Y.A.R.E
(A Maze Solving Robot)

Presented by:

Dominic Bergeron
2064116

Georges Daoud
2173098

Bruno Daoust
2029977

Erick Duchesneau
2030910

Mathieu Mallet
1976865

Martin Hurtubise
2032730

Presented to

Dr. Emil Petriu

For CEG 4392

University of Ottawa
Thursday, December 4th, 2003

Table of Contents

FOREWORD ... **4**

THEORY ... **5**

BACKGROUND AND INTRODUCTION ... 5
ADOPTED SOLUTION ... 6
ALGORITHMS USED ... 11
Data acquisition system ... *12*
Data transmission system ... *12*
Data analysis system ... *13*

DESIGN ... **16**

COMPONENT DESCRIPTION ... 16
Data acquisition system ... *16*
Data transmission system ... *19*
Data analysis system ... *23*
Miscellaneous hardware modules ... *25*
CONTROL ALGORITHMS ... 27
Artificial Intelligence Core description ... *30*
IMPLEMENTED SOLUTION ... 31
Implementation Variations ... *31*
Overall System Specifications ... *33*

IMPLEMENTATION RESULTS ... **34**

HARDWARE SIMULATIONS AND ANALYSIS ... 34
WIRELESS COMMUNICATION LINK TESTS ... 37
OVERALL SYSTEM TEST ... 38

PROJECT ORGANIZATION ... **40**

SCHEDULING ... 40
MEMBER PARTICIPATION ... 45

CONCLUSION ... **47**

POSSIBLE IMPROVEMENTS AND FURTHER STUDIES ... **47**

REFERENCES ... **48**

Table of figures

FIGURE 1 : SAMPLE GRAPH 5
FIGURE 2: OVERALL SYSTEM SOLUTION 6
FIGURE 3: SPECIFIC SYSTEM SOLUTION 7
FIGURE 4: FIRST DATA PACKET PROTOTYPE 9
FIGURE 5: SECOND OUTGOING DATA PACKET PROTOTYPE 9
FIGURE 6: SECOND INCOMING DATA PACKET PROTOTYPE 10
FIGURE 7: THIRD OUTGOING DATA PACKET PROTOTYPE 10
FIGURE 8: THIRD INCOMING DATA PACKET PROTOTYPE 11
FIGURE 9: A* ALGORITHM EXAMPLE 14
FIGURE 10: COLLISION SENSOR HARDWARE IMPLEMENTATION 17
FIGURE 11: INFRARED SIGNAL 17
FIGURE 12: INFRARED SENSOR HARDWARE REPRESENTATION 17
FIGURE 13: PULSE WIDTH MODULATION 18
FIGURE 14: SERVO CONTROLLER HARDWARE REPRESENTATION 19
FIGURE 15: TRANSMITTER STATE DIAGRAM 19
FIGURE 16: TRANSMITTER HARDWARE REPRESENTATION 21
FIGURE 17: SERIAL DECODER STATE DIAGRAM 21
FIGURE 18: SERIAL DECODER HARDWARE REPRESENTATION 22
FIGURE 19: RECEIVER STATE DIAGRAM 22
FIGURE 20: CLOCK DIVIDER HARDWARE REPRESENTATION 25
FIGURE 21: SEVEN SEGMENT DECODER HARDWARE REPRESENTATION 26
FIGURE 22: DEBUGGING DECODER HARDWARE REPRESENTATION 26
FIGURE 23: AUTONOMOUS MODE STATE DIAGRAM 27
FIGURE 24: DESCISION PROCESS 28
FIGURE 25: COMPENSATION ALGORITHM 29
FIGURE 26: ARTIFICIAL INTELLIGENCE CORE HARDWARE REPRESENTATION 30
FIGURE 27: STATE DIAGRAM OF IMPLEMENTED AI CORE 32
FIGURE 28: SERVO MOTOR SIMULATION 34
FIGURE 29: RF TRANSMITTER SIMULATION 35
FIGURE 30: SERIAL DECODER SIMULATION 35
FIGURE 31: RF RECEIVER SIMULATION 36
FIGURE 32: TEST MAZE 38
FIGURE 33: SHORTEST PATH 39

Foreword

The purpose of this document is to explain to the reader the process used to design and implement a robot that must perform the following task:

"A mobile robot controlled by the Altera UP -2 board and/or the HC12 microprocessor will have to traverse a maze-based map in order to get from one end to the other of the maze. The main objective is to utilize remotely-embedded knowledge to solve the task at hand. IR sensors should be used to locate the openings within the maze, while a wireless communications scheme should be utilized to communicate with the knowledge base residing on a remote PC. A lookup table may be initially used, but as the task gets more complicated, a knowledge base should be utilized. The autonomous mobile robotic platform will acquire and interpret the data, in accordance with the remotely-embedded knowledge, and can update the knowledge base with newly learnt factoids. Contacts sensors should be used for handling accidental collision situations."[1]

Six engineers were assigned to solve this problem. These engineers are: Dominic Bergeron, Georges Daoud, Bruno Daoust, Erick Duchesneau, Mathieu Mallet and Martin Hurtubise. This document will explain their design and how they implemented a robot capable of solving this task.

The result of their efforts is Y.A.R.E. Yare, which the oxford dictionary defines as, moving lightly and easily, is a recursive acronym that stands for YARE Automaton for Revealing Exits. The acronym will be used in this report to designate the mobile robot.

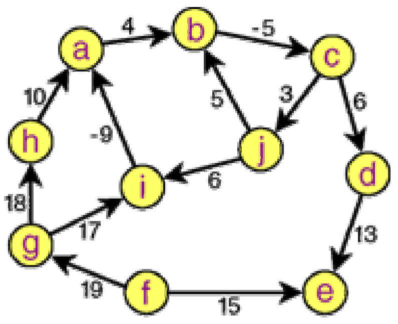
Theory

Background and Introduction

The problem the team had to solve was theoretically vast. As it has been explored by many researchers in the past, the task was to find the least costly option among a choice of many. In other words, a task needed to be performed while minimizing a certain parameter. This is a common topic in graph theory, which has a wide variety of applications such as packet routing.

In graph theory, each point in a graph represents a certain step (for this particular application it is a spatial location) between the starting point and the desired point. There is a cost associated with moving between points. This is demonstrated in the following sample graph (obtained from [3]).

Figure 1 : Sample Graph



As can be observed from the graph, the cost to travel between point A and point B would be four. This represents the distance required to travel between the two points. The objective of graph theory is to develop mathematical methods capable of minimizing the cost of traveling between two designated points. Detailed discussions of such methods can be found in referenced works by Stallings and Rensselaer [2][3].

Applying this theory to the application was difficult. The application involved putting a mobile robot in an unknown maze environment. The robot had to find its way out of the maze while communicating the maze's characteristics to a base station. The base station then analyzed the maze data and computed the optimal path that the robot should have traveled. If the robot was put into the same maze again it would then solve the maze using the optimized path. Thus, the goal was to have the robot start at the entrance of a maze and find its exit while minimizing the distance traveled.

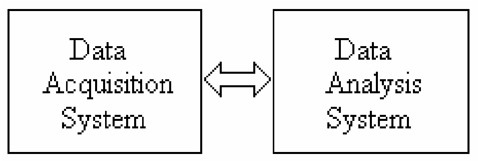
The importance of solving such a problem is obvious from the numerous examples that can be found in daily situations. For example, if it is crucial to traverse an environment that is hazardous to humans, it may desirable to minimize exposure to this environment. In order to do so, the shortest path through the environment would need to be determined. Thus, Y.A.R.E. could be sent to find the shortest path, and ensure that it is safe to use. An example of such a situation would be a mine shaft that is known to be unstable. If it is necessary to go into the mine (to rescue trapped miners for example), Y.A.R.E could search the tunnels for the workers instead of risking additional human casualties. The rescuers could then use the shortest path from the entrance of the mine to the trapped miners, found by Y.A.R.E., in order to rescue them.

Adopted Solution

This section of the report will attempt to explain to the reader how the selected solution resolved the problem at hand.

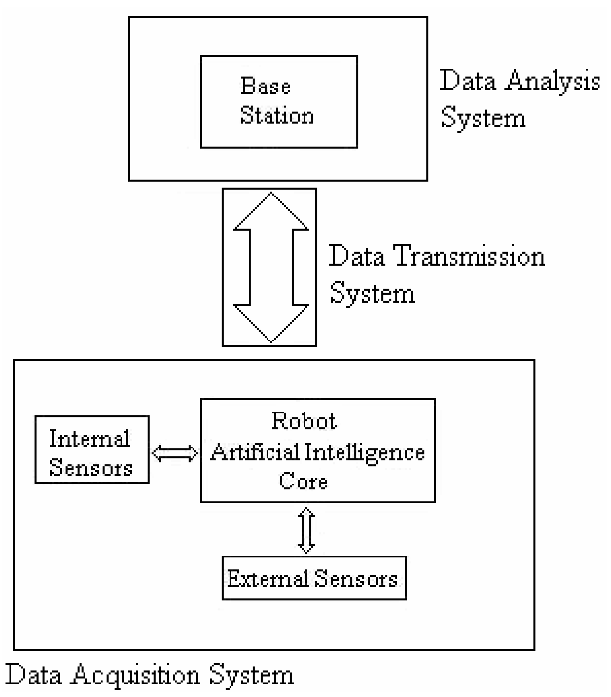
The overall problem can easily be separated into two major systems: the data acquisition system and the data analysis system. The data acquisition system was required to obtain information about the maze. This is where the different nodes and vertices of the graph, as well as their costs, are obtained. The calculation of the shortest path would be performed by the data analysis system.

Figure 2: Overall System Solution



These two major systems can be broken down into several sub-systems. In this particular case, the data acquisition system represents the mobile robot and the data analysis system represents the base station. However, the robot and the base station do not communicate directly. This introduces a new system: the data communication system. The following figure is a graphical representation of each major system, divided for our particular case.

Figure 3: Specific System Solution



As mentioned earlier, the primary task of the data acquisition system is to gather information from the robot's environment. This is accomplished through a series of external sensors.

The first type of external sensors is used to detect accidental collisions. If the robot comes in contact with an object these sensors are triggered and a signal is sent to the artificial intelligence core. The core stops the robot and tells it to wait for instructions from the base station. This procedure will be explained in greater detail in another section of this report.

Infrared sensors are the second type of sensor that are used on the mobile robot. These sensors are located in carefully selected areas on the front, right and left of the robot. The sensors emit a beam of infrared light. If the light hits an obstacle, it is reflected and the sensor detects the returning beams' angle. From this angle, it is possible to obtain a distance code from the sensors. After decoding, Y.A.R.E. is capable of calculating the distance that separates it from walls and other obstacles. These sensors are linked directly to the artificial intelligence core in order to allow the robot to make decisions based on its environment.

In order to accurately obtain information about its surroundings the mobile robot also uses a series of internal measurements. These measurements include a timer and the speed of its DC motors. These two readings are needed to compute the distance traveled by the mobile robot. Although the data is transmitted to the artificial intelligence core it is not used by the robot itself, but rather by the base station's software.

The final element of the data acquisition system is the robot's artificial intelligence core. This is the most complex hardware component that was designed. All of its functionalities are described in detail in another section of the report, and to avoid redundancy, they will not be repeated here. The reader should simply note that the artificial intelligence core decides how the robot will respond to its environment. For example, it is the core's responsibility to keep the robot at an acceptable distance from the walls.

Although Y.A.R.E. is capable of reacting to its environment using the artificial intelligence core, it is not capable of analyzing the maze itself. To do this, it relies on the base station's software. In order to communicate with the base station, Y.A.R.E uses a wireless communication link. The wireless link is, in fact, the data transmission system shown in figure 3.

The wireless link consists of two pairs of transmitters and receivers. One set is used to transfer maze information from the robot to the base station, and the other is used to transfer commands from the base station to the robot. To avoid interference between the two pairs of transmitters and receivers, they each operate on different channels. One set uses channel two, which corresponds to 907.87 MHz, while the other set uses channel three, which corresponds to 909.37 MHz. Both sets of transmitters/receivers use frequency shift keying to transmit the data.

The data blocks that are transmitted and received are both of constant size. However, incoming and outgoing data blocks (from the robot's point of view) are different sizes. This is because communication from the robot to the base station requires more data to be sent, than is required for communications from the base station to the robot.

By analyzing the information that had to be sent, data blocks were determined to be 96 bit wide for robot to base transfer and 16 bit in length for base to robot transfer. For both types of data blocks (96-bit block and 16-bit block), an 8-bit checksum is used to validate the data.

Originally, to minimize errors during transfer, the data block was sent three times (Data A, B and C are identical) along with the checksum of that block as suggested by "Application Note AN-00160" (see [4] for further details). This resulted in the first packet prototype, as shown below:

Figure 4: First Data Packet Prototype

Start1	**Start2**	**Data A**	**Data B**	**Data C**	**Checksum**
(8 bits)	(8 bits)	(96 bits)	(96 bits)	(96 bits)	(8 bits)

Wireless outgoing packet (robot's point of view) – prototype 1

Start1	**Start2**	**Data A**	**Data B**	**Data C**	**Checksum**
(8 bits)	(8 bits)	(16 bits)	(16 bits)	(16 bits)	(8 bits)

Wireless incoming packet (robot's point of view) – prototype 1

According the application note, the two start bytes FF (Hex) and 00 (Hex) are a good choice since noise has a very low probability of manifesting itself as a series of eight ones or zeros.

Unfortunately, upon further analysis, it was realized that if the checksum was damaged, our data would automatically be rejected since it would not be correctly verifiable. Our packet prototype was changed to send the checksum three times. The second prototype is shown below:

Figure 5: Second Outgoing Data Packet Prototype

Start	**Data A**	**Chksm A**	**Data B**	**Chksm B**	**Data C**	**Chksm C**
(16 bits)	(96 bits)	(8 bits)	(96 bits)	(8 bits)	(96 bits)	(8 bits)

Wireless outgoing packet – prototype 2

After modifying the data packets for a second time, it was realized that it was crucial for the mobile robot to receive its commands from the base station. To minimize the chances of getting incorrect results, the data was sent five times instead of three. The new incoming wireless packet is shown below:

Figure 6: Second Incoming Data Packet Prototype

Start	**Data A**	**Chksm A**	**Data B**	**Chksm B**	**Data C**	**Chksm C**	**Data D**	**Chksm D**	**Data E**	**Chksm E**
(16 bits)	(16 bits)	(8 bits)	(16 bits)	(8 bits)	(16 bits)	(8 bits)	(16 bits)	(8 bits)	(16 bits)	(8 bits)

Wireless incoming packet – prototype 2

Since the base station's receiver, transfers the data to the computer through the serial port, the packet needed to be encapsulated within a serial packet (in order to ensure compatibility with the serial port). The new packet structure is shown below:

Figure 7: Third Outgoing Data Packet Prototype

Info bit	**Data bit 6**	**Data bit 5**	**Data bit 4**	**Data bit 3**	**Data bit 2**	**Data bit 1**	**Data bit 0**

Representation of our 8-bit serial packet – prototype 3
The information bit is used to specify if the packet contains start byte or data

Start	**Data A**	**Chksm A**	**Data B**	**Chksm B**	**Data C**	**Chksm C**
(7 bits)	(96 bits)	(8 bits)	(96 bits)	(8 bits)	(96 bits)	(8 bits)

Wireless outgoing packet – prototype 3
Global representation of our information before encapsulation
Length: 319 bits + 3 bits of padding = 322 bits (multiple of 7)

As it can be seen in the preceding figure, our wireless outgoing packet was encapsulated within an 8-bit serial packet. The start byte has been changed to "0000001" (7-bit). Additionally, the most significant bit of the serial packet is used to tell if the packet contains a start byte (0) or actual data (1). To make our packet a multiple of 7, a padding bit is sent after each checksum. This makes our packet 322 bits wide which permits it to be encapsulated within 46 serial packets.

Figure 8: Third Incoming Data Packet Prototype

Start	**Data A**	**Chksm A**	**Data B**	**Chksm B**	**Data C**	**Chksm C**	**Data D**	**Chksm D**	**Data E**	**Chksm E**
(7 bits)	(16 bits)	(5 bits)	(16 bits)	(5 bits)	(16 bits)	(5 bits)	(16 bits)	(5 bits)	(16 bits)	(5 bits)

Wireless incoming packet – prototype 3
Representation of our packet before encapsulation
Length: 112 bits (multiple of 7)

The final version (as shown in the above figure) of the incoming packet was also encapsulated within an 8-bit serial packet. The start byte was changed to "0000001" (7-bit). The most significant bit of the serial packet was used to tell if the packet contains a start byte (0) or actual data (1). To make our packet a multiple of 7, a padding bit was sent after each checksum. Since the data block is only 16 bits wide, the checksum was reduced to 5 bits. This makes the packet 112 bits wide encapsulated within 16 serial packets.

The final system that was needed to solve the task was the data analysis system. This system is implemented by the base station. The base station receives the data sent by the robot and computes the shortest path using specially designed software. For a complete explanation of this software's functionality the reader is invited to consult the user manual attached with this report.

Algorithms Used

This section of the report will discuss the various algorithms that were used to implement our final solution. The algorithms will be divided between the three major systems.

Data acquisition system

The algorithms implemented in the data acquisition system affect the robot's behavior. Such algorithms are called control algorithms and will be discussed in a subsequent section of this report.

Data transmission system

The only algorithms used in the data transmission system are the majority algorithm and checksum. The majority algorithm is computed only by the receiver while the checksum is computed by both the transmitter and receiver.

<u>Majority algorithm</u>

The majority algorithm is quite simple. It examines each data block (they are supposed to be identical) and compares each bit. If the bits compared are not identical, the one that is repeated most often is kept, and a new data block resulting from the majority of each data block is constructed. This process greatly reduces errors. However, if the majority of bits are in error, the resulting data will be bad.
Data and checksum received by the base station are to be reconstructed as follows:

new_bit → (A&B) | (A&C) | (B&C) where | represents a logical OR operation and & represents a logical AND operation

Data and checksum received by the robot are to be reconstructed as follows:

new_bit → (A&B&C) | (A&B&D) | (A&B&E) | (A&C&D) | (A&C&E) | (A&D&E) | (B&C&D) | (B&C&E) | B&D&E | (C&D&E)

<u>Checksum</u>

The checksum can be resumed as the sum of several groups of bits. It was chosen to validate the data since it gives considerably good results and a low complexity (compared to the CRC). Although it is not as accurate as a cyclic redundancy check, it is sufficient for this application because the probability of a bad packet passing the checksum is quite minute.

For our outgoing packet (from robot's point of view), an 8-bit checksum is computed. The 96-bit data is thus separated into chunks of 8-bits denoted as data_part(i) and added together to give sum. The 8 least significant bits of the sum are kept and become the checksum.

$$checksum = \left[\sum_{i=1}^{12} Data\_part(i)\right](7 \quad downto \quad 0)$$

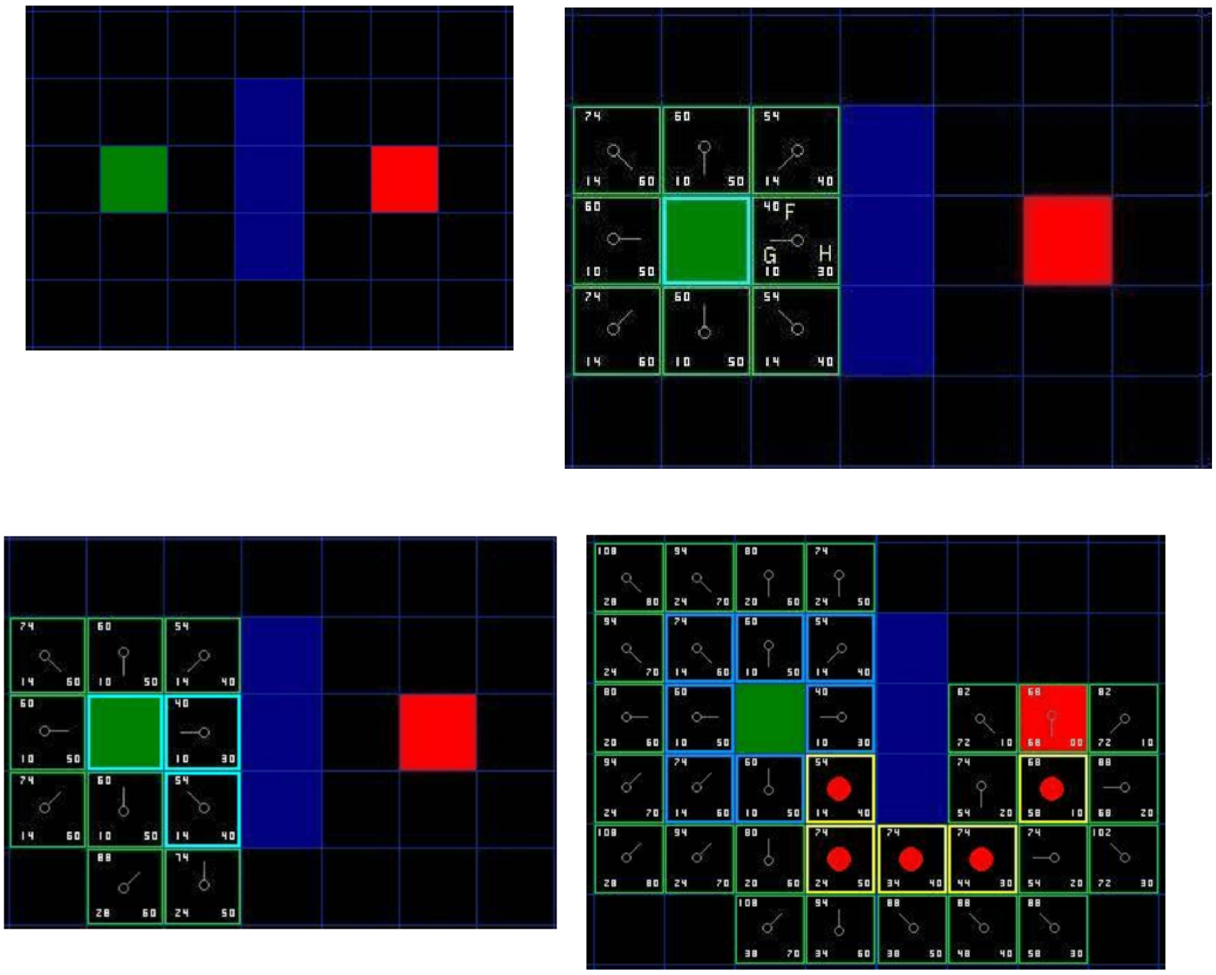
For the incoming packet (from robot's point of view), a 5-bit checksum is computed. The 16-bit data is separated into 4 chunks of 5-bits denoted as data_part(i) and added together to give sum. The 5 least significant bits of the sum are kept and become the checksum. Since the chunks are created from least to most significant bits and the data is not a multiple of 5 bits, the most significant data bit will become the least significant bit of one of a chunk and four 0s will be added (if necessary) to that chunk.

Data analysis system

The base station's software contains an implementation of the A* algorithm. The A* algorithm was developed in 1968 to combine the advantages of Djikstra's algorithm (it always finds an optimal path) and the advantages of the BFS algorithm (its speed). A* takes into consideration the distance traveled, as well as an estimate the distance left to reach the goal. This means that A* is computationally quick and it *can* (depending on the implementation) guarantee a shortest path. The A* algorithm is also very flexible since the values of G(distance traveled so far) and H(estimate of the distance) can be varied in order to modify the behavior. Since the A* relies on F to know which square to choose (and F = G + H), if G is set to 0 then A* becomes the BFS algorithm. If H is set to 0, then A* turns into Djikstra's algorithm. In other words, the effects of G (on F) can be attenuated in order to make the algorithm faster. Or the effects of H (on F) can be attenuated in order to make the algorithm more accurate. In addition, if H is lower than the cost to get to the goal, the shortest path will be found. This flexibility makes the A* algorithm very powerful.

In order to better explain the algorithm, an example (represented by figure 9) will be used:

Figure 9: A* Algorithm example



In order to get from the green square to the red square (see upper left section of Figure 9) as quickly as possible without hitting any obstacles (ie. the blue wall), the following procedure would be used.

To implement the algorithm, two lists are required, an OpenList and a ClosedList. First, the starting point of our map (ie. the green square) is added to the ClosedList. Then, the adjacent squares are examined. In this case, there are 8 squares that are reachable, as can be seen in the upper right section of Figure 9. For each square, the starting square is set as the new square's parent. The new square is added to the OpenList. This can be seen in the upper right section of Figure 9, as a pointer that points to the parent of that square (in this case, the starting square). Afterwards, the values of F, G, and H are calculated for each square as described below. Also, when adding squares that are reachable from the current square, if a certain square is already on the OpenList it's G value must be checked and compared to the G value that would be obtained if it were reached from the current square. If the current square's G value is smaller than the original value, the current square is set as the observed square's parent.

F = G + H
where G is the cost attributed to getting to a specific square from the beginning square, following the path that was generated earlier.
H is the estimated cost of getting from a specific square to the final destination. It is an estimate since we ignore obstacles.

By looking once again at the upper right section of Figure 9, it can be seen that the values of F, G, and H have been calculated for all of the squares surrounding the starting square. Note that a horizontal or vertical displacement has been chosen to represent a movement of 10, while a diagonal displacement has been chosen to represent a movement of 14 (ie. $(10^2 + 10^2)^{0.5}$ is about equal to 14), since this simplifies the calculations. Furthermore, the method used in this case, to find an estimate of the distance left to travel, assumes that we can only perform horizontal or vertical displacements (this is called the Manhattan method). Once all of these values have been calculated, the square in the OpenList that has the minimum F value is removed from the OpenList and added to the ClosedList. If two squares have the same F value, either can be chosen as long as the selection method stays consistent.

To obtain the optimal path, the algorithm is repeated (the result after two iterations is shown in the lower left section of figure 9) until the destination point is added to the ClosedList. Once the destination point has been added to the ClosedList, the optimal path is found by backtracking. The destination point's parent is added to the path (the destination point) and then get this point's parent, and then the parent's parent, and then the parent's parent's parent, etc. until the starting point is added. This can be observed in the lower right section of Figure 9, where the red dots clearly indicate the optimal path found by backtracking. Also, the reader should notice that the square positioned two squares below the starting square's parent has been changed (ie. it used to point to the square to the top and to the right of it). This happened because it is shorter to get to this point from the square just above it (ie. a G value of 20 is smaller than a G value of 28) than from the square above it and to the right.

Summary of the A* Method:

```
1 - Add the starting square to the closed list. This will be our current square

2 - LOOP UNTIL (the destination point is found or we can no longer find a path)
        - Find all of the squares that are reachable from the current square and that are not
        already on the ClosedList. Add all of them to the OpenList (while also setting the
        current square as the parent). At the same time calculate the F, G, and H values.
        If a point found is already on the OpenList, then check to see if it's G value would
        be smaller if it was calculated from the current square. If that is the case, the
        observed square's parent is changed (ie. the current square becomes it's parent)
        and then recalculate the F, G, and H values.
        - Find, from the OpenList, the square that has the minimum F value, remove it
        from the OpenList and add it to the ClosedList. This becomes the new current
        square.

   END LOOP

3 - Start at the last square (ie. the destination) and backtrack by following the list of
parents in order to find the optimal path.
```

Design

Component Description

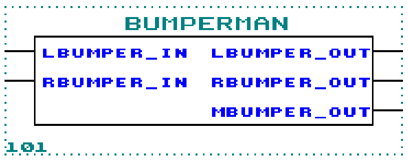
This section of the report will describe each of the hardware and software components that were necessary to implement the chosen solution. The components' functionality will also be described. The components will be separated into the three major systems identified in the theoretical part of the report.

Data acquisition system

BumperMan

BumperMan is a simple VHDL module created to handle bumper events on the robot. It converts bumper hits (LOW signals) to bumper events (HIGH signals). It also creates a third signal, middle bumper, which is active when both left and right bumpers are triggered. This is the hardware representation of the BumperMan module (VHDL):

Figure 10: Collision Sensor Hardware Implementation

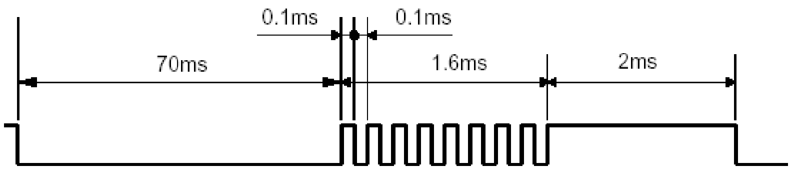


The complete code for this module can be found in Appendix A.

IR sensor manager

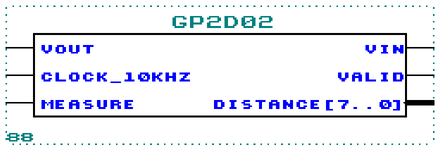
Existing code was used to control the IR sensors. The original code was taken from [5]. Instead of detailing the implementation of the module, its use will be explained. The input Vout is connected to the IR sensor output. The output Vin is connected to the IR sensor input. A clock of 10KHz must be fed to the module. Once started, the module will send this modulated signal to the IR sensor:

Figure 11: Infrared Signal



Using this signal, the sensor will return one bit of data each time the input signal returns to 0. The data will be valid for 0.2 ms, until the next bit starts. Once the 8 bits have been received by the GP2D02 module, they will be available on the Distance signal. The Valid signal will also be set to high to indicate that data can now be read. It should be noted that the module will only send the input signal if the Measure signal is set to high and if a measure is not already being taken. In order to take measures continuously, one simply has to set the Measure pin to High. This is the hardware representation of the IR Sensor module.

Figure 12: Infrared Sensor Hardware Representation

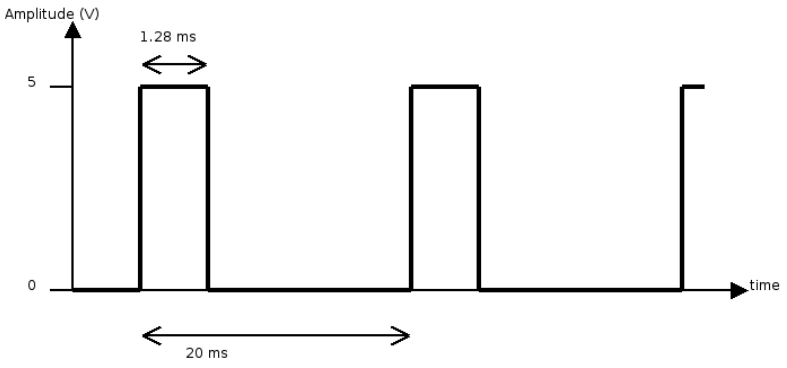


Since there are three IR sensors mounted on the robot, three of these modules have been used.
The complete code for this module can be found in Appendix A.

Servo controls

The servo control module is another simple VHDL module that controls the speed of the motors by sending them a pulse-width-modulated signal. Depending on the height of the length of the duty-cycle, the server will either move forward or backwards.

Figure 13: Pulse Width Modulation



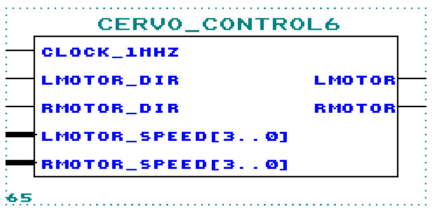
The previous diagram displays a sample of the PWM signal used to control the servos. The period of the signals needs to be precisely 20ms. The duty-cycle, or high-time, needs to be approximately 1.28 ms in order to have the motor stop. A shorter duty-cycle, down to a minimum of 0.8ms, means that the robot will turn clockwise. A longer duty-cycle, up to a maximum of 2.2ms, means that the robot will turn counter-clockwise. Since the motors are mounted opposite to each other on the robot, in order to move forward one motor turns clockwise and the other motor turns counter-clockwise.

The Servo Control module abstracts the complexity of controlling the motors by simply taking the speed and direction of each wheel and converting those to pulse-width-modulated signals that are sent to the two servos. One bit per motor is used to signal the desired direction of the wheel. Four bits are used to specify the desired speed of the wheels. Though only four different speeds have been implemented in the module, a few bits of margin were put in case more speeds were required.

The Servo Control module is implemented using simple conditional logic and a counter. Every clock cycle, the counter is incremented by one. When the counter is below 17800, which corresponds to 17.8ms on a 1MHz clock, a low signal is sent to the servos. Once that limit is passed, the program checks the desired direction and speed of the wheels. Up until the counter reaches a predefined valued, chosen to represent a specific speed, a high signal is sent to the servos. After that value is passed, a low signal is sent to the servos. Once the counter reaches 20000, which corresponds to a period of 20ms, the counter is reset. All that really matters is that the speeds are matched: when the robot goes forward at, say, the speed 'fast', both motors must turn at the 'fast' speed. This means that the left motor will turn counter-clockwise at 'fast' speed and that the right motor will turn clockwise at the same speed.

The hardware representation of the Servo Control module (VHDL) is:

Figure 14: Servo Controller Hardware Representation

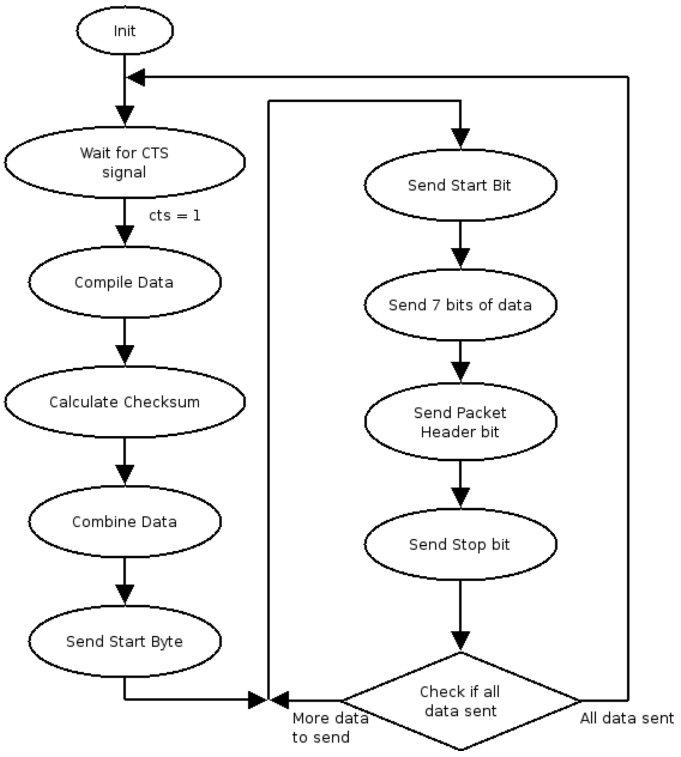


Data transmission system

rf_tx

This module simulates the hardware implementation of the transmitter. Here is a state diagram followed by a summary of the module's functionality.

Figure 15: Transmitter State Diagram



The module takes a series of useful inputs along with two different clocks. Most of the module operates at serial compatible bit rate (19200 kbps) but the timer process operates faster. The output of our modules is simply data output.

The cycle begins by waiting for the CTS command to be issued. That command is sent automatically by the transmitter hardware when the frequency synthesizer has locked on the proper channel frequency. Once this is done, the data is compiled from the various interesting inputs and from the internal timer. That data is compiled in a block of 96 bits.

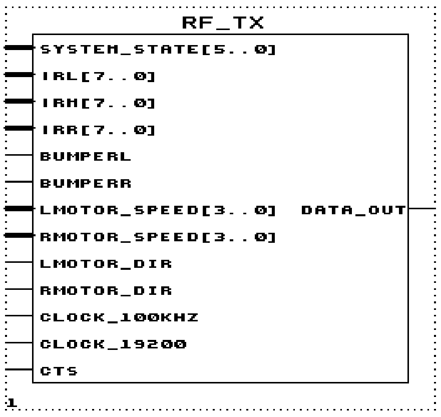
The 8-bit checksum is then calculated for that data block. A new block denoted as “combined data” is then compiled and includes the data block, the checksum and a padding bit.

Next, the start byte is sent and the serialization of data begins. Seven data bits are stored in the serial packet and the most significant bit becomes the header bit (denoted earlier as information bit). That header bit is used to identify if the serial packet contains a start byte or