOL check_collission);
BOOL turn(int angle, char omega, BOOL check_collission);
BOOL actual_turn(int angle, char omega, BOOL check_collission);
BOOL turnWithBall(int angle, char omega, BOOL check_collission);
BOOL actual_turnWithBall(int angle, char omega, BOOL check_collission);
BOOL check_whiskers(void);

#endif
```

Drive.c

```
/*Filename:- drive.c
Custom build functions to aquire robot locomotion
*/
```

```c
#include "drive.h"

/*Initialize the handlers for motor and encoders*/
void initMotion(void){
   quadL=QUADInit(QUAD_LEFT);
   quadR=QUADInit(QUAD_RIGHT);
   motorL=MOTORInit(MOTOR_LEFT);
   motorR=MOTORInit(MOTOR_RIGHT);
   whiskers_flag = FALSE;
   collisionMotion_flag=FALSE;
}

/*Turn off motors and release the handlers for motor and encoders*/
void releaseMotion(void){
   MOTORDrive(motorL,0);
   MOTORDrive(motorR,0);
   MOTORRelease(motorL);
   MOTORRelease(motorR);
   QUADRelease(quadL);
   QUADRelease(quadR);
}


/*
  Moves the robot to the given distance and with given speed in forward or
    backward direction
  Builds up the speed
  Also keep a check if collision occurs
*/
BOOL actual_goStraight(int distance, char speed, BOOL check_collission){
   int dis_ticksL, dis_ticksR;
   int sp_ticksL1, sp_ticksR1,sp_ticksL2, sp_ticksR2;
   int target_dis, target_sp;
   int speedL=0, speedR=0;
   int time=OSGetCount();
   int direction=1;
   if(distance<0)   direction=-1;
   initMotion();

   //calculate distance in terms of clicks
   target_dis=abs(distance*CLICKS_PER_CM);

   //calculate speed in terms of clicks
   target_sp=(speed*CLICKS_PER_MM*ST_SPEED_DELAY)/100;

   //reset encoders
   QUADReset(quadL);
   QUADReset(quadR);

start_maneavour:

   sp_ticksL1=abs((int)(QUADRead(quadL)));
   sp_ticksR1=abs((int)(QUADRead(quadR)));

   //integrating effect is achieved by increasing speed by a factor
   speedL+=ST_SPEED_FACTOR;
   speedR+=ST_SPEED_FACTOR;

   speedL=speedL*direction;
   speedR=speedR*direction;
   MOTORDrive(motorL,speedL);
   MOTORDrive(motorR,speedR);
```

```
    //neutralize variable so that addition can be performed in future
    speedL=speedL*direction;
    speedR=speedR*direction;

    //give some time to the robot to achieve the speed
    OSWait(ST_SPEED_DELAY);

    sp_ticksL2=abs((int)(QUADRead(quadL)))-sp_ticksL1;
    sp_ticksR2=abs((int)(QUADRead(quadR)))-sp_ticksR1;

calc_distance:

    //if there was a collision stop the robot
    if(check_collission){
        if(!whiskers_flag)   check_whiskers();
        if(whiskers_flag==TRUE && collisionMotion_flag==FALSE)
            goto end_maneavour;
    }

    //timeout for locomotion
    if(OSGetCount()-time >TIME_OUT_1)   goto end_maneavour;
    dis_ticksL=(int)(QUADRead(quadL));
    dis_ticksR=(int)(QUADRead(quadR));

    //if required distance is acheived stop the robot
    if(abs(dis_ticksL)>=target_dis || abs(dis_ticksR)>=target_dis)
    goto end_maneavour;

    //if required speed is not acheived keep increasing it
    if(sp_ticksL2<target_sp && sp_ticksR2<target_sp)
        goto start_maneavour;

    goto calc_distance;

end_maneavour:

    //braking effect
    MOTORDrive(motorL,-1*direction*speedL);
    MOTORDrive(motorR,-1*direction*speedR);
    OSWait(5);

    //turn off motors
    MOTORDrive(motorL,0);
    MOTORDrive(motorR,0);

    releaseMotion();
    OSWait(70);
    return TRUE;
}


/*
  Turns the robot to the given angle and speed in positive and negative
    direction
  Builds up the speed
  Also checks for collision on the way
*/
BOOL actual_turn(int angle, char omega, BOOL check_collission){
    int an_ticksL, an_ticksR;
    int om_ticksL1, om_ticksR1,om_ticksL2, om_ticksR2;
    int target_an, target_om;
    int speedL=0, speedR=0;
    int time=OSGetCount();
```

```
    int direction=1;
    if(angle<0)    direction=-1;
    initMotion();

    //calculate angle in terms of clicks
    target_an=abs(angle*CLICKS_PER_DEGREE);

    //calculate speed in terms of clicks
    target_om=(omega*CLICKS_PER_DEGREE*TR_SPEED_DELAY)/100;

    //reset encoders
    QUADReset(quadL);
    QUADReset(quadR);

start_tr_maneavour:

    om_ticksL1=abs((int)(QUADRead(quadL)));
    om_ticksR1=abs((int)(QUADRead(quadR)));

    //integrating effect is achieved by increasing speed by a factor
    speedL+=TR_SPEED_FACTOR;
    speedR+=TR_SPEED_FACTOR;

    speedL=speedL*direction*-1;
    speedR=speedR*direction;
    MOTORDrive(motorL,speedL);
    MOTORDrive(motorR,speedR);

    //neutralize variable so that addition can be performed in future
    speedL=speedL*direction*-1;
    speedR=speedR*direction;

    //give some time to the robot to achieve the speed
    OSWait(TR_SPEED_DELAY);
    om_ticksL2=abs((int)(QUADRead(quadL)))-om_ticksL1;
    om_ticksR2=abs((int)(QUADRead(quadR)))-om_ticksR1;

calc_angle:

    //if there was a collision stop the robot
    if(check_collission){
        if(!whiskers_flag)    check_whiskers();
        if(whiskers_flag==TRUE && collisionMotion_flag==FALSE)
            goto end_tr_maneavour;
    }

    //timeout for locomotion
    if(OSGetCount()-time >TIME_OUT_1) goto end_tr_maneavour;

    an_ticksL=(int)(QUADRead(quadL));
    an_ticksR=(int)(QUADRead(quadR));

    //if required angle is acheived stop the robot
    if(abs(an_ticksL)>=target_an || abs(an_ticksR)>=target_an)
        goto end_tr_maneavour;

    //if required speed is not acheived keep increasing it
    if(om_ticksL2<target_om && om_ticksR2<target_om)
        goto start_tr_maneavour;
    goto calc_angle;

end_tr_maneavour:
```

```
    //braking effect
    MOTORDrive(motorL,direction*speedL);
    MOTORDrive(motorR,-1*direction*speedR);
    OSWait(5);

    //turn off motors
    MOTORDrive(motorL,0);
    MOTORDrive(motorR,0);

    releaseMotion();
    OSWait(70);
    return TRUE;
}


/*
  Turns the robot to the given angle and speed in positive and negative
    direction when robot has possession of the ball
  The only difference is that while turning with ball only one wheel is actuated
  Builds up the speed
  Also checks for collision on the way
*/
BOOL actual_turnWithBall(int angle, char omega, BOOL check_collission){
    int an_ticksL, an_ticksR;
    int om_ticksL1, om_ticksR1,om_ticksL2, om_ticksR2;
    int target_an, target_om;
    int speedL=0, speedR=0;
    int time=OSGetCount();
    int direction=1;
    if(angle<0)   direction=-1;
    initMotion();

    //calculate angle in terms of clicks
    target_an=abs(angle*CLICKS_PER_TRWB_DEGREE);

    //calculate speed in terms of clicks
    target_om=(omega*CLICKS_PER_TRWB_DEGREE*TR_SPEED_DELAY)/100;

    //reset encoders
    QUADReset(quadL);
    QUADReset(quadR);

    an_ticksL1=(int)(QUADRead(quadL));
    an_ticksR1=(int)(QUADRead(quadR));

start_trwb_maneavour:

    om_ticksL1=abs((int)(QUADRead(quadL)));
    om_ticksR1=abs((int)(QUADRead(quadR)));

    //integrating effect is achieved by increasing speed by a factor
    speedL+=TR_SPEED_FACTOR;
    speedR+=TR_SPEED_FACTOR;

    //based on the direction one of the speed variables becomes zero
    speedL=speedL*((direction-1)/2)*-1;
    speedR=speedR*((direction+1)/2);
    MOTORDrive(motorL,speedL);
    MOTORDrive(motorR,speedR);

    //neutralize variable so that addition can be performed in future
    speedL=speedL*((direction-1)/2)*-1;
    speedR=speedR*((direction+1)/2);
```

```
    //give some time to the robot to achieve the speed
    OSWait(ST_SPEED_DELAY);
    om_ticksL2=abs((int)(QUADRead(quadL)))-om_ticksL1;
    om_ticksR2=abs((int)(QUADRead(quadR)))-om_ticksR1;
    calc_trwb_angle:

    //if there was a collision stop the robot
    if(check_collission){
        if(!whiskers_flag)   check_whiskers();
        if(whiskers_flag==TRUE && collisionMotion_flag==FALSE)
            goto end_trwb_maneavour;
    }

    //timeout for locomotion
    if(OSGetCount()-time >TIME_OUT_1) goto end_trwb_maneavour;

    an_ticksL2=(int)(QUADRead(quadL));
    an_ticksR2=(int)(QUADRead(quadR));

    //if required angle is acheived stop the robot
    if(abs(an_ticksL2)>=target_an || abs(an_ticksR2)>=target_an)
        goto end_trwb_maneavour;

    //if required speed is not acheived keep increasing it
    if(om_ticksL2<target_om && om_ticksR2<target_om)
        goto start_trwb_maneavour;

    goto calc_trwb_angle;

end_trwb_maneavour:
    //braking effect
    MOTORDrive(motorL,direction*speedL);
    MOTORDrive(motorR,-1*direction*speedR);
    OSWait(5);

    //turn off motors
    MOTORDrive(motorL,0);
    MOTORDrive(motorR,0);

    releaseMotion();
    OSWait(70);
    return TRUE;
}


/*Check the status of the whiskers and update whisker_flag*/
BOOL check_whiskers(void){
    /*use 8 bits for digital inputs
    bits 1-4 not used
    bits 5-8 used for whiskers
    i.e. 11100000 => Whisker_1 is closed*/

    latch=OSReadInLatch(0);

    //mask to use only bit 5 to 8
    latch=latch & 0xF0;

    if(latch==0xF0)      {whiskers_flag = FALSE; }
    else whiskers_flag = TRUE;
    return TRUE;
}
```

```
/*Performs the respective collision motion based on the status of whiskers*/
BOOL collisionMotion(void){
   switch (latch){
      //Front_Left & Front_Right & Rear_Left => 00100000
      case  0x20:
      actual_turn(-45,25,FALSE);
      actual_goStraight(-15,20,FALSE);
      //reset flags
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Front_Left & Front_Right & Rear_Right => 10000000
      case 0x80:
      actual_turn(45,25,FALSE);
      actual_goStraight(-15,20,FALSE);
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Front_Left & Rear_Left & Rear_Right => 00010000
      case 0x10:
      actual_turn(45,25,FALSE);
      actual_goStraight(15,20,FALSE);
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Front_Right & Rear_Left & Rear_Right => 01000000
      case 0x40:
      actual_turn(-45,25,FALSE);
      actual_goStraight(15,20,FALSE);
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Front_Left & Front_Right => 10100000
      case 0xA0:
      roller(ROLLER_ON);
      actual_goStraight(-30,20,FALSE);
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Front_Left & Rear_Left => 00110000
      case 0x30:
      actual_turn(90,25,FALSE);
      actual_goStraight(15,20,FALSE);
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Front_Left & Rear_Right => 10010000
      case 0x90:
      actual_turn(45,25,FALSE);
      actual_goStraight(-15,20,FALSE);
      whiskers_flag = FALSE;
      collisionMotion_flag=FALSE;
      break;

      //Rear_Left & Front_Right => 01100000
      case 0x60:
```

```
        actual_turn(-45,25,FALSE);
        actual_goStraight(-15,20,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        //Front_Right & Rear_Right => 11000000
        case 0xC0:
        actual_turn(-90,25,FALSE);
        actual_goStraight(15,20,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        //Rear_Left & Rear_Right => 01010000
        case 0x50:
        actual_goStraight(15,25,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        //Front_Left => 10110000
        case 0xB0:
        roller(ROLLER_ON);
        actual_goStraight(-30,20,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        //Front_Right => 11100000
        case 0xE0:
        roller(ROLLER_ON);
        actual_goStraight(-30,20,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        //Rear_Left => 01110000
        case 0x70:
        actual_goStraight(15,20,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        //Rear_Right => 11010000
        case 0xD0:
        actual_goStraight(15,20,FALSE);
        whiskers_flag = FALSE;
        collisionMotion_flag=FALSE;
        break;

        default:
        break;
    }
    return TRUE;
}


/*
  goStraight() is called by the main routine
  actual_gostraight() actually performs the locomotion
  This is done to avoid recursion when collision occurs because
    collisionmotion() also performs locomotion
```

```
*/
BOOL goStraight(int distance, char speed, BOOL check_collission){
   actual_goStraight(distance, speed, check_collission);
   if(whiskers_flag){
      collisionMotion_flag=TRUE;
      collisionMotion();
   }
   return TRUE;
}


/*
  turn() is called by the main routine
  actual_turn() actually performs the locomotion
  This is done to avoid reccursion when collision occurs because
    collisionmotion() also performs locomotion
*/
BOOL turn(int angle, char omega, BOOL check_collission){
   actual_turn(angle, omega, check_collission);
   if(whiskers_flag){
      collisionMotion_flag=TRUE;
      collisionMotion();
   }
   return TRUE;
}


/*
  turnWithBall() is called by the main routine
  actual_turnWithBall() actually performs the locomotion
  This is done to avoid reccursion when collision occurs because
    collisionmotion() also performs locomotion
*/
BOOL turnWithBall(int angle, char omega, BOOL check_collission){
   actual_turnWithBall(angle, omega, check_collission);
   if(whiskers_flag){
   collisionMotion_flag=TRUE;
   collisionMotion();
   }
   return TRUE;
}
```

High level strategy

Strategy.h

```
#ifndef STRATEGY_H
#define STRATEGY_H

#include "eyebot.h"
#include "classify.h"
#include "drive.h"

#define ROLLER_ON 0x80
#define ROLLER_OFF 0x00
#define DIGITAL_OUT_PORT 0x7F
#define POSSESSION_DISTANCE 1  //changed form 1
#define POSSESSION_ANGLE 30
#define POSSESSION_ANGLE_POS 30
#define POSSESSION_ANGLE_NEG -34
```

```
#define SAFE_ROTATE_ANGLE 10 //able to rotate with roller off w/o loosing ball
#define POSSIBLE_TO_SCORE_MIN_ANGLE 10
#define MIN_OFFSET_ANGLE 10
#define MIN_OFFSET_DISTANCE 1
#define MIN_GOAL_OFFSET_ANGLE 6
#define MIN_BALL_SCORE_OFFSET_ANGLE 20
#define MAX_ERR_ANGLE 0
#define MAX_ERR_DISTANCE 10
#define LEFT -1
#define RIGHT 1
int direction;

void roller(char action);
BOOL doISee(location * object);
BOOL locate(location * object);
BOOL getPossession(void);
BOOL goToBall(void);
BOOL rotateTowardsGoal(location * goal);
//BOOL curveAdvance(char goal);
BOOL check_whiskers(void);
void goAroundBall(void);
void goAroundBall2(void);
BOOL ballInReach(void);
BOOL doWeHavePossession(void);

#endif
```

Strategy.c

```
#include "strategy.h"
#include "drive.h"
#include "classify.h"


extern location ball,ygoal,bgoal;
extern BOOL possession;
extern int ygoalmaxy; //define the line where we think goal starts.
extern int bgoalmaxy;

void roller(char action)
{
      OSWriteOutLatch(0, DIGITAL_OUT_PORT, action);
}

BOOL doISee(location * object){

      getImage();
      findLocations();

      if(object->x == NOT_FOUND)
            return FALSE;

      return TRUE;
}


//rotates to find an object
BOOL locate(location * object){
```

```
	int i = 0;
	//if(possession == FALSE)
	if(doISee(object)){
		return TRUE;
	}
	else{
		LCDPrintf("\nLocating...");
		while(!doISee(object)){
			turn((direction * 40),20,FALSE);
			i++;
			if(i>9)
				return FALSE;
		}
	}
	return TRUE;
}

//ROTATES WITH BALL TOWARDS THE DEFINED GOAL
BOOL rotateTowardsGoal(location * goal)
{

	if(goal->x == NOT_FOUND)
		return FALSE;

	while(1){
		turnWithBall(goal->angle,20,TRUE);

		getImage();
		findLocations();

		if(!ballInReach()){
			LCDPrintf("NOT Reach\n");
			return FALSE;
		}

		if(abs(goal->angle) < POSSIBLE_TO_SCORE_MIN_ANGLE){
		    //robot, ball and goal on the same line
			return TRUE;
		}
	}

	return TRUE;
}

//this function enables us to establish if the ball is in front of us
//and if we will kick it going straight
//we defined a region where it's possible to say "ball is in reach"
//and expressed it like a function of angle and distance
//
//   angle  |           + identifies the mentioned region
//          |
//          |                                                      _____
//          |\                                                    |+++++|
//          |+\                                                   |+++++|
//          |++\                                                ||=======||
//          |+++\                                               ||robot  ||
//          |++++|                                              ||chassis||
//          |++++|---------> distance                           ===========
BOOL ballInReach(void)
{
	if(ball.distance > 6)
		return FALSE;
```

```
	if(abs(ball.angle) > ((-2)*(ball.distance)+22))
		return FALSE;

	return TRUE;
}


BOOL goToBall(void){
	LCDPrintf("\ngoing to ball");
	//Check if we have possession
	if(doWeHavePossession())
		return TRUE;

	roller(ROLLER_OFF);

	while(ball.distance > POSSESSION_DISTANCE ||
		abs(ball.angle) > POSSESSION_ANGLE){


		//Did we mess things up and loose the ball?
		if(!doISee(&ball)){ //yes we messed things up
			LCDPrintf("we messed up");
			return FALSE;
		}

		if(ball.distance > MIN_OFFSET_DISTANCE &&
				abs(ball.angle) < 10){
			//Max distance without error = 30cm
			//if ball.distance > 30 => distance is wrong
			if(ball.distance > 30){
			goStraight(20,50,TRUE); //goStraight to have better measure
			}
			else{ //distance is correct
				goStraight(ball.distance-1,20,TRUE);
			}
		}

		//Now we do not have possession
		if(abs(ball.angle) >= MIN_OFFSET_ANGLE){
			turn(ball.angle/2 ,20,TRUE);
		}

	}
	return TRUE;
}

BOOL getPossession(void)
{
	if(doWeHavePossession())
		return TRUE;
	goStraight(4,8,FALSE);

	getImage();
	findLocations();

	if(!doWeHavePossession())
		return FALSE;

	return TRUE;
}

BOOL doWeHavePossession(void)
{
```

```
        if(ball.distance == 0){
              if(ball.angle > POSSESSION_ANGLE_NEG &&
                 ball.angle < POSSESSION_ANGLE_POS){
                 return TRUE;
           }
        }
        return FALSE;
}

//Assumes that we have run the goToBall function
//Moves the robot in a pre-defined movement such that it ends
//on the other side of the ball without touching the ball
//USED WHEN WE DID NOT HAVE GOOD-ROLLER
void goAroundBall(void){
        direction = selectRotationalAngle();

        goStraight(-5,20,TRUE);
        turn(direction*90,25,TRUE);
        goStraight(15,20,TRUE);
        turn(((-1)*direction)*90,25,TRUE);
        goStraight(30,20,TRUE);
        turn(((-1)*direction)*90,25,TRUE);
        goStraight(15,20,TRUE);
        turn(((-1)*direction)*90,25,TRUE);

        LCDPrintf("\ngo around completed");
}
//USED WITH A GOOD ROLLER
void goAroundBall2(void){
        direction = selectRotationalAngle();
        roller(ROLLER_ON);
        turnWithBall(direction*135,15,TRUE);
        LCDPrintf("\ngo around completed");
}

/*
//used in lab1
BOOL curveAdvance(char goal)
{
        int angle =0;
        int wasItFound = NOT_FOUND;

        if(ball.angle > 15){
              turn(40 ,15);
              goStraight(40,30);
              turn(-35 ,15);
        }
        else if(ball.angle < -15){
              turn(-40 ,15);
              goStraight(40,30);
              turn(35 ,15);
        }

        return TRUE;
}
*/
```

Main function

Ras5.c

```c
/*
| ras5.c
| main function for Zidane in Robotics course
| RAS5 group
| 15. may 2007
*/

#include "eyebot.h"
#include "classify.h"
#include "drive.h"
#include "strategy.h"



BOOL possession;
extern int direction;

QuadHandle quadL, quadR;
MotorHandle motorL, motorR;


int main()
{
	int cam;
	int end_flag = 0;
	int key;
	location * opponentGoal = &bgoal;
	location * ourGoal = &ygoal;
	BOOL no_go = TRUE; //can we drive?
	BOOL no_goal = TRUE;
	possession=FALSE;


	/**** Startup Procedures *****/
	cam = initCamera();
	fillClassificationMatrix();
	fillLengthArray();
	direction = RIGHT;
	//startup motors & encoders


	//initMotion();

	//Print the menu
	LCDMenu("BLU","yel","GO","End");

	while (!end_flag)
	{

		key = KEYRead();
		switch(key)
		{
			case KEY1:
				opponentGoal = &bgoal;
				ourGoal = &ygoal;
				LCDMenu("BLU","yel","","");
			break;
```

```
		case KEY2:
			opponentGoal = &ygoal;
			ourGoal = &bgoal;
			LCDMenu("blu","YEL","","");
		break;
		case KEY3:
		no_go = FALSE;
		break;
		case KEY4:
		end_flag = TRUE;
		break;
	}//end switch

	//we are still geting images and classifying them
	//before starting.  Just warming up the camera
	getImage();
	findLocations();
	drawLocations();
	printLocations();
	LCDMenu("","","","");

	//if there is no_go there is no use to continue.
	if(no_go)
		continue;

	LCDClear();

	/*OUR STRATEGY STARTS HERE BELOW*/
	//if(doISee(opponentGoal) && doISee(BALL)) //Used in lab 1
	//    curveAdvance(opponentGoal);  //To have a better angle at goal


	while(no_goal){
		//printLocations();

		if(!locate(&ball)){
		    //defence or attack strategy TO BE DEFINED
			LCDPrintf("\nunable to find ball");
			LCDPrintf("\nDEFENCE STRATEGY");
	    continue;
	}

		LCDPrintf("I see the ball\n");

		if(!goToBall()){
			LCDPrintf("Cannot go to ball\n");
			roller(ROLLER_OFF);
			continue;
		}

		LCDPrintf("Checking for our goal\n");
		if(doISee(ourGoal)){
			LCDPrintf("I see our goal\n");
			goAroundBall2();
			LCDPrintf("Got around the ball\n");
			continue;
		}

		if(opponentGoal->x == NOT_FOUND){
			roller(ROLLER_ON);
			LCDPrintf("Opponent Goal not found\n");
			if(!getPossession())
				continue;
```

```
                        turnWithBall((direction * 30),20,TRUE);
                        LCDPrintf("Looking for opponent goal\n");
                        continue;
                }
                else if(opponentGoal->angle > SAFE_ROTATE_ANGLE){
                        roller(ROLLER_ON);
                        LCDPrintf("Opp goal angle>safe rotate\n");
                        if(!getPossession())
                                continue;
                }

                roller(ROLLER_ON);
                if(!rotateTowardsGoal(opponentGoal)){
                LCDPrintf("Rotating towards opp goal\n");
                        continue;}

                roller(ROLLER_OFF);
                goStraight(opponentGoal->distance/2,80,FALSE);

                LCDPrintf("We think we scored\n");

                goStraight(-70,80,FALSE);

                continue;


        }//end while(no_goal)

    }//end while (!end_flag)
    releaseMotion();
    return 0;

}
```

References

[1] Lecture notes of course 2D1426
[2] "Autonomous Mobile Robots", R. Siegwart, I. R. Nourbakhsh
[3] "Embedded Robotics - Mobile Robot Design and Applications with Embedded Systems", T. Bräunl
[4] On-line documentation http://robotics.ee.uwa.edu.au/eyebot/
[5] On-line documentation http://www.joker-robotics.com/eyebot/