BRADLEY University

Cooperative Control of Heterogeneous Mobile Robots Network

Gregory A. Bock, Brittany J. Dhall, Ryan T. Hendrickson, & Jared A. Lamkin
Dr. Jing Wang & Dr. In Soo Ahn
Department of Electrical and Computer Engineering

May 12th, 2016

Abstract

Cooperative control of mobile robots has been a rapidly growing area of research and development (R&D) for industry and academia during the past years. Such R&D activities are inspired by cooperative systems found in nature, for example, a flock of birds or a swarm of insects. In this project, the objective was to design and implement cooperative control algorithms on different types of robotic platforms. With the proposed cooperative control structure, several tasks are performed autonomously by a fleet of robot agents, which include point convergence, trajectory following, formation control, and heading alignment. The completion of the tasks is based on the consensus of the heterogeneous robot agents through the exchange of local information. MATLAB was used to conduct simulations of different control structures, and determine how a large number of robot agents can interact with one another. Applications of cooperative control strategies are significant and far reaching. This emerging technology can be used for intelligence, surveillance and reconnaissance (ISR) in military missions and civilian applications as well.

Acknowledgement

Our sincere gratitude goes to Dr. Jing Wang and Dr. In Soo Ahn for their guidance throughout the capstone project. Their expertise on dynamic systems analysis has helped us explore challenging cooperative control problems and implement complex algorithms on many state-of-the-art robotic platforms, with a lot of fun.

We also would like to thank Mr. Chris Mattus, ECE Lab Director, and Mr. Nick Schmidt, ECE Assistant Lab Director, for their technical support.

Last, but not least, our thanks go to our parents for the unconditional support they have given us. Without their support, we could not have come this far.

This work was supported in part by Air Force Research Laboratory under grant agreements FA8750-13-1-0190 and FA8750-15-1-0143.

Contents

Abstract i
Acknowledgement ii
Contents iii
Table of Figures vi
Chapter 1: Introduction 1
1.1 Problem Description 2
1.1.1 Kinematic Model 2
1.1.2 Control Problems 3
1.2 Control Design 5
Chapter 2: E-puck 7
2.1 Overview of E-puck 7
2.1.1 Hardware 7
2.1.2 Software 8
2.1.3 System Setup 9
2.2 Design and Implementation 9
2.2.1 Motor Control 9
2.2.2 Localization via Odometry 9
2.2.3 Information exchange through communication 10
2.3 Experiments 11
2.3.1 Heading Alignment 11
Chapter 3: Kilobot 13
3.1 Overview of Kilobot 13
3.1.1 Hardware 13
3.1.2 Software 14
3.1.3 System Setup 16
3.2 Design and Implementation 16
3.2.1 Motor Control 16
3.2.2 Information Exchange through communication 18
3.2.3 Localization Via Communication 19
3.3 Experiments 20
3.3.1 Gradient 21

3.3.2 Orbiting 22
3.3.3 Asynchronous Consensus 25
3.3.4 Light Following 26
3.3.5 Sending Messages from an Outside Source / Controllable Node 28
Chapter 4: QBot 2 29
4.1 Overview of QBot 2 29
4.1.1 Hardware 29
4.1.2 Software 30
4.2 Design and Implementation 30
4.2.1 Localization Using Kinect Sensor 30
4.2.2 Information Exchange Through Communication 32
4.2.3 Motor Control 33
4.3 Experiments 33
4.3.1 Rendezvous Control 34
4.3.2 Formation Control 37
4.3.3 Trajectory Following 39
4.3.4 Object Avoidance 40
Chapter 5: Conclusion and Future Work 46
Appendix A 47
E-puck Code 47
Color Detection 47
searchball.c 50
searchball.h 60
runballfollow.c 61
runballfollow.h 67
Vicsek and Odometry 68
Bluetooth Communication 82
btcom.c 85
btcom.h 88
Appendix B 89
E-puck Unbricking Guide 89
Appendix C 93
Kilobot Code 93
Gradient 93
Orbiting 98

Light Following 104
Asynchronous Consensus 109
Random ID Generator 127
Color Consensus 131
Ambient Light Sensor Calibration 136
Follow Leader 139
Leader 144
Fixed Reference Consensus 147
Multiple Agent Orbiting 154
Atmega128 Messaging 162
myLibrary.h 167
myLibrary.c 168
Appendix D 171
Program Kilobots 171
Appendix E 172
Flash Kilobot Firmware 172
Appendix F 177
Overall Simulink Model 177
Localization 177
Communication 179
Motor Control 180
Data Acquisition 182
Appendix G 190
Find Object 190
Kinect Initialize 192
Kinect Get Image 193
Kinect Get Depth 195
Stream Server 197
Stream Client 199

Table of Figures

Figure 1.1 A Nonlinear Kinematic Model 2
Figure 1 2 A Linearized Kinematic Model 3
Figure 1.3 Heading Alignment 4
Figure 1.4 Rendezvous Control 4
Figure 1.5 Formation Control 4

Figure 2.1 An E-puck Robot 7
Figure 2.2 E-puck Infrared Sensor Locations 8
Figure 2.3 Basic Motor Function Call 9
Figure 2.4 Motor Initialize Function 9
Figure 2.5 E-Puck Message Structure 11
Figure 2.6 Vicsek Model Flowchart 11
Figure 2.7 E-puck Implementation of Vicsek Model 12

Figure 3.1 A Kilobot Robot 13
Figure 3.2 Kilobot Underside 14
Figure 3.3 Kilobot Controller Window 15
Figure 3.4 An Example of Motor Control 17
Figure 3.5 setMotion Function 18
Figure 3.6 Motion Type Definition 18
Figure 3.7 Kilobot IR Communication 19
Figure 3.8 Kilobot Messaging Function 19
Figure 3.9 Gradient Flowchart 21
Figure 3.10 Gradient Implementation 22
Figure 3.11 Orbiting Zones 23
Figure 3.12 Orbiting Flowchart 24
Figure 3.13 Orbiting Implementation 24
Figure 3.14 Asynchronous Consensus Flowchart 25
Figure 3.15 Asynchronous Consensus Implementation 26
Figure 3.16 Light Following Flowchart 27
Figure 3.17 Light Following Implementation 28
Figure 3.18 Messaging Circuit 28

Figure 4.1 A QBot 2 at Bradley University 29
Figure 4.2 Overall Simulink Model 30
Figure 4.3 Above-View of QBot 2 Localization 31
Figure 4.4 Expanded Above-View of QBot 2 Localization 32
Figure 4.5 Rendezvous Implementation, ki = 2 34
Figure 4.6 Rendezvous X Position vs Time, ki = 2 35
Figure 4.7 Rendezvous Y Position vs Time, ki = 2 35
Figure 4.8 Rendezvous Implementation, ki = 6 36
Figure 4.9 Rendezvous X Position vs Time, ki = 6 36
Figure 4.10 Rendezvous Y Position vs Time, ki = 6 37

Figure 4.11 Formation Control Implementation 38
Figure 4.12 Formation Control X Position vs Time 38
Figure 4.13 Formation Control Y Position vs Time 39
Figure 4.14 Object Avoidance Flowchart 41
Figure 4.15 Object Avoidance Input Variables 41
Figure 4.16 “Left” Input Membership Function 42
Figure 4.17 “Center” Input Membership Function........ 42
Figure 4.18 “Right” Input Membership Function 43
Figure 4.19 “Vl” Output Membership Function 43
Figure 4.20 "Vr" Output Membership Function 44
Figure 4.21 Object Avoidance Implementation 45

Figure B.1 E-puck with Top Removed........ 89
Figure B.2 E-puck Connection Pins for Debugger 90
Figure B.3 Select Programmer Window 90
Figure B.4 Program Memory Pane 91
Figure B.5 Power Pane 92

Figure F.1 Overall Simulink Model........ 177
Figure F.2 Localization Subsystem........ 178
Figure F.3 Color Detection Subsystem 178
Figure F.4 Determine Position Subsystem........ 179
Figure F.5 Communication Subsystem 179
Figure F.6 Motor Control Subsystem........ 180
Figure F.7 Fuzzy Subsystem 181
Figure F.8 Write to Motors Subsystem........ 181
Figure F.9 Data Acquisition Subsystem 182
Figure F.10 Write to Workspace Subsystem 183
Figure F.11 Get Image Subsystem 183
Figure F.12 Get Depth Subsystem 184
Figure F. 13 Get Basic Data Subsystem 184
Figure F.14 QBot Basic Subsystem 185
Figure F.15 QBot 2 Basic IO Subsystem 186
Figure F.16 Encoder to Velocity Subsystem 187
Figure F.17 Encoder Subsystem 187
Figure F.18 Encoder to Distance Subsystem 187
Figure F.19 QBot 2 Full Kinematics Subsystem 188
Figure F.20 QBot 2 Differential Drive Kinematics Subsystem 188
Figure F.21 QBot 2 Kinematics Subsystem 189

Introduction 1

Distributed control of multiple mobile robots has received a great deal of attention in recent years. This growing area of research finds its inspiration from different systems that exist in nature. There are many examples of such systems, for instance, a flock of birds or a swarm of insects. Each agent, in these systems, is able to obtain local sensory information, but together the agents are able to perform complex tasks. Numerous applications of cooperative control structures exist. This technology can be used in a variety of military missions such as surveillance and reconnaissance, or search and rescue [12][15]. Civilian applications exist as well, for example, environmental sensing and monitoring, or cooperative transportation may utilize this technology.

In general, the design of distributed control of multiple robots relies on local interactions and information exchange among robots in the group. Through this exchange, the whole group will be controlled to achieve desired tasks cooperatively. The control design is challenging because interactions among robots are often local, time-varying, directional and intermittent due to an individual robots' sensing and communication capabilities. Thorough study has been done addressing this challenge by assuming simple linear models for robots [15][12][2][17]. For instance, formation control of multi-robots was studied in [4][11] by assuming a fixed sensing and communication structure among robots. For time varying sensing and communication, the neighboring control rule was proposed in [18] and rigorously proved in [7]. It was shown that all systems in the group will converge to the same value if the underlying undirected sensing communication topologies among systems are connected. More complicated time-varying and directed sensing and communication topologies were considered in [14][8][17][13][21]. By explicitly considering robot dynamics, a discontinuous control was proposed in [5] and stability was analyzed using nonsmooth Lyapunov theory. Time-varying controls were designed and analyzed using average theory in [9]. A number of experimental results have been reported in recent literature which deal with multi-robot coordination [10], leader-follower flocking [6], formation control [1][16], and containment control for multiple vehicles [3]

The objective of this research is to present simple distributed control designs for multiple mobile robots. The control designs are constrained through a kinematic model, and are validated through experimentation on three different mobile robot platforms. In particular, experiments focus on heading alignment, rendezvous control, and formation control/Following. Each of these experiments are addressed with the consideration of the sensing and communication capabilities of each robot platform. The mobile robot platforms used for experimentation are the following: E-puck, Kilobot, and QBot 2. In each experiment, the mobile robot platforms only utilize local information. For the E-puck and QBot 2, the position with respect to their local coordinate frame is determined through wheel encoders. The Kilobot can only determine its position with respect to their local coordinate frame through communication

The thesis is organized as follows. chapter 1.1 discusses the Kinematic model and the control problems in detail. Chapter 2 discusses the E-puck mobile robot, chapter 3 provides information on the Kilobot robot, and chapter 4 examines the QBot 2 mobile robot. In each chapter, the implementation of the robot and its experimental results are provided.

1.1 Problem Description

1.1.1 Kinematic Model

The E-puck and the QBot 2 are typical differential drive mobile robots. The kinematic model of the mobile robot can be described using the following nonlinear equations:

$$\dot{x}_i = v_i \cos\theta_i\,,\ \ \dot{y}_i = v_i \sin\theta_i,\ \ \dot{\theta}_i = \omega_i \tag{1}$$

$$v_i = \frac{v_{iR}+v_{iL}}{2},\ \ \ \omega_i = \frac{v_{iR}-v_{iL}}{d} \tag{2}$$

where $i \in \Omega \triangleq \{1,\cdots,n\}$, $[x_i \quad y_i]^T \in \mathfrak{R}^2$denotes the position of the center of the ith robot, θ_i is the orientation, and $v_i \in \ \mathfrak{R}$ is the driving velocity. $\omega_i \in \ \mathfrak{R}$ is the steering velocity, d is the distance between wheel centers, and v_{iR} and v_{iL} are the linear speeds of the right and left wheel, respectively. Let the robot's radius be r. The nonlinear kinematic model is shown in the figure below.

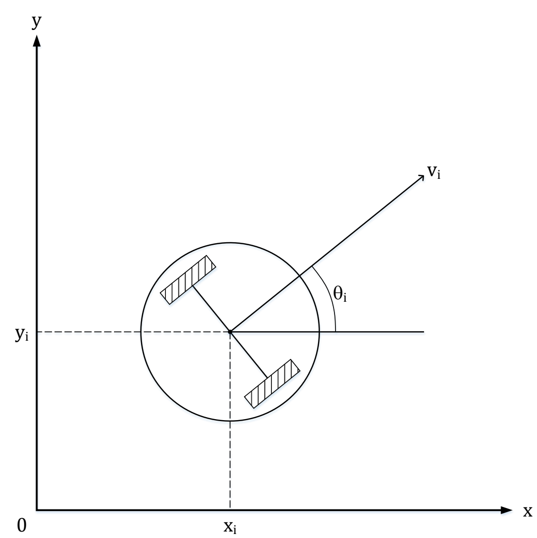


Figure 1.1 A Nonlinear Kinematic Model

The control design is based on the linearized model of robots. Define the front end of the robot as a reference point $p_i = [\mathrm{p_{ix}} \quad \mathrm{p_{iy}}]^{\mathrm{T}}$, that is, it is the point along the sagittal axis of the robot at a distance r from the center of the robot i, and it follows:

$$p_{ix} = x_i + r\cos\theta_i,\ p_{iy} = y_i + r\sin\theta_i \tag{3}$$

This creates a linearized kinematic model, shown in Figure 1 2. It follows the definition of the reference point p_i in (3) that its time derivative is.

$$\begin{bmatrix}\dot{p}_{ix}\\ \dot{p}_{iy}\end{bmatrix} = \begin{bmatrix}\cos\theta_i & -r\sin\theta_i\\ \sin\theta_i & r\cos\theta_i\end{bmatrix}\begin{bmatrix}v_i\\ \omega_i\end{bmatrix} \tag{4}$$

Using the following input transformation

$$\begin{bmatrix}v_i\\ \omega_i\end{bmatrix} = \begin{bmatrix}\cos\theta_i & \sin\theta_i\\ -\sin\theta_i/r & \cos\theta_i/r\end{bmatrix}\begin{bmatrix}u_{i1}\\ u_{i2}\end{bmatrix} \tag{5}$$

Equation (9) can be converted into the form

$$\begin{cases}\dot{p}_{ix} = u_{i1}\\ \dot{p}_{iy} = u_{i2}\end{cases} \tag{6}$$

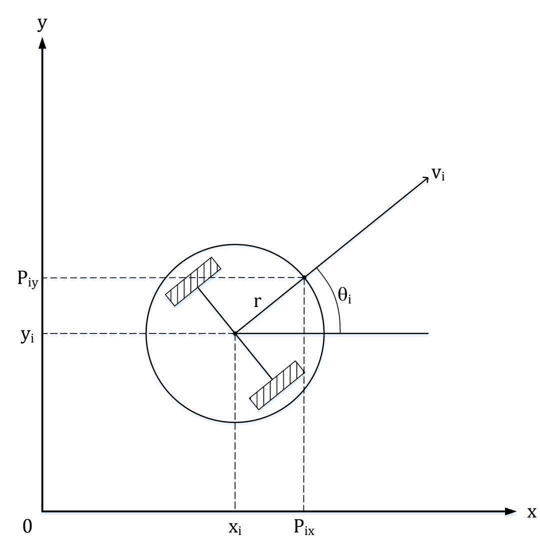


Figure 1 2 A Linearized Kinematic Model

1.1.2 Control Problems

In this research, we designed cooperative control algorithms to solve several coordination tasks, which are listed below.

Problem 1: **Heading alignment**.

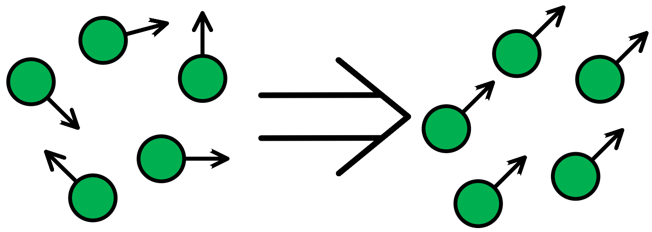


Figure 1.3 Heading Alignment

As shown in Figure 1.3, the heading alignment task is to design local control for each robot such that starting with different headings, all robots will eventually move towards the same direction. All robots will eventually move towards the same direction. Mathematically, it can be described using the following.

$$\lim_{t\to\infty}\left\|\theta_i(t)-\theta_j(t)\right\| = 0\,,\forall i,j \tag{7}$$

Problem 2: **Rendezvous control.**

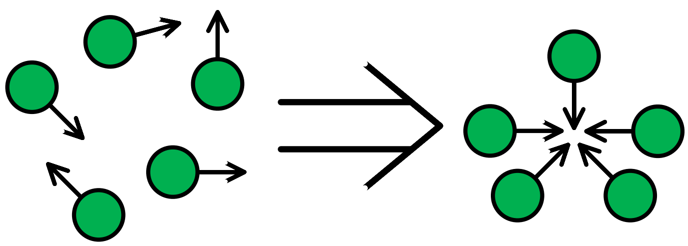


Figure 1.4 Rendezvous Control

In the Rendezvous Control**,** all robots will be controlled to move to a common location. As shown in *Figure 1.4*, initially, robots are located at different planes, and move until they reach position consensus. This problem is also known as point consensus, and can be described by the following equation.

$$\lim_{t\to\infty}\left\|p_i(t)-p_j(t)\right\| = 0\,,\forall i,j \tag{8}$$

Problem 3: **Formation control/Following.**

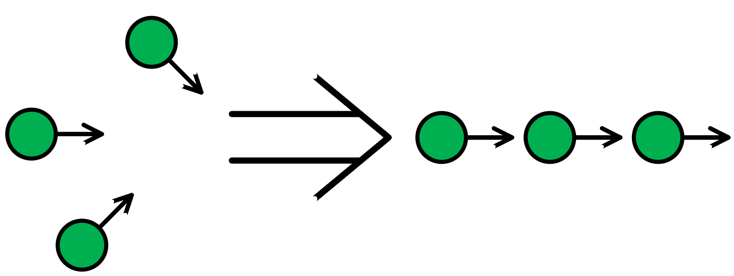


Figure 1.5 Formation Control

In Figure 1.5, shown above, agents move into a formation and continue on a desired trajectory. Formation control/Following is described by the following equations.

$$\lim_{t\to\infty}[p_i(t) - p_j(t)] = \begin{bmatrix} C_{ix} - C_{jx} \\ C_{iy} - C_{jy} \end{bmatrix}, \forall i,j \tag{9}$$

$$\lim_{t\to\infty}\|p_i(t) - d_p(t)\| = 0\,, \forall i,j \tag{10}$$

where $[C_{ix} \quad C_{iy}]^T$ is an offset vector for agent i. The offset vector specifies the relative position of the agent in a desired formation shape. In (10), $d_p(t)$ is a desired trajectory point at time t.

1.2 Control Design

In this section, we present the desired cooperative control algorithms for solving the problems listed in section 1.1.2 Control Problems. The design is based on local information exchange among robot agents. The connections between robots are determined through a communication matrix $S(t)$.

$$S(t) = \begin{bmatrix} 1 & s_{12}(t) & \cdots & s_{1n}(t) \\ s_{21}(t) & 1 & \cdots & s_{2n}(t) \\ \vdots & \vdots & \ddots & \vdots \\ s_{n1}(t) & s_{n2}(t) & \cdots & 1 \end{bmatrix} \tag{11}$$

where $s_{ij}(t) > 0$ if robot j is within the sensing/communication range of robot i at time instant t, otherwise, $s_{ij}(t) = 0$.

To solve problem 1, the distributed control is of the form

$$\theta_i(k+1) = \frac{\theta_i(k) + \sum_{j=1}^{n} \theta_j(k)}{n} \tag{12}$$

where $\theta_i(k)$ is the current heading of the robot, $\theta_i(k+1)$ is the next heading of the robot, n is the total number of neighboring robots, and $\theta_j(k)$ is the current heading of a neighboring robot.

To solve problem 2, the distributed control is of the form

$$u_{i1}(t) = k_i \sum_{j=1}^{n} s_{ij}(t)\left(p_{jx}(t) - p_{ix}(t)\right) \tag{13}$$

$$u_{i2}(t) = k_i \sum_{j=1}^{n} s_{ij}(t)\left(p_{jy}(t) - p_{iy}(t)\right) \tag{14}$$

where $k_i > 0$ is the control gain, and $s_{ij}(t)$ is the value in the current sensing/communication matrix $S(t)$.

To solve problem 3, the distributed control is of the form

$$u_{i1}(t) = k_i \sum_{j=1}^{n} s_{ij}(t)\big(d_{px}(t) - C_{jx} - p_{ix}(t) + C_{ix}\big) \tag{15}$$

$$u_{i2}(t) = k_i \sum_{j=1}^{n} s_{ij}(t)\big(d_{py}(t) - C_{jy} - p_{iy}(t) + C_{iy}\big) \tag{16}$$

where $k_i > 0$ is the control gain, and $[C_{ix} \quad C_{iy}]^T$ defines the formation shape.

E-puck

2

This chapter provides the implementation of cooperative control algorithms using the E-puck. An E-puck robot is shown in Figure 2.1. E-pucks were developed at the Ecole Polytechnique Fédérale de Lausanne, to be used for educational purposes.

Figure 2.1 An E-puck Robot

2.1 Overview of E-puck

2.1.1 Hardware

The E-puck uses a dsPIC 30F6014A, a 16-bit microcontroller with a DSP core. The dsPIC contains 8 kB of RAM, 144 kB of flash, and a 64 MHz internal clock. The 144 kB of flash is used to store user programs, as well as the bootloader. The 64 MHz is scaled down to 30 MHZ for user programs.

The E-puck is a differential-drive wheeled robot with a maximum speed of 15 cm/s. The motors are permanent magnet stepper motors with a gearbox having a reduction ratio of 1/50. The motors have a step angle of 0.36 degrees. This results in the motors having a resolution of 1000 steps/rev with no load. The additional load of the E-puck body and wheels, gives the motors a resolution of 1300 steps/rev. A wheel is attached to each motor, and has a diameter of 41 mm and a circumference of 128.8 mm. The distance between the wheels is 53 mm.

Eight infrared sensors are mounted onto the E-puck. The infrared sensor locations can be seen in Figure 2.2

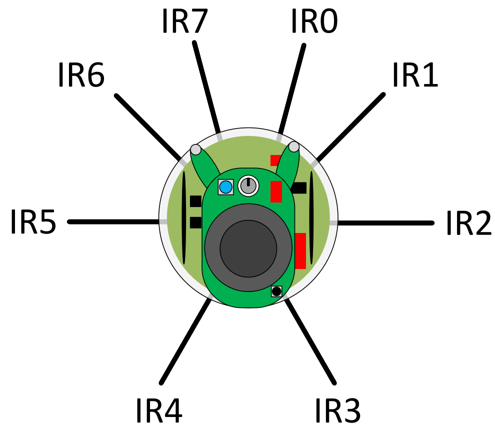


Figure 2.2 E-puck Infrared Sensor Locations

The infrared sensors can be used as proximity sensors, or can be used for communication purposes. In proximity mode, the infrared sensors have a range of 4 cm. In communication mode, the infrared sensors have a range of 25 cm. Communication is achieved through the infrared sensors acting as emitters and receivers to send messages. Proximity information can be obtained while the infrared sensors are in communication mode.

Communication can also be achieved through the use of a Bluetooth module on the E-puck. The Bluetooth module can connect to other E-pucks, as well as a computer. User programs can be uploaded onto the E-puck using a Bluetooth link with a computer.

Three independent microphones, one speaker, ten LEDs, a 3D accelerometer, and a CMOS camera are also built onto the E-puck. The CMOS camera has a pixel resolution of 640x480, and can be configured to operate in either color or gray-scale mode. Although the camera has a resolution of 640x480, the on board microcontroller does not have enough processing power or memory to operate on a picture of that size. However, the camera can be configured to lower resolutions, that way images can be processed in a timely manner.

2.1.2 Software

To create projects for the E-pucks, the MPLAB X IDE version 3.25 was used. All programs were written in the C programming language and used various libraries that were provided with the E-pucks, which can be found in Appendix A. Once a project is completed, a HEX file is generated, which will be uploaded to the E-puck. To upload a program to an E-puck, the ICD-3 programmer from Microchip is used, and using MPLAB X the ICD-3 uploads the hex file to the connected E-puck.

On occasion, an E-puck may no longer perform the program stored in its memory, and will also not be able to receive new programs. To fix this, The ICD-3 must be connected to the E-puck, and using MPLAB X, erase the flash memory and reset the fuses. A detailed guide on how to fix the E-pucks can be found in Appendix B.

2.1.3 System Setup

In all experiments, 1 to 3 E-pucks are used. On the computer, a program is developed to perform a desired behavior or task and a HEX file is generated from it. The Hex file is then bootloaded onto the E-pucks using the ICD-3 programming cable.

2.2 Design and Implementation

This section provides information on how controls are implemented using the E-puck.

2.2.1 Motor Control

The stepper motors can be controlled individually by two function, `e_set_speed_left(int x)` and `e_set_speed_right(int x)`, which can be found in Appendix A. The function takes a value ranging from -1000 to 1000, where each value has a unit of steps/s. This gives the E-puck a maximum speed of 1000 steps/s, or 15 cm/s. When using the motors, it is important to temporarily pause all other interrupts. A delay before calling the motor functions, and another delay after the motor function calls must be included. This can be seen in Figure 2.3. In the code example, a delay of 400 ms is applied before and after the motors are called. Before a speed is applied to the motors, the step count for both the left and right motors are reset to zero. In this example, the motors are set to a speed of 200 steps/s. To stop the motors, a value of 0 steps/s need to be applied to the set speed functions.

```
myWait(400);

e_set_steps_left(0);
e_set_steps_right(0);
e_set_speed_left(200);
e_set_speed_right(200);

myWait(400);
```

Figure 2.3 Basic Motor Function Call

Before any value can be applied to the motors, the motor initialize function must be called. This function can be seen in Figure 2.4, and is called at the beginning of the main program.

```
init_motors();
```

Figure 2.4 Motor Initialize Function

2.2.2 Localization via Odometry

The E-pucks continuously update how many steps the stepper motors have traveled since initialization. By intermediately using the change in steps from two points in time, the Epuck can compute its position and orientation. The odemetetry algorithms are given below.

$$\Delta\theta = \frac{(\Delta R - \Delta L)}{2} \quad (17)$$

$$\Delta S = \frac{(\Delta R + \Delta L)}{2} \quad (18)$$

$$\Delta x = \Delta S * cos\left(\theta + \frac{(\Delta\theta)}{2}\right) \quad (19)$$

$$\Delta y = \Delta S * sin\left(\theta + \frac{(\Delta\theta)}{2}\right) \quad (20)$$

$$x(k+1) = x(k) + \Delta x \quad (21)$$

$$y(k+1) = y(k) + \Delta y \quad (22)$$

$$\theta(k+1) = \theta + \frac{(\Delta\theta)}{3} \quad (23)$$

$\Delta\theta$ is the change in orientation of the E-puck and is calculated by taking the average of the difference of the change in the step count of the right, ΔR, and left, ΔL, motors (17). The average change in step count of both motors, ΔS (18), is used to calculate the changes in the E-puck's change in x direction, Δx, and y direction, Δy (19)(20). The change in x and y directions are then added to the previous known values of x and y to update its current position (21) (22). The orientation is updated by taking the sum of the previous known orientation and $\Delta\theta$ divided by three. $\Delta\theta$ is divided by three to convert it from steps to degrees (23).

2.2.3 Information exchange through communication

The E-puck's infrared proximity sensors can be configured to act as an infrared messaging system, while still retaining its ability to act as a proximity sensor. The provided libraries for IR communication allow for a 4-byte message to be sent, the bytes must be combined into a long integer data type. The proximity sensors when configured to messaging mode, are set to receive messages if the receiver is activated by an incoming signal. Incoming messages are checked at a sample rate of 100 µs. Received messages are stored in a data structure called *IrcomMessage*, which can be seen in Figure 2.5. The data structure holds the value of the message, the distance to the sender, the angle the message was received at, a value corresponding to the sensor that received the message, and an error check.

```
typedef struct
{
	long int value;
	float distance;
	float direction;
```

```
        int receivingSensor;
        int error;
} IrcomMessage;
```

Figure 2.5 E-Puck Message Structure

Messages are then stored in a stack, with the oldest messages on the bottom, and the newest message on the top. The data structure and functions related to infrared messaging can be found in Appendix A.

2.3 Experiments

2.3.1 Heading Alignment

In this section, we present the experimental implementation of the heading alignment algorithm, using the so called Vicsek model Agents share their heading information with neighboring agents, and determine common heading.

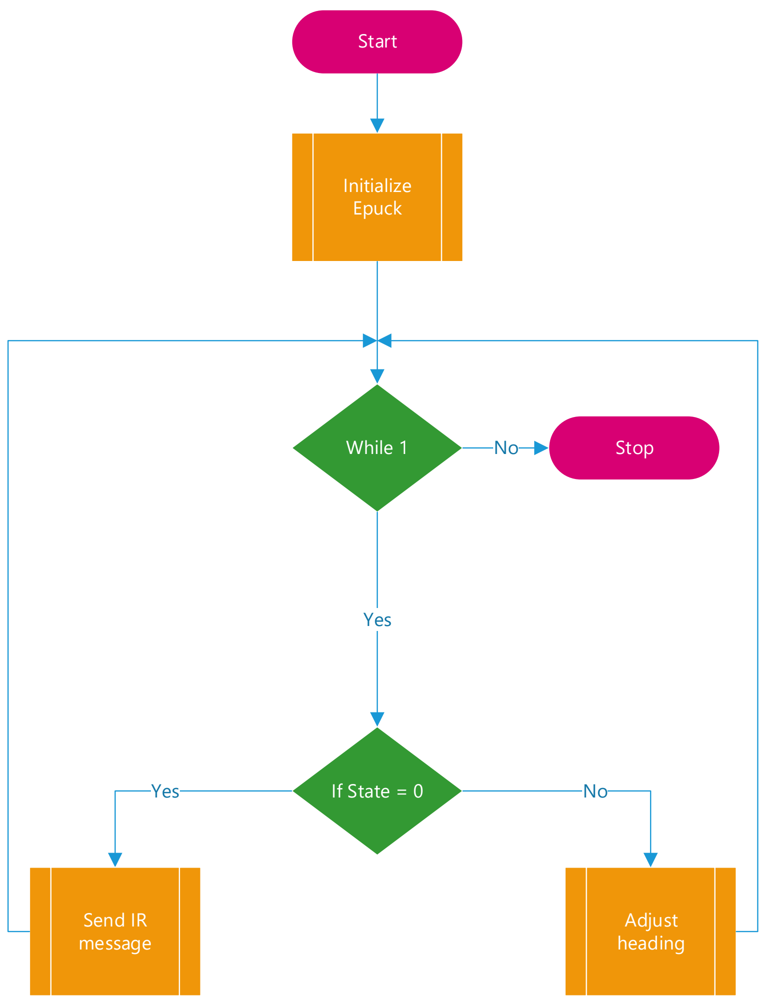


Figure 2.6 Vicsek Model Flowchart

In this experiment each E-puck is given an initial orientation at start up, and then transmit its orientation to nearby agents and also receive the neighboring agents' orientations.

Received messages are checked for any errors, and if none are found, the received orientation is stored into a buffer, when the buffer is full, the E-puck can begin to compute its new heading.

Once the buffer has been filled, the sum of the stored values is determined, and added to the agent's current orientation. The new value is then divided by the size of the buffer, giving the new heading. The difference between the new heading and the previous heading is then calculated, and the E-puck rotates by that amount. Once the E-puck has rotated, it then drives forward a small distance and the process begins again.

The corresponding snapshots with time stamps from the video is shown in Figure 2.7. It can be seen that two E-pucks find a common heading, and move together.

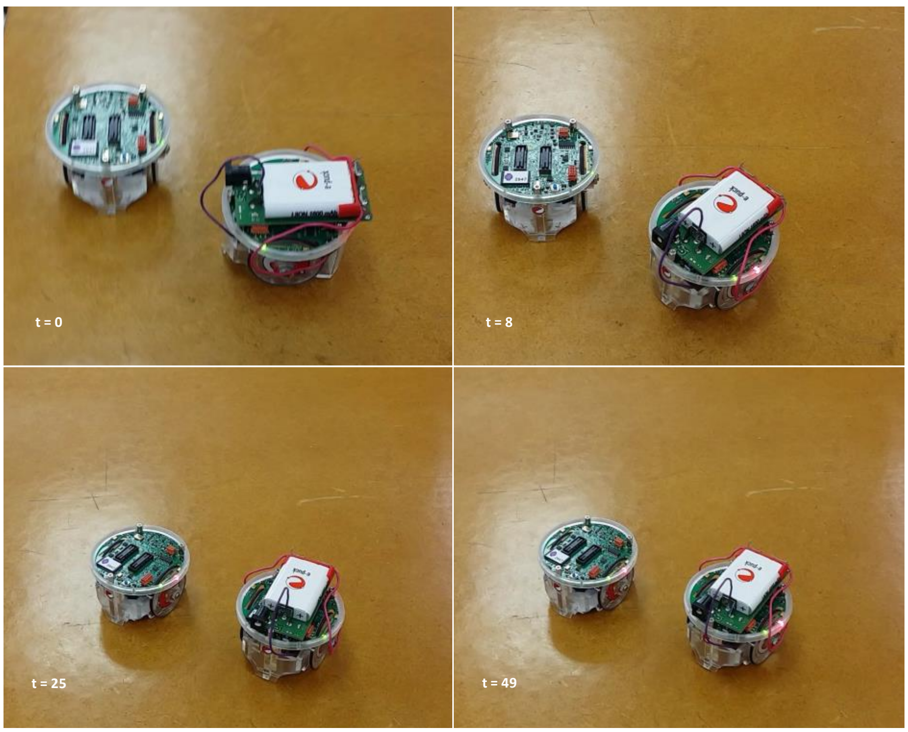


Figure 2.7 E-puck Implementation of Vicsek Model

Kilobot

3

In this chapter, we describe the implementation of cooperative control algorithms using the Kilobot. A Kilobot is shown in Figure 3.1. The Kilobot was developed by Harvard University as a low-cost platform for swarm robotics research. A Kilobot is 34 mm in height (including the legs), and has a diameter of 33 mm.

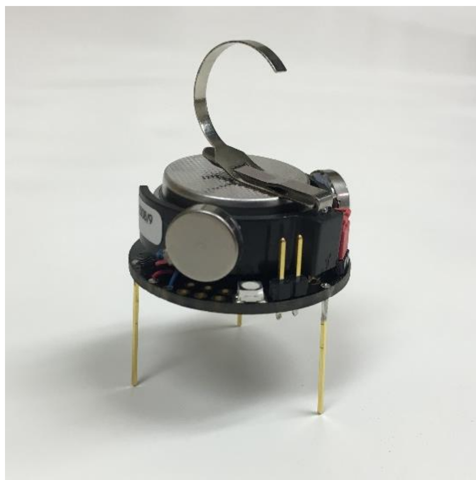


Figure 3.1 A Kilobot Robot

3.1 Overview of Kilobot

This section provides a brief overview of the Kilobot's hardware and software, as well as the system set up for experimentation.

3.1.1 Hardware

An 8-bit Atmega328p microcontroller is employed by the Kilobot, and contains 32kB of program memory, 1kB of EEPROM, and operates at a frequency of 8 MHz. The 32kB of program memory is used to store a user program as well as the bootloader. The 1kB of EEPROM is used to store important non-volatile data such as the motor calibration values.

The Kilobot robot uses two differential vibration motors for movement, and is capable of a maximum speed of 1 cm/s. The differential vibration motors are independently controllable, with 255 different power levels. For optimum performance, the differential vibration motors must be frequently calibrated.

An infrared receiver and an infrared LED is located on the underside of the Kilobot body. The underside of the Kilobot can be seen in Figure 3.2. The infrared LED is used to transmit messages to neighboring agents, while the infrared receiver is used to accept messages from neighboring agents

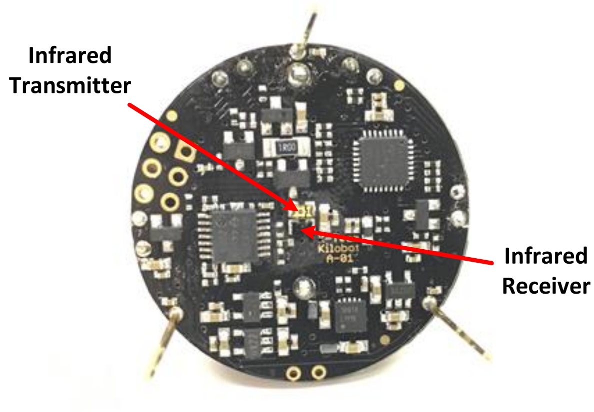


Figure 3.2 Kilobot Underside

Messages are sent at a rate of 32 kb/s, and are composed of 3 bytes (24 bits), but the least significant bit is reserved as a new message flag. Kilobots receiving a message can determine the distance to the sender based on the strength of the infrared signal. When a signal is below a threshold strength, the message will not be accepted. The rated communication distance is up to 7 cm, but under ideal conditions the maximum distance has been observed to be up to 12 cm.

Each Kilobot is also equipped with a RGB LED and a light intensity sensor. The RGB LED is capable of displaying 64 different colors, with each of the three colors having 4 different possible values. The light intensity sensor returns a value in the range 0 to 1000. The greater the value, the more intense the light.

The small legs of the Kilobots are easy to get stuck on the surface they are traversing. This can sometimes be overcome by having the motors briefly pulse to maximum power, but only in an ideal environment.

3.1.2 Software

AVR Studio 4 software is used to edit and build Kilobot projects. All programs are written in the C programming language and use the standard libraries provided with the Kilobots, which can be found in Appendix C. Once a project is built, a hex file is generated by AVR Studio 4 which is then used by a program called Kilobot Controller. The window for the Kilobot Controller can be seen in Figure 3.3. The Kilobot Controller software is used to upload hex files onto the overhead controller.

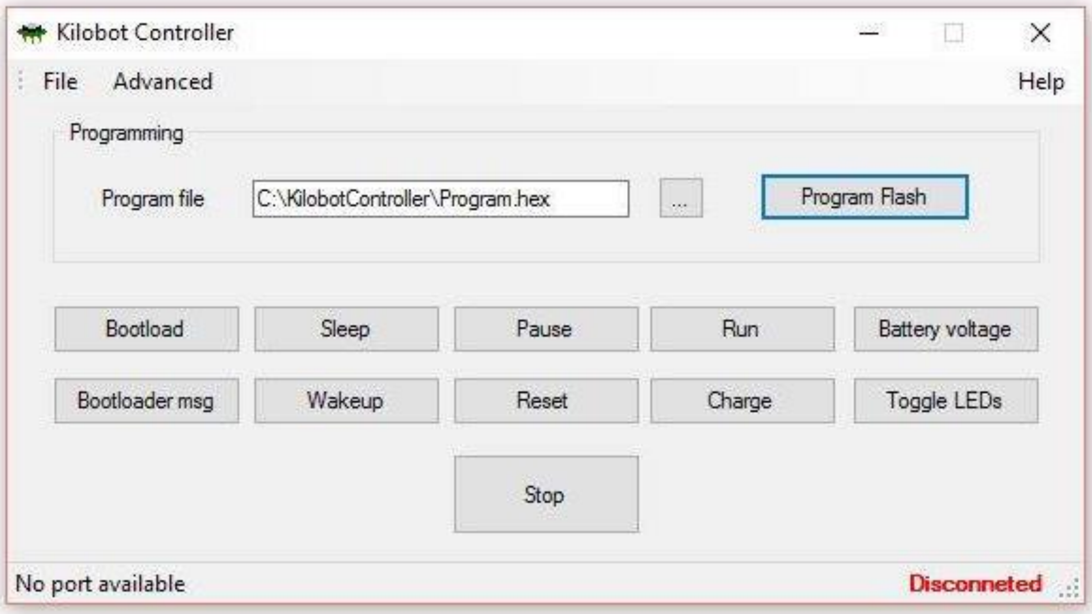


Figure 3.3 Kilobot Controller Window

The Kilobot Controller has a number of other commands, such as sleep, pause, and check battery voltage. Table 3.1 describes each command for the Kilobot Controller software.

Table 3.1 Kilobot Controller Commands

Command	Description
...	Browse through projects to choose a hex file to be uploaded onto the Kilobots.
Program Flash	Programs the OHC (Overhead controller) with the selected hex file. A black window will briefly appear, showing the progress of the programming.
Bootload	Flashes Kilobots with program stored on the OHC. Kilobots will quickly flash red, green, and then blue to show they have entered programming mode, and will then pulse blue until programming is completed. Continuous function.
Sleep	Sets Kilobots to sleep mode. Kilobots will periodically flash white while in sleep mode. Continuous function.
Pause	Sets Kilobots to pause mode. Kilobots will frequently flash yellow while in pause mode.
Run	Runs the current program on the Kilobots.
Battery Voltage	Kilobot's LED displays a color dependent on current battery charge. Green: battery voltage over 4 V Blue: battery voltage over 3.75 V Yellow: battery voltage over 3.5 V Red: battery voltage less than 3.5 V
Bootloader	Sends a message to the Kilobots to exit current program.

msg	
Wake-up	(While in sleep mode) Sets Kilobots to pause mode. Continuous function.
Reset	Restarts current program on Kilobots
Charge	Sets Kilobots to charge mode. While in charge mode, the Kilobot's LED will blink red while charging, otherwise the LED will be off.
Toggle LEDs	Toggles LEDs on OHC.
Stop	Stops whatever the OHC is currently doing. Used to end continuous functions.

On occasion, a Kilobot may need to have its firmware re-flashed onto the Atmega328p. The need to re-flash can be caused by a faulty program being flashed onto them by the user, a static discharge, a low battery while using the motors set at higher power levels, or failing to follow proper procedure when flashing a new program onto the Kilobots.

To re-flash the firmware, the Kilobot must be connected to the debugging cable, and using AVR studio, flash the Kilobot firmware hex file. This proved to be problematic as the provided materials were missing crucial steps in the process. The correct procedure was documented and is now available to the general public, and is included in Appendix E.

3.1.3 System Setup

Experiments were set up on a sleek surface. This ensures correct movements with the Kilobots. The surface was also reflective, allowing for maximum communication distance. The experiment area can be seen in the figure below.

In all experiments, 1 to 20 Kilobots are used. On the computer, a program is developed to perform a desired behavior or task and a HEX file is generated from it. The Hex file is then bootloaded onto the Kilobot controller and then flashed onto the awaiting Kilobots.

3.2 Design and Implementation

This section provides information on how controls are implemented using the Kilobot robot.

3.2.1 Motor Control

The Kilobots use two differential motors that cause vibrations in the robot's legs allowing them to move. The motors are controlled by the standard function `set_motor(char L, char R)`, with a range of input values from 0 to 255. The motors must be spun up before the desired input value can be applied. This can be seen in Figure 3.4. A value of 0xA0 must be applied to the motor(s) for 15 ms before the desired power level can be set.

```
set_motor(0xA0,0xA0);
_delay_ms(15);
set_motor(cw_in_straight,ccw_in_straight);
```

Figure 3.4 An Example of Motor Control

Although the motors can be set to custom power levels, there are four constant values defined in the EEPROM that can be used to ensure a desired action. The four values are as follows:

- *cw_in_place*
- *ccw_in_place*
- *cw_in_straight*
- *ccw_in_straight*

Combinations of these four values can be applied to the motors to allow the Kilobot to move in a forward, counter clockwise, or clockwise motion. The numerical value of the above constants is determined through calibration of the motors. It is important to note that the motors need to be calibrated frequently to insure proper behavior.

To allow for the easy use of the defined constants, a set motion function was created. The `setMotion` function will set the Kilobot's motors to perform one of the following: stop, forward, left, or right. The function `setMotion` can be seen in Figure 3.5.

```
void SetMotion(motion newMotion)
{
    if(currentMotion != newMotion)
    {
        currentMotion = newMotion;
        switch(currentMotion)
        {
            case stop:
                set_motor(0,0);
                break;
            case forward:
                set_motor(0xA0,0xA0);
                _delay_ms(15);
                set_motor(cw_in_straight,ccw_in_straight);
```

```
                    break;
                case left:
                    set_motor(0,0xA0);
                    _delay_ms(15);
                    set_motor(0,ccw_in_place);
                    break;
                case right:
                    set_motor(0xA0,0);
                    _delay_ms(15);
                    set_motor(cw_in_place,0);
                    break;
                default:
                    set_motor(0,0);
                    break;
        }
    }
}
```

Figure 3.5 setMotion Function

For ease of use, an enumerated datatype (motion) was created as the input for the `setMotion` function. Figure 3.6 shows the code for the motion type definition.

```
typedef enum {stop = 0, forward = 1, left = 2, right = 3}
motion;
```

Figure 3.6 Motion Type Definition

More on the `set_motor` function and its implementation can be found in the appendix.

3.2.2 Information Exchange through communication

The Kilobots utilize infrared light for communication. The infrared light is bounced off the ground and is received by any nearby Kilobot. This can be seen in Figure 3.7.

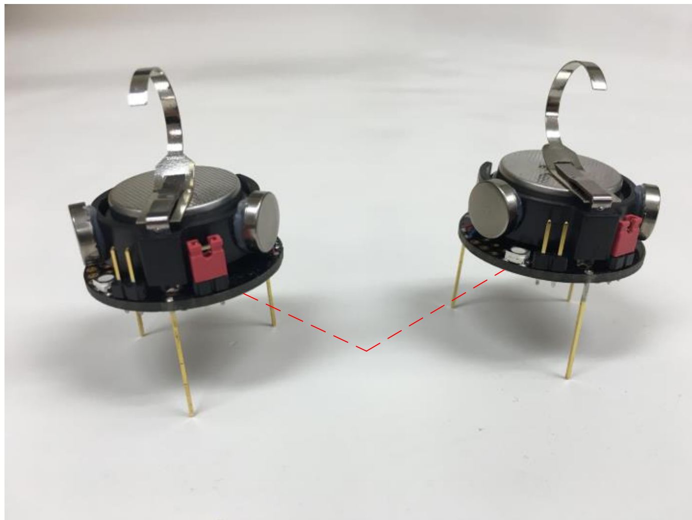


Figure 3.7 Kilobot IR Communication

The Kilobot firmware allows for 23 bits to be transmitted as a single message. This is equivalent to sending three 8-bit characters. Where the last character has an even value. The messaging function call can be seen in Figure 3.8 below.

```
message_out(0,0,0);
enable_tx =1;
```

Figure 3.8 Kilobot Messaging Function

Messages are transmitted every 200 ms, and messages are received when the IR receiver detects an incoming signal. Before a message can be sent, a series of operations must be performed on the data. First a fourth byte, which serves as a checksum, is appended to the data. The fourth byte has a value equal to the sum of the three data bytes and 128. Each of the four bytes are then operated on at the bit level. Once completed, the message is ready to be transmitted. At the start of each transmission, the IR LED is turned on for a period of 0.75 µs and then turned off for 92.25 µs. The 32b that make up the message processed, a value of 1 turns the IR LED on, while a 0 sets the IR LED off. Between each bit the IR LED is set low for 13.875 microseconds. The total time to transmit a message, from the initial IR LED flash to the last bit, is 537 microseconds.

3.2.3 Localization Via Communication

The Kilobots lack a means of observing the surrounding environment, and do not know their own orientation. The only be possible way to determine local information is via communication. Equation (1) shows the method for Kilobot localization. This method of localization is known as the gradient. G_i is the gradient value of the current agent.

$$G_i = min(messages) + 1 \tag{24}$$

By designating a single Kilobot as a leader, any other Kilobot can determine the number of Kilobots away from the leader. The leader is also known as a root node. The root node transmits a value of zero, while non-root Kilobots search for the minimum value in the messages they receive. When the minimum value is found, the non-root Kilobot increments the value by one. The value of G_i corresponds to the number of Kilobots away from the root node.

Another more advanced method of localization, employs a distributed method of trilateration. Unlike the gradient method, this method requires a minimum of three Kilobots to be configured as fixed reference points for the remaining agents to calculate their current location. The remaining agents are given the coordinates (0,0) as their initial position.

$$C_i = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{25}$$

$$V_i = \left(\frac{x_i - x_j}{C_i}, \frac{y_i - y_j}{c_i}\right) \tag{26}$$

$$N_i = (x_j - D_{ij} * Vx_i, y_j - D_{ij} * Vy_i) \tag{27}$$

$$(x, y) = \left(x_i - \frac{(x_i - Nx_i)}{4,} y_i - \frac{(y_i - Ny_i)}{4}\right) \tag{28}$$

After initialization, every reference agent transmits their coordinates to all non-localized agents in range, from these message the distance D_{ij}, can be calculated based on the light intensity of the message. Non-localized agents store the received information and calculated distance until three unique reference points are detected. Once three unique points of reference are found, the non-localized agent calculates the distance from its alleged current position to the reference points [2]. Unit direction vectors, V_i, are then generated, with the tails located at the reference points and the heads at the current position of the non-localized agent (26). By taking the difference of the reference agents and the product of the measured distances, D_{ij}, and the unit direction vectors, V_i,a new set of coordinates is generated that represent where the agent believes it is, N_i, with reference to each individual reference point (27). Finally, the non-localized agents position is updated by taking the difference of its previous position with that of the a fourth of the difference of previous position and N_i (28). By iteratively performing these steps, the non-localized agent coordinates quickly converge to the correct values.

3.3 Experiments

In this section, the experimental results for the Kilobot are presented. Each subsection contains a flowchart describing the method of implementation, as well as photos captured during runtime. The photos are timestamped with the variable t, in seconds.

3.3.1 Gradient

As mentioned in section 3.2.3 Localization Via Communication, the Gradient is a one dimensional method of localization that allows for a Kilobot to determine how many Kilobots away it is from a root node. A Kilobot was predetermined to be the root node. The other Kilobots generated their own ID by using a random number generator, and would then proceed to perform the gradient algorithm as shown in Figure 3.9.

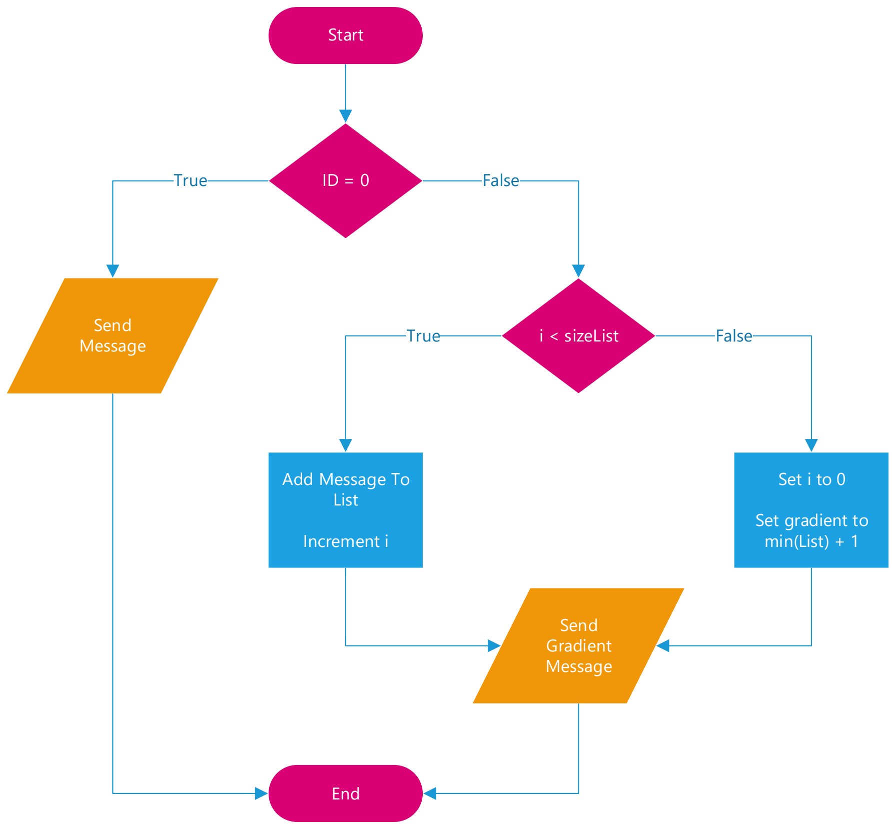


Figure 3.9 Gradient Flowchart

Figure 3.10 shows snapshots from a video taken during implementation of the gradient function. It can be seen that the gradient value cascades through the Kilobots until localization is achieved. In this experiment, the total run time was 12 seconds, but the amount of time it takes for localization to be achieved increases as the number of Kilobots increases.

Figure 3.10 Gradient Implementation

3.3.2 Orbiting

As mentioned in section 3.1.1 Hardware, the Kilobots determine the distance from one another based on the strength of incoming messages. Using the distance information, and a simple set of rules, a Kilobot can perform an orbiting motion around another Kilobot. The rules for orbiting are determined from three zones.

A zone is an area of space that an orbiting Kilobot may or may not occupy. The area for a zone is defined by a distance to the stationary Kilobot from a point in space. The three zones are the following: zone of repulsion, zone of orientation, and zone of attraction. Figure 3.11 shows the three zones.

When a Kilobot is in the zone of repulsion, the orbiting Kilobot is notified that it is too close to the stationary Kilobot. The orbiting Kilobot will then move away from the stationary Kilobot. When a Kilobot is in the zone of orientation, the orbiting Kilobot is notified that it can move in a forward motion. When a Kilobot is in the zone of attraction, the orbiting Kilobot is notified that it is too far from the stationary Kilobot. The orbiting Kilobot will then move towards the stationary Kilobot.

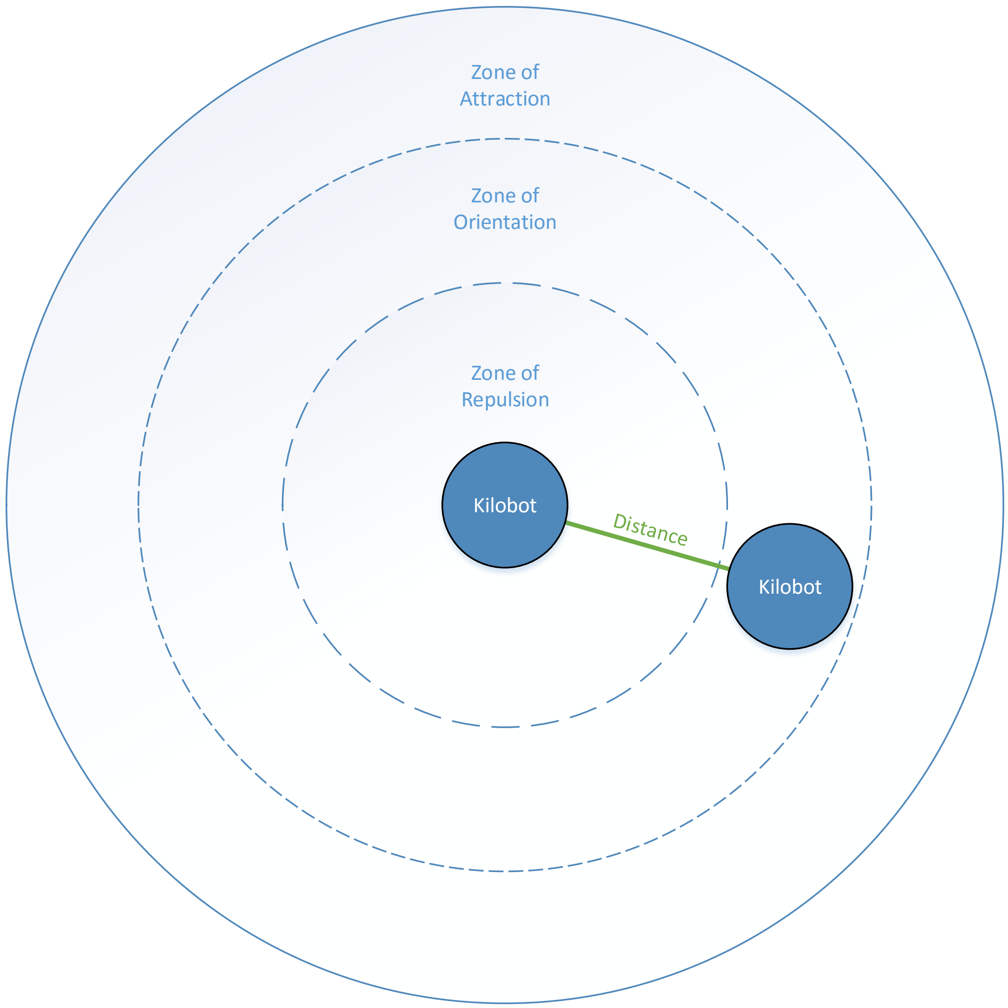


Figure 3.11 Orbiting Zones

Figure 3.12 shows the control flow diagram that was implemented on the Kilobots. If the identification number given to the Kilobot is zero, then messages will only be sent out by the Kilobot. The Kilobot with this identification number is known as a root.

The root sends messages to allow any non-root Kilobot to determine its distance to the root. A non-root Kilobot will compare the computed distance with the distance defining the zone of repulsion. If the Kilobot's calculated distance is less than the zone of repulsion distance, then the Kilobot will turn right. If the Kilobot's calculated distance is greater than the zone of repulsion distance, then the Kilobot will compare its distance to the zone of orientation distance. If the Kilobot's distance is greater than the zone of orientation distance, then the Kilobot will turn left. If this comparison is false, then the Kilobot will move in a straight line.

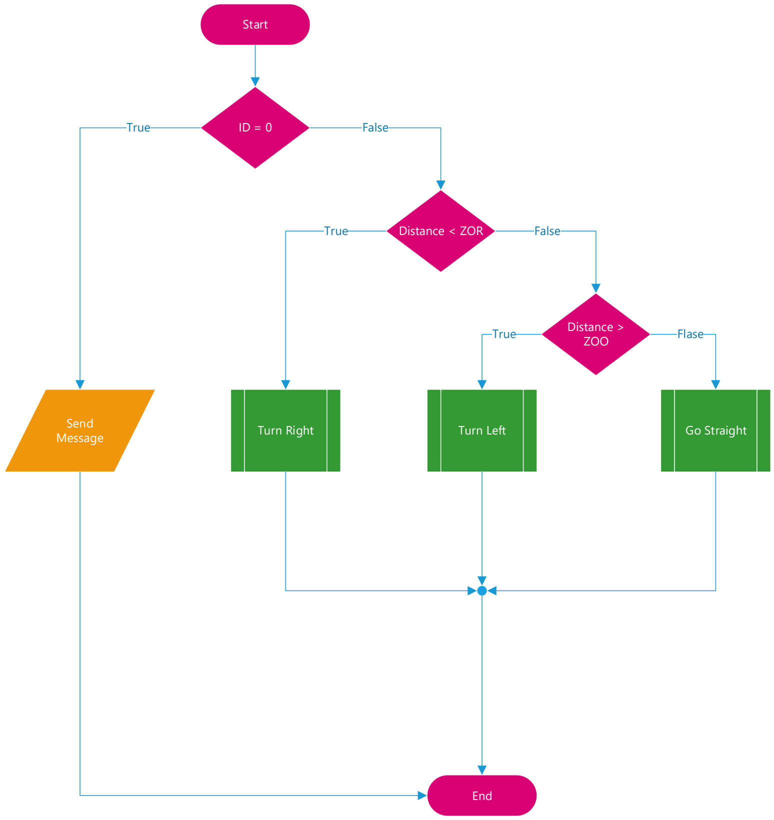


Figure 3.12 Orbiting Flowchart

A video of the orbiting implementation was taken. Snapshots are shown in Figure 3.13.

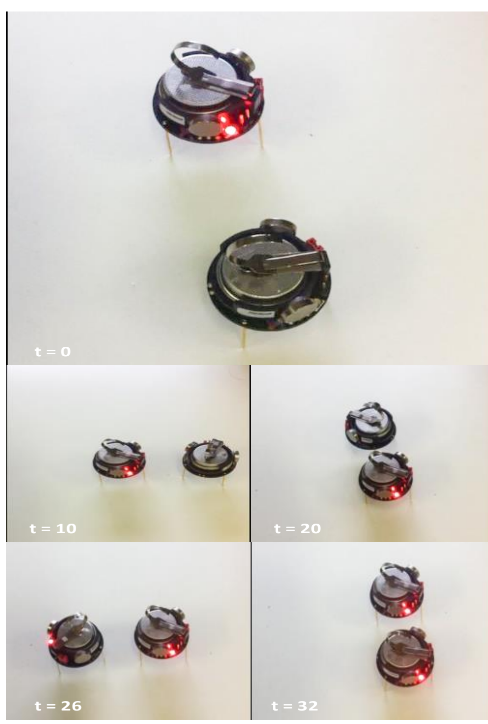


Figure 3.13 Orbiting Implementation

3.3.3 Asynchronous Consensus

By combining the gradient and orbiting algorithms, it is possible to have the Kilobots converge to a signal fixed location. An agent is designated as a root node, which is placed in a desired location. This location will be the convergence point for the other agents. A flowchart for asynchronous consensus can be seen in Figure 3.14. First the gradient algorithm is performed until a timer flag is thrown. Then the agents perform orbiting, but the radii of the three zones decreases at specified time intervals. Over time, the decaying orbit causes the agents to converge at the root node. The converging agents are constantly aware of the gradient vales directly above and below them, and only perform any orbiting movement if the Kilobot with a gradient value above their own is within range.

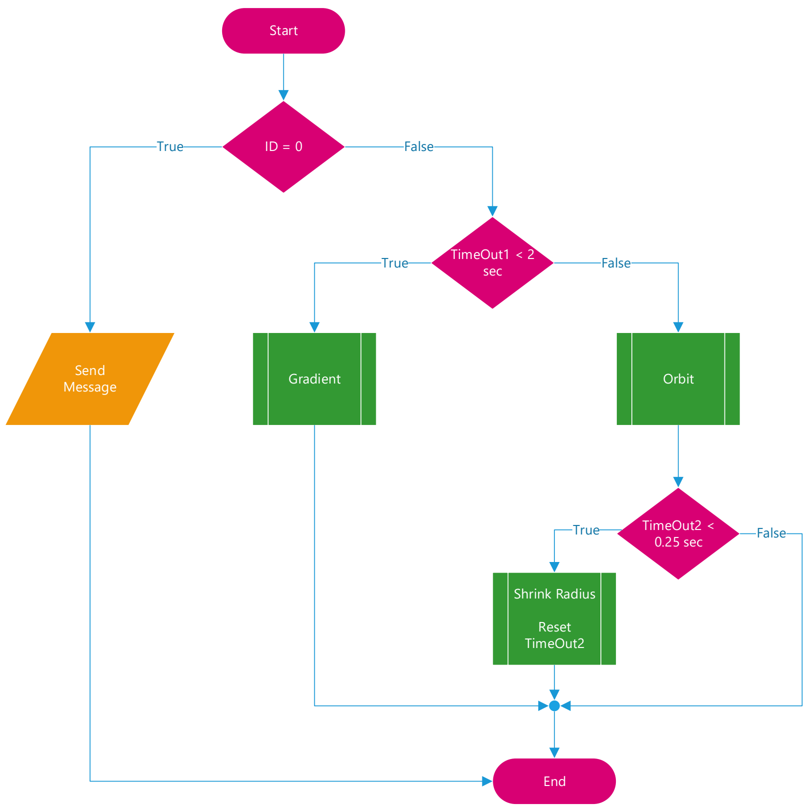


Figure 3.14 Asynchronous Consensus Flowchart

Figure 3.15 shows four photos captured during an implementation of asynchronous consensus. In this experiment the root agent is displaying the color red. At the first time stamp, all three agents are dispersed, and the gradient algorithm has been completed by each Kilobot. The next time stamp (t =13), shows that the blue agent has moved next to the green agent. This signals the green agent to start moving. The third image shows that the green agent has moved to the root, with the blue agent is following behind the green agent. The final image shows all agents at the desired location.

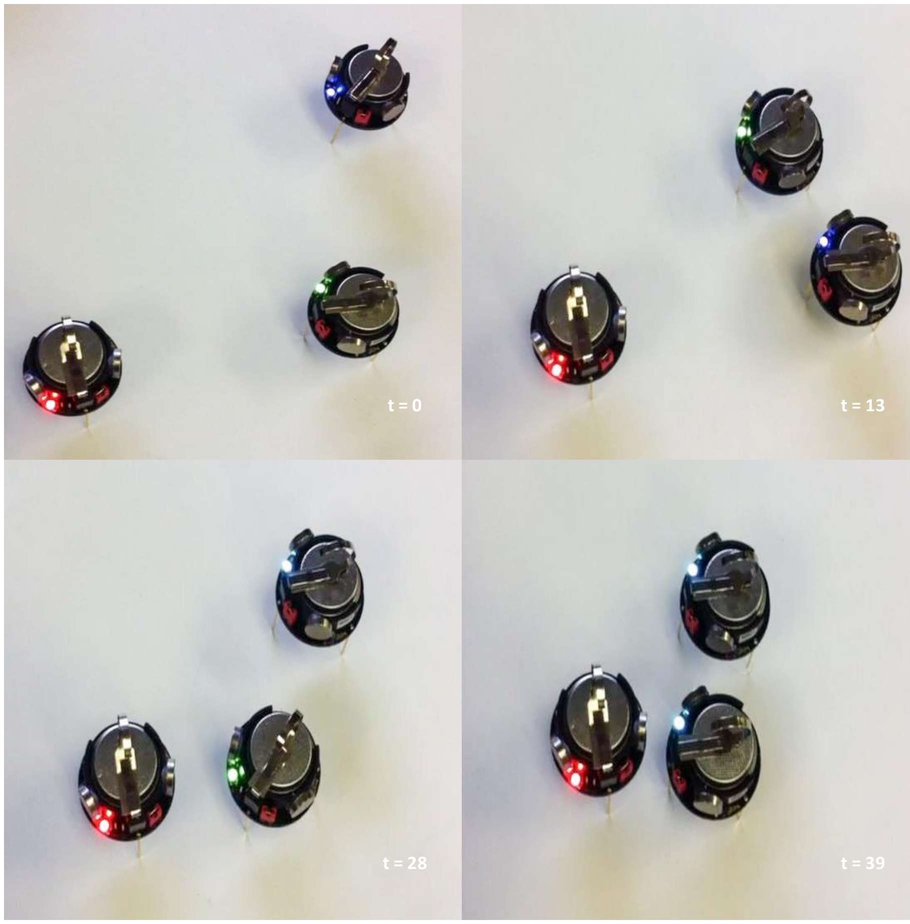


Figure 3.15 Asynchronous Consensus Implementation

3.3.4 Light Following

Using values from the Kilobot's ambient light sensor, the Kilobots can be made to follow a light source. A control flow diagram for the implementation of light following is shown in Figure 3.16. Multiple readings from the ambient light sensor are taken, and then the average of the readings is calculated. This average is compared against two threshold values. The threshold values were determined by testing the light sensor in different lighting conditions. Sensor values were measured in a room with natural lighting, a light directly on the sensor, and inside a sealed box. If the average value is less than or equal to the lower threshold, then the Kilobot turns to the left. If the average value is greater than or equal to the higher threshold, then the Kilobot turns right. As the Kilobot turns left or right, the light sensor attempts to center the light source. This constant centering makes the agent move towards the light source.

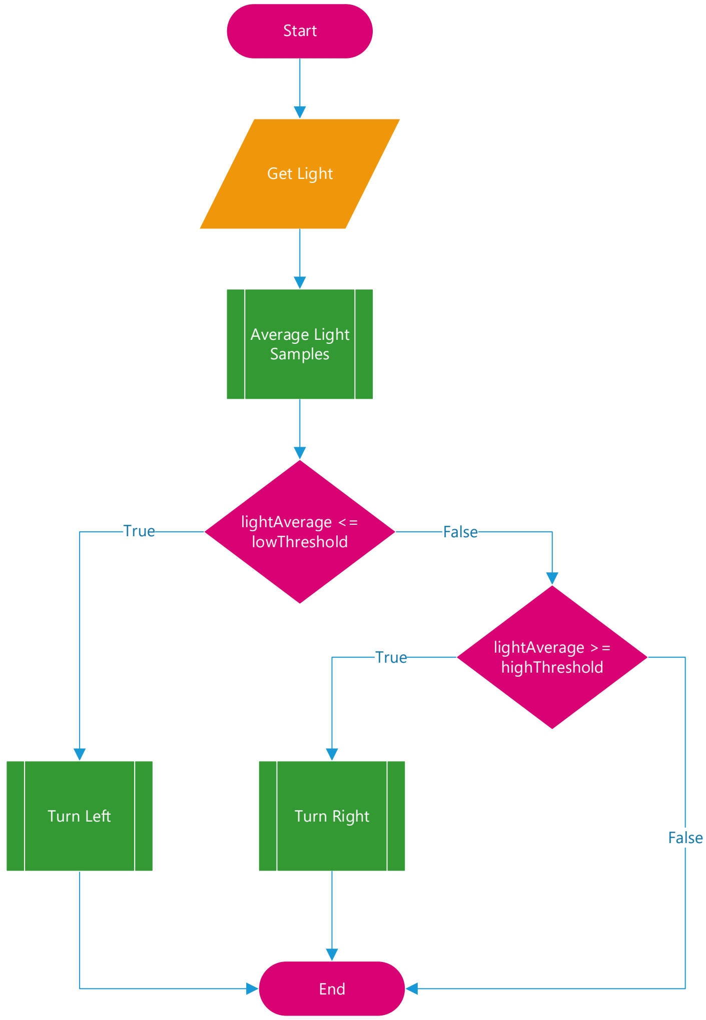


Figure 3.16 Light Following Flowchart

Figure 3.17, shows four Kilobots moving to the light source, which is situated directly behind the camera.

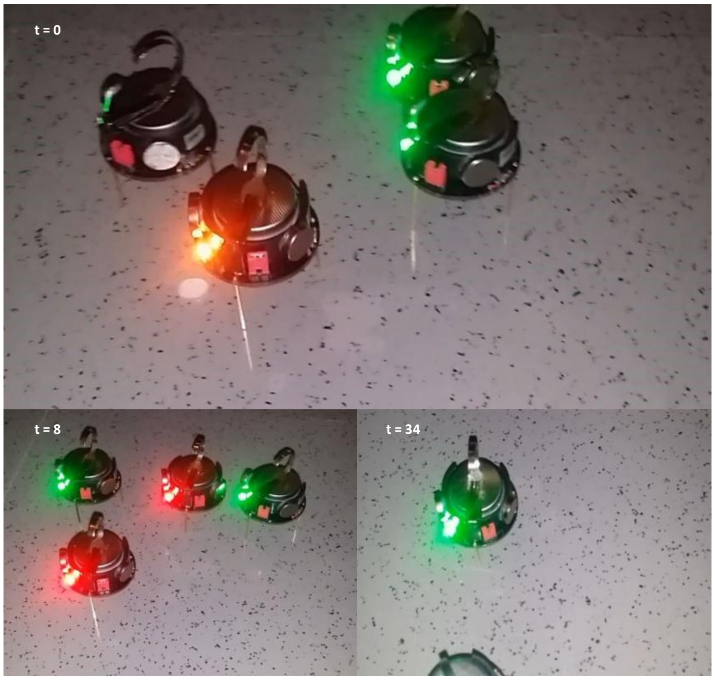


Figure 3.17 Light Following Implementation

3.3.5 Sending Messages from an Outside Source / Controllable Node

As mentioned in section 3.1.2 Software, the Kilobot Controller software allows users to perform several different tasks, but it lacks the ability to send user generated messages to the Kilobots. By using an Atmega128, an infrared LED, a 330 Ω resistor, and Atmel Studio 6.1, a program was written that mimics how the Kilobots send messages. The Atmega128's system clock was configured to 8 MHz to match the speed of the Kilobots. The program used a timer based interrupt that triggered every 200 ms and performed the same messaging protocol as the Kilobots.

The messaging program was verified by using a Kilobot that was programmed to perform specific actions depending on the values contained in the message being sent. For example, setting the Kilobot to a root mode, moving in a given direction, or performing light following. Because the Kilobot's behaviors can be changed on the fly, it is known as a controllable node.

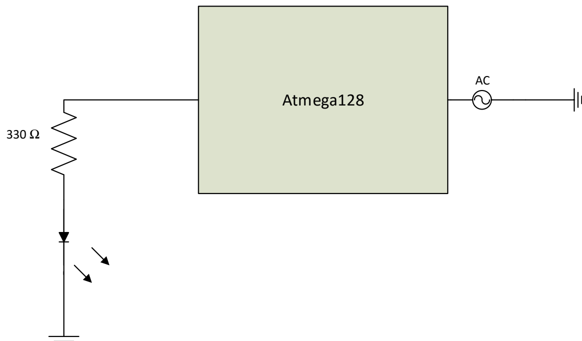


Figure 3.18 Messaging Circuit

QBot 2

4

In this chapter, we describe the implementation of cooperative control algorithms using the QBot 2. A figure showing a QBot 2 is shown below.

Figure 4.1 A QBot 2 at Bradley University

4.1 Overview of QBot 2

4.1.1 Hardware

A QBot 2 is composed of a Kobuki robot base by Yujin Robot, a Microsoft Kinect RGB camera and depth sensor, and a Quanser DAQ with a wireless embedded target computer. The Kobuki robot platform has two differential drive wheels, with a maximum speed of 0.7 m/s. The differential drive wheels contain built in encoders. The height of the Kobuki platform, including the Kinect sensor, is 27 cm, and the diameter is 35 cm. Three digital bump sensors, three digital wheel drop sensors, three analog and digital cliff sensors, and a 3-axis gyroscope are also part of the Kobuki platform.

The Microsoft Kinect sensor is mounted on top of the Kobuki robot, allowing for different viewing orientations. The minimum viewing angle is 21.5 downwards. The Kinect has a horizontal field of view limited to 57◦, and a vertical field of view limited to 43◦ data can be captured and processed, as well as 11-bit depth. RGB image data. The RGB image has a minimum resolution of 640×480 pixels and a maximum resolution of 1280×1024 pixels. The depth image has a resolution of 640 × 480 pixels, and has a range of 0.5 to 6 meters.

The embedded target computer uses the Gumstix DuoVero computer which contains 1 GB of RAM, and uses a Texas Instruments CPU with a base clock speed of 1 GHz. The

Gumstix DuoVero computer runs a real-time control software, known as QUARC to interface with QBot 2 data acquisition card (DAQ) for all sensor data processing. QUARC also supports additional IO configurations, allowing users to customize the QBot 2. Additional IO includes: four PWM outputs, four analog inputs, eight reconfigurable digital I/O, one UART, one SPI, and one I2C.

4.1.2 Software

MATLAB/Simulink software integrated with QUARC is used to interface the target computer. A Simulink model can be seen in Figure 4.2. Controllers are developed in Simulink with QUARC on the host computer, and then code can be generated and downloaded to the target computer wirelessly. Several main QUARC blocks used to communicate with the QBot 2 include Hardware in the Loop (HIL) initialize block, which configures the drivers and hardware interface for QBot 2; HIL Read/Write, which are used to read sensory data and drive motors; Kinect Initialize; Kinect Get Image; and Kinect Get Depth.

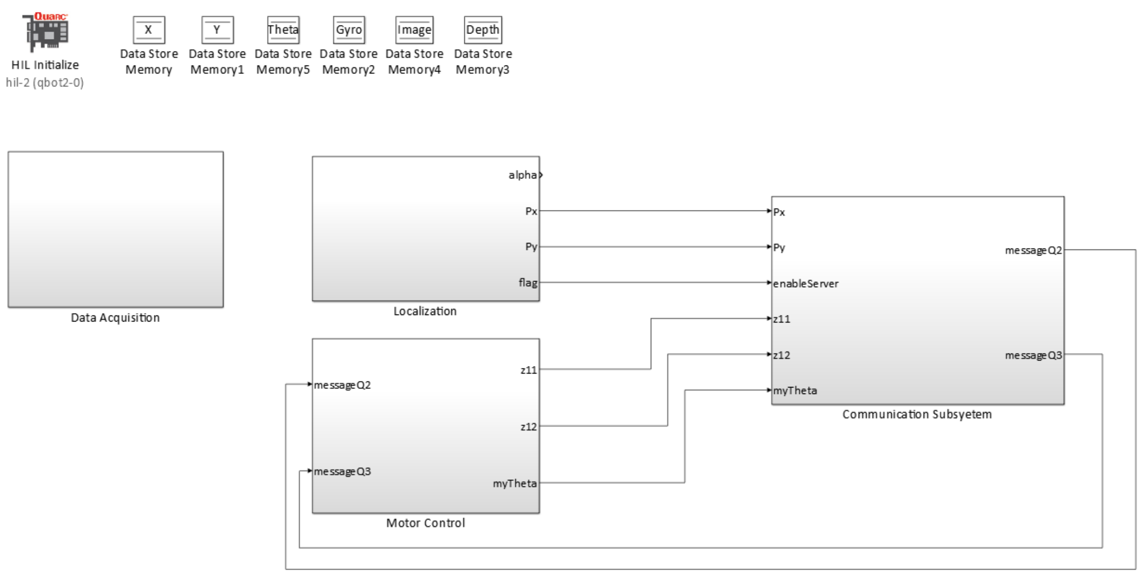


Figure 4.2 Overall Simulink Model

4.2 Design and Implementation

4.2.1 Localization Using Kinect Sensor

At start up, the QBot 2 is initialized to a local reference frame, with the origin at the center of the QBot 2. The local reference frame initializing becomes a problem when multiple QBot 2s are being used. To overcome this issue, two of the three QBot 2s positions are determined with reference to the remaining QBot. These two locations are then used to translate the two QBot 2s reference frames to the other QBot 2s local reference frame, creating a global coordinate system. The Kinect sensor of a QBot 2 can be used to determine the coordinates of an object. This means that a QBot 2 that is within the global reference frame can determine the coordinates of another QBot 2 with reference to the global frame.

Before the Kinect sensor can calculate the position of a QBot 2, it must first identify it. Identification is possible through the use of a QUARC Simulink block called Find Object. A description of the Find Object block can be found in Appendix G. Objects are determined by adjusting the RGB values in the parameter window. The threshold parameter gives an allowable error for acceptable RGB values. The Find Object block also has a minimum size parameter, which estimates the minimum size of the desired object in number of pixels. From Figure 4.1, it can be seen that the standard QBot 2 is a black color.

This presents a problem; the QBot 2 blends into the background when color identification is trying to be completed. To overcome this issue, the QBot was outfitted with colored construction paper. An example of this can be seen in Figure 4.3 below. The Find Object block outputs the center of mass of the desired object. The center of mass is given as two outputs, an x value (the image matrix's column value) and a y value (the image matrix's row value). These values can be used in conjunction with a depth image to determine the distance to the object.

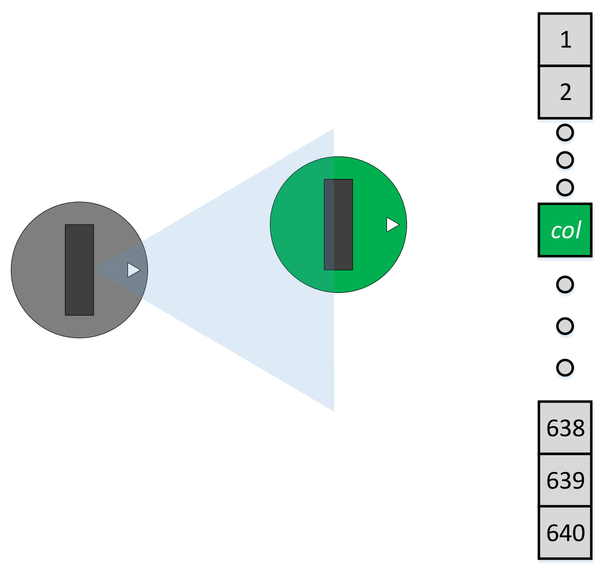


Figure 4.3 Above-View of QBot 2 Localization

The QUARC Simulink library provides a block, called Kinect Get Depth, which captures a 640 × 480 depth image. The depth image contains a distance value, in mm, that can be used with the previously calculated center of mass values to determine the distance in the x direction to a desired object. The distance in the y direction must be determined by calculating the angle of the object relative to the center of the image. The angles needed are determined from equation (29).

$$\alpha = (320 - pixel)\frac{57}{640}\frac{\pi}{180} \quad (29)$$

Where the number 320 refers to the center of the captured image (640 × 480), 57 refers to the Kinects field of view, and dividing the field of view by 640 gives an angle value per pixel. Pixel refers to a current pixel in the range of 1 to 640. By doing this an angle is determined for each individual pixel with regards to the center of the image. Because the pixel variable ranges from 1 to 640, α is returned as an array, with units of radians. A visualization of α can be found in Figure 4.4. These angles are calculated in the model properties of Simulink as a post load function. The calculated angles, the depth image, the center of mass of the object, and the gyroscope value are then used to determine the coordinates in the global reference frame, which will be transmitted to the other QBot 2s for the implementation of distributed controls.

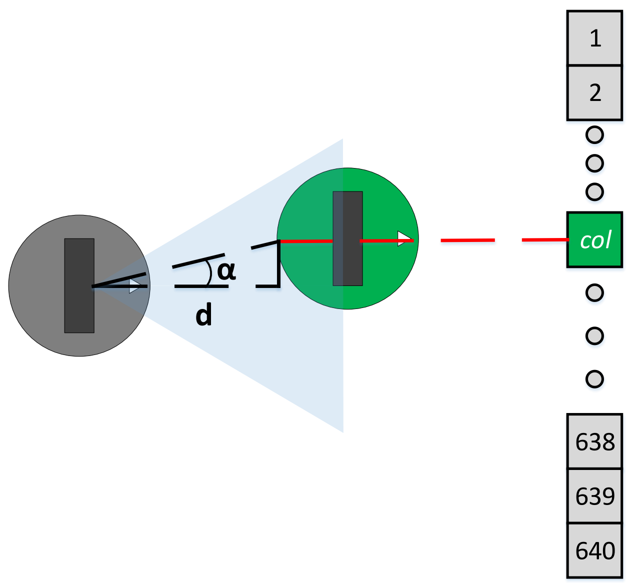


Figure 4.4 Expanded Above-View of QBot 2 Localization

4.2.2 Information Exchange Through Communication

The QBot 2 utilizes IEEE 802.11 b/g/n protocol for communication. The QUARC Simulink library for communication provides basic, intermediate, and advanced blocks. This allows for a number of different communication topologies. The simplest way to set up communication between QBots is by utilizing QUARCs basic communication blocks. The basic communication blocks are the stream server block and the stream client block.

QUARC allows for easy implementation of different communication protocols, where each are specified by the URI parameter. In this case, the protocol being utilized is TCP/IP, and each QBot is given a unique IP address to be identified with. The stream server block sends its input to the stream client block, as well as receives output from the stream client block. The input and output values are a single value or an array, where the length of the data is determined by the default output value in both parameter windows.

The stream client block works much like the stream server block, but the URI parameter is set to the same value that is used in the stream server block. This notifies the stream client block that it should search and connect to a host with that URI. In the experiments, we adopt the time-varying communication topologies following a time sequence $\{t_k, k = 0,1, \dots\}$ as

$$S_1(t) = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}, t \in [t_{2k}, t_{2k+1}) \tag{30}$$

$$S_1(t) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, t \in [t_{2k+1}, t_{2(k+1)}\big) \tag{31}$$

That is, for time intervals $[t_{2k}, t_{2k+1})$, the communication topology $S_1(t)$ is strongly connected, and messages that are sent between the QBots contain information about their coordinates values x_i, y_i, p_{ix} and p_{iy}. For time intervals $[t_{2k+1}, t_{2(k+1)})$, there is no information sent among QBot 2s. Nonetheless, the overall communication pattern consisting of $S_1(t)$ and $S_2(t)$ is still strongly connected, and satisfies the network connectivity condition for coordination of multiple dynamical systems [13].

4.2.3 Motor Control

The basis for the motor control is the HIL Read and HIL Write blocks. These blocks allow Simulink to access the input and output ports of the QBot 2. The port numbers for the left and right motors are 2000 and 2001 respectively. The HIL write block is used to write to the motors, but can also be used to write to different outputs such as the PWMs. The HIL read block is used to read the encoders, gyroscope, bumper sensors, and any other sensor inputs for the QBot 2. The encoder values for the QBots are used to calculate the left and right wheel velocities, then these velocities are used to calculate the QBots current x and y positions, as well as its angle. All this information along with the received communication information is sent to a distributed control module which is designed based on the algorithms in (13) (14) and (15)-(16).

4.3 Experiments

In this section, we report the experimental testing results for solving problem 2 and problem 3 using distributed controls. Three QBot 2s are used in the experiments, and their IP addresses are 192.168.2.49, 192.168.2.50, and 192.168.2.51, respectively. In the results presented below, the robot trajectory data from the real run were recorded and plotted using MATLAB. In the plots, a blue line represents the first QBot 2 to be activated. A green line represents the second QBot 2, which is identified by the first QBot 2 for localization. A red line represents the third QBot 2, which is identified by the second QBot

2 for localization. Each QBot 2 is also numbered by the last two digits of their IP address in the legend. Squares on plots represent the starting position of a QBot 2, and a circle represents the QBot 2 final location.

4.3.1 Rendezvous Control

Different control gains k_i were used to test the control algorithms in (13)-(14). Figure 4.5,Figure 4.6, and Figure 4.7 depict the phase plot and individual trajectories for robots for the case of k_i = 2. The corresponding snapshots with time stamps from the video clip are shown in figure 8. It can be seen that rendezvous is achieved. With the control gain k_i = 6, the results are illustrated in Figure 4.8, Figure 4.9, and Figure 4.10. It can be seen that convergence can be reached quickly at about t = 20s.

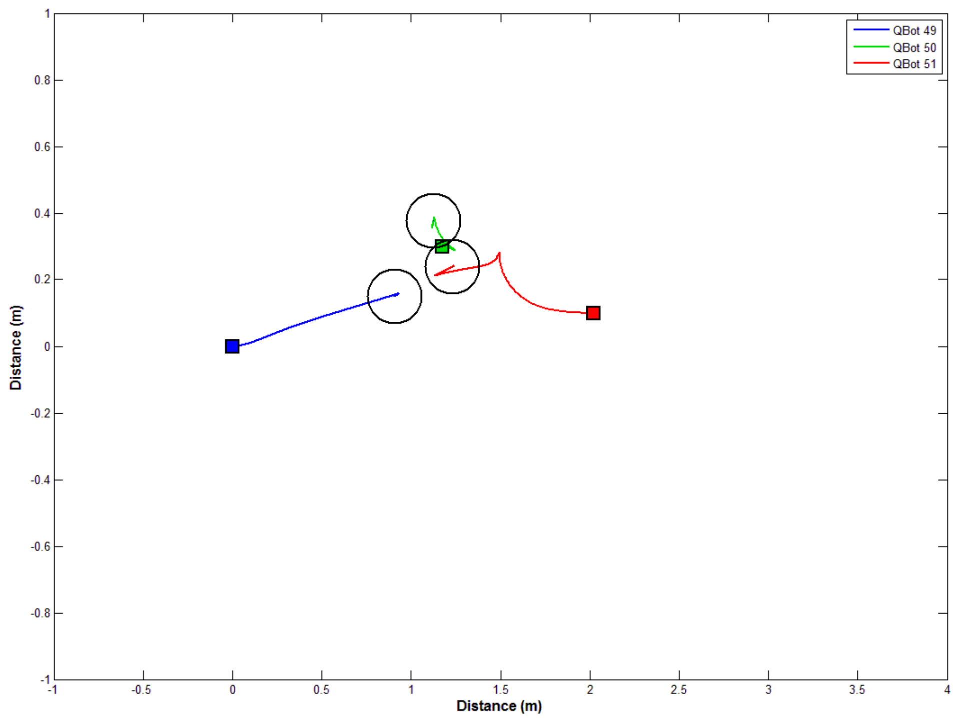


Figure 4.5 Rendezvous Implementation, k_i = 2

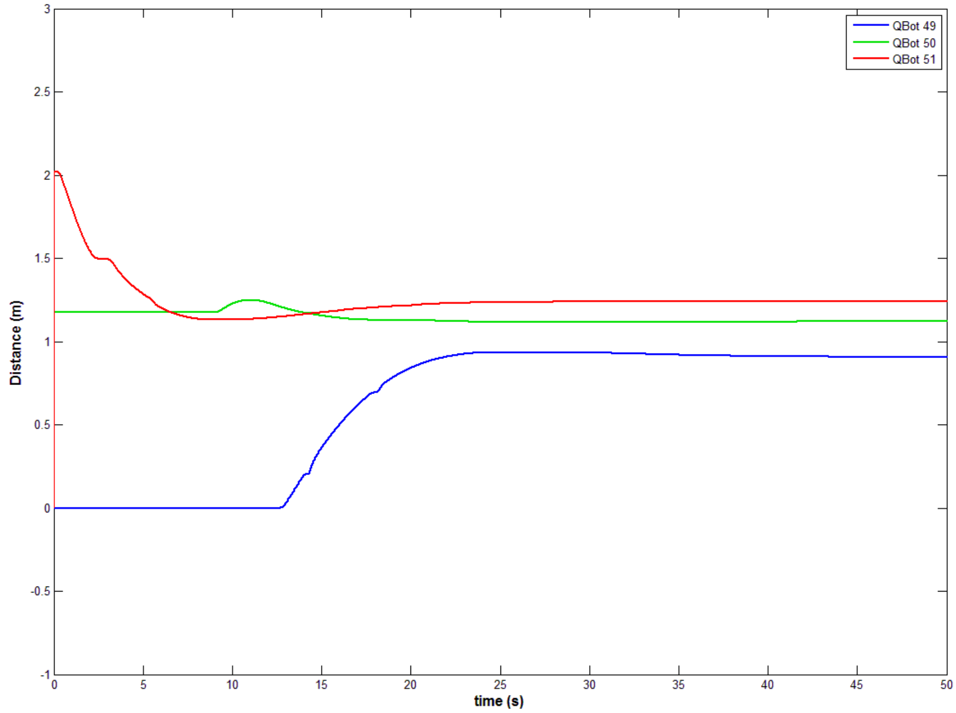


Figure 4.6 Rendezvous X Position vs Time, k_i = 2

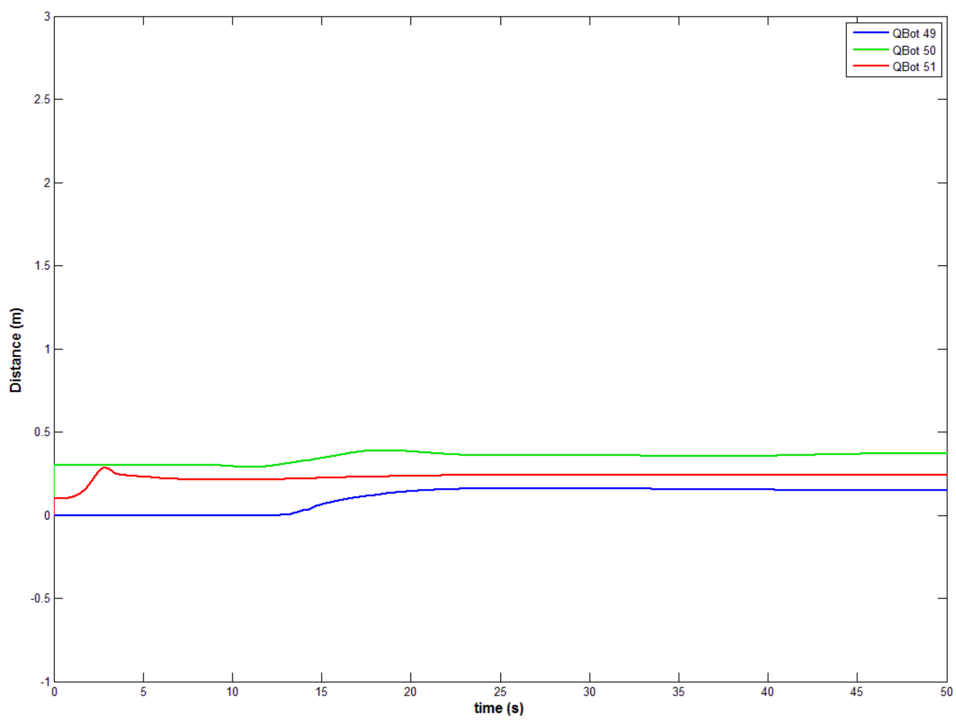


Figure 4.7 Rendezvous Y Position vs Time, k_i = 2

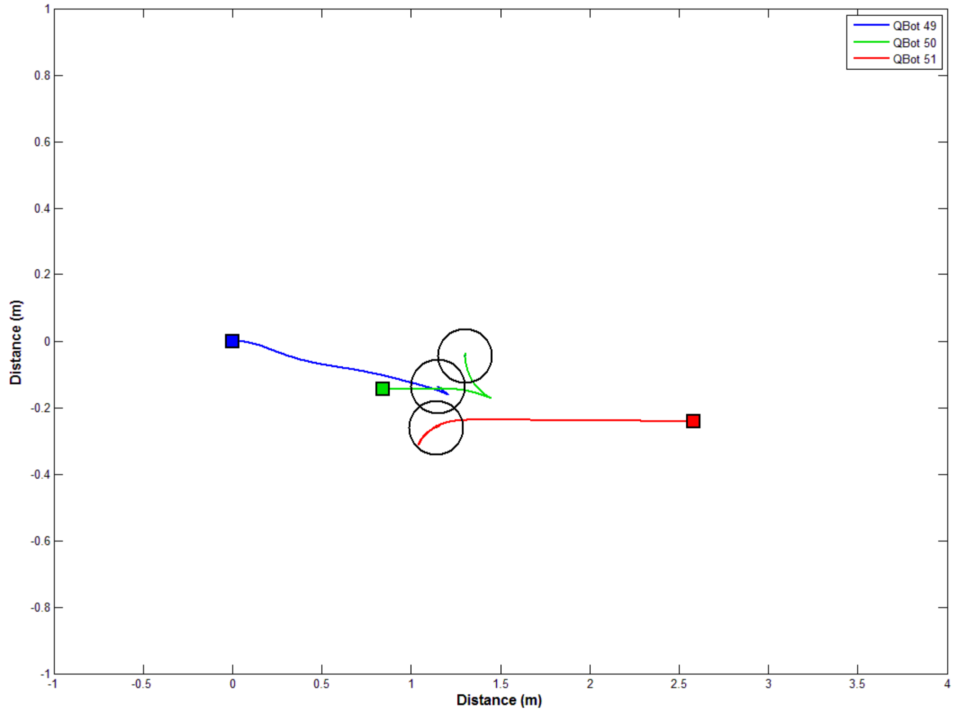


Figure 4.8 Rendezvous Implementation, $k_i = 6$

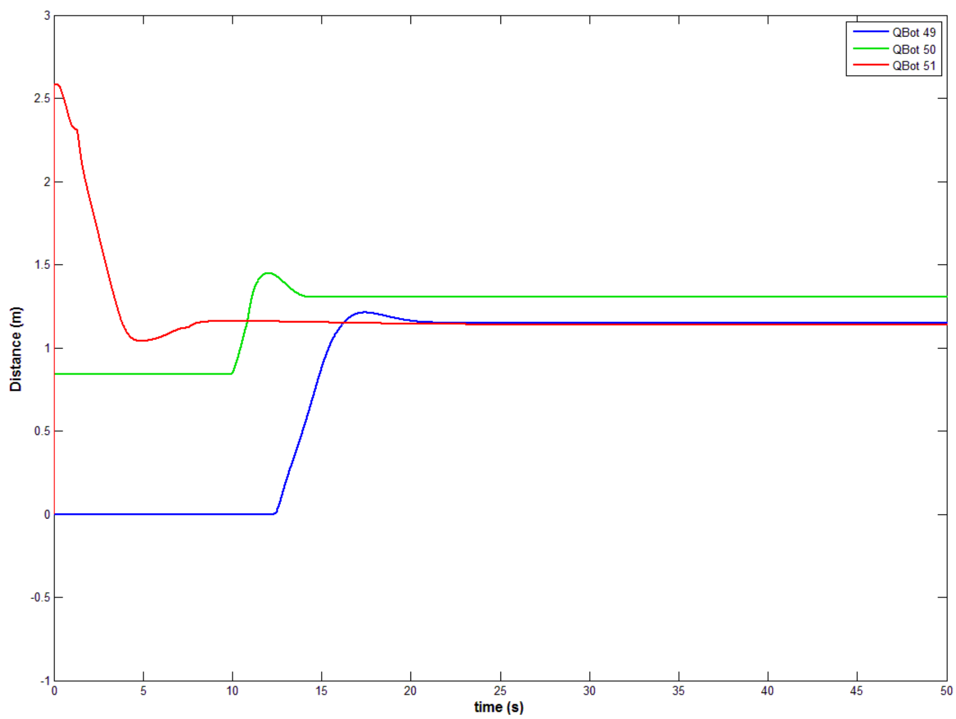


Figure 4.9 Rendezvous X Position vs Time, $k_i = 6$

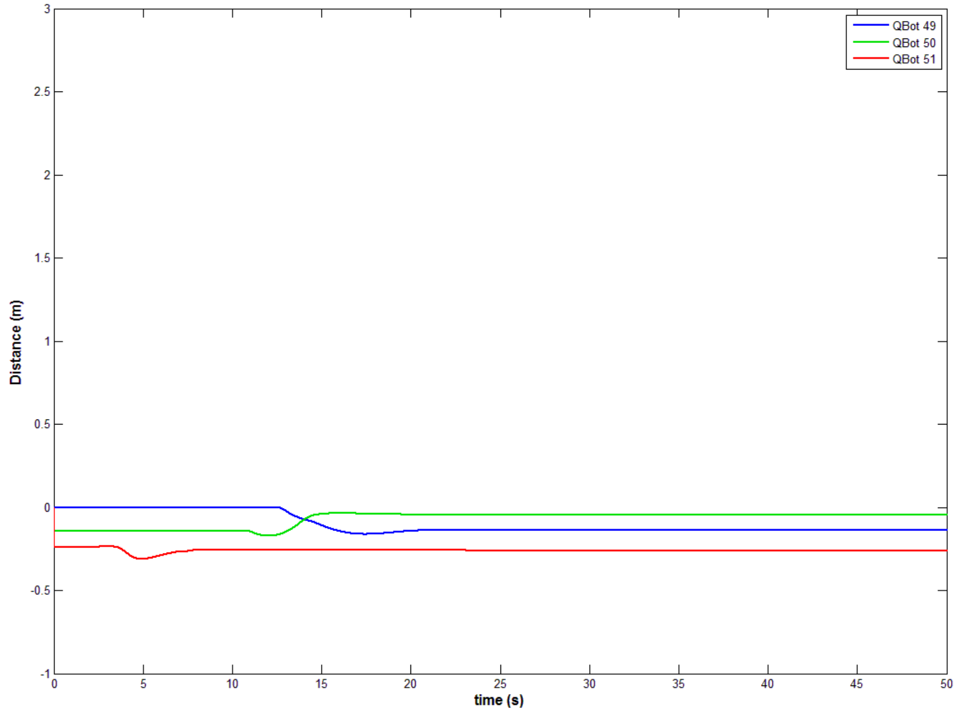


Figure 4.10 Rendezvous Y Position vs Time, k_i = 6

4.3.2 Formation Control

The formation algorithm was tested by giving the QBot 2s a shape to move into. In this case, the QBot 2s were controlled to form a triangle. Figure 4.11, Figure 4.12, and Figure 4.13 show the phase plot and system responses. The snapshots are illustrated in figure 15. It can be seen that robots converge to the desired right triangle formation.

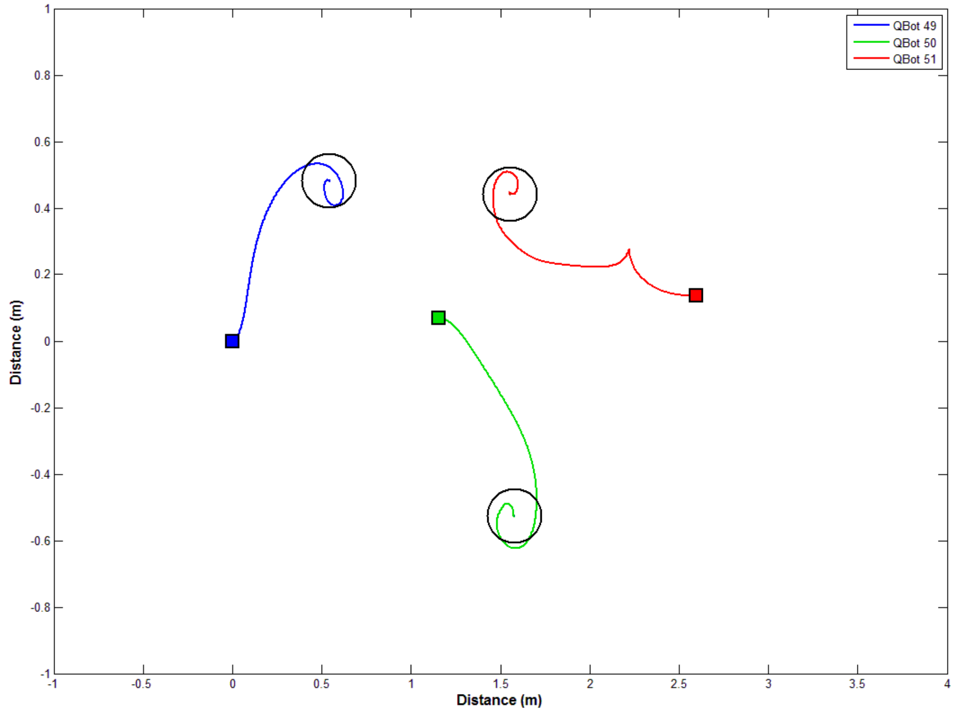


Figure 4.11 Formation Control Implementation

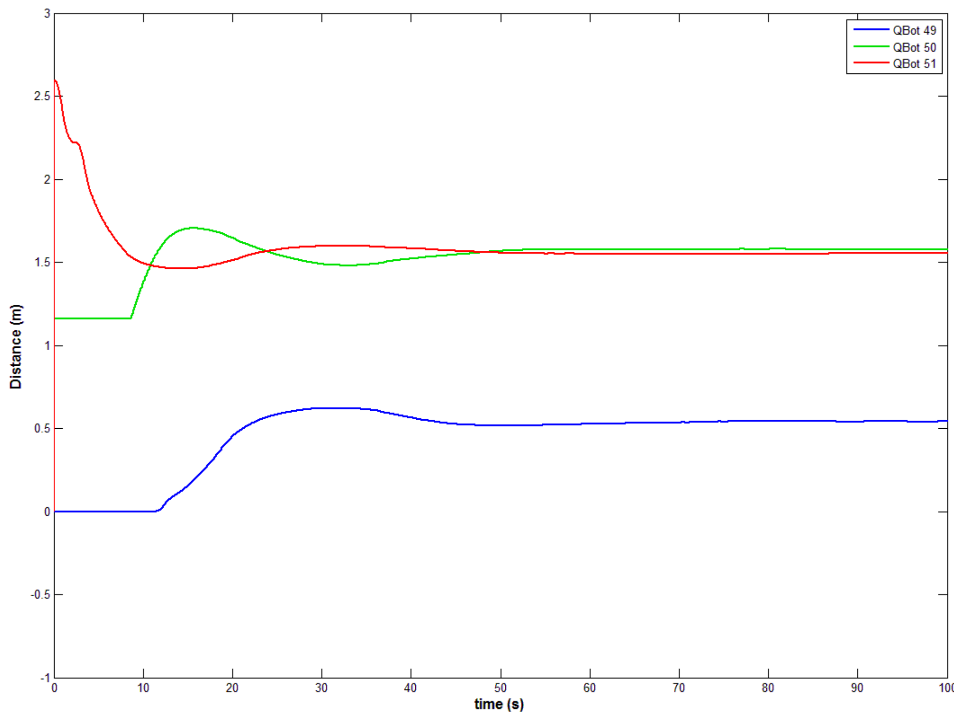


Figure 4.12 Formation Control X Position vs Time

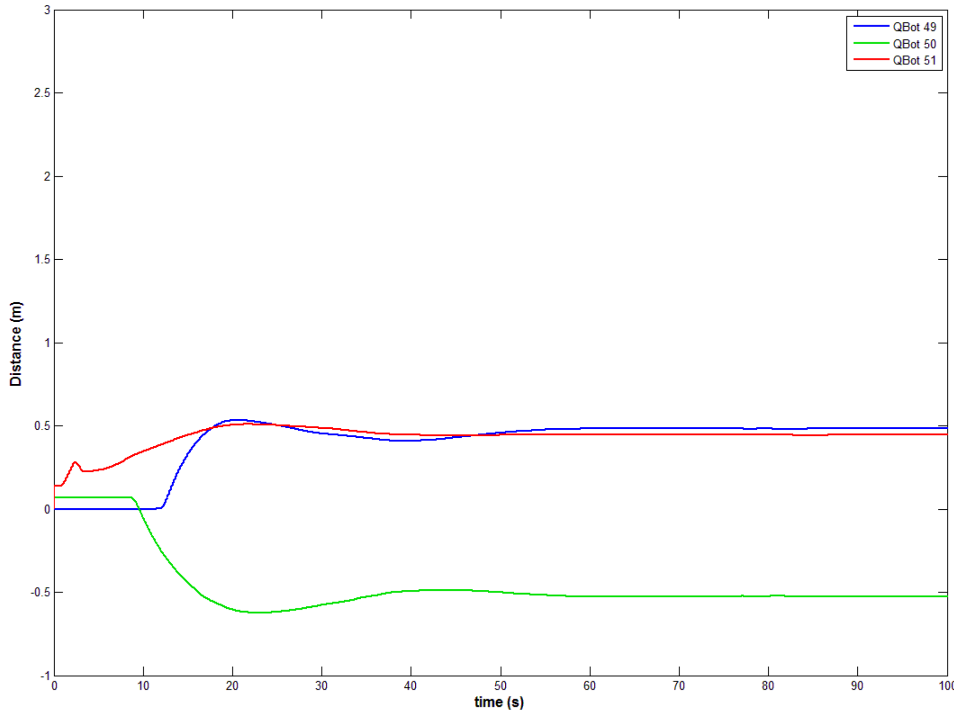


Figure 4.13 Formation Control Y Position vs Time

4.3.3 Trajectory Following

Trajectory following is tracking a specific route based on the corresponding equations. Any path is realizable if the shape can be modeled by equations. The equations generate a continuously updating point to which the QBots try to move. If the point moves too quickly for the QBot, the path of the QBot will deviate from the desired path. If the point moves very slowly, the QBot will move very slowly as well.

The first shape attempted to trace was a sine wave. The equations used to model a sine wave were as follows, where ω is angular velocity, t is time, and x_d and y_d are the desired x and y position.

$$\dot{x} = \omega \tag{32}$$

$$x_d = \omega t \tag{33}$$

$$\dot{y} = \omega \cos x_d \tag{34}$$

$$y_d = \sin x_d \tag{35}$$

$$v_1 = -k(z_1 - x_d) + \dot{x} \tag{36}$$

$$v_2 = -k(z_2 - y_d) + \dot{y} \tag{37}$$

The first tests for the sine wave trajectory following resulted in an increasingly flattened sine wave as the QBot moved. The sine wave became more and more damped because the desired point was moving much faster than the QBot. The QBot did not have to turn much to face the distant desired point. Therefore, the QBot moved in a slightly curvy line.

Another trajectory following experiment used the equations for a circle. The corresponding equations were as follows, where (*a*,*b*) is the center of the circle on a coordinate plane, *r* is the radius of the circle, *t* is time, *n* is the time scaler, x_d and y_d are the desired x and y position, and *k* is a constant.

$$x_d = a + r\cos\left({}^{t}/_{n}\right) \tag{38}$$

$$y_d = b + r\sin\left({}^{t}/_{n}\right) \tag{39}$$

$$\dot{z}_{11} = \dot{v}_{11} \tag{40}$$

$$\dot{z}_{12} = \dot{v}_{12} \tag{41}$$

$$v_{11} = -k(z_{11} - x_d) + \dot{x}_d \tag{42}$$

$$v_{12} = -k(z_{12} - y_d) + \dot{y}_d \tag{43}$$

When the time scaler *n* and the radius *r* were both one, the QBot moved in a circle, but its radius was not one meter. In fact, the radius was about one-third of a meter. Like the sine wave experiment, the desired point moved too quickly for the QBot. When *n* was changed to 16, the radius was much closer to the desired one meter. As *n* increases, the closer the QBot will trace the desired shape. However, a larger *n* will also result in a slower velocity. It is crucial to balance a fast velocity with the accuracy of the tracking.

Even after increasing *n* to 16, the radius of the circle did not quite reach a full meter. The QBot is not told to match the equation point for point. It tries to move to the desired point at each time instance. Therefore, the QBot makes little “shortcuts” to the desired point at every time instant instead of connecting the desired points together. This phenomenon is similar to how a semi-truck turns. The back wheels of the truck follow the cab when moving straight. However, when the truck turns, the rear wheels do not track perfectly with the front wheels. The back of the truck will be inside of the cab’s turn. Consequently, the QBots will never perfectly trace the equations’ curves.

4.3.4 Object Avoidance

Object avoidance is key when using autonomous systems. The agents need to account for unforeseen objects and other agents that may block their way. Fuzzy logic was implemented to improve the QBots’ object avoidance algorithm.

Fuzzy logic takes the inputs, and based on their values, assigns output values. Instead of hard cutoffs in the logic, fuzzy logic uses transitions for input and output definitions. This will result in smooth transitions from state to state. Abrupt changes in QBot speed and turning are unwanted, so using a fuzzy logic block is preferred. A flowchart detailing the object avoidance in located in Figure 4.14.

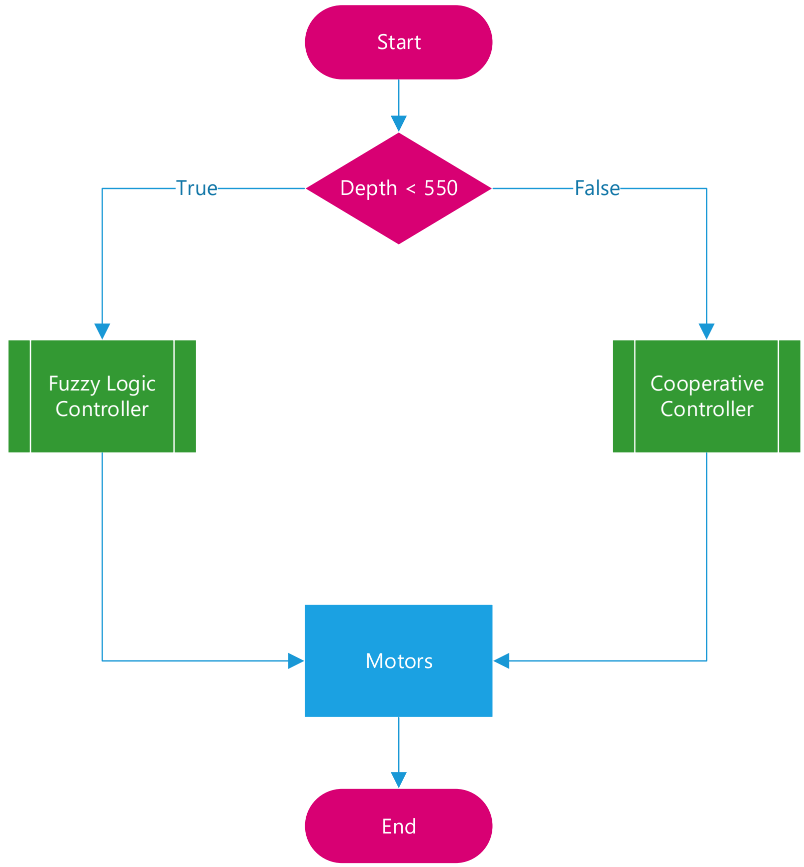


Figure 4.14 Object Avoidance Flowchart

The fuzzy logic block contained three inputs: left side, center, and right side, as shown in Figure 4.15. Each input is the distance (in millimeters) to the closest object in that third of the Kinect image. The left and right inputs are divided into two states: clear (more than 2000mm) and not clear (less than 700mm), which can be seen in Figure 4.16 and Figure 4.18. The center input is divided into three states: close (less than 600mm), middle (600mm to 3000mm), and far (more than 3000mm), as shown in Figure 4.17 “Center” Input Membership Function. The block had two outputs: left motor and right motor speeds. The output membership functions are divided into five states: stop (0 m/s), slow (0.2 m/s), medium (0.4 m/s), fast (0.6 m/s), and negative slow (-0.2 m/s), which can be seen in Figure 4.19 “Vl” Output Membership Function and Figure 4.20 "Vr" Output Membership Function.

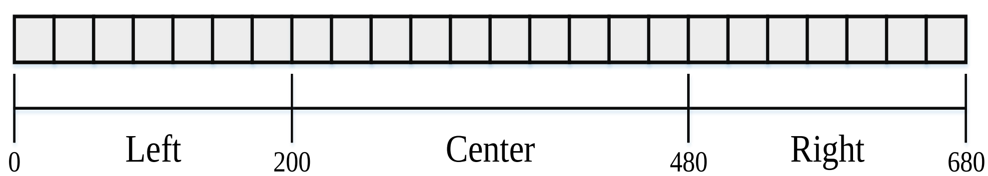


Figure 4.15 Object Avoidance Input Variables

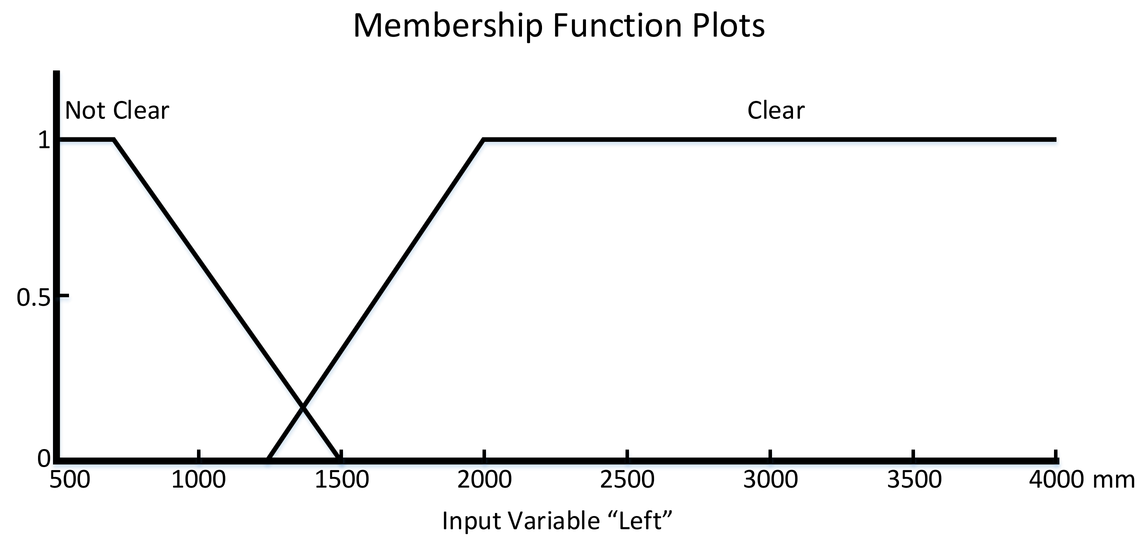


Figure 4.16 "Left" Input Membership Function

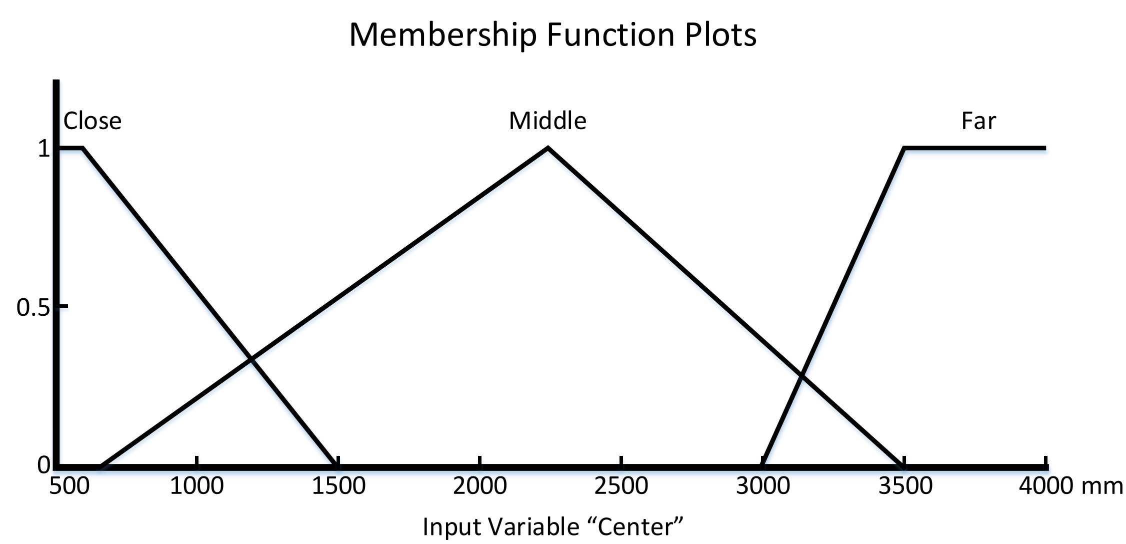


Figure 4.17 "Center" Input Membership Function

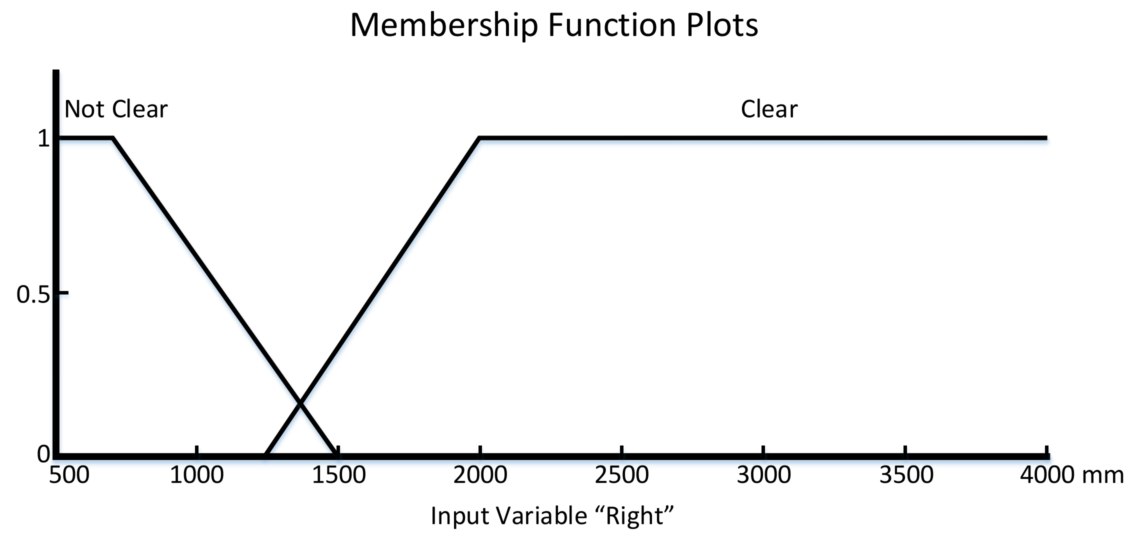


Figure 4.18 "Right" Input Membership Function

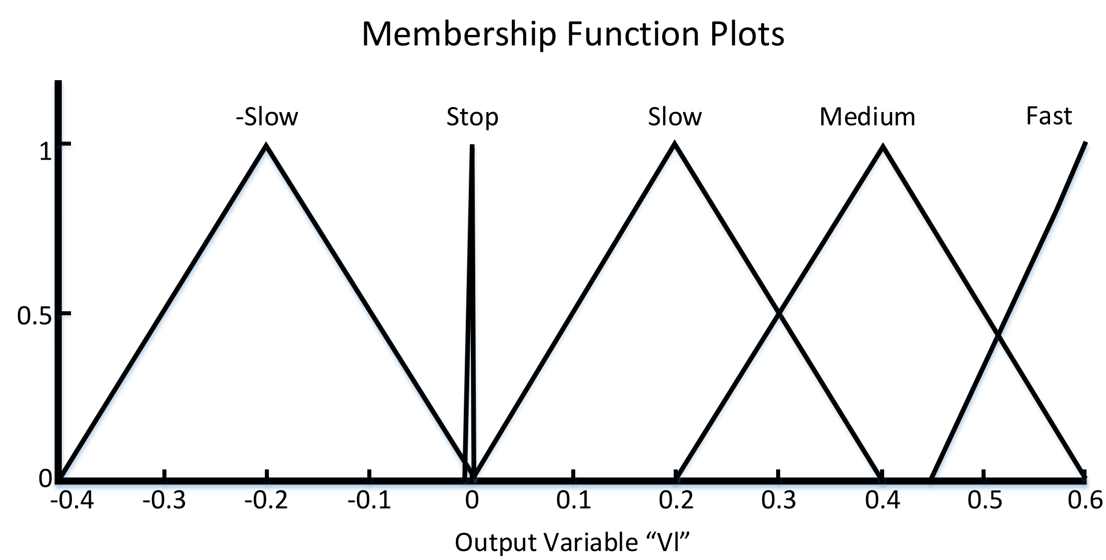


Figure 4.19 "Vl" Output Membership Function

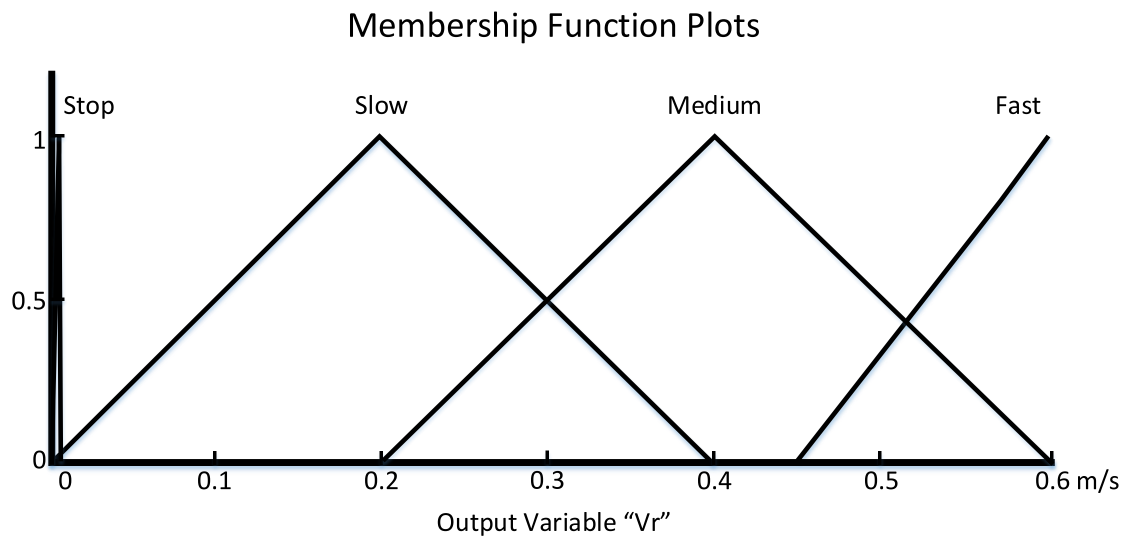


Figure 4.20 "Vr" Output Membership Function

The distance values were taken from the center horizontal line of the Kinect image. Using the center line ignores any object above and the ground directly below the QBot. If an object is located on the center line and the QBot drives to it, the QBot is guaranteed to collide with it. Because the Kinect Get Distance block erroneously reads zero at times, all zero values were disregarded. The smallest nonzero value for each third of the Kinect image was used as the inputs to the fuzzy logic block.

The rules within the fuzzy logic block were the decision-making portion of the object avoidance algorithm. The input-output logic used is displayed in Table 4.1 below.

Table 4.1 Fuzzy Logic Rules

Input			**Output**	
Left Side	Middle	Right Side	V_R	V_L
Far	Far	Far	Medium	Medium
Far	Middle	Far	Slow	Slow
Close	Close	Close	Negative Slow	Slow
Middle	Close	Close	Stop	Slow
Close	Close	Middle	Slow	Stop
Middle	Close	Middle	Stop	Slow

Because fuzzy logic uses transitions instead of hard values, the final outputs the block calculates is the centroid of the possible outputs. For example, if the center distance reading is located in the transition between the middle and far distances, and the left and right inputs are far, then the first two rules in the table above apply. The fuzzy logic block will assign the centroid of the resulting triangles for each output as the actual block outputs. In this case, the motor speeds will be between medium and slow. The closer the input fits a rule, the more the block will favor its matching outputs.

To test the object avoidance, a QBot was programmed to follow a counter-clockwise circular trajectory, as shown in Figure 4.21. After a complete revolution, a trashcan was placed in the path of the QBot. Upon approach, the QBot turned left to avoid the trashcan. Once the trashcan was out of frame, the QBot resumed the circular trajectory. Despite the trashcan never moving, the QBot started its avoidance behavior at a different position for every loop around the circle. The inconsistency is due to the relatively slow refresh rate of two hertz of the object avoidance flag. The flag changes to logic high when an object is within 600mm and shifts the logic to the object avoidance algorithm. When the object is out of frame, the flag switches back to logic low, and the QBot resumes trajectory following.

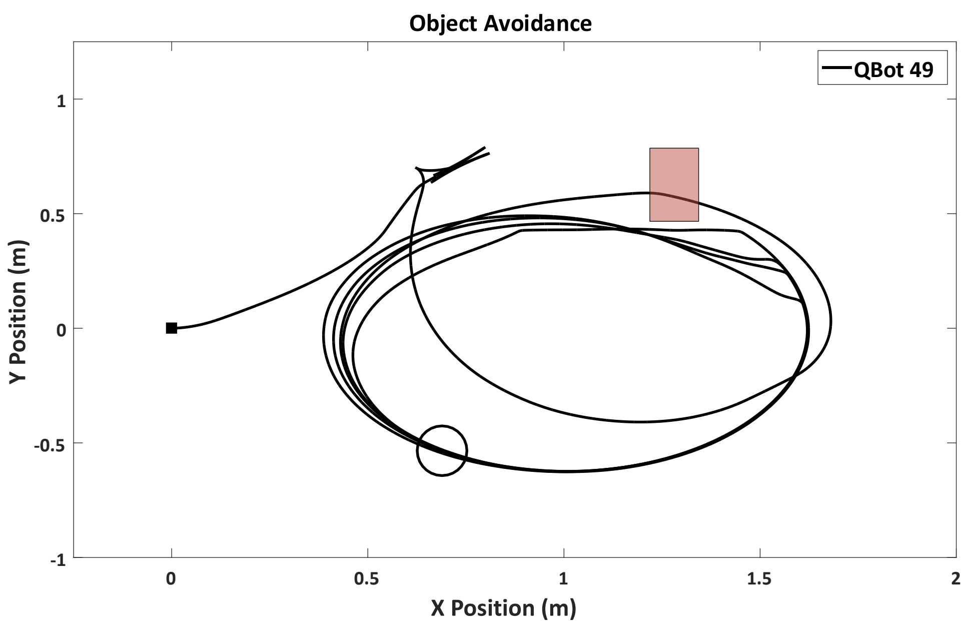


Figure 4.21 Object Avoidance Implementation

Conclusion and Future Work 5

In this project, cooperative control algorithms were designed and implemented on a network of mobile robots so that the robots can converge to maintain the same heading, rendezvous in an area, or form maneuvering patterns like filing, toroidal motions, flocking, and swarming. Control algorithms were obtained by linearizing robot models with the assumption of local information exchange through sensing and communication among neighboring robots. Experimental results validated the effectiveness and robustness of the proposed cooperative controls.

This is a multi-year project, and as such, there are many areas of research. For example, future work will include the study of target tracking problem by a network of heterogeneous robots, when communication capabilities of some neighboring robots are impaired. Future work may also include the improvement of existing features implemented on the robots, such as object avoidance, color detection, and communication capabilities. Communication capabilities may be improved to allow for the sending of data between the different robot platforms.

Appendix A

E-puck Code

Color Detection

```
#include <p30f6014A.h>
#include <stdlib.h>//for random numbers

#include "stdio.h"
#include "string.h"
#include "math.h"
#include "e_poxxxx.h"
#include "e_epuck_ports.h"
#include "e_init_port.h"
#include "e_motors.h"
#include "utility.h"
#include "e_led.h"
#include "e_prox.h"
#include "e_ad_conv.h"
#include "e_uart_char.h"
#include "e_randb.h"
#include "btcom.h"
#include "e_remote_control.h"
#include "e_agenda.h"
#include "searchball.h"
#include "runfollowball.h"

void indicateDirectionLED(double bearing);//turns on the LED
corresponding to bearing bearing

```

```
char debugMessage[80];//this is some data to store screen-bound debug messages
int seeSomething;//boolean for forward facing prox sensors

int main(void)
{
	char buffer[240];
	int selector;
	unsigned char *tab_start = buffer;
	e_init_port();
	e_init_uart1();
	e_init_motors();
	selector = getselector();

	if (selector == 1)
	{
		e_poxxxx_init_cam();
		select_cam_mode(1);
		e_poxxxx_launch_capture((char *)tab_start);
		while (!e_poxxxx_is_img_ready());
		LED1 = 1;
	}
	else if (selector == 2)
	{
		//run_follow_ball();
		e_set_speed_left(500);
		e_set_speed_right(500);
	}
	else if (selector == 3)
	{
```

```
                //run_follow_ball_green();
        }
        else
        {
                //LED0 =1;
        }
        while (1);
        return(0);
}
```

searchball.c

```
//search ball library
#include <p30f6014a.h>
#include <stdlib.h>
#include "searchball.h"
#include "e_motors.h"

#define PIC_SIZE_MIN 3
static float ui_lin = 0.0;

int get_average(unsigned char arr[], int start, int end);
int calc_peak_left(int *width_L, int *center_L, unsigned char buffer[], int nb_val);
int calc_peak_right(int *width_R, int *center_R, unsigned char buffer[], int nb_val);
void epuck_init(Epuck *epuck);
int calc_lin_speed(int distance, int gain);
int calc_angle_speed(int pos_pic, int gain);
void ARW();

void e_set_speed(int linear_speed, int angular_speed)
{
	if (abs(linear_speed) + abs(angular_speed) > 1000)
		return;
	else
	{
		e_set_speed_left(linear_speed - angular_speed);
		e_set_speed_right(linear_speed + angular_speed);
	}
}
```

```c

//this function calculates the average of an array from a given start point, to a given end point
int get_average(unsigned char arr[], int start, int end)
{
	int i;
	int avg = 0;
	if (start == end) //if one element average just return the element. duh!
	{
		return(arr[start]);
	}
	for (i = start; i<end; i++) // find the sum of the elements
	{
		avg += arr[i];
	}
	if (avg == 0) // if the sum was 0 just return 0
	{
		return(0);
	}
	return(avg / (end - start)); //return the average
}

int calc_peak_left(int *width_L, int *center_L, unsigned char buffer[], int nb_val)
{
	static int nb_avg = 10;
	int pic1, pic2;
	int difference;

	pic1 = nb_avg + 1;
	difference = 0;
```

```
        while (pic1 < nb_val - 1)
        {
                difference = get_average(buffer, pic1 - nb_avg - 1, pic1 - 1) - ((int)buffer[pic1] + (int)buffer[pic1 + 1]) / 2;
                if (difference > PIC_SIZE_MIN)
                {
                        break;
                }
                pic1++;
        }
        //check to see if we have an edge that is within expected parameters
        if (pic1 >= nb_val || difference <= PIC_SIZE_MIN)
        {
                return(PIC_NOT_FOUND);
        }

        pic2 = pic1 + 1;
        difference = 0;

        while (pic2 < nb_val)
        {
                difference = ((int)buffer[pic2] + (int)buffer[pic2]) / 2 - get_average(buffer, pic1, pic2);
                if (difference > PIC_SIZE_MIN)
                {
                        break;
                }
                pic2++;
        }

```

```
        *width_L = pic2 - pic1;

        //calculate the center of the object
        if (pic2 >= nb_val)
        {
                *center_L = nb_val / 2;
        }
        else
        {
                *center_L = pic1 + (pic2 - pic1) / 2 - nb_val / 2;
        }
        return(PIC_FOUND);
}

int calc_peak_right(int *width_R, int *center_R, unsigned char buffer[],
int nb_val)
{
        static int nb_avg = 10;
        int pic1, pic2;
        int difference;
        pic1 = nb_val - (nb_avg + 1);
        difference = 0;

        while (pic1 >0)
        {
                difference = get_average(buffer, pic1 + 1, pic1 + nb_avg +
                1) - ((int)buffer[pic1] + (int)buffer[pic1 - 1]) / 2;
                if (difference > PIC_SIZE_MIN)
                {
                        break;
                }
```

```
        pic1--;
    }

    if (pic1 == 0 || difference <= PIC_SIZE_MIN)
    {
        return(PIC_NOT_FOUND);
    }

    pic2 = pic1 - 1;
    difference = 0;

    while (pic2 >0)
    {
        difference = ((int)buffer[pic2] + (int)buffer[pic2 - 1]) /
        2 - get_average(buffer, pic2 + 1, pic1);
        if (difference > PIC_SIZE_MIN)
        {
            break;
        }
        pic2--;
    }

    *width_R = pic1 - pic2;
    if (pic2 <= 0)
    {
        *center_R = -nb_val / 2;
    }
    else
    {
        *center_R = pic2 + (pic1 - pic2) / 2 - nb_val / 2;
```

```
        }
        return(PIC_FOUND);
}

void epuck_init(Epuck *epuck)
{
        epuck->state = IS_SEARCHING_BALL;
        epuck->dist_ball = -1;
        epuck->angle_ball = -1;
        epuck->lin_speed = 0;
        epuck->angle_speed = 300;
}

void normalize(unsigned char buffer[], int nb_val)
{
        int avg = get_average(buffer, 0, nb_val);
        int i;

        if (avg == 0)
                return;
        for (i = 0; i<nb_val; i++)
        {
                buffer[i] = (10 * buffer[i]) / avg;
        }
}

int search_ball(Epuck *epuck, unsigned char buffer[], int nb_val)
{
        int center_L, center_R;
        int width_L, width_R;
```

```
        char pic_found_l, pic_found_r;


        pic_found_l = calc_peak_left(&width_L, &center_L, buffer, nb_val);
        pic_found_r = calc_peak_right(&width_R, &center_R, buffer,
        nb_val);

        if (pic_found_l == PIC_NOT_FOUND && pic_found_r == PIC_NOT_FOUND)
                return PIC_NOT_FOUND;

        else if (pic_found_l == PIC_FOUND && pic_found_r == PIC_NOT_FOUND)
        {
                epuck->dist_ball = width_L;
                epuck->angle_ball = center_L;
                return PIC_FOUND;
        }
        else if (pic_found_l == PIC_NOT_FOUND && pic_found_r == PIC_FOUND)
        {
                epuck->dist_ball = width_R;
                epuck->angle_ball = center_R;
                return PIC_FOUND;
        }
        else
        {
                epuck->dist_ball = (width_L + width_R) / 2;
                epuck->angle_ball = (center_L + center_R) / 2;
                return PIC_FOUND;
        }
}

void goto_ball(Epuck *epuck)
```

```
{
        int lin_speed = 0;
        int angle_speed = 0;
        int gain_lin = 35;
        int gain_angle = 6;

        lin_speed = calc_lin_speed(epuck->dist_ball, gain_lin);
        angle_speed = calc_angle_speed(epuck->angle_ball, gain_angle);

        epuck->lin_speed = lin_speed;
        epuck->angle_speed = angle_speed;
        e_set_speed(lin_speed, angle_speed);
}

int calc_lin_speed(int distance, int gain)
{
        int consigne = 50;
        float h = 0.1;
        int ti = 3;
        int ecart = consigne - distance;
        int lin_speed;

        ui_lin = ui_lin + h * ecart / ti;
        lin_speed = (ecart + ui_lin) * gain;

        if (lin_speed >= 1000)
        {
                ui_lin = 999 / gain - ecart;
                if (ui_lin > 60)
                        ui_lin = 60.0;
```

```
			lin_speed = 999;
		}
		else if (lin_speed <= -1000)
		{
			ui_lin = -999 / gain + ecart;
			if (ui_lin < -10)
				ui_lin = -10.0;
			lin_speed = -999;
		}
		return lin_speed;
}

int calc_angle_speed(int pos_pic, int gain)
{
		int consigne = 0;
		int angle_speed = 0;
		int ecart = consigne - pos_pic;

		angle_speed = ecart*gain;

		if (angle_speed >= 1000)
			angle_speed = 999;
		else if (angle_speed <= -1000)
			angle_speed = -999;

		return angle_speed;
}

void ARW()
{
```

```
        ui_lin = 0.0;
}
```

searchball.h

```
#define PIC_FOUND 1
#define PIC_NOT_FOUND -1
#define IS_SEARCHING_BALL 0
#define IS_FOLLOWING_BALL 1

#ifndef Epuck
typedef struct
{
	char state;
	int dist_ball;
	int angle_ball;
	int lin_speed;
	int angle_speed;
} Epuck;
#endif

void epuck_init(Epuck *epuck);
void normalize(unsigned char buffer[], int nb_val);
int search_ball(Epuck *epuck, unsigned char buffer[], int nb_val);
void goto_ball(Epuck *epuck);
void ARW();
```

runballfollow.c

```
#include "searchball.h"
#include "e_epuck_ports.h"
#include "e_init_port.h"
#include "e_uart_char.h"
#include "e_agenda.h"
#include "e_motors.h"
#include "e_poxxxx.h"

#define NB_VAL 240
#define VIT_ROT_search 300

unsigned char buffer[NB_VAL];
int line_thickness_cam = 4;
int pos_line1 = ARRAY_WIDTH/2 - 4/2;

void run_follow_ball_red(void);
void execute(unsigned char *buffer_execute, Epuck *epuck);
void follow_red(unsigned char *buf, int size);
void select_cam_mode(int mode);
void follow_green(unsigned char *buf, int size);
void run_follow_ball(void);
void run_follow_ball_green(void);
void run_follow_ball_red(void);

void select_cam_mode(int mode)
{
       e_poxxxx_config_cam(pos_line1, 0, line_thickness_cam,
       ARRAY_HEIGHT, 4, 4, mode);
       e_poxxxx_set_mirror(1, 1);
```

```
        e_poxxxx_write_cam_registers();
}

void execute(unsigned char *buffer_execute, Epuck *epuck)
{
        char pic_found;

        normalize(buffer_execute, NB_VAL / 2);
        pic_found = search_ball(epuck, buffer_execute, NB_VAL / 2);

        if (pic_found == PIC_FOUND)
        {
                if (epuck->state == IS_SEARCHING_BALL) {
                        ARW();
                }
                epuck->state = IS_FOLLOWING_BALL;
                BODY_LED = 1;
                goto_ball(epuck);
        }
        else
        {
                ARW();
                epuck->state = IS_SEARCHING_BALL;
                BODY_LED = 0;
                epuck->lin_speed = 0;

                if (epuck->angle_ball > 0)
                {
                        e_set_speed_left(VIT_ROT_search);
                        e_set_speed_right(-VIT_ROT_search);
```

```
            }
            else
            {
                e_set_speed_left(-VIT_ROT_search);
                e_set_speed_right(VIT_ROT_search);
            }
        }
}

void follow_red(unsigned char *buf, int size)
{
        int i;
        unsigned char green;
        for (i = 0; i<size / 2; i++)
        {
                green = (((buf[2 * i] & 0x07) << 5) | ((buf[2 * i + 1] &
                0xE0) >> 3));
                //blue = ((buf[2*i+1] & 0x1F) << 3)
                buf[i] = green;
        }
}

void follow_green(unsigned char *buf, int size)
{
        int i;
        unsigned char red;
        for (i = 0; i<size / 2; i++)
        {
                red = (buf[2 * i] & 0xF8);
                //blue = ((buf[2*i+1] & 0x1F) << 3);
```

```
                buf[i] = red;
        }
}

void run_follow_ball(void)
{
        unsigned char *tab_start = buffer;
        unsigned char *tab_middle = buffer + NB_VAL / 2;

        Epuck epuck;

        epuck_init(&epuck);
        e_init_port();    // configure port pins
        e_start_agendas_processing();
        e_init_motors();
        e_init_uart1();   // initialize UART to 115200 Kbaud
        e_poxxxx_init_cam();
        select_cam_mode(GREY_SCALE_MODE);

        while (1)
        {
                e_poxxxx_launch_capture((char *)tab_start);
                execute(tab_middle, &epuck);
                while (!e_poxxxx_is_img_ready());
                e_poxxxx_launch_capture((char *)tab_middle);
                execute(tab_start, &epuck);
                while (!e_poxxxx_is_img_ready());
        }
}

```

```
void run_follow_ball_green(void)
{
        unsigned char *tab_start = buffer;
        //      unsigned char *tab_middle = buffer + NB_VAL/2;

        Epuck epuck;

        epuck_init(&epuck);
        e_init_port();     // configure port pins
        e_start_agendas_processing();
        e_init_motors();
        e_init_uart1();    // initialize UART to 115200 Kbaud
        e_poxxxx_init_cam();
        select_cam_mode(RGB_565_MODE);

        while (1)
        {
                LED0 = 1;
                e_poxxxx_launch_capture((char *)tab_start);
                LED2 = 1;
                while (!e_poxxxx_is_img_ready());
                LED4 = 1;
                follow_green(tab_start, NB_VAL);
                LED6 = 1;
                execute(tab_start, &epuck);
        }
}

void run_follow_ball_red(void)
{
```

```
	unsigned char *tab_start = buffer;
	//	unsigned char *tab_middle = buffer + NB_VAL/2;

	Epuck epuck;

	epuck_init(&epuck);
	e_init_port();    // configure port pins
	e_start_agendas_processing();
	e_init_motors();
	e_init_uart1();   // initialize UART to 115200 Kbaud
	e_poxxxx_init_cam();
	select_cam_mode(RGB_565_MODE);

	while (1)
	{
		e_poxxxx_launch_capture((char *)tab_start);
		while (!e_poxxxx_is_img_ready());
		follow_red(tab_start, NB_VAL);
		execute(tab_start, &epuck);
	}
}
```

runballfollow.h

```
#ifndef _FOLLOW_BALL
#define _FOLLOW_BALL

void run_follow_ball(void);
void run_follow_ball_green(void);
void run_follow_ball_red(void);

#endif
```

Vicsek and Odometry

```
#include <p30f6014A.h>

#include <stdlib.h>//for random numbers
#include "stdio.h"
#include "string.h"
#include "math.h"

//#include "e_poxxxx.h"

#include "e_epuck_ports.h"
#include "e_init_port.h"
#include "e_motors.h"
#include "utility.h"
#include "e_led.h"
#include "e_prox.h"
#include "e_ad_conv.h"
#include "e_uart_char.h"
#include "e_randb.h"
#include "btcom.h"
#include "e_remote_control.h"
#include "e_agenda.h"

#include <ircom.h>



struct p
{
```

```
        int x;
        int y;
        int theta;
};


//the epuck wheels have a diameter of 41mm
//the distance between wheels is approx 53mm (important for kinematics)
//max speed is 1000 steps/sec
//one revolution is equal to  128 mm
// 1000 steps = 1 revolution

#define PI 3.14159

static int prevStepL = 0, prevStepR = 0, stepL, stepR;

static int deltatheta, deltaL, deltaR, deltaS, dx, dy;

static float uilin = 0.0;

//this function calculates the epucks current position and orientation based on
//information from the wheel encoders

void reset()
{
        uilin = 0.0;
}

void updateposition(struct p *old)
```

```
{
        stepL = e_get_steps_left(); //get our steps
        stepR = e_get_steps_right();

        deltaL = stepL - prevStepL; //calculate change
        prevStepL = stepL;   // update info

        deltaR = stepR - prevStepR;
        prevStepR = stepR;

        deltatheta = (deltaR - deltaL) / 2;
        deltaS = (deltaR + deltaL) / 2;

        dx = deltaS + cos(old->theta + deltatheta / 2);
        dy = deltaS + sin(old->theta + deltatheta / 2);

        old->x = old->x + dx;
        old->y = old->y + dy;
        old->theta = old->theta + (deltatheta / 3);


}

void turntoangle2(int angle, struct p *epuck)
{
        int c = 0;
        int state = 0;
        int turnangle = 0;

        int theta = angle;// - epuck->theta;
```

```
        theta = (theta * 3) + 60;   /// actually susposed to be angle*1000/360, but with wheel slippage a slight gain is required

        int oldright = e_get_steps_right();
        int oldleft = e_get_steps_left();

        //LED0 =1;
        while (c == 0)
        {
               //  LED2 =1;
               switch (state)
               {
               case 0:
                      e_set_steps_left(0);
                      e_set_steps_right(0);
                      e_set_speed_left(-200);
                      e_set_speed_right(200);
                      state = 1;
                      break;
               case 1:
                      turnangle = e_get_steps_left();
                      if (turnangle < -theta)
                      {
                             e_set_speed_left(0);
                             e_set_speed_right(0);
                             state = 0;
                             c = 1;
                      }
                      break;
               }
```

```
        }
        //LED0 =1;
        e_set_steps_left(oldleft);
        e_set_steps_right(oldright);
        //epuck->theta =
}

void turntonegativeangle(int angle, struct p *epuck)
{
        int c = 0;
        int state = 0;
        int turnangle = 0;

        int theta = angle;// - epuck->theta;
        theta = (theta * 3) + 60;   /// actually susposed to be angle*1000/360, but with wheel slippage a slight gain is required

        int oldright = e_get_steps_right();
        int oldleft = e_get_steps_left();

        //LED0 =1;
        while (c == 0)
        {
                //  LED2 =1;
                switch (state)
                {
                case 0:
                        e_set_steps_left(0);
                        e_set_steps_right(0);
                        e_set_speed_left(200);
```

```
                        e_set_speed_right(-200);
                        state = 1;
                        break;
                case 1:
                        turnangle = e_get_steps_right();  //right
                        if (turnangle < -theta)  //changed to negative
                        {
                                e_set_speed_left(0);
                                e_set_speed_right(0);
                                state = 0;
                                c = 1;
                        }
                        break;
                }
        }

        e_set_steps_left(oldleft);
        e_set_steps_right(oldright);

}

void drive_distance(long int d) //d is in mm
{
        long int steps = d / 128;  // distance (mm) * (1 step/ 128mm)
        int c = 0;
        int state = 0;
        int oldL = e_get_steps_left();
        int oldR = e_get_steps_right();
        int stepsdone = 0;

```

```
        while (c == 0)
        {
                switch (state)
                {
                case 0:     e_set_steps_left(0);
                        e_set_steps_right(0);
                        e_set_speed_left(200);
                        e_set_speed_right(200);
                        state = 1;
                        break;
                case 1:
                        stepsdone = e_get_steps_right();
                        if (stepsdone >= steps)
                        {
                                e_set_speed_left(0);
                                e_set_speed_right(0);
                                state = 0;
                                c = 1;
                        }
                        break;
                }
        }
        e_set_steps_left(oldL);
        e_set_steps_right(oldR);
}

double calculatedistance(struct p *current, struct p *goal)
{
        double distance = 0;
```

```
        distance = ((goal->x - current->x)*(goal->x - current->x)) +
((goal->y - current->y)*(goal->y - current->y));
        distance = sqrt(distance);
        return(distance);
}

int calculateangle(struct p *current, struct p *goal)
{
        double deltay = goal->y - current->y;
        double deltax = goal->x - current->x;
        double angleindegrees = atan2(deltay, deltax) * 180 / PI;

        return((int)angleindegrees);
}

int calculatelinearvelocity(double distance, int gain)
{
        int cosine = 50;
        float h = 0.1;
        int ti = 3;
        int gap = cosine - distance;
        int linspeed;


        uilin = uilin + h * gap / ti;
        linspeed = (gap + uilin)*gain;

        if (linspeed >= 1000)
        {
                uilin = 999 / gain - gap;
```

```
		if (uilin >60)
		{
			uilin = 60.0;
		}
		linspeed = 999;
	}
	else if (linspeed <= -1000)
	{
		uilin = -999 / gain + gap;
		if (uilin < -10)
		{
			uilin = -10.0;
		}
		linspeed = -999;
	}

	return(linspeed);
}

int calculateangularvelocity(int angle, int gain, struct p *epuck)
{
	//	int cosine = 0;
	int angle_velocity = 0;
	int gap = (angle - epuck->theta);

	angle_velocity = gap*gain;
	if (angle_velocity >= 1000)
	{
		angle_velocity = 999;
	}
```

```
        else if (angle_velocity <= -1000)
        {
                angle_velocity = -999;
        }
        return(angle_velocity);
}

void e_set_speed(int linear_speed, int angular_speed)
{
        if (abs(linear_speed) + abs(angular_speed) > 1000)
                return;
        else
        {
                e_set_speed_left(linear_speed - angular_speed);
                e_set_speed_right(linear_speed + angular_speed);
        }
}

static int init = 0;

int main(void)
{
        int selector;

        e_init_port();
        e_init_motors();
        e_init_ad_scan();
        e_start_agendas_processing();

        ircomStart();
```

```
        ircomEnableContinuousListening();
        ircomListen();

        if (RCONbits.POR)    //Reset if Power on (some problem for few
robots)
        {
                RCONbits.POR = 0;
                __asm__ volatile ("reset");
        }
        //selector = getselector();
        struct p epuck;
        epuck.x = 0;
        epuck.y = 0;
        epuck.theta = 0;

        int i = 0;
        int j;
        int k;
        int state = 0;
        int angle;
        int diffangle = 0;
        int buffer[3]; //was 5
        int sumtheta = 0;


        epuck.theta = 0;
        IrcomMessage imsg;

        for (;;)
        {
```

```
		if (state == 0)
		{
			updateposition(&epuck);
			ircomSend(epuck.theta);		//send out our current heading
			while (ircomSendDone() == 0); //wait until done sending
			state = 1;
			LED0 = 1;
			LED2 = 0;

		}
		else if (state == 1)
		{
			LED0 = 0;
			LED2 = 1;

			ircomPopMessage(&imsg); // pop message off of stack to be processed
			if (imsg.error == 0) //check to see if message was recieved correctly
			{

				//	LED0=1;
				if (i <1) // lets fill a buffer of angles
				{
					buffer[i] = (int)imsg.value; //fill buffer
					i++; //increment count
				}
				else	//if the buffer is full
				{
					i = 0; //reset count
```

```
						//	LED6=1;
					sumtheta = buffer[0];// + buffer[1] + buffer[2];//+buffer[3] +buffer[4]; //summation of all recieved angles
					angle = (epuck.theta + sumtheta) / 2; // current angle plus the sum of theta divided by number of angles +1
					diffangle = angle - epuck.theta; //how much do we really have to turn?
					if (diffangle <0) //what direction?
					{
						ircomPause(1); //stop communication, causes issues with motors
												//	for(k=0;k<200;k++) asm("nop");
						myWait(400);
						turn(diffangle, 100);
						//for(k=0;k<2000;k++) asm("nop");
						myWait(400);
						ircomPause(0);
					}
					else if (diffangle>0) {
						ircomPause(1);
						//for(k=0;k<200;k++) asm("nop");
						myWait(400);
						turn(diffangle, 100);
						//for(k=0;k<2000;k++) asm("nop");
						myWait(400);
						ircomPause(0);
					}
					else
					{

					}
```

```
                                    epuck.theta = angle;

                                    ircomPause(1);
                                    // for(k=0;k<200;k++) asm("nop");
                                    myWait(400);
                                    move(50, 100);
                                    //for(k=0;k<2000;k++) asm("nop");
                                    myWait(400);
                                    ircomPause(0);

                                    state = 0;

                             }
                      }

                      state = 0;
               }
               myWait(300); //400
                                     //for(j=0;j<20000;j++) asm("nop"); //need a long delay between states roughly 20000
                                     // LED4=0;
        }
        return(0);

}
```

Bluetooth Communication

```
#include <p30f6014A.h>

#include <stdlib.h>//for random numbers
#include "stdio.h"
#include "string.h"
#include "math.h"

#include "e_poxxxx.h"

#include "e_epuck_ports.h"
#include "e_init_port.h"
#include "e_motors.h"
#include "utility.h"
#include "e_led.h"
#include "e_prox.h"
#include "e_ad_conv.h"
#include "e_uart_char.h"
#include "e_randb.h"
#include "btcom.h"
#include "e_remote_control.h"
#include "e_agenda.h"
#include "e_bluetooth.h"

void indicateDirectionLED(double bearing);//turns on the LED
corresponding to bearing bearing

char debugMessage[80];//this is some data to store screen-bound debug
messages
int seeSomething;//boolean for forward facing prox sensors
```

```
int main(void)
{

	char buffer[100] = { 0 };
	int selector; //switch
	char error = 1;
	int masterdone = 0;
	int i;

	e_init_port();
	e_start_agendas_processing();
	e_init_uart1();
	e_init_uart2();

	selector = getselector();

	if (selector == 1) //master role
	{
		while (1)
		{
			if (masterdone == 0)
			{
				masterdone = 1;
				i = 1;
				do
				{
					i = e_bt_find_epuck();

				} while (i != 0);
```

```


                        do
                        {
                              LED0 = 1;
                              error = e_bt_connect_epuck;
                              LED0 = 0;

                        } while (error != 0);

                  }
                  e_bt_send_SPP_data("12", 2);
            }

      }
      else { //slave role
            while (1)
            {
                  //memset(buffer,0,100);
                  buffer[0] = 0;
                  buffer[1] = 0;
                  e_bt_recv_SPP_data(buffer);
                  if (buffer[0] == '1' && buffer[1] == '2');
                  {
                        LED0 = 1;
                        //LED1=1;
                        //LED2=1;
                        //LED4=1;
                        //LED5=1;
                        //LED6=1;
```

```
                        }

                }


        }
        return(0);
}
```

btcom.c

```
#ifndef BTCOM_C
#define BTCOM_C

#include "btcom.h"
#include "e_uart_char.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Don't forget to initialize hardware before using it when debugging. Library to use on the e-puck
// maximum size of messages is set to 255 bytes


void btcomSendStringStatic(char* buffer)
{
        e_send_uart1_char(buffer, sizeof(*buffer) - 1);
        while (e_uart1_sending());
}

void btcomSendString(char* buffer)
```

```
{
        e_send_uart1_char(buffer, strlen(buffer));
        while (e_uart1_sending());
}

void btcomSendInt(long int x)
{
        char msg[BTCOM_MAX_MESSAGE_LENGTH];
        sprintf(msg, "%ld", x);
        btcomSendString(msg);
}

void btcomSendFloat(double x)
{
        char msg[BTCOM_MAX_MESSAGE_LENGTH];
        sprintf(msg, "%lf", x);
        btcomSendString(msg);
}

void btcomSendChar(char c)
{
        e_send_uart1_char(&c, 1);
        while (e_uart1_sending());
}

void btcomWaitForCommand(char trigger)
{
        char msg;
        do
        {
```

```
                e_getchar_uart1(&msg);
        } while (msg != trigger);

        // sleep a bit
        long int count;
        for (count = 0; count < 1000000; count++)
                asm("nop");
}

// BTCOM_C
#endif
```

btcom.h

```
#ifndef BTCOM_H
#define BTCOM_H

#define BTCOM_MAX_MESSAGE_LENGTH 256

void btcomSendStringStatic(char* buffer);
void btcomSendString(char* buffer);
void btcomSendInt(long int x);
void btcomSendFloat(double x);
void btcomSendChar(char c);
void btcomWaitForCommand(char trigger);

// BTCOM_H
#endif
```

Appendix B

E-puck Unbricking Guide

Note: When Unbricking an E-puck, it should be powered off.

1. Open MPLAB IDE v8.30

2. Remove the top portion of the E-puck

Figure B.1 E-puck with Top Removed

3. Connect the ICD 3 in-circuit debugger to the computer

4. Connect the ICD 3 in-circuit debugger to the E-puck

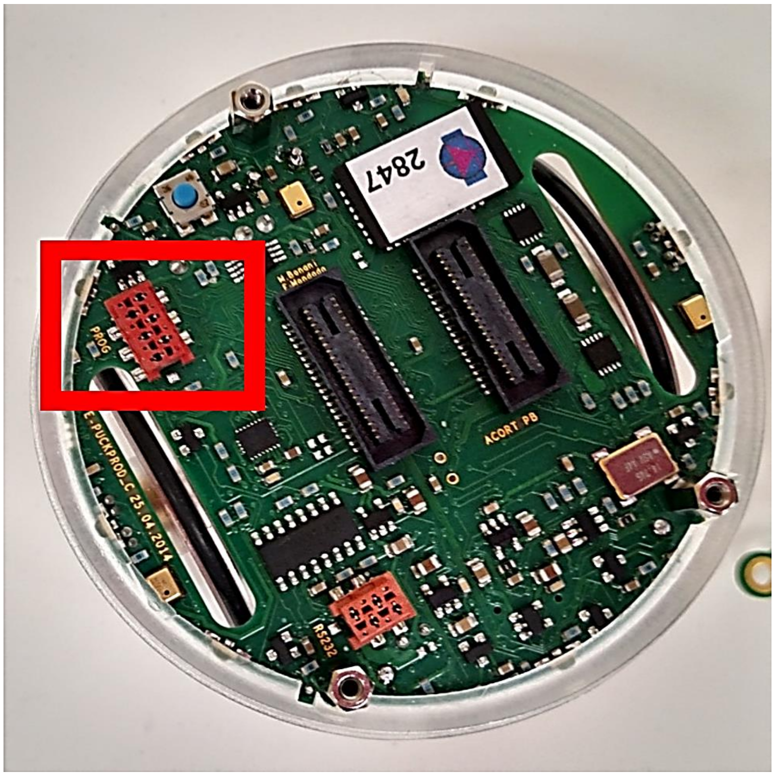


Figure B.2 E-puck Connection Pins for Debugger

5. In MPLAB IDE v8.30, go to Programmer > Select Programmer

6. Select MPLAB ICD 3

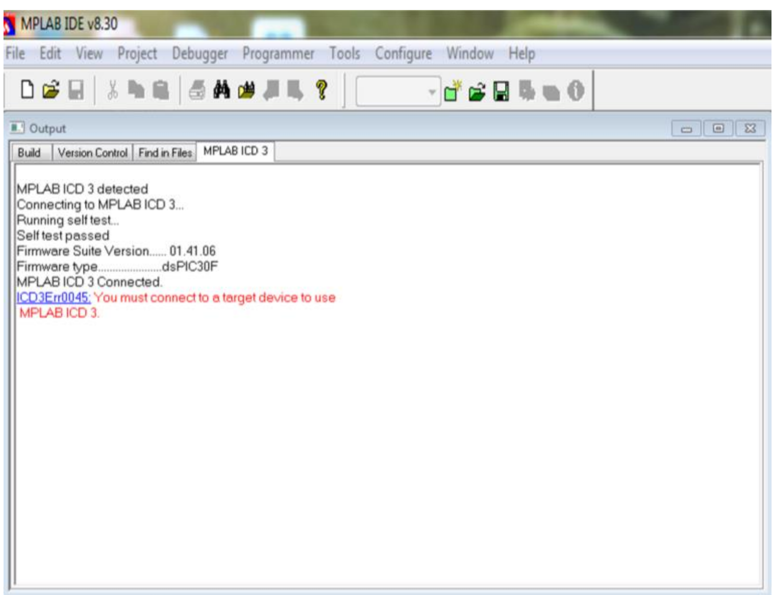


Figure B.3 Select Programmer Window

7. Go to Programmer > Settings

8. In the Settings window, click the Program Memory tab. Select the "Manually select memories and ranges" checkbox. Then select the checkboxes below.

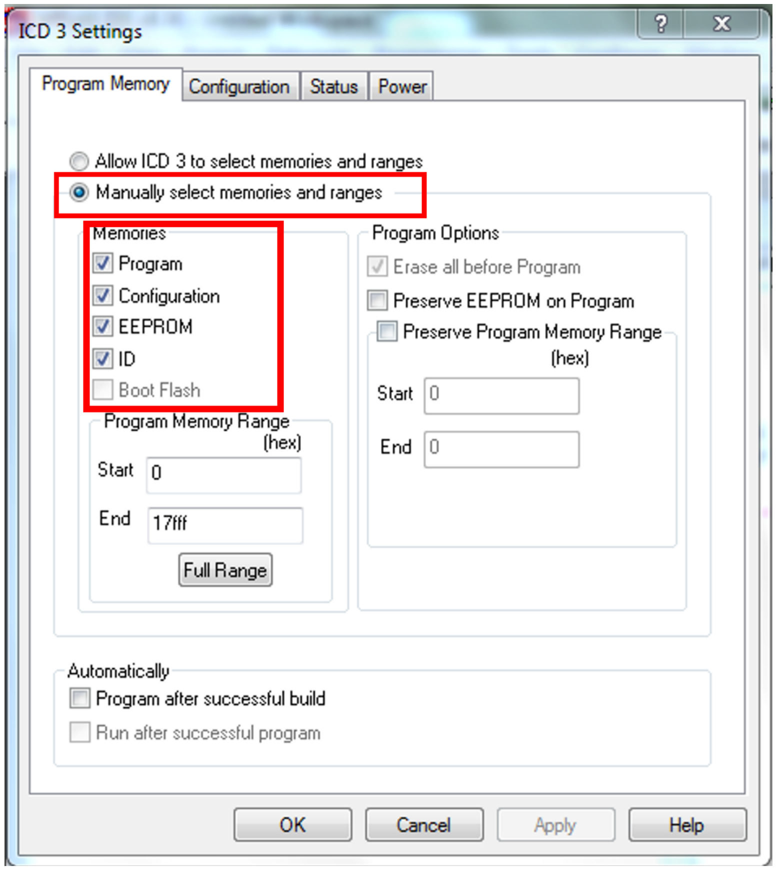


Figure B.4 Program Memory Pane

9. Click the Full Range button

10. Click Apply

11. In the Settings window, select the Power tab.

12. Set Voltage to 5.5

13. Select Power target circuit from MPLAB ICD 3 checkbox

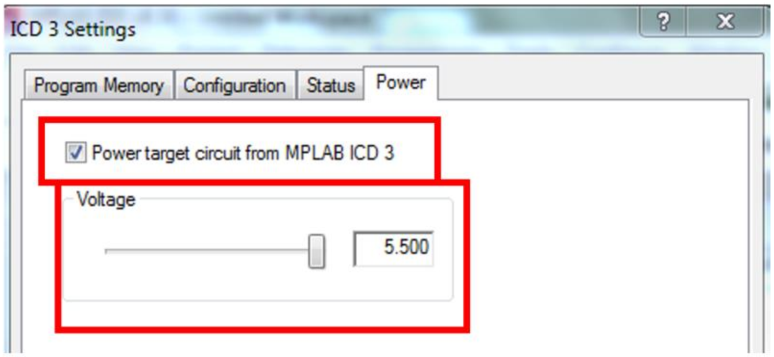
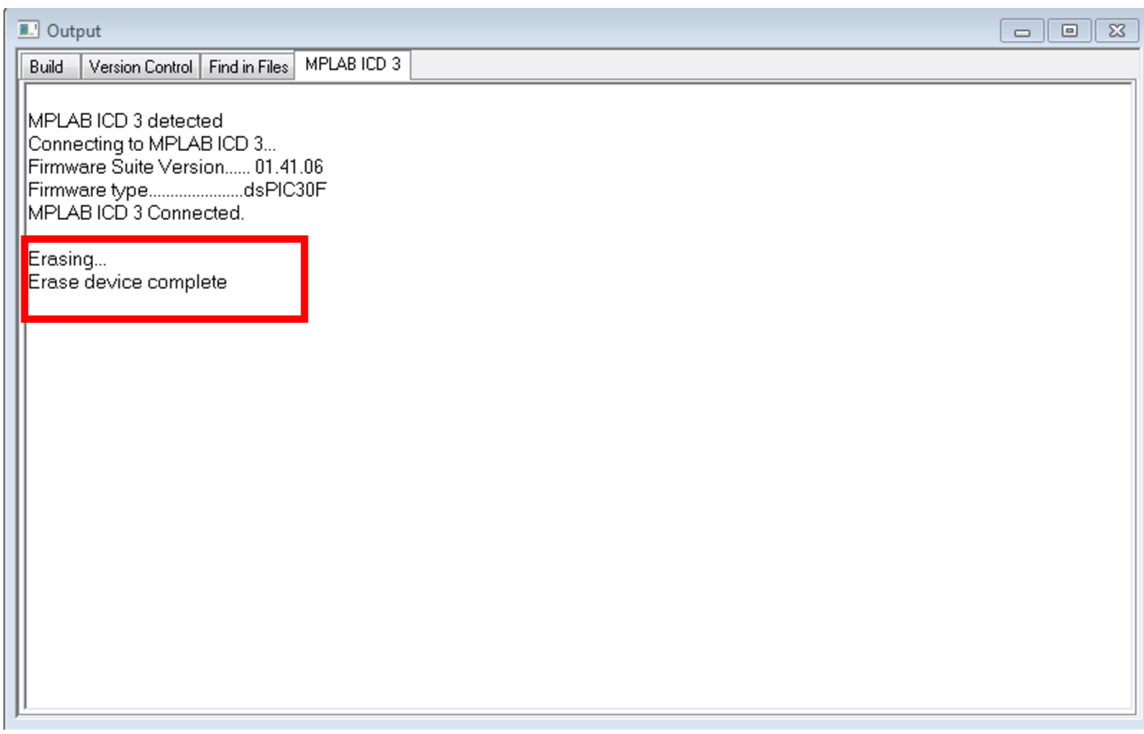


Figure B.5 Power Pane

14. Click Apply, then click OK

16. Go to Programmer > click Erase Flash Device

17. The Output window will display erasing status. When erasing is complete, disconnect and power on the E-puck.



Note: If the E-puck is still bricked, repeat steps 1-15

Appendix C

Kilobot Code

Gradient

```
#include "libKilobot.h" // include Kilobot library file

#define MAX  (70)
#define MSIZE  (5)
#define ASIZE (3)
#define MAXGRADIENT (10)

uint8_t ID = 0;
uint8_t message_buffer[5] = { 0 };
uint8_t average_buffer[3] = { 0 };

uint8_t gradient = MAXGRADIENT;
uint8_t avg_gradient = 0;
uint8_t min = 0;
uint8_t i, k = 0;
uint8_t full = 0;

uint8_t init = 1;

// user program function
void user_program(void)
{
       if (init)
       {
```

```
            int randseed = 0;
            // generate random seed (must be placed AFTER init_robot()

            for (int i = 0; i<30; i++)
                randseed += get_ambient_light();//generate some random sensor data

            srand(randseed);//seed random variable with some sensor data

                                    // generate robot id
            ID = rand() & 255;
            //ID = 0;
            init = 0;
        }

        if (ID == 0)
        {
            message_out(0, 0, 0);
            enable_tx = 1;
            set_color(3, 0, 0);
        }
        else
        {
            get_message();
            if (message_rx[5] == 1)
            {
                if (message_rx[3] < MAX)
                {
                    if (i < MSIZE)
                    {
```

```
                    message_buffer[i] = message_rx[1];
                    i++;
                }
                else
                {
                    i = 0;
                    min = message_buffer[0];
                    for (int j = 1; j < MSIZE; j++)
                    {
                        if (message_buffer[j] < min)
                        {
                            min = message_buffer[j];
                        }

                    }
                    average_buffer[k] = min;
                    k++;
                    if (k >= ASIZE)
                    {
                        k = 0;
                        avg_gradient = 0;
                        for (int avg = 0; avg < ASIZE; avg++)
                        {
                            avg_gradient += average_buffer[avg];
                        }
                        avg_gradient = avg_gradient / ASIZE;
                        gradient = avg_gradient + 1;
                    }
```

```
                        }
                }
        }

        message_out(gradient, gradient, gradient);
        enable_tx = 1;

        switch (gradient)
        {
        case 0:  set_color(3, 0, 0);
                break;
        case 1:  set_color(0, 3, 0);
                break;
        case 2:  set_color(0, 0, 3);
                break;
        case 3:  set_color(3, 3, 0);
                break;
        case 4:  set_color(0, 3, 3);
                break;
        case 5:  set_color(3, 3, 3);
                break;
        case 6:  set_color(1, 3, 0);
                break;
        case 7:  set_color(0, 3, 1);
                break;
        default: set_color(0, 0, 0);
                break;

        }
```

```
        }

}

// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();

        // loop and run each time the user program
        main_program_loop(user_program);
}
```

Orbiting

```
#include "libKilobot.h" // include Kilobot library file

#define ROOT               (0)
#define STOP               (0)
#define FORWARD                   (1)
#define LEFT               (2)
#define RIGHT              (3)
#define NORMAL                    (1)
#define LOWERBOUND         (94)
#define UPPERBOUND   (95)
#define D                         (40)

static int init = 1;
static int robot_id = 0;

static int currentMotion = 0;
static int currentDistance = 0;
static int distance = 0;


void SetMotion(int newMotion);
void CheckBounds(void);

// user program function
void user_program(void)
{
       /////////////////////////////////////////////////////////////////
////////////////////
```

```
        //user program code goes here.  this code needs to exit in a resonable amount of time
        //so the special message controller can also run
        ///////////////////////////////////////////////////////////////////////////////////

        // if the first time the loop is called, initialise the robot id
        if (init)
        {
                int randseed = 0;
                // generate random seed (must be placed AFTER init_robot()

                for (int i = 0; i<30; i++)
                        randseed += get_ambient_light();//generate some random sensor data

                srand(randseed);//seed random variable with some sensor data

                                                // generate robot id
                robot_id = rand() & 255;
                init = 0;
                //robot_id = 0;
        }

        if (robot_id != ROOT)
        {


                message_out(robot_id, 3, 0);
                enable_tx = 1;

```

```
            get_message();

            if (message_rx[5] == 1)
            {
                    if (message_rx[0] == ROOT)
                    {
                            currentDistance = message_rx[3];
                            CheckBounds();

                    }
                    else
                    {
                            distance = message_rx[3];
                    }
            }
            else if (currentDistance == 0)
            {
                    return;
            }
    }
    else
    {
            message_out(robot_id, 5, 0);
            enable_tx = 1;
            set_color(3, 0, 0);
    }

}

```

```
// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();


        // loop and run each time the user program
        main_program_loop(user_program);


}

void SetMotion(int newMotion)
{
        if (currentMotion != newMotion)
        {
                currentMotion = newMotion;
                switch (currentMotion)
                {
                case STOP:                                      // Stop
                        set_motor(0, 0);
                        set_color(0, 0, 0);
                        break;

                case FORWARD:                                   // Forward
```

```
                set_motor(0xA0, 0xA0);
                _delay_ms(15);
                set_motor(cw_in_straight, ccw_in_straight);
                set_color(0, 3, 0);
                break;

        case LEFT:                                  // Left
                set_motor(0, 0xA0);
                _delay_ms(15);
                set_motor(0, ccw_in_place);
                set_color(3, 0, 0);
                break;

        case RIGHT:                          // Right
                set_motor(0xA0, 0);
                _delay_ms(15);
                set_motor(cw_in_place, 0);
                set_color(0, 0, 3);
                break;

        default:
                set_motor(0, 0);
                set_color(0, 0, 0);
                break;

        }
    }
}


```

```
void CheckBounds()
{
        if (currentDistance < LOWERBOUND)
        {
                SetMotion(RIGHT);
        }
        else if (currentDistance > UPPERBOUND)
        {
                SetMotion(LEFT);
        }
        else
        {
                SetMotion(FORWARD);
        }
}
```

Light Following

```
#include "libKilobot.h" // include Kilobot library file
//#include "myLibrary.h"


#define THRESH_LO 500
#define THRESH_HI 700
#define THRESH_STOP 700

#define STOP 0
#define FORWARD 1
#define LEFT 2
#define RIGHT 3


int current_motion = STOP;
int current_light = 0;
uint8_t prev = LEFT;

uint8_t ID = 0;
uint8_t init = 1;



void set_motion(int new_motion)
{
	// Only take an action if the motion is being changed.
	if (current_motion != new_motion)
	{
```

```
                current_motion = new_motion;

                if (current_motion == STOP)
                {
                        set_motor(0, 0);
                }
                else if (current_motion == FORWARD)
                {
                        set_motor(255, 255);
                        _delay_ms(75);
                        set_motor(ccw_in_straight, cw_in_straight);
                }
                else if (current_motion == LEFT)
                {
                        set_motor(255, 0);
                        _delay_ms(75);
                        set_motor(ccw_in_place, 0);
                }
                else if (current_motion == RIGHT)
                {
                        set_motor(0, 255);
                        _delay_ms(75);
                        set_motor(0, cw_in_place);
                }
        }

}


void sample_light()
```

```
{
        // The ambient light sensor gives noisy readings. To mitigate this,
        // we take the average of 300 samples in quick succession.

        int number_of_samples = 0;
        int sum = 0;


        // while (number_of_samples < 300)
        {
                int sample = get_ambient_light();

                // -1 indicates a failed sample, which should be discarded.
                if (sample != -1)
                {
                        //       sum = sum + sample;
                        //    number_of_samples = number_of_samples + 1;
                        current_light = sample;
                }
        }


        // Compute the average.
        //current_light = sum / number_of_samples;
}


// user program function
void user_program(void)
```

```
{
	if (init)
	{
		int randseed = 0;
		// generate random seed (must be placed AFTER init_robot()

		for (int i = 0; i<30; i++)
			randseed += get_ambient_light();//generate some random sensor data

		srand(randseed);//seed random variable with some sensor data

						// generate robot id
		ID = rand() & 255;
		//ID = 0;
		init = 0;
		//	set_motion(LEFT);

	}

	sample_light(); //hail the sunshine! let the sunshine in!
	if (current_light <= THRESH_LO)
	{
		set_motion(LEFT);
		set_color(3, 0, 0);
	}
	else if (current_light >= THRESH_HI)
	{
		set_motion(RIGHT);
		set_color(0, 3, 0);
```

```
        }//else if(current_light >= THRESH_STOP)
         //    {
         //           set_motion(STOP);
         //           set_color(3,0,3);
         //    }


}

// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();

        // loop and run each time the user program
        main_program_loop(user_program);
}
```

Asynchronous Consensus

```
#include "libKilobot.h" // include Kilobot library file

typedef enum { false = 0, true = 1 } bool;
typedef enum { stop = 0, forward = 1, left = 2, right = 3 } motion;
typedef enum { done = 0, start = 2, wait = 4 } state;

#define ROOT                (0)
#define MAX                        (100)
#define UNDEFINED           (-1)
#define MINDISTANCE         (37)
#define MAXDISTANCE         (50)
#define BOUNDRANGE          (1)
#define SEC                        (32)
#define HALFSEC                    (16)
#define QUARTSEC            (8)

/* MYDATA */
int8_t myGradient = MAX;
int8_t myID = 1;
state myState = wait;
state myLastState = start;

/* NEIGHBORDATA */
int8_t nextDistance = MAX;  //UNDEFINED   // distance to next kilobot
int8_t prevDistance = UNDEFINED;          // distance to previous kilobot
int8_t nextID = UNDEFINED;                       // ID of next kilobot
int8_t prevID = UNDEFINED;                       // ID of previous kilobot
```

```
state nextState = wait;                               // state of next kilobot
state prevState = wait;                               // state of previous kilobot
int8_t nextGradient = MAX;
int8_t prevGradient = MAX;
//--------------------------------------------


uint8_t lowerBound = 94; //static
uint8_t upperBound = 95; //static

                                          /* Gradient Variables */
uint8_t recGrad = MAX;            //static

static int init = 1;

motion currentMotion = 0;   //static
                                                //static int16_t currentDistance = 0;

                                                /* Delay Variables */
uint32_t lastChanged = 0;   //static
uint32_t lastTime = 0;            //static

                                                /* FLAGS */
bool initGrad = false;  //static
bool initData = false;  //static
bool initNext = false;  //static
bool initPrev = false;  //static
bool timeOut = false;   //static
bool initBound = false; //static
bool avoidFlag = false;
```

```

/* Functions */
void SetMotion(motion newMotion);
void CheckBounds(void);
void InitGrad(void);
void Grad(void);
void SetGradColor(void);
void InitData(void);
void InitTimeOut(void);
void UpdateData(void);
void InitBound(void);
void CheckState(void);
void StateMachine(state currentState);
void MyTimer(void);
void ReduceBounds(void);
void ChangeBounds(void);
//void Avoid(void);
void RootStateMachine(void);
motion RandMotion(void);
// user program function
void user_program(void)
{
	////////////////////////////////////////////////////////////////////////////////////
	//user program code goes here.  this code needs to exit in a resonable amount of time
	//so the special message controller can also run
	////////////////////////////////////////////////////////////////////////////////////

	// if the first time the loop is called, initialise the robot id
	if (init)
```

```
	{
		int randseed = 0;
		// generate random seed (must be placed AFTER init_robot()

		for (int i = 0; i<30; i++)
			randseed += get_ambient_light();//generate some
random sensor data

		srand(randseed);//seed random variable with some sensor
data

						// generate robot id
		myID = (rand() & 127);
		myID = 0;
		//myGradient = 0;
		//myState = wait;
		while (myID <= 0)
		{
			myID = (rand() & 127);
		}

		init = 0;
		lastChanged = kilotick;
		lastTime = kilotick;
	}

	message_out(myGradient, myID, myState);
	enable_tx = 1;

	if (!timeOut)
	{
```

```
                InitTimeOut();
        }
        else
        {
                MyTimer();
        }
        get_message();

        if (myGradient == 0)
        {
                initGrad = true;
        }

        if (message_rx[5] == 1)
        {
                if (!timeOut)
                {
                        InitGrad();   // initialize gradient
                        InitData();   // initialize distance, state, and neighbor IDs
                }
                else
                {
                        UpdateData();

                        if (myID != ROOT)
                        {
                                InitBound();

                                if (myLastState == wait || myLastState == done)
```

```
                        {
                                ChangeBounds();
                        }

                        CheckState();
                        StateMachine(myState);
                        //Avoid();
                        //CheckBounds();
                }
                else
                {
                        CheckState();
                        set_color(3, 0, 0);
                        RootStateMachine();
                }
        }
    }

}


// main
int main(void)
{
    // no instruction should be placed before init_robot();
    // because nothing is already initialised  !!

    // initialise the robot
    init_robot();

```

```
        // loop and run each time the user program
        main_program_loop(user_program);


}

void SetMotion(motion newMotion)
{
        if (currentMotion != newMotion)
        {
                currentMotion = newMotion;
                switch (currentMotion)
                {
                case stop:                                  // Stop
                        set_motor(0, 0);
                        break;

                case forward:                               // Forward
                        set_motor(0xA0, 0xA0);
                        _delay_ms(15);
                        set_motor(cw_in_straight, ccw_in_straight);
                        break;

                case left:                                  // Left
                        set_motor(0, 0xA0);
                        _delay_ms(15);
                        set_motor(0, ccw_in_place);
                        break;

                case right:                         // Right
```

```
                        set_motor(0xA0, 0);
                        _delay_ms(15);
                        set_motor(cw_in_place, 0);
                        break;

                default:
                        set_motor(0, 0);
                        break;

                }
        }
}

void CheckBounds()
{
        if (nextDistance < lowerBound)
        {
                SetMotion(right);
        }
        else if (nextDistance > upperBound)
        {
                SetMotion(left);
        }
        else
        {
                SetMotion(forward);
        }
}

void Grad()
```

```
{
        recGrad = message_rx[0];

        if (myGradient > recGrad + 1)
        {
                myGradient = recGrad + 1;
                SetGradColor();
                initGrad = true;
        }
}

void SetGradColor()
{
        if (myGradient == 1)
        {
                set_color(0, 3, 0);
        }
        else if (myGradient == 2)
        {
                set_color(0, 0, 3);
        }
        else if (myGradient == 3)
        {
                set_color(0, 3, 3);
        }
        else if (myGradient == 4)
        {
                set_color(3, 0, 3);
        }
        else if (myGradient == 5)
```

```
        {
                set_color(3, 3, 0);
        }
        else if (myGradient == 6)
        {
                set_color(0, 3, 1);
        }
        else if (myGradient == 7)
        {
                set_color(3, 1, 1);
        }
        else
        {
                set_color(3, 3, 3);
        }
}

void InitGrad()
{
        if (!initGrad)
        {
                Grad();
        }
}

void InitData()
{
        if (!initData && initGrad)
        {
```

```
		if (!initNext && ((myGradient - 1) == (int8_t)message_rx[0]))
		{
			nextGradient = (int8_t)message_rx[0];
			nextID = (int8_t)message_rx[1];
			nextState = message_rx[2];
			nextDistance = message_rx[3];
			initNext = true;
		}
		else if (!initPrev && ((myGradient + 1) == (int8_t)message_rx[0]))
		{
			prevGradient = (int8_t)message_rx[0];
			prevID = (int8_t)message_rx[1];
			prevState = message_rx[2];
			prevDistance = message_rx[3];
			initPrev = true;
		}

		if (initNext && initPrev)
		{
			initData = true;
		}
	}
}

void InitTimeOut()
{
	if (kilotick > lastChanged + 2 * SEC)
	{
		timeOut = true;
```

```

                if (!initNext)
                {
                        nextGradient = UNDEFINED;
                        nextID = UNDEFINED;
                        nextState = wait;
                        nextDistance = UNDEFINED;
                }

                if (!initPrev)
                {
                        prevGradient = UNDEFINED;
                        prevID = UNDEFINED;
                        prevState = done;
                        prevDistance = UNDEFINED;
                }

        }

}

void UpdateData()
{
        if (nextID == (int8_t)message_rx[1])
        {
                nextState = message_rx[2];
                nextDistance = message_rx[3];
        }

        if (prevID == (int8_t)message_rx[1])
```

```
        {
                prevState = message_rx[2];
                prevDistance = message_rx[3];
        }

        kprinti(myID);
        kprinti(nextID);
        kprinti(nextState);
        kprinti(nextDistance);
        kprinti(prevID);
        kprinti(prevState);
        kprinti(prevDistance);
        kprints("     ");
}

void InitBound()
{
        if (!initBound)
        {
                lowerBound = nextDistance - BOUNDRANGE;
                upperBound = nextDistance + BOUNDRANGE;
                initBound = true;
        }
}

void CheckState()
{
        myLastState = myState;

        if (prevDistance > MAXDISTANCE)
```

```
        {
                myState = wait;
                //set_color(0,0,3);
        }
        else
        {
                if (nextDistance <= MINDISTANCE && nextDistance != UNDEFINED)
                {
                        myState = done;
                        //set_color(0,3,3);
                }
                else if ((nextState == wait && prevState == done) || (nextState == done && prevState == done && nextDistance > MINDISTANCE))
                {
                        myState = start;
                        //set_color(0,3,0);
                }
        }
}

void StateMachine(state currentState)
{
        switch (currentState)
        {
        case done:
                SetMotion(stop);
                //set_color(0,3,3);
                break;

        case start:
```

```
				CheckBounds();
				//set_color(0,3,0);
				break;

		case wait:
				SetMotion(stop);
				//set_color(0,0,3);
				break;

		default:
				SetMotion(stop);
				//set_color(3,3,3);
				break;
		}

}
void MyTimer()
{
		if (kilotick > lastTime + QUARTSEC)
		{
				lastTime = kilotick;
				ReduceBounds();
		}

}

void ReduceBounds()
{
		if (lowerBound > MINDISTANCE)
		{
```

```
                lowerBound -= 1;
                upperBound -= 1;
        }
        else
        {
                if (nextDistance > MINDISTANCE)
                {
                        ChangeBounds();
                }
        }

}

void ChangeBounds()
{
        lowerBound = nextDistance - BOUNDRANGE;
        upperBound = nextDistance + BOUNDRANGE;
}

/*void Avoid()
{
static int8_t currentDistance;
static int8_t prevCurrentDistance;

if(nextID != (int8_t)message_rx[1])
{
currentDistance = message_rx[3];

if(!avoidFlag)
{
```

```
if(currentDistance <= MINDISTANCE)
{
SetMotion(right);
avoidFlag = true;
prevCurrentDistance = currentDistance;
}
}
else if(currentDistance <= MINDISTANCE)
{
if(currentDistance < prevCurrentDistance)
{
SetMotion(left);
}
else
{
SetMotion(right);
}

prevCurrentDistance = currentDistance;
}
else
{
avoidFlag = false;
SetMotion(forward);
}
}

}*/

void RootStateMachine()
```

```
{
        if (myState == start)
        {
                SetMotion(forward);//SetMotion(RandMotion());
        }
        else
        {
                SetMotion(stop);
        }
}

motion RandMotion()
{
        motion myMotion;

        do
        {
                myMotion = rand() & 3;
        } while (myMotion != stop);

        return(myMotion);
}
```

Random ID Generator

```
#include "libKilobot.h" // include Kilobot library file
#include "myLibrary.h"

#define FALSE       (0)
#define TRUE        (1)

static int init = 1;
//static int robot_id=0;

//uint8_t generatedID = 0;

void idGenerator(int ID);



// user program function
void user_program(void)
{
       ///////////////////////////////////////////////////////////////////////////////////////
       //user program code goes here.  this code needs to exit in a resonable amount of time
       //so the special message controller can also run
       ///////////////////////////////////////////////////////////////////////////////////////

       // if the first time the loop is called, initialise the robot id
       if (init)
       {
              int randseed = 0;
```

```
        // generate random seed (must be placed AFTER init_robot()

        for (int i = 0; i<30; i++)
            randseed += get_ambient_light();//generate some random sensor data

        srand(randseed);//seed random variable with some sensor data

                        // generate robot id
                        //robot_id = rand() & 255;
                        //robot_id = rand() & 7;
        robot_id = 0;
        init = 0;
    }

    if (robot_id == 7)
    {
        robot_id = 0;
    }

    robot_id += 1;
    send(robot_id, 0, 0);


    //get_message();
    //if(message_rx[5]==1)
    //{
    //idGenerator(message_rx[0]);
    //}

```

```
        switch (robot_id)
        {
        case 0: set_color(3, 0, 0);
                break;
        case 1:         set_color(3, 3, 0);
                break;
        case 2: set_color(0, 3, 0);
                break;
        case 3:         set_color(0, 3, 3);
                break;
        case 4:         set_color(0, 0, 3);
                break;
        case 5:         set_color(3, 0, 3);
                break;
        case 6: set_color(1, 3, 0);
                break;
        case 7: set_color(3, 3, 3);
                break;
        default: set_color(0, 0, 0);
        }

        _delay_ms(500);
}

// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

```

```
        // initialise the robot
        init_robot();


        // loop and run each time the user program
        main_program_loop(user_program);


}

void idGenerator(int ID)
{
        if (generatedID == FALSE)
        {
                robot_id = rand() & 7;
                generatedID = TRUE;
        }
        else
        {
                if (robot_id == ID)
                {
                        generatedID = FALSE;
                }
        }
}
```

Color Consensus

```
#include "libKilobot.h" // include Kilobot library file
//#include "myLibrary.h"


void color_calc();
void color_change();

#define bsize (20)

uint8_t color = 0;
uint8_t buffer[20] = { 0 };
uint8_t i = 0;
uint8_t full = 0;
uint8_t r, g, b;
uint8_t init = 1;

// user program function
void user_program(void)
{
        if (init == 1)
        {
                int randseed = 0;
                // generate random seed (must be placed AFTER init_robot()

                for (int i = 0; i<30; i++)
                        randseed += get_ambient_light();//generate some
random sensor data

```

```
		srand(randseed);//seed random variable with some sensor data

		color = (rand() & 3); //gnerate a random start color between 0 and 3
		if (color > 2)
		{
			color = 2;
		}

		if (color < 0)
		{
			color = 0;
		}
		color_change();    //set color of led
		_delay_ms(5000);
		init = 0;
	}

	message_out(color, 0, 0); //broadcast my color
	enable_tx = 1;

	//_delay_ms(500);
	get_message(); //listen for other colors

	if (message_rx[5] == 1)
	{
		buffer[i] = message_rx[0]; //store the colors i can see
		i++;
		if (i >= bsize) // restart buffer from the begining
		{
```

```
                        i = 0;
                        full = 1; //begin to new calc
                }
        }

        if (full == 1)
        {
                color_calc(); //figure out most common color
                color_change(); //change my color
                                            //     full =0;

        }
}

// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();

        // loop and run each time the user program
        main_program_loop(user_program);
}

void color_calc()
{
```

```
        for (int k = 0; k < bsize; k++) //take the values in buffer and
tally them up
        {
                switch (buffer[k])
                {
                case 0:  r++;
                        break;
                case 1:  g++;
                        break;
                case 2:  b++;
                        break;
                default: break;

                }
        }

        if ((r > g && r > b) || (r >= g && r >b) || (r > g && r >= b))
        {
                color = 0;
        }
        else if ((g > r && g >b) || (g >= r && g >b) || (g > r && g >=
b))
        {
                color = 1;
        }
        else if ((b > r && b >g) || (b >= r && b >g) || (b > r && b >=
g))
        {
                color = 2;
        }
        else
        {
```

```
                //color = 0;
        }

        r = 0;
        g = 0;
        b = 0;


}


void color_change() // change led to corresponding color
{
        switch (color)
        {
        case 0: set_color(3, 0, 0);
                break;
        case 1: set_color(0, 3, 0);
                break;
        case 2: set_color(0, 0, 3);
                break;
        default: set_color(0, 0, 0);
                break;


        }


}
```

Ambient Light Sensor Calibration

```
#include "libKilobot.h" // include Kilobot library file
//#include "myLibrary.h"



#define MAX  (70)
#define MSIZE  (5)
#define ASIZE (3)
#define MAXGRADIENT (10)

uint8_t ID = 0;
uint8_t message_buffer[5] = { 0 };
uint8_t average_buffer[3] = { 0 };

uint8_t gradient = MAXGRADIENT;
uint8_t avg_gradient = 0;
uint8_t min = 0;
uint8_t i, k = 0;
uint8_t full = 0;

uint8_t init = 1;
int light = 0;
// user program function
void user_program(void)
{
       if (init)
       {
              int randseed = 0;
```

```
		// generate random seed (must be placed AFTER init_robot()

		for (int i = 0; i<30; i++)
			randseed += get_ambient_light();//generate some random sensor data

		srand(randseed);//seed random variable with some sensor data

						// generate robot id
		ID = rand() & 255;
		//ID = 0;
		init = 0;
	}

	set_color(3, 3, 0);
	light = get_ambient_light();
	kprinti(light);
	_delay_ms(500);


}

// main
int main(void)
{
	// no instruction should be placed before init_robot();
	// because nothing is already initialised  !!

	// initialise the robot
	init_robot();
```

```

        // loop and run each time the user program
        main_program_loop(user_program);
}
```

Follow Leader

```
#include "libKilobot.h" // include Kilobot library file
#include "myLibrary.h"

#define root                (0)
#define goalDistance        (40)
#define TOOFAR              (60)

#define STOP                (0)
#define FORWARD             (1)
#define TURN                (2)

static int init = 1;
static int robot_id = 0;

static uint8_t previousState = STOP;
static uint8_t state = FORWARD;

static uint8_t prevDistance = 2;
static uint8_t Distance = 1;

static uint8_t start = 1;

static uint32_t messageBuffer = 0;


void stateSpinUp()
{
       if (start == 1)
```

```
        {
                spinUp();
                start = 0;
        }
}

void updateDistance()
{
        if (message_rx[5] == 1)
        {
                message_rx[5] = 1;
                prevDistance = Distance;
                Distance = message_rx[3];
        }
}

// user program function
void user_program(void)
{


        // if the first time the loop is called, initialise the robot id
        if (init)
        {
                int randseed = 0;
                // generate random seed (must be placed AFTER init_robot()

                for (int i = 0; i<30; i++)
                        randseed += get_ambient_light();   //generate some random sensor data
```

```

		srand(randseed);						//seed random variable with some sensor data


	// generate robot id
	//robot_id = rand() & 255;
		robot_id = root;

		//set_motor(0xA0,0xA0);
		init = 0;
	}

	send(robot_id, 0, 0);

	if (robot_id != root)
	{
		get_message();
		if (message_rx[5] == 1)
		{
			prevDistance = Distance;
			Distance = message_rx[3];
			messageBuffer = 0;
		}
		else
		{
			if (messageBuffer != 3000)
			{
				messageBuffer++;
			}
		}
```

```
		if (Distance < goalDistance)
		{
			stop();
			state = STOP;

		}
		else if (messageBuffer == 3000)
		{

			turnAround();
			updateDistance();
			start = 1;
			stop();
			_delay_ms(50);
			forward();
			_delay_ms(4000);
			state = TURN;

		}
		else //if( (prevDistance >= Distance) && (Distance >= goalDistance) )
		{
			forward();
			state = FORWARD;
		}

		if (previousState != state)
		{
			start = 1;
```

```
                }

                previousState = state;
        }

}



// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();


        // loop and run each time the user program
        main_program_loop(user_program);


}
```

Leader

```
#include "libKilobot.h" // include Kilobot library file
//#include "myLibrary.h"
uint8_t ID = 0;
uint8_t i = 0;
uint8_t k = 0;

uint8_t init = 1;

// user program function
void user_program(void)
{
        if (init)
        {
                int randseed = 0;
                // generate random seed (must be placed AFTER init_robot()

                for (int i = 0; i<30; i++)
                        randseed += get_ambient_light();//generate some random sensor data

                srand(randseed);//seed random variable with some sensor data

                                                // generate robot id
                ID = rand() & 255;
                ID = 0;
                init = 0;
        }

```

```
        set_color(3, 0, 0);
        enable_tx = 1;
        message_out(0, 1, 0);
        get_message();



        if (message_rx[5] == 1)
        {
                if (message_rx[0] == 1)
                {
                        if (message_rx[3] <= 60)
                        {
                                set_motor(60, 60);
                        }
                        else
                        {
                                set_motor(0, 0);
                        }
                }
        }
        else
        {
                set_color(0, 3, 0);
                set_motor(60, 60);
        }


}

```

```
// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();

        // loop and run each time the user program
        main_program_loop(user_program);
}
```

Fixed Reference Consensus

```
include "libKilobot.h" // include Kilobot library file

#define ROOT                (0)
#define STOP                (0)
#define FORWARD                    (1)
#define LEFT                (2)
#define RIGHT               (3)
#define NORMAL                     (1)


int LOWERBOUND = 94;
int UPPERBOUND = 95;

static int init = 1;
static uint8_t initBound = 1;
static int robot_id = 0;

static int currentMotion = 0;
static int currentDistance = 0;
uint32_t lastChanged = 0;

static int stopFlag = 0;

void SetMotion(int newMotion);
void CheckBounds(void);

// user program function
void user_program(void)
```

```
{
	///////////////////////////////////////////////////////////////////////////////////////
	//user program code goes here.  this code needs to exit in a resonable amount of time
	//so the special message controller can also run
	///////////////////////////////////////////////////////////////////////////////////////

	// if the first time the loop is called, initialise the robot id
	if (init)
	{
		int randseed = 0;
		// generate random seed (must be placed AFTER init_robot()

		for (int i = 0; i<30; i++)
			randseed += get_ambient_light();//generate some random sensor data

		srand(randseed);//seed random variable with some sensor data

						// generate robot id
		robot_id = rand() & 255;
		init = 0;
		//robot_id = 0;
		lastChanged = kilotick;
	}

	if (robot_id != ROOT)
	{
		if (kilotick > (lastChanged + 8))
		{
```

```
                    lastChanged = kilotick;

                    if (LOWERBOUND > 32)
                    {
                           LOWERBOUND -= 1;
                           UPPERBOUND -= 1;
                    }
                    else
                    {
                           stopFlag = 1;
                    }
             }

             get_message();

             if (message_rx[5] == 1)
             {
                    if (message_rx[0] == ROOT)
                    {
                           currentDistance = message_rx[3];

                           if (initBound == 1)
                           {
                                  LOWERBOUND = currentDistance - 1;
                                  UPPERBOUND = currentDistance + 1;
                                  initBound = 0;
                           }

                           if (currentDistance <= 33)
                           {
```

```
                                        SetMotion(STOP);
                                        stopFlag = 1;
                                }
                                else if (stopFlag == 1 && currentDistance > 
33)
                                {
                                        LOWERBOUND = currentDistance - 1;
                                        UPPERBOUND = currentDistance + 1;
                                        stopFlag = 0;
                                }
                                else
                                {
                                        CheckBounds();
                                }
                        }
                }
                else if (currentDistance == 0)
                {
                        return;
                }
        }
        else
        {
                message_out(robot_id, 5, 0);
                enable_tx = 1;
                set_color(3, 0, 0);
        }

}
```

```
// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();


        // loop and run each time the user program
        main_program_loop(user_program);


}

void SetMotion(int newMotion)
{
        if (currentMotion != newMotion)
        {
                currentMotion = newMotion;
                switch (currentMotion)
                {
                case STOP:                                  // Stop
                        set_motor(0, 0);
                        set_color(0, 0, 0);
                        break;

                case FORWARD:                               // Forward
```

```
                        set_motor(0xA0, 0xA0);
                        _delay_ms(15);
                        set_motor(cw_in_straight, ccw_in_straight);
                        set_color(0, 3, 0);
                        break;

                case LEFT:                                      // Left
                        set_motor(0, 0xA0);
                        _delay_ms(15);
                        set_motor(0, ccw_in_place);
                        set_color(3, 0, 0);
                        break;

                case RIGHT:                             // Right
                        set_motor(0xA0, 0);
                        _delay_ms(15);
                        set_motor(cw_in_place, 0);
                        set_color(0, 0, 3);
                        break;

                default:
                        set_motor(0, 0);
                        set_color(0, 0, 0);
                        break;

                }
        }
}
```

```
void CheckBounds()
{
        if (currentDistance < LOWERBOUND)
        {
                SetMotion(RIGHT);
        }
        else if (currentDistance > UPPERBOUND)
        {
                SetMotion(LEFT);
        }
        else
        {
                SetMotion(FORWARD);
        }
}
```

Multiple Agent Orbiting

```
#include "libKilobot.h" // include Kilobot library file

#define ROOT                (0)
#define STOP                (0)
#define FORWARD                    (1)
#define LEFT                (2)
#define RIGHT               (3)
#define NORMAL                     (1)
#define LOWERBOUND          (54)
#define UPPERBOUND   (55)
#define RANGE            (40)


static int init = 1;
static int flag = 0;
static int go = 0;
static int robot_id = 0;

static int currentMotion = 0;
static int currentDistance = 0;
static int previousDistance = 0;
void SetMotion(int newMotion);
void CheckBounds(void);

// user program function
void user_program(void)
{
```

```
	/////////////////////////////////////////////////////////////////////////////////////
	//user program code goes here.  this code needs to exit in a resonable amount of time
	//so the special message controller can also run
	/////////////////////////////////////////////////////////////////////////////////////

	// if the first time the loop is called, initialise the robot id
	if (init)
	{
		int randseed = 0;
		// generate random seed (must be placed AFTER init_robot()

		for (int i = 0; i<30; i++)
			randseed += get_ambient_light();//generate some random sensor data

		srand(randseed);//seed random variable with some sensor data

						// generate robot id
		while (robot_id == 0) //make sure the blasted thing is never 0
		{
			robot_id = rand() & 255;
		}
		init = 0;
		//robot_id = 0;
	}


	if (robot_id != ROOT)
```

```
	{
		message_out(robot_id, go, 0); // broadcast my id and go wether the other guy can go
		enable_tx = 1;


		get_message();

		if (message_rx[5] == 1)
		{

			if (message_rx[0] == 1) //check if it ok to move
			{
				flag = 0; //ok to move
				go = 0;
			}

			if (flag == 0) //we're ok to move
			{
				CheckBounds(); //standard orbiting proecedure
			}
			else if (flag != 2) // not ok to move
			{
				SetMotion(STOP); //stop
			}

			if (message_rx[0] == ROOT) //if the message is from root
			{
				currentDistance = message_rx[3]; //update distance from root
```

```

            }
            else
            {
                if (message_rx[3] < RANGE && flag == 0) //are we too close to another bot?
                {
                    previousDistance = message_rx[3]; //store this distance
                    flag = 1; //update flag
                    go = 0;  //make sure other bit is not moving

                    if (robot_id < message_rx[0]) // am i the lower bot?
                    {
                        flag = 2; // if i am, movement is allowed
                    }

                }
                else if (flag == 2) //if im the lower id, i can move
                {
                    CheckBounds();
                    if (previousDistance >= message_rx[4] + 5) //check to see if ive gotten closer, with in a tolerance
                    {
                        flag = 1; // o no!, i have!, i better stop
                        go = 1; // better tell the other guy to move though
                    }
                    else
                    {
```

```
                                                flag = 0; // ok my distanc eis increasing, i can move freely
                                                go = 1; //tell the other guy its ok now
                                        }
                                }
                                else // my range is greater than the range, i can move
                                {
                                        flag = 0;
                                        go = 1;

                                }


                        }


                }
                else if (currentDistance == 0)
                {
                        return;
                }
        }
        else
        {
                message_out(robot_id, 0, 0);
                enable_tx = 1;
                set_color(3, 0, 0);
        }

```

```
}


// main
int main(void)
{
        // no instruction should be placed before init_robot();
        // because nothing is already initialised  !!

        // initialise the robot
        init_robot();


        // loop and run each time the user program
        main_program_loop(user_program);


}

void SetMotion(int newMotion)
{
        if (currentMotion != newMotion)
        {
                currentMotion = newMotion;
                switch (currentMotion)
                {
                case STOP:                                  // Stop
                        set_motor(0, 0);
                        set_color(0, 0, 0);
                        break;
```

```

			case FORWARD:					// Forward
				set_motor(0xA0, 0xA0);
				_delay_ms(15);
				set_motor(cw_in_straight, ccw_in_straight);
				set_color(0, 3, 0);
				break;

			case LEFT:					// Left
				set_motor(0, 0xA0);
				_delay_ms(15);
				set_motor(0, ccw_in_place);
				set_color(3, 0, 0);
				break;

			case RIGHT:				// Right
				set_motor(0xA0, 0);
				_delay_ms(15);
				set_motor(cw_in_place, 0);
				set_color(0, 0, 3);
				break;

			default:
				set_motor(0, 0);
				set_color(0, 0, 0);
				break;

			}
		}
}
```

```


void CheckBounds()
{
	if (currentDistance < LOWERBOUND)
	{
		SetMotion(RIGHT);
	}
	else if (currentDistance > UPPERBOUND)
	{
		SetMotion(LEFT);
	}
	else
	{
		SetMotion(FORWARD);
	}
}
```

Atmega128 Messaging

```
/*
* kilobot_message_send.c
*
* Created: 10/15/2015 9:37:19 AM
*  Author: jlamkin
*/

#define F_CPU  (8000000L)
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/delay.h>

static uint8_t tx_mask = 1;  //0

int send_message(int a, int b, int c);

int main(void)
{
       //XDIV = 0x00;
       //XDIV = 0x10;
       // 1 means output, 0 input
       //1 means high 0 low
       DDRB = 1;
       PORTB = 0;

       while (1)
       {
              //PORTB = 0x01;
```

```
			send_message(100, 0, 0);
			//_delay_ms(200);
			//PORTB ^= 0x01;


		}

}

int send_message(int a, int b, int c)
{
		sei();



		//any messages already being received



		uint16_t data_out[4];
		uint8_t data_to_send[4] = { a,b,c,255 };



		//prepare data checksum to send
		data_to_send[3] = data_to_send[2] + data_to_send[1] + 
data_to_send[0] + 128;

		//prepare data to send
		for (int i = 0; i<4; i++)
```

```
        {
                data_out[i] = (data_to_send[i] & (1 << 0)) * 128 +
                        (data_to_send[i] & (1 << 1)) * 32 +
                        (data_to_send[i] & (1 << 2)) * 8 +
                        (data_to_send[i] & (1 << 3)) * 2 +
                        (data_to_send[i] & (1 << 4)) / 2 +
                        (data_to_send[i] & (1 << 5)) / 8 +
                        (data_to_send[i] & (1 << 6)) / 32 +
                        (data_to_send[i] & (1 << 7)) / 128;

                data_out[i] = data_out[i] << 1;
                data_out[i]++;
        }

        uint8_t collision_detected = 0;
        cli();//start critical



                  //send start pulse
        DDRB = 1; //DDRB |= tx_mask;
        PORTB = 1;    //PORTB |= tx_mask;
        asm volatile("nop\n\t");
        asm volatile("nop\n\t");
        PORTB = 0;    //PORTB&= ~tx_mask;

                                //wait for own signal to die down
        for (int k = 0; k<53; k++) //53
                asm volatile("nop\n\t");


```

```
        //check for collision
        for (int k = 0; k<193; k++)
        {
        }
        if (collision_detected == 0)
                for (int byte_sending = 0; byte_sending<4; byte_sending++)
                {
                        int i = 8;
                        while (i >= 0)
                        {
                                if (data_out[byte_sending] & 1)
                                {
                                        PORTB = 1; //PORTB |= tx_mask; 1
                                        asm volatile("nop\n\t");
                                        asm volatile("nop\n\t");
                                }
                                else
                                {
                                        PORTB = 0;//PORTB &= ~tx_mask; 0
                                        asm volatile("nop\n\t");
                                        asm volatile("nop\n\t");
                                }
```

```
                        PORTB = 0; //PORTB &= ~tx_mask;
                        for (int k = 0; k<35; k++) //3500
                        {
                            asm volatile("nop\n\t");
                        }

                        data_out[byte_sending] = data_out[byte_sending] >> 1;
                        i--;
                    }

                }//end of safe

                //ensure led is off
        PORTB = 0;//PORTB &= ~tx_mask;
        DDRB = 0;//DDRB &= ~tx_mask;
        sei();//end critical
        return(0);
}
```

myLibrary.h

```
#ifndef __myLibrary__
#define __myLibrary__

//*****************************************
//                           Variables
//*****************************************
#define FALSE        (0)
#define TRUE         (1)

static int robot_id;
uint8_t generatedID;

//*****************************************
//                           Functions
//*****************************************
void spinUp(void);
void stop(void);
void forward(void);
void left(void);
void right(void);
void send(uint8_t, uint8_t, uint8_t);
void turnAround(void);
void clockDelay(uint32_t);
void idGenerator(int ID);

#endif
```

myLibrary.c

```
// myLibrary.c

//*****************************************************************
//     I'm gonna make my own library with blackjack and hookers
//*****************************************************************

#include "libKilobot.h"
#include "myLibrary.h"

#define FALSE       (0)
#define TRUE        (1)

static int robot_id = 0;
uint8_t generatedID = 0;

void spinUp()
{
       set_motor(0xA0, 0xA0);
       _delay_ms(15);
}

void stop()
{
       set_motor(0, 0);
}

void forward()
{
```

```
        spinUp();
        set_motor(cw_in_straight, ccw_in_straight);
}


void left()
{
        spinUp();
        set_motor(cw_in_place, 0);
}

void right()
{
        spinUp();
        set_motor(0, ccw_in_place);
}

void send(uint8_t a, uint8_t b, uint8_t c)
{
        message_out(a, b, c);
        enable_tx = 1;
}

void turnAround()
{
        right();
        clockDelay(6400);
}

void clockDelay(uint32_t duration)
```

```
{
        int time = clock + duration;
        while (clock <= time)
        {
                get_message();
                if (message_rx[5] == 1)
                {
                        break;
                }
        }
}

void idGenerator(int ID)
{
        if (generatedID == FALSE)
        {
                robot_id = rand() & 7;
                generatedID = TRUE;
        }
        else
        {
                if (robot_id == ID)
                {
                        generatedID = FALSE;
                }
        }
}
```

Appendix D

Program Kilobots

1. Remove Kilobots from box
2. Open "Kilobot controller" shortcut is on desktop
3. Connect Kilobot controller to PC
4. Click "..." and select a hex file to download onto the Kilobots
5. Previous experiments are located in C:\KilobotController\Experiments
6. Select the project you want, then double click the "default" folder
7. Select the HEX file in the folder.
8. Click Open
9. Click Wake Up in the Kilobot controller GUI. The Kilobots should flash yellow
10. Click Pause. The Kilobots should still flash yellow
11. Click Program Flash. A command window will open, wait until it closes
12. Click Boatload. The Kilobots will flash red, green, and blue for a second, and then continue to flash blue
13. Wait until the Kilobots are done flashing blue
14. Click Stop
15. Click Run to start the Kilobots
16. To stop the Kilobots: click Pause
17. To put away the Kilobots: click Pause, once they flash yellow, click Sleep
18. The Kilobots will now flash white very slowly, signifying they are in sleep mode
19. The Kilobots can then be put away

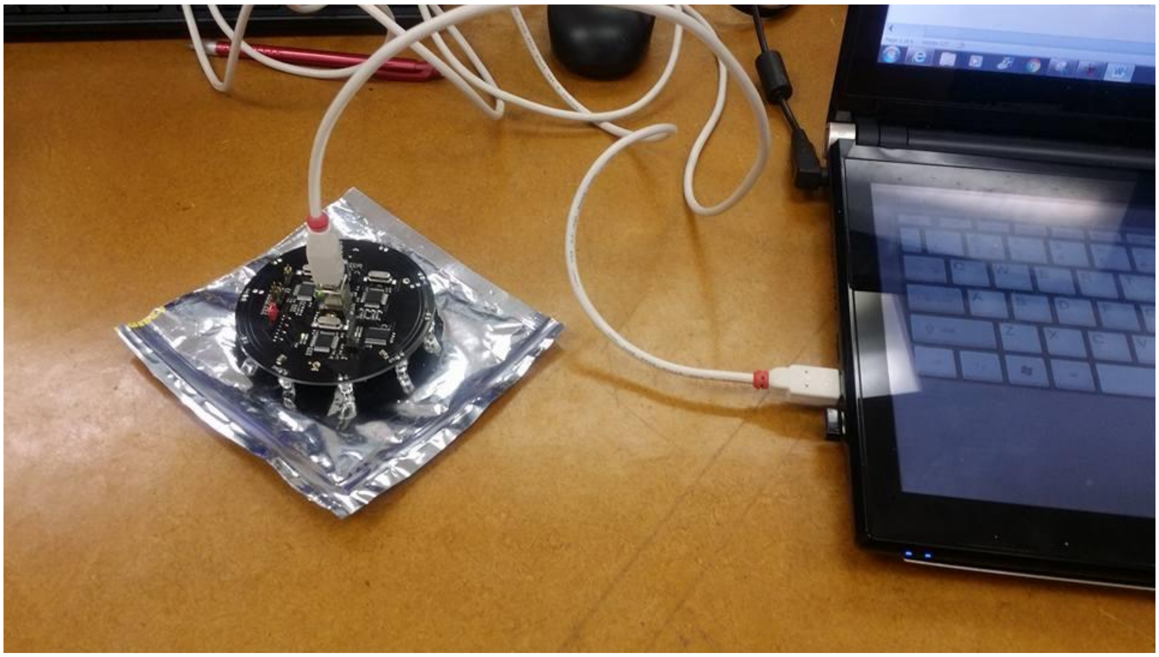
Note: for Gradient and Orbiting programs, a root is needed. So program a Kilobot or two with the root project.

Note: Kilobots must be in pause mode when uploading a program.

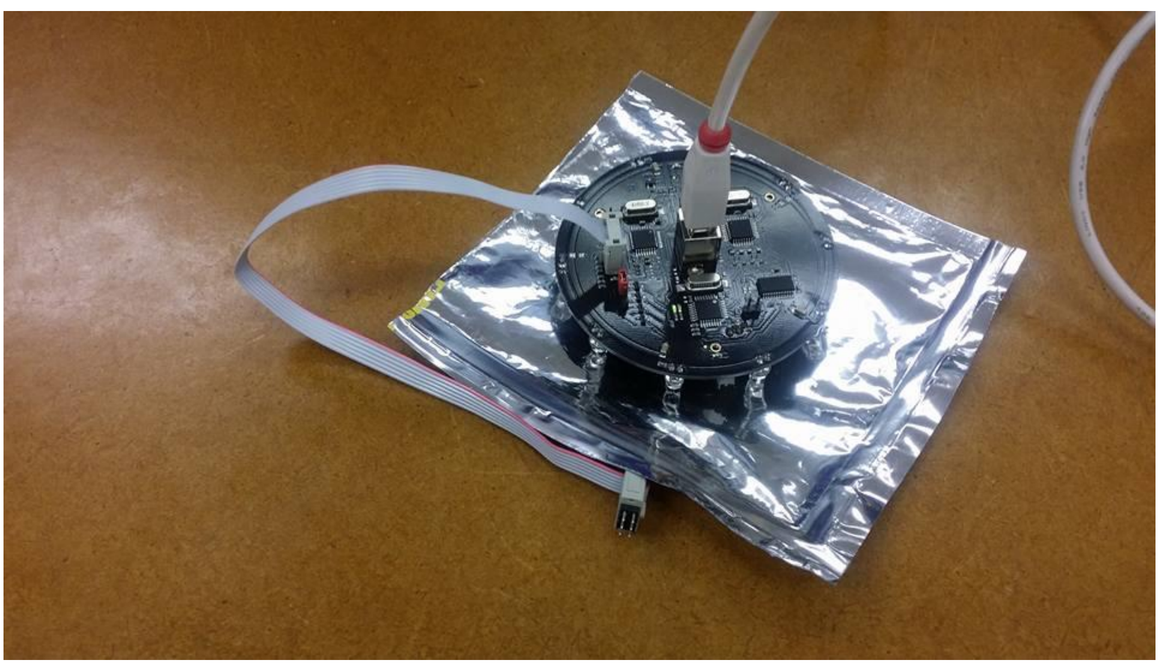
Appendix E

Flash Kilobot Firmware

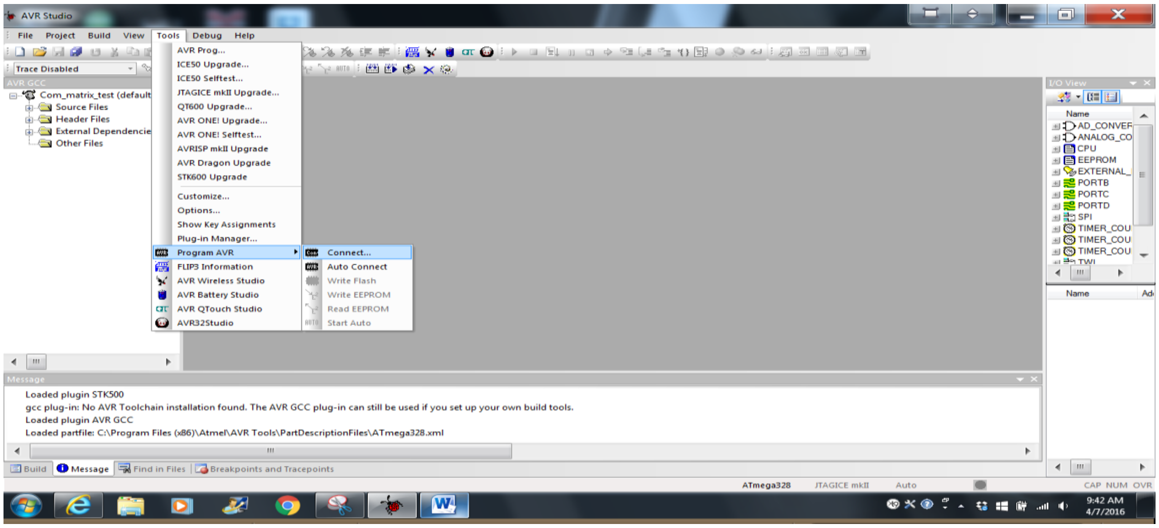
1. Connect controller to pc



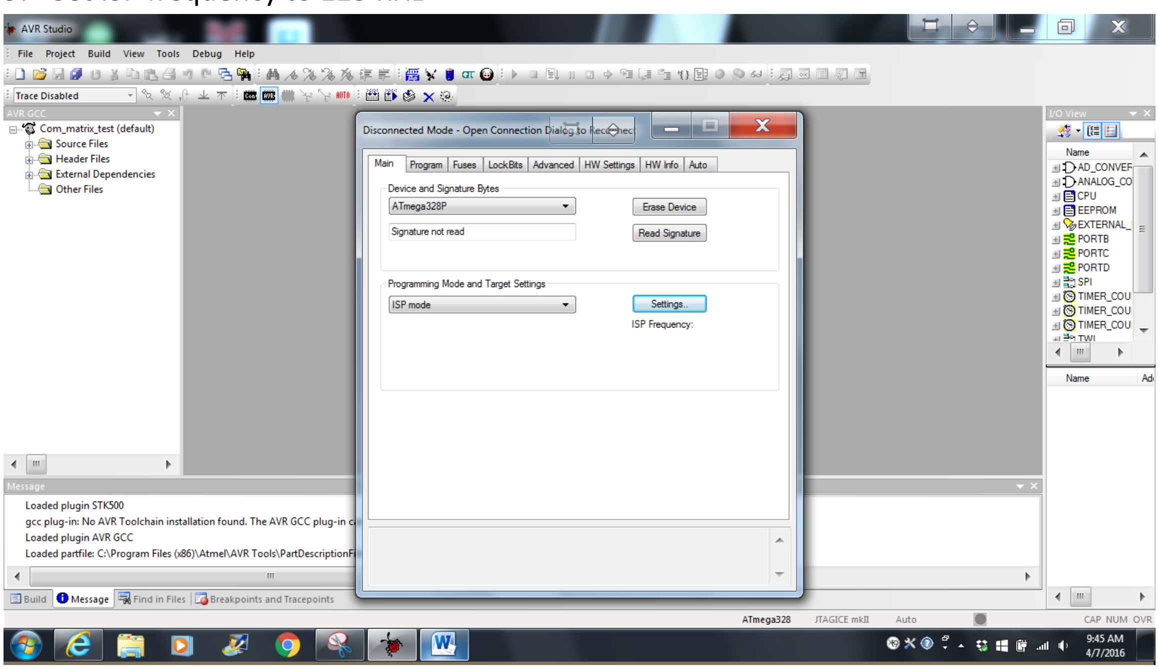
2. Hook firmware cable to controller



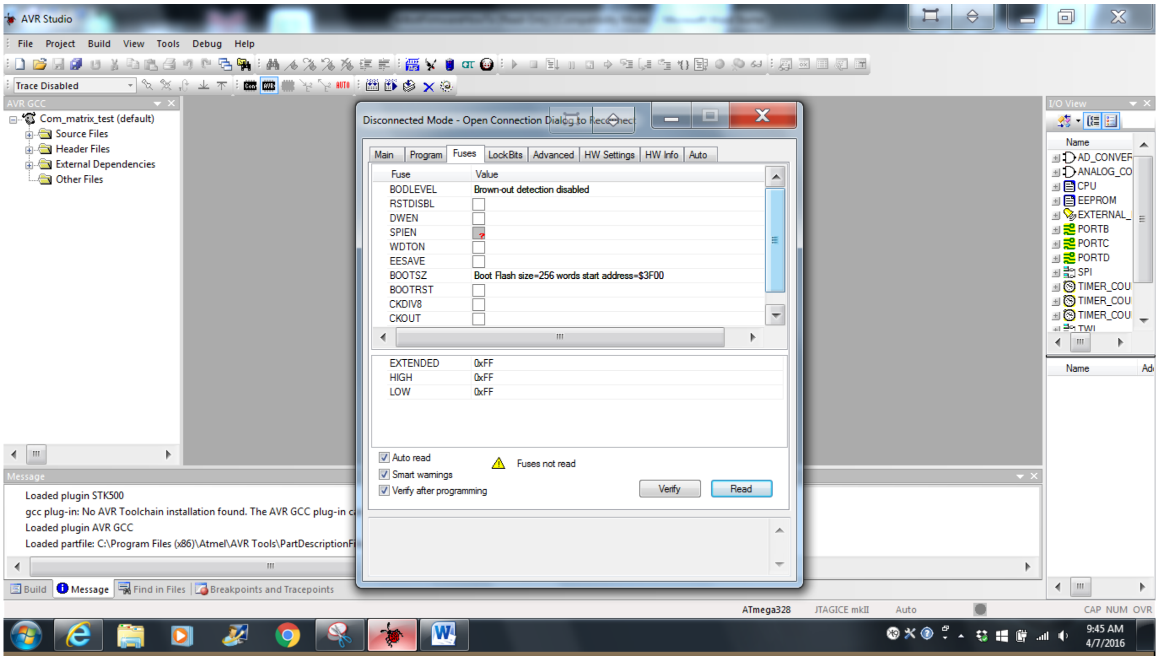
3. In AVR Studio, go to Tools > Program AVR > Connect…



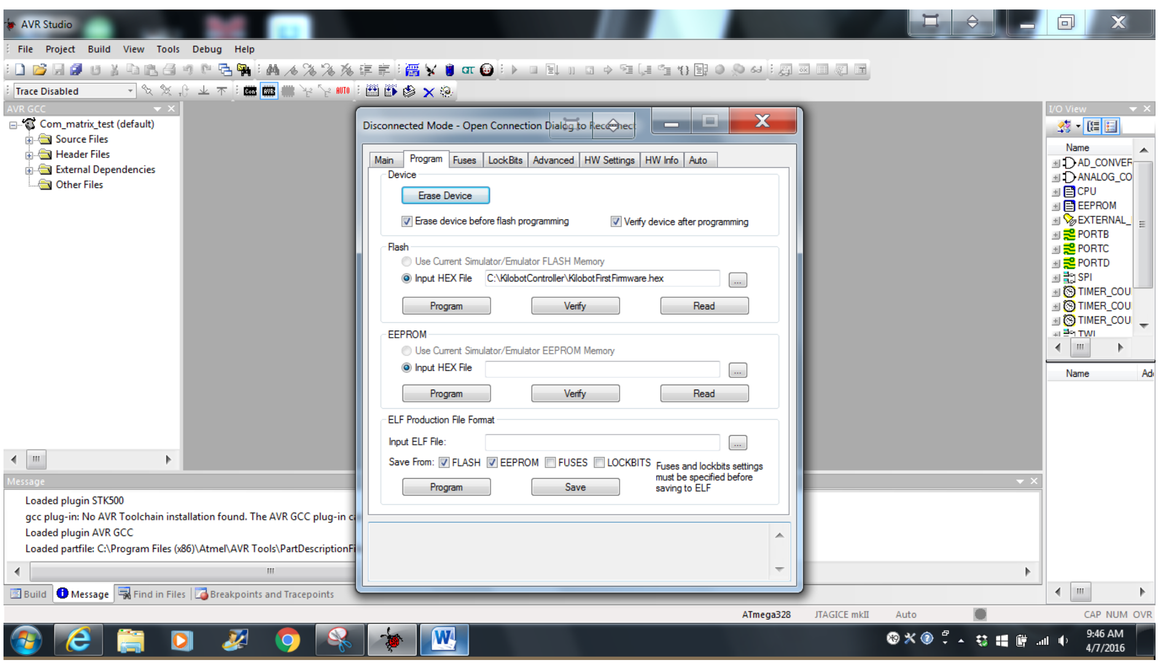
4. Select AVRISP MKII
5. Click USB
6. Click Connect
7. In the Main tab, select ATmega328p
8. Atmega328 for OHC
9. Set ISP frequency to 125 KHz



10. Click Fuses tab
11. In Fuses pane, set fuses to:
    - EXTENDED: 0xFF
    - HIGH: 0xD1
    - LOW: 0xE2

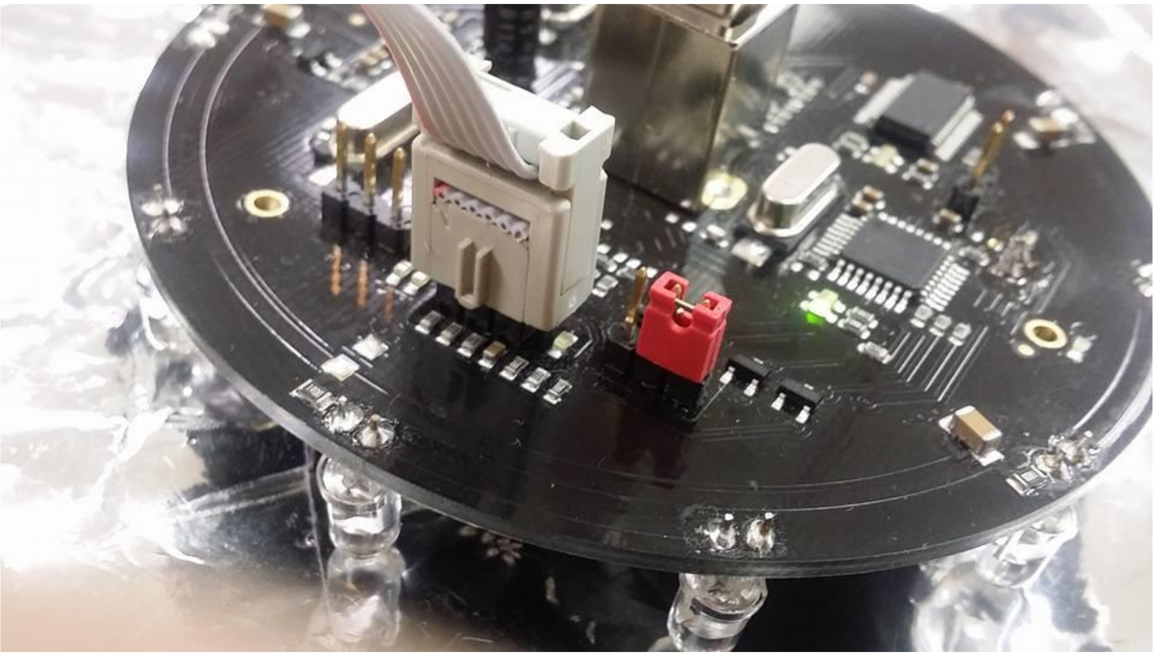


12. Select Program tab
13. In Program pane, select “Input HEX File” checkbox
14. Include `firmware.hex`

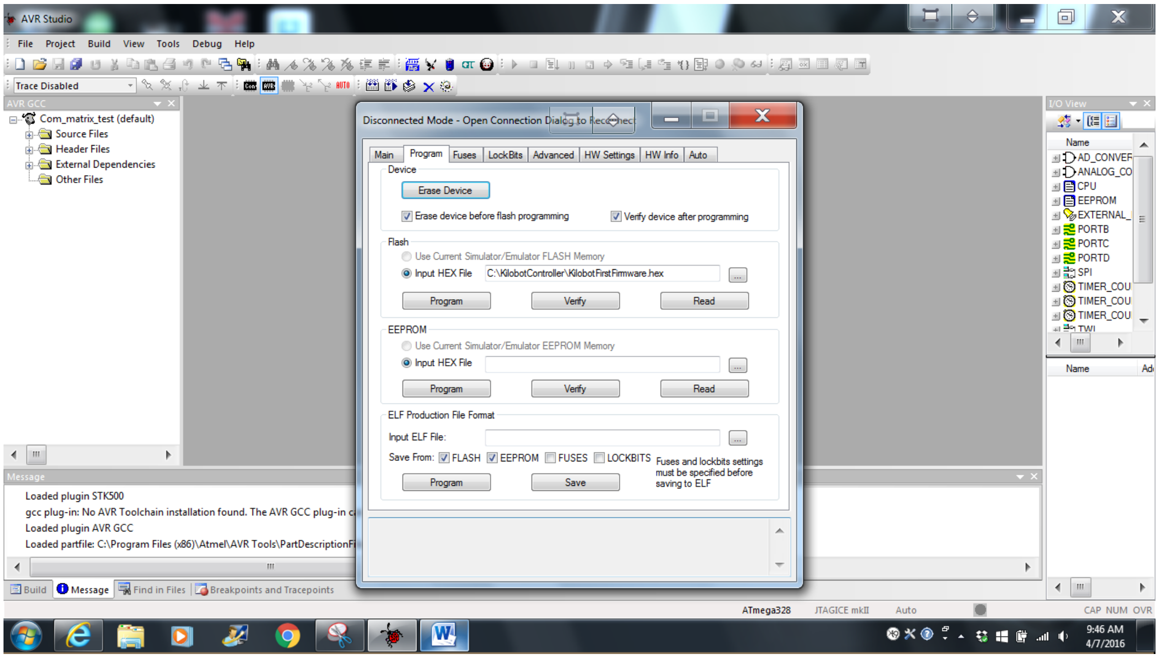


15. Connect a powered on kilobot to the firmware cable. Cable should be held at a slight angle to ensure the pins are touching the connection points.

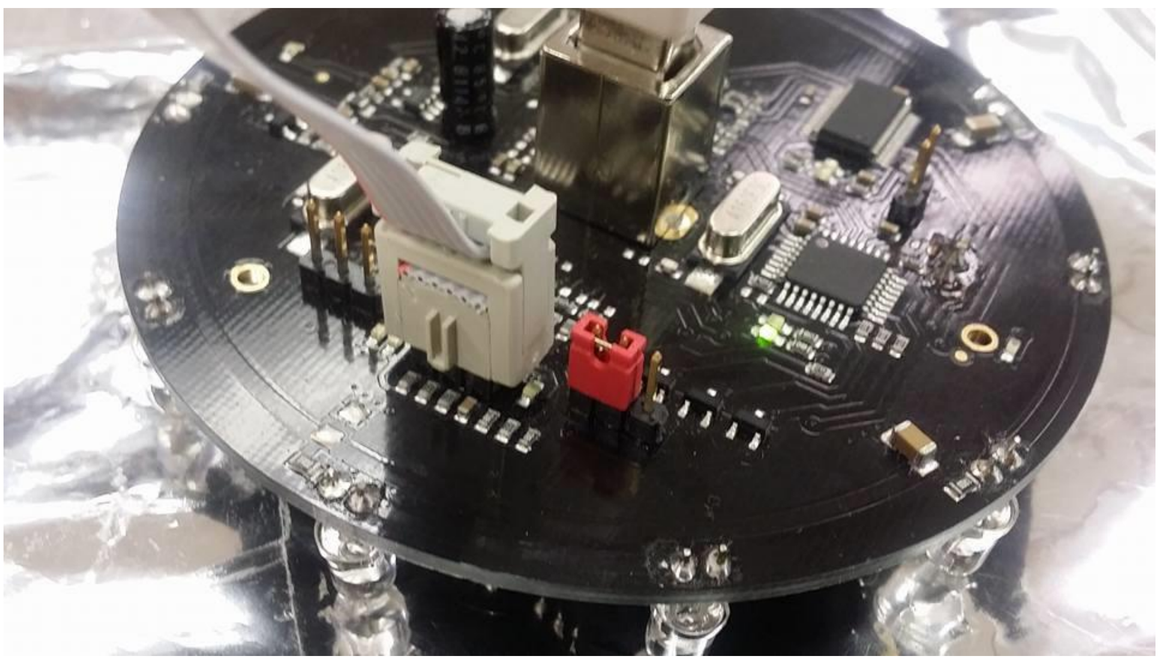
16. Change the connection pin on the controller (red cover)



17. Click the Program button in the Program pane under Flash



18. Remove the Kilobot
19. Set controller pin back



Appendix F

QBot 2 Simulink Model

This section provides information on a Simulink model used during experimentation with the QBot 2.

Overall Simulink Model

Figure F.1 shows the overall Simulink model for the QBots. It is comprised of the HIL Initialize block, global variables, and four subsystems: Localization, Communication, Motor Control, and Data Acquisition.

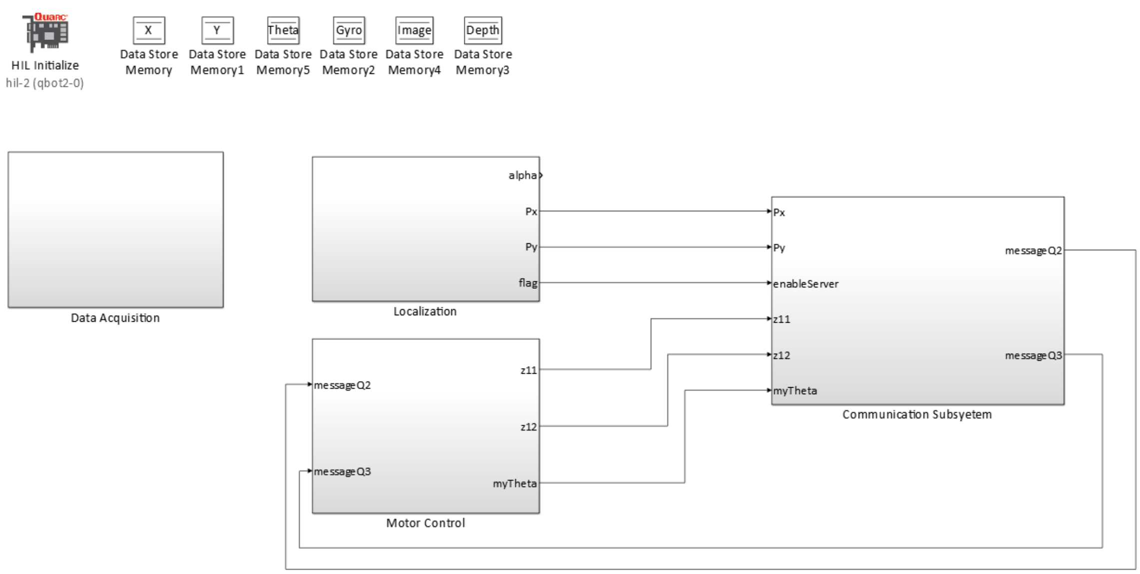


Figure F.1 Overall Simulink Model

Localization

The following Simulink blocks are located in the localization subsystem block shown in the figure above.

Localization Subsystem

The localization subsystem is divided into two distinct parts, color detection and determine position. This can be seen in Figure F.2.The localization subsystem outputs the calculated xy- coordinates, the angle from the Kinect sensor, and a flag. If the flag is high, then the localization process has been completed.

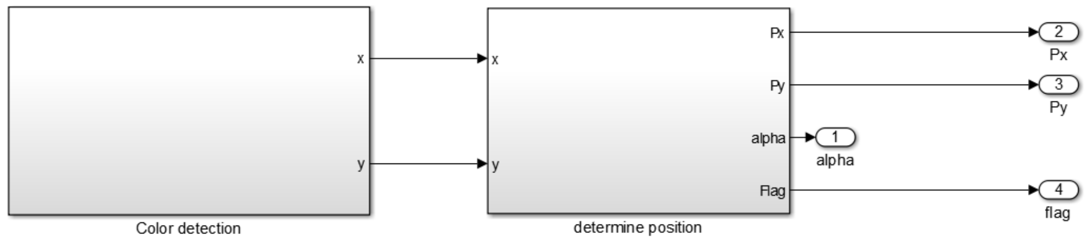


Figure F.2 Localization Subsystem

Color Detection Subsystem

The color detection subsystem can be seen in the figure below, and utilizes the Quanser Simulink Find Object block. This block will output the row and column number of a detected objects center of mass. The inputs used are an image taken from the Kinect sensor, and a flag. This flag controls when the block is active. If the flag is low, then the image will be processed. If the flag is high, then no processing will occur. In this model the block is active for the first three seconds of runtime. For more information on the Find Object block see Appendix G.

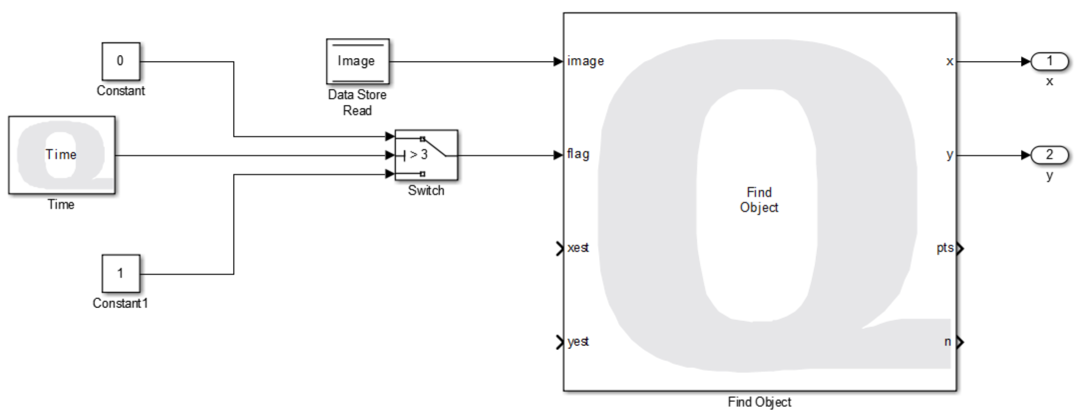


Figure F.3 Color Detection Subsystem

Determine Position Subsystem

The determine position subsystem performs the calculations mentioned in section 4.2 through the use of a MATLAB function. The inputs to the MATLAB function are the obtained angles (α), gyroscope readings, a depth image from the Kinect sensor, and the center of mass of the identified object. The outputs are the new xy-coordinates, the angle from the Kinect sensor, and the flag described in the section Localization Subsystem.

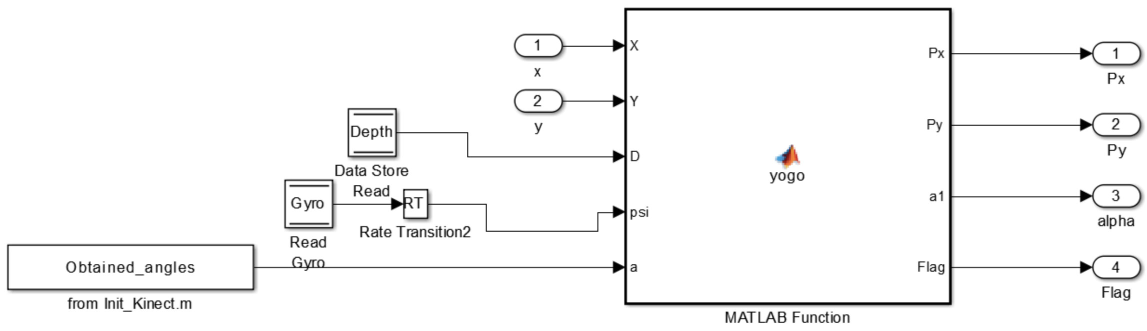


Figure F.4 Determine Position Subsystem

Communication

The following Simulink blocks are located in the communication subsystem block seen in Figure F.1

Communication Subsystem

The communication subsystem establishes a server connection and a client connection with other QBots. Data is put into an array in the message convert MATLAB function block. Messages are sent, or received, through a Stream Server Simulink block and Stream Client Simulink block. The Stream Server and Stream Client Simulink blocks are described in Appendix G.

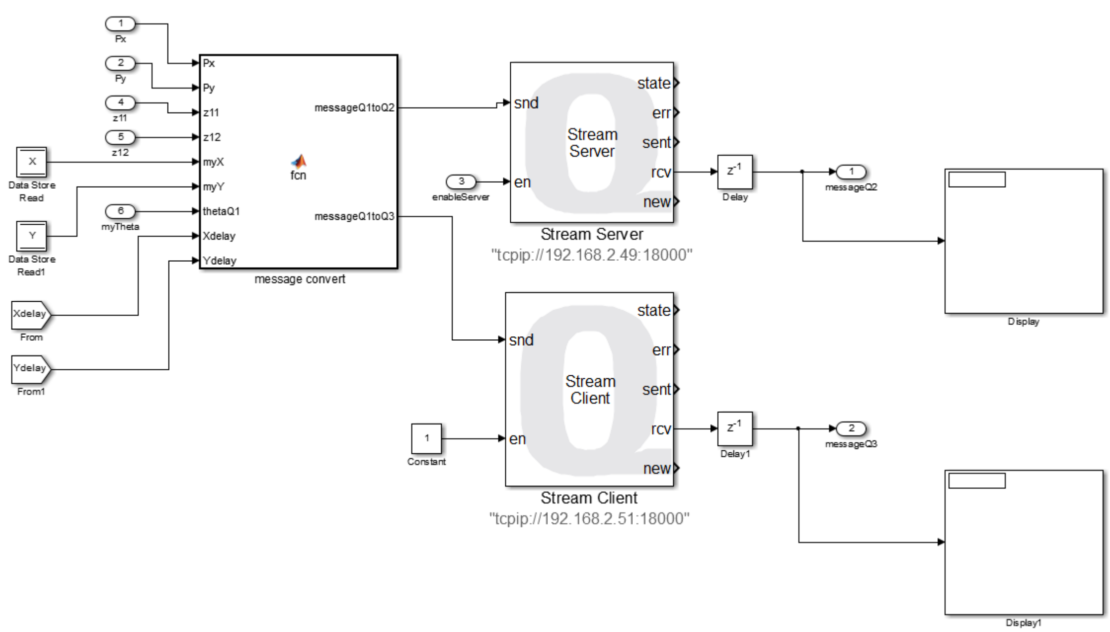


Figure F.5 Communication Subsystem

Motor Control

The following Simulink blocks are located in the motor control subsystem block seen in Figure F.1

Motor Control Subsystem

The Motor Control subsystem contains all control logic for a QBot 2. Inputs to the MATLAB function block, motor logic control, include the QBot's position and orientation information, messages received from other QBots, and time. The constant value blocks were used to specify trajectory information. Outputs of the MATLAB function block include the left and right wheel motor velocities, and the linearized position information of the QBot 2. The motor velocities are connected to a switch. The switch also takes the fuzzy logic control output. The value of a control signal outputted by a fuzzy subsystem will determine which value to send to the motors.

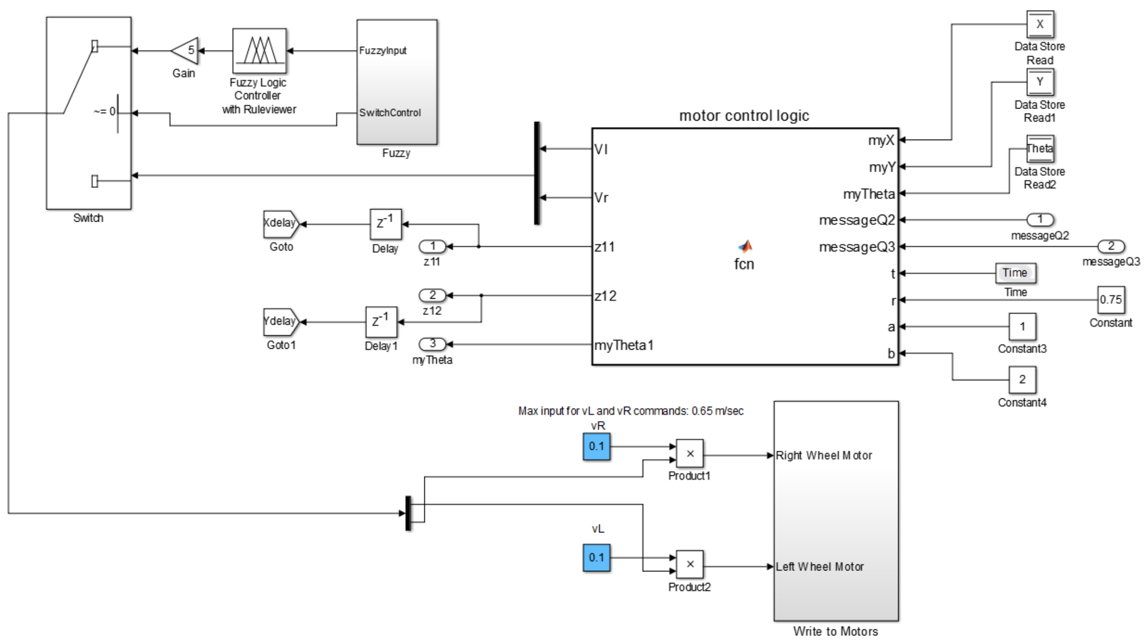


Figure F.6 Motor Control Subsystem

Fuzzy Subsystem

The fuzzy subsystem contains two MATLAB function blocks. Each block takes a depth image as an input. The Depf finder function outputs either one or zero based on the range of objects in the depth image. The Partition function takes the depth image and separates it into three separate sections. The function then outputs the minimum non-zero value found in each section.

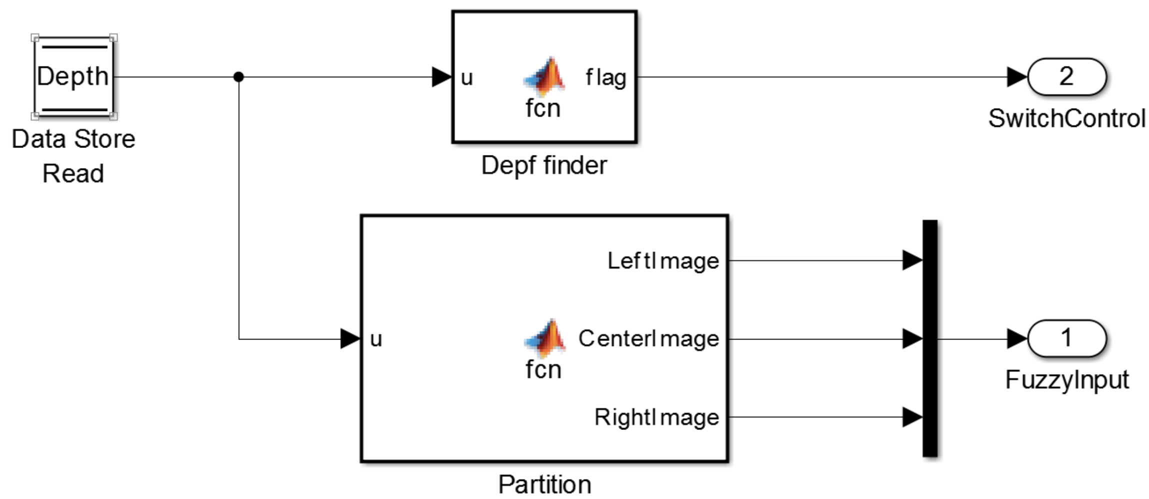


Figure F.7 Fuzzy Subsystem

Write to Motors Subsystem

The write to motors subsystem contains the HIL Write Simulink block. The HIL Write block takes The right and left wheel velocities as inputs.

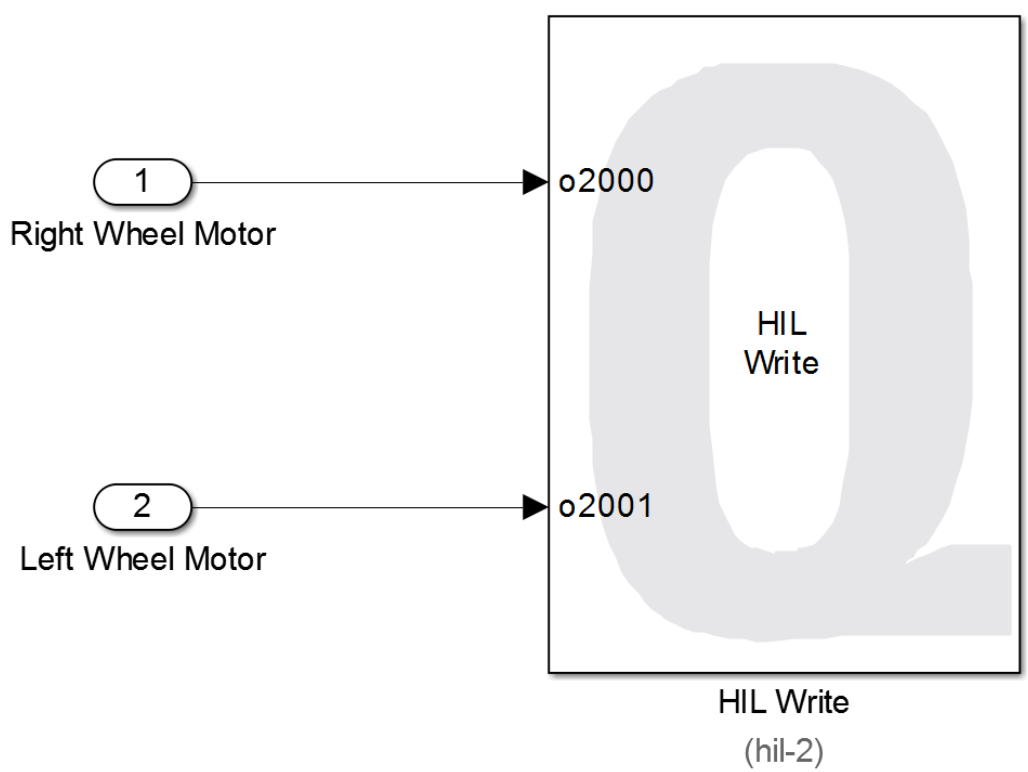


Figure F.8 Write to Motors Subsystem

Data Acquisition

The following Simulink blocks are located in the data acquisition subsystem block seen in Figure F.1.

Data Acquisition Subsystem

The Data Acquisition Subsystem collects all data information for a QBot 2. This information includes the Kinect image and depth data, as well the QBot's current position and orientation data.

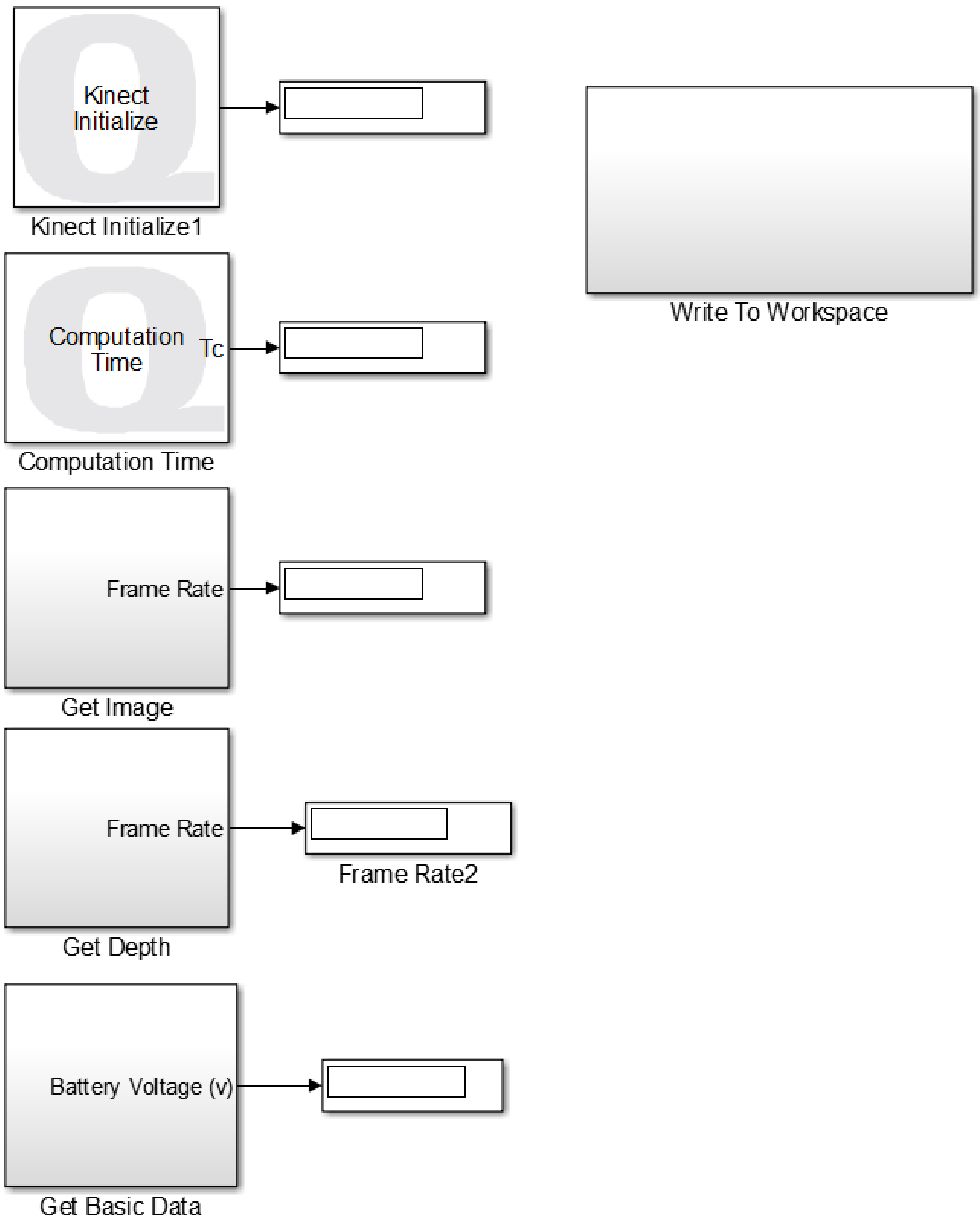


Figure F.9 Data Acquisition Subsystem

To Workspace Subsystem

This subsystem collects the QBot's position information as well as simulation time and saves the data into a file. The file type can be specified in the block parameters.

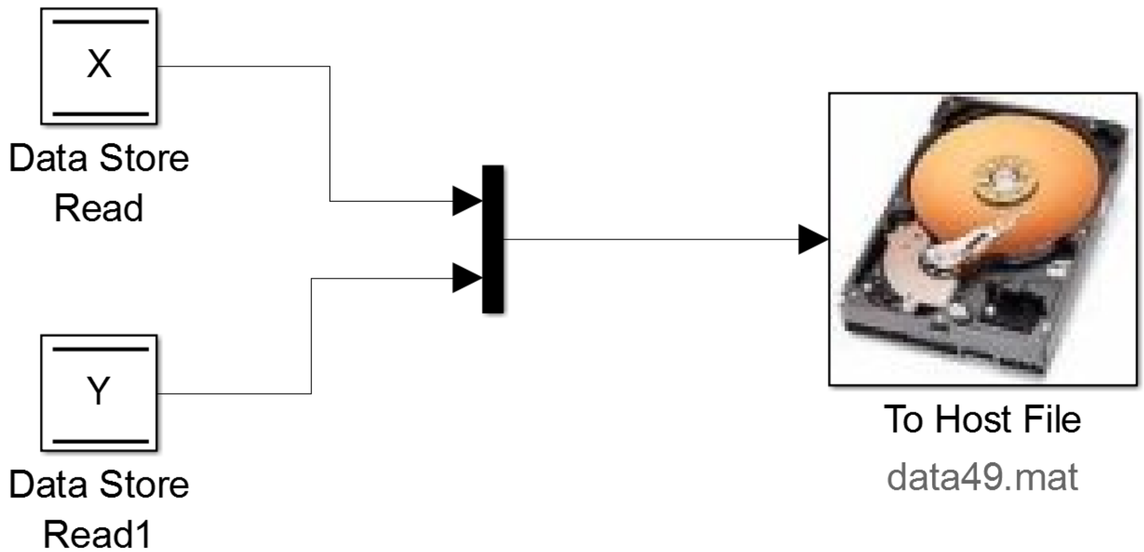


Figure F.10 Write to Workspace Subsystem

Get Image Subsystem

The get image subsystem accesses the Kinect camera and resizes the image data acquired. This data is sent to a global variable called image.

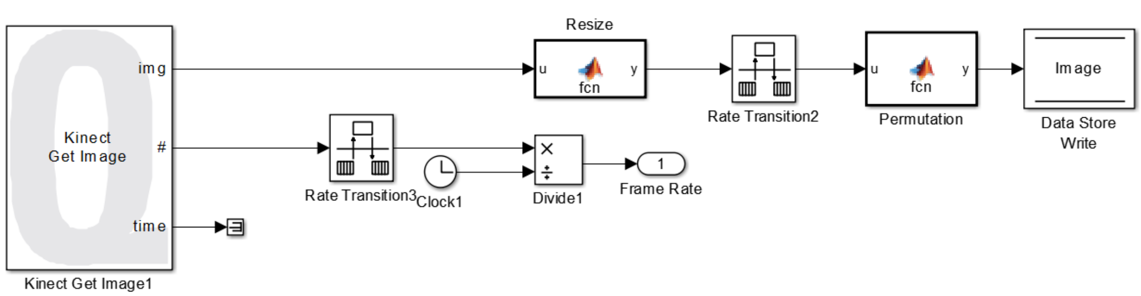


Figure F.11 Get Image Subsystem

Get Depth Subsystem

The get depth subsystem accesses the Kinect depth sensor and sends this data to a global variable called depth.

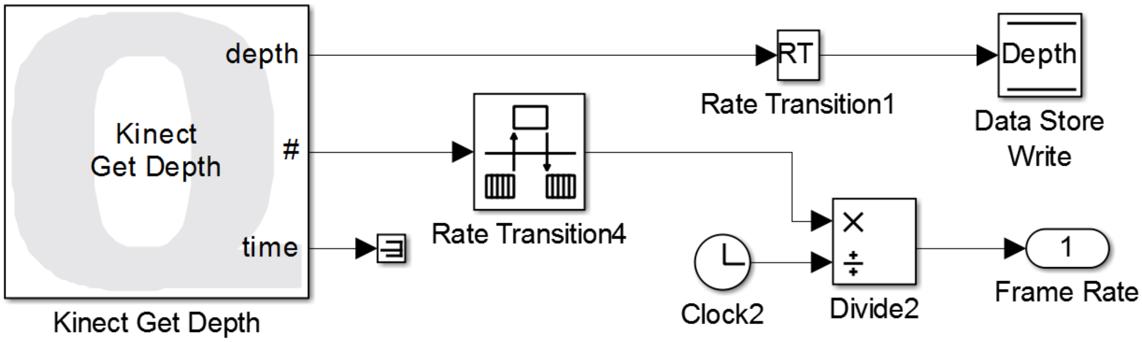


Figure F.12 Get Depth Subsystem

Get Basic Data Subsystem

This subsystem acquires all basic data, such as the QBot's position and orientation information.

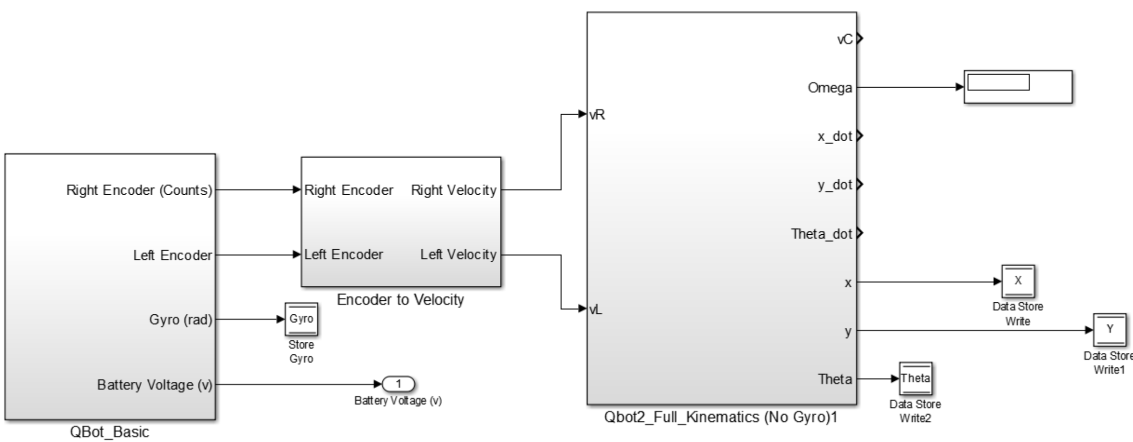


Figure F. 13 Get Basic Data Subsystem

QBot Basic Subsystem

The QBot basic subsystem acquires the right and left wheel encoder data, as well as the gyroscope data.

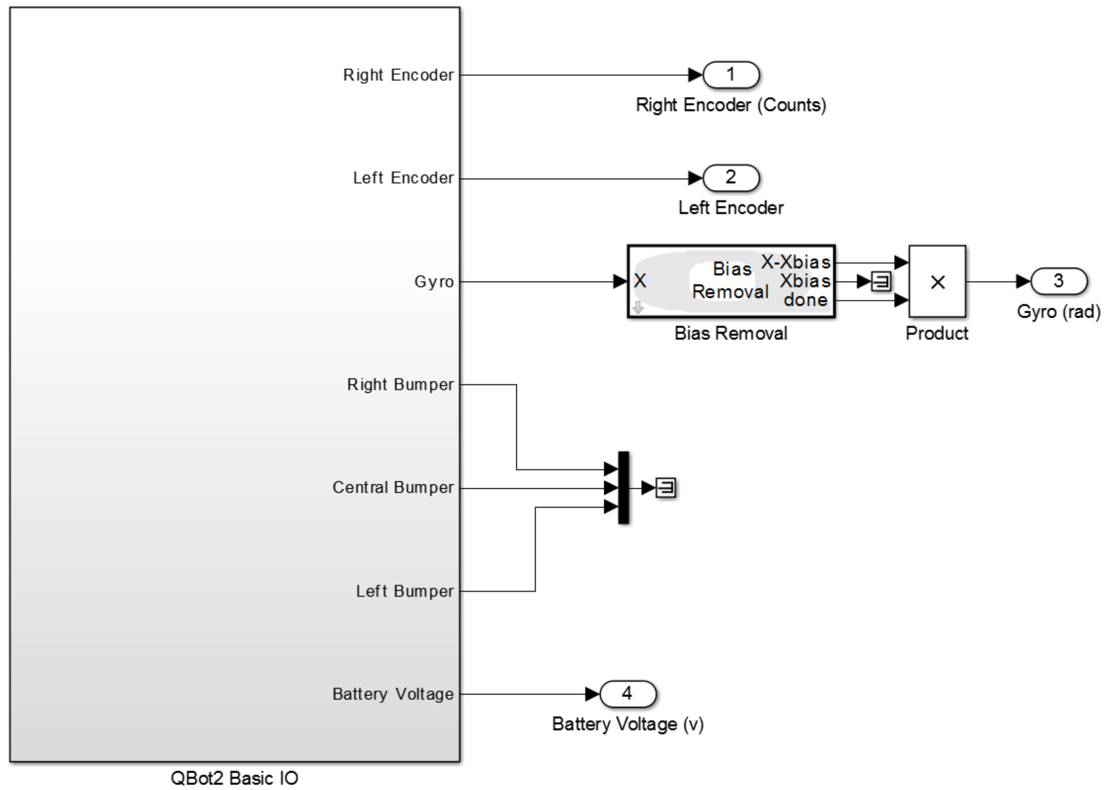


Figure F.14 QBot Basic Subsystem

Basic IO Subsystem

The basic IO subsystem utilizes the HIL Read Simulink block to acquire sensor data.

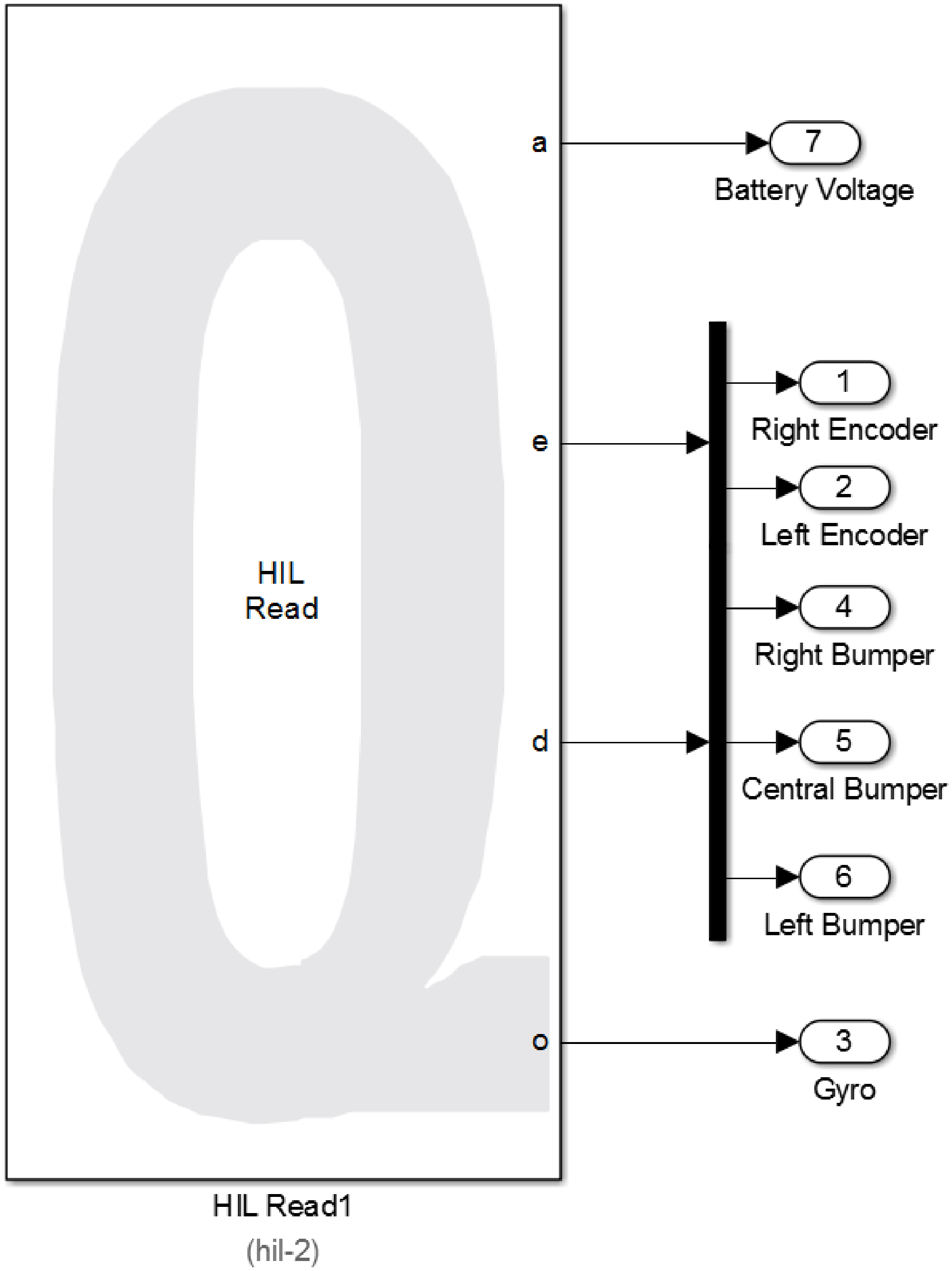


Figure F.15 QBot 2 Basic IO Subsystem

Encoder to Velocity Subsystem

The following blocks are contained in the encoder to velocity subsystem. A figure of the subsystem can be seen below.

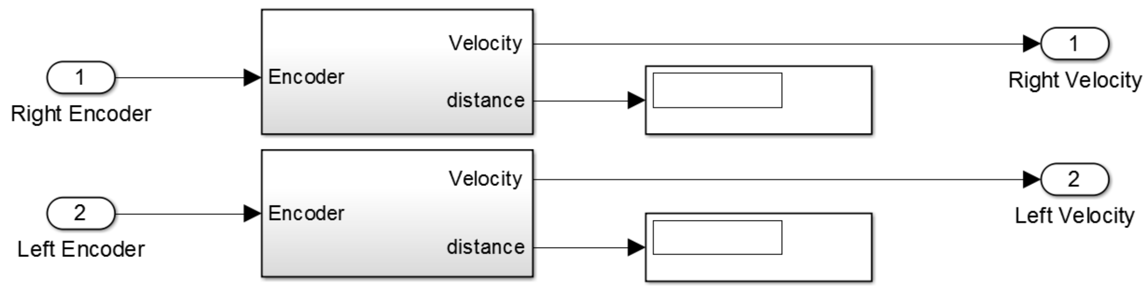


Figure F.16 Encoder to Velocity Subsystem

Encoder Subsystem

The encoder subsystem takes the encoder count of the right or left wheel, and transforms the data into a wheel distance. This distance is sent through a low-pass filter to obtain velocity information.

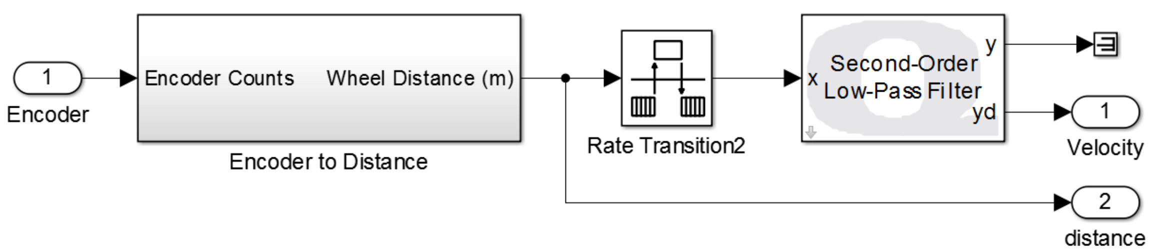


Figure F.17 Encoder Subsystem

Encoder to Distance Subsystem

The encoder to distance subsystem converts the encoder count into distance information.

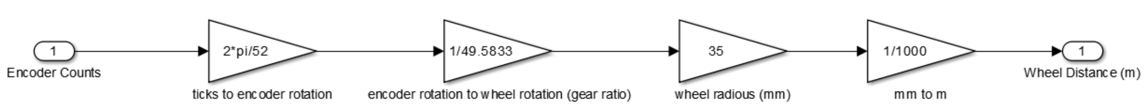


Figure F.18 Encoder to Distance Subsystem

Full Kinematics Subsystem

This subsystem contains all the blocks that calculate the QBot's current position and orientation. The map theta MATLAB function bounds the orientation data in the range of $[-\pi, \pi]$.

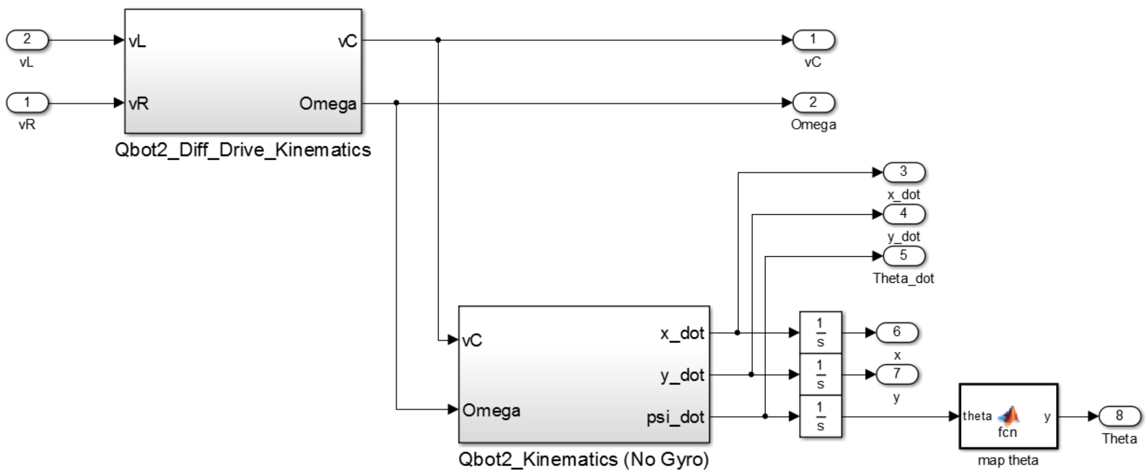


Figure F.19 QBot 2 Full Kinematics Subsystem

Differential Drive Kinematics Subsystem

This subsystem calculates the radial and angular velocity of the QBot 2.

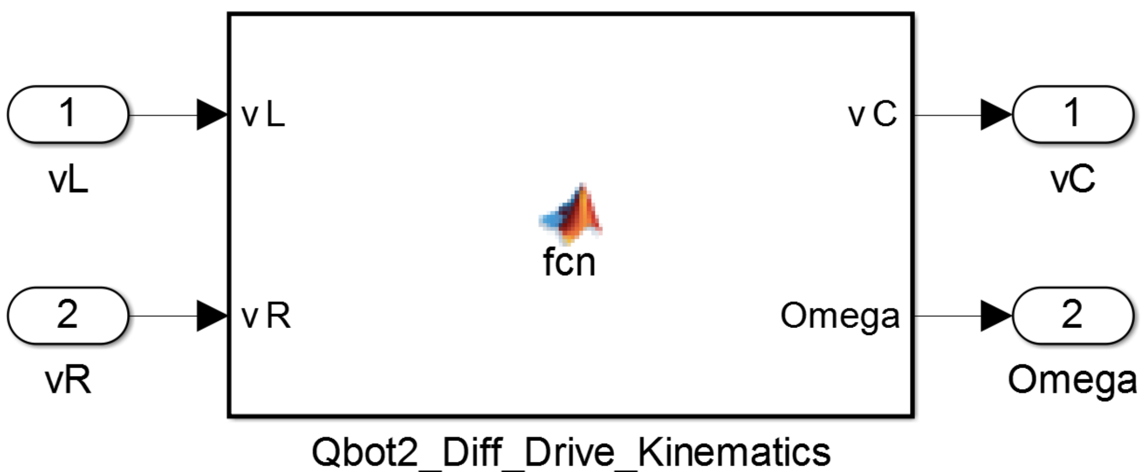


Figure F.20 QBot 2 Differential Drive Kinematics Subsystem

QBot 2 Kinematics Subsystem

This subsystem calculates the current xy-coordinates and orientation of the QBot 2.

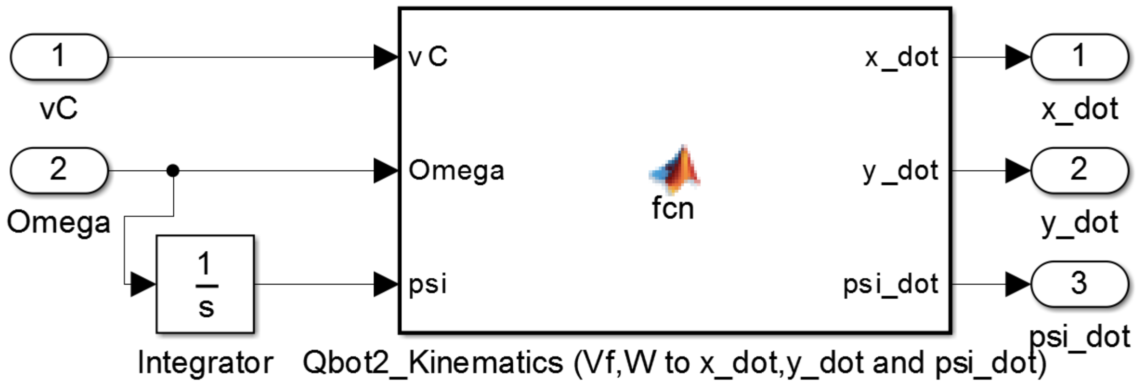


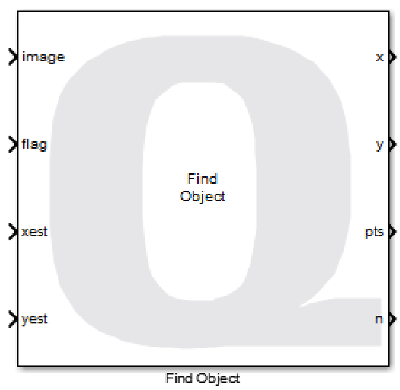
Figure F.21 QBot 2 Kinematics Subsystem

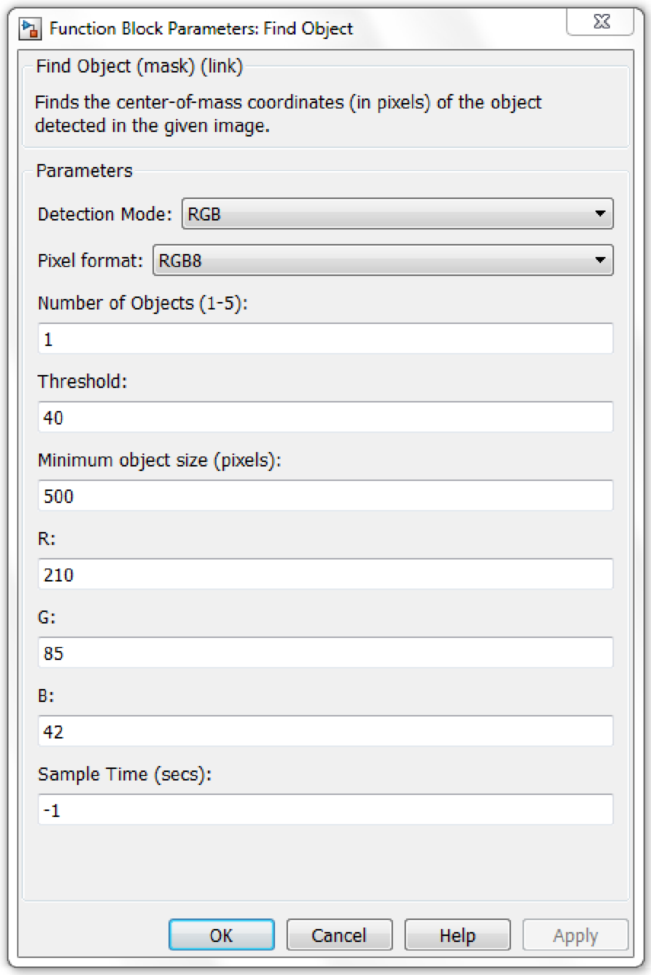
Appendix G

Quanser Simulink Blocks

This section presents information on the inputs and outputs of select Quanser Simulink blocks. A picture of each block and its parameter window is shown, when applicable.

Find Object



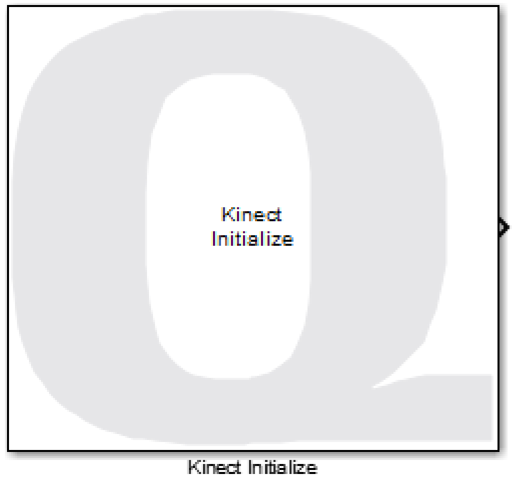


Input	Description
image	Matrix of type uint8 that represents a source image to be searched. Three dimensions for color image (3x640x480), two dimensions for greyscale (640x480).
flag	Enabling signal. 1 enables, others disable
xest	(Optional) Vector of type double that corresponds to the estimate x coordinate for each object
yest	(Optional) Vector of type double that corresponds to the estimate y coordinate for each object

Output	Description
x	Vector of type double containing the x-axis coordinates of the center-of-mass of the objects found matching the searched criteria.
y	Vector of type double containing the y-axis coordinates of the center-of-mass of the objects found matching the searched criteria.
pts	Matrix of type uint8 that is a filtered version of the source image based on the block parameters. Matching pixels are black, and all others are white.
n	Vector of type double that contains the number of pixels of each found object. The length of the vector equals the number of objects specified in the *Number of Objects* parameter.

Parameter	Description
Detection Mode	Determines if block is to search for white, black, or colored objects. The *R*,*G*, and *B* parameters are only enabled when the mode is set to RGB.
Pixel format	The format of the source image. Use RGB8 and BGR8 formats for color images depending on the correct order. Greyscale is used for black and white images.
Number of objects	Scaler integer from one to five defining the number of objects to find. The block will stop looking for images after all desired objects are found.
Threshold	Scaler value for tuning luminosity (greyscale) or RGB values (color). Any pixel within the threshold amount from the defined luminosity/RGB value (inclusive) will constitute as a matching pixel.
Minimum object size	Minimum number of grouped pixels that constitute an object. Groups with less than this amount of pixels are not considered an object. Used to filter out color noise.
R	Desired red value of pixels to search for; ignored if *Detection Mode* is not set to RGB.
G	Desired green value of pixels to search for; ignored if *Detection Mode* is not set to RGB.
B	Desired blue value of pixels to search for; ignored if *Detection Mode* is not set to RGB.
Sample time	Sample time of the block. Zero means continuous, positive indicates discrete block with given sample time, and negative one designates inherited sample time.

Kinect Initialize

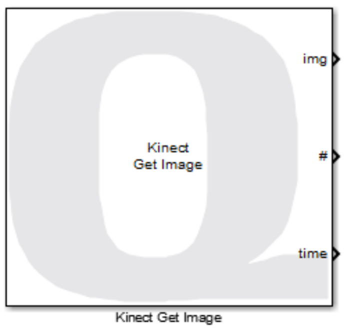


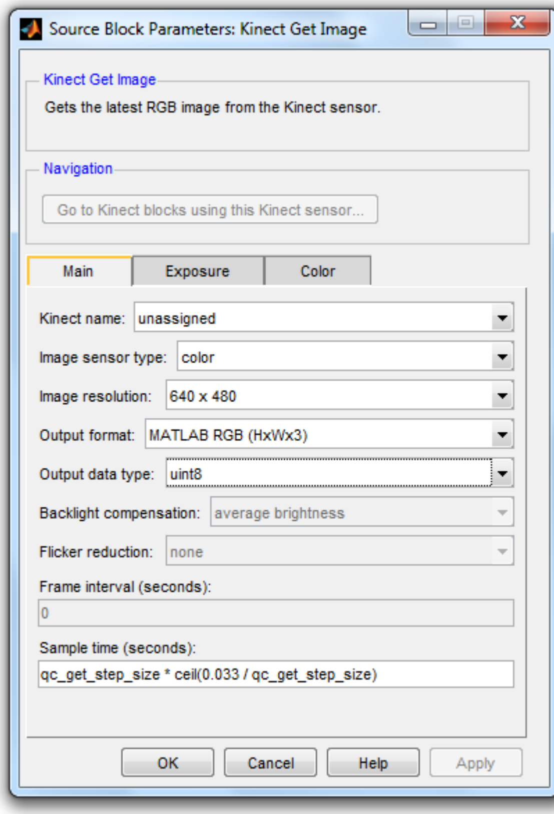
Input	Description
-	N/A

Output	Description
status	Status code of type int32 indicating current status of the Kinect sensor. Status codes can be found on Quanser's website

Parameter	Description
Kinect name	Name to identify this Kinect sensor. Other Kinect blocks will use this name to refer to this sensor.
Kinect type	Selects the type of Kinect sensor. Only used to control which features are enabled.
Kinect identifier	Identifies which Kinect sensor to use if there are multiple Kinect sensors committed to system. If the identifier is an integer, then it designates the number of the Kinect, starting from zero. If the identifier is a string, then the string corresponds to the identifier of the Kinect sensor itself.
Sample time	Sample time of the block. Zero means continuous, positive indicates discrete block with given sample time, and negative one designates inherited sample time. Because this is a source block, it can only inherit a sample time if the block is within a conditionally executed subsystem.
Active during normal simulation	Indicates if this block should communicate during normal simulation. Unless checked, other Host blocks in the model that are associated with this connection will not do anything.

Kinect Get Image





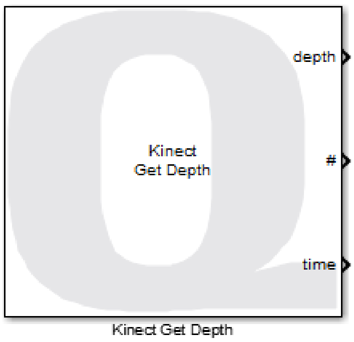
Input	Description
-	N/A

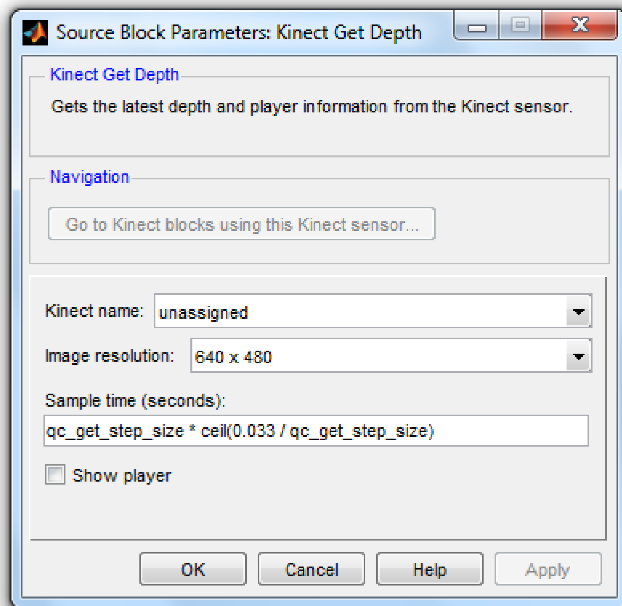
Output	Description
img	Data type depends on *Output data type* parameter. If the *Output format* is set to *MATLAB RGB*, the output is a three-dimensional MxNx3 matrix. It contains the RGB values for each pixel. If the *Output format* is set to *MATLAB Greyscale*, the output is a two-dimensional MxN matrix. In each case, M is the image height, and N is the image width. **Note: You will need to perform a permutation between this block and the Find Object block because the matrix formats do not line up correctly. The Find Object block wants 3xNxM or NxM.**
#	The frame number of the current frame.
time	Timestamp associated with current frame. May not be related to the start time of the model.

Parameter	Description
Kinect name	Name to identify this Kinect sensor. Must be associated with a Kinect Initialize block.
Image sensor type	Type of sensor where the image will be called. Choose between the RGB camera and the infrared camera.

Image resolution	The resolution of image to retrieve. The 640x480 resolution is supported on all targets that support the Kinect, but some targets may not support the other resolutions.
Output format	Format of the output image. *MATLAB RGB* outputs a MxNx3 matrix. *MATLAB Greyscale* outputs MxN matrix.
Output data type	Data type to use for the output image. Integral and floating-point data types are supported.
Backlight compensation	This option allows the Kinect to capture data to be adjusted according to environmental conditions.
Flicker reduction	This option can reduce the flicker caused by the frequency of the power line.
Frame interval (seconds)	Frame rate that images are taken. Zero uses the default frame rate.
Sample time	Sample time of the block. Zero means continuous, positive indicates discrete block with given sample time, and negative one designates inherited sample time. Because this is a source block, it can only inherit a sample time if the block is within a conditionally executed subsystem.

Kinect Get Depth



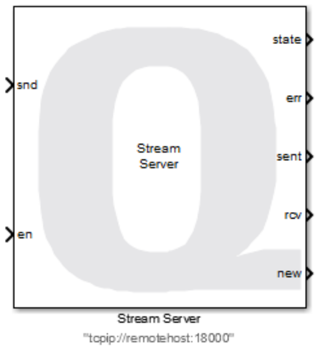


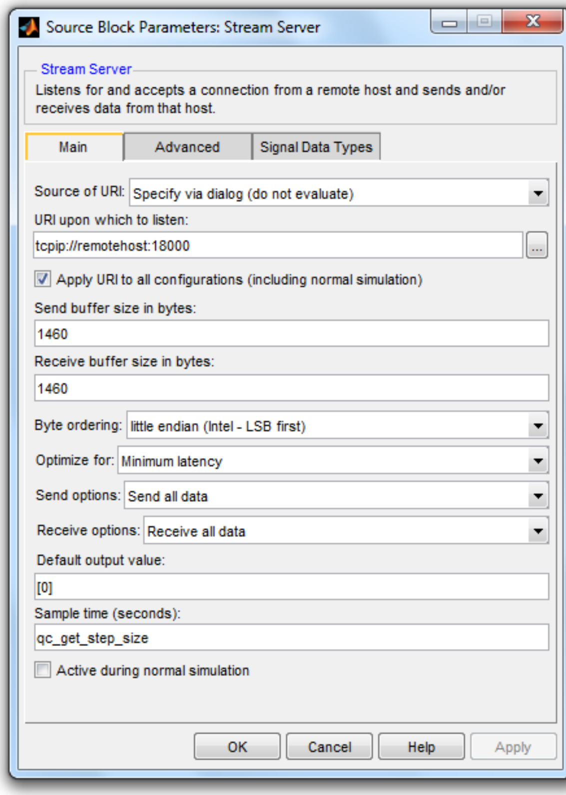
Input	Description
-	N/A

Output	Description
Depth	Two-dimensional matrix of type uint16 representing the depth image. Elements of the matrix represents the distance between the object at that pixel and the camera in millimeters.
#	Frame number of current depth frame.
time	Timestamp associated with current frame. May not be related to the start time of the model.
player	(Optional) Two-dimensional matrix of type uint16 representing the players in the scene. Every element of the matrix has the number of players visible at that pixel. The element is zero when there is no player at that pixel. This output is only available when the *Show player* parameter is checked.

Parameter	Description
Kinect name	Name to identify this Kinect sensor. Must be associated with a Kinect Initialize block.
Image resolution	The resolution of image to retrieve. The 640x480 resolution is supported on all targets that support the Kinect, but some targets may not support the other resolutions.
Sample time	Sample time of the block. Zero means continuous, positive indicates discrete block with given sample time, and negative one designates inherited sample time. Because this is a source block, it can only inherit a sample time if the block is within a conditionally executed subsystem.
Show player	Player output is created when this is checked, but not all targets support this option.

Stream Server



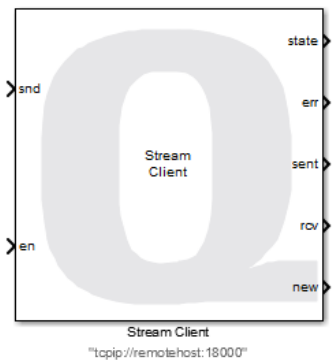


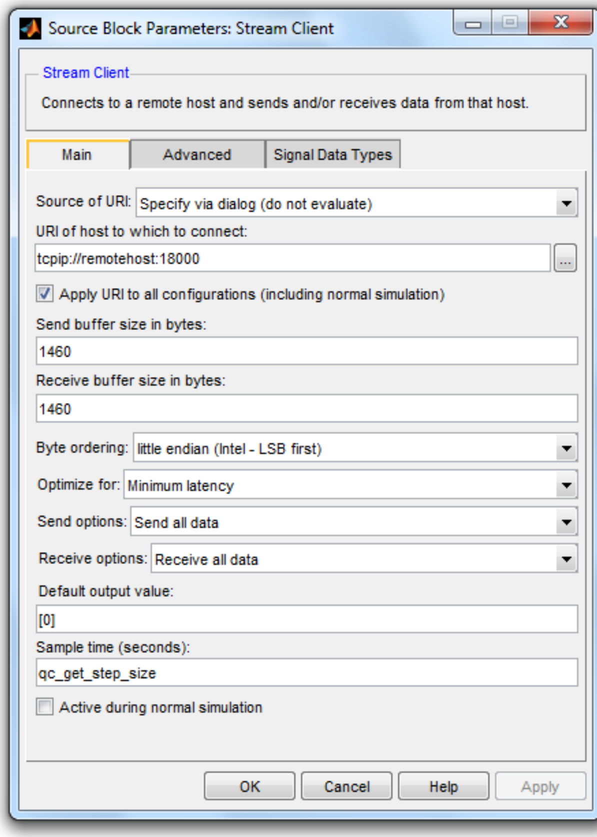
Input	Description
snd	Data to be sent to peer. Data is sent each sampling instant when enabled. Bus and multi-dimensional inputs are supported.
en	Enable signal. When non-zero, data at snd port is transmitted. When zero, data is ignored.
uri	String specifying the URI to listen to and service client connections. Input is only available is the *Source of URI* parameter is set to "External input port."

Output	Description
state	Current status of connection. Zero means stream is not connected. One means stream is waiting to accept connection. Two means stream is connected to remote host. Three means stream is closing the connection to the host.
err	Negative QUARC error code if an error occurs sending and receiving data. Zero if no error occurred. Zero can also mean data could not be sent or received without blocking. Check sent and new outputs to verify data is actually sent or received.
sent	Optional Boolean value representing if the input signal was successfully written to the stream buffer. Non-zero (true) means data was written successfully. Otherwise, it is zero. This output just indicates data was written to the stream buffer, not that it was delivered to the successfully remote peer.
rcv	Optional output contains data received from peer. If no data has been received yet, this value is the *Default output value* parameter. If no new data is received, it will retain the last value received. The *Default output value* parameter determines the dimensions of the output. Bus and multi-dimensional outputs are supported.

new	Optional Boolean output that indicates if the rcv output is new or old data. Non-zero (true) means new data has been received. Zero (false) means data could not be received without blocking.

Stream Client





Input	Description
snd	Data to be sent to peer. Data is sent each sampling instant when enabled. Bus and multi-dimensional inputs are supported.
en	Enable signal. When non-zero, data at snd port is transmitted. When zero, data is ignored.
uri	String specifying the URI to listen to and service client connections. Input is only available is the *Source of URI* parameter is set to "External input port."

Output	Description
state	Current status of connection. Zero means stream is not connected. One means stream is waiting to accept connection. Two means stream is connected to remote host. Three means stream is closing the connection to the host.
err	Negative QUARC error code if an error occurs sending and receiving data. Zero if no error occurred. Zero can also mean data could not be sent or received without blocking. Check sent and new outputs to verify data is actually sent or received.
sent	Optional Boolean value representing if the input signal was successfully written to the stream buffer. Non-zero (true) means data was written successfully. Otherwise, it is zero. This output just indicates data was written to the stream buffer, not that it was delivered to the successfully remote peer.
rcv	Optional output contains data received from peer. If no data has been received yet, this value is the *Default output value* parameter. If no new data is received, it will

	retain the last value received. The *Default output value* parameter determines the dimensions of the output. Bus and multi-dimensional outputs are supported.
new	Optional Boolean output that indicates if the rcv output is new or old data. Non-zero (true) means new data has been received. Zero (false) means data could not be received without blocking.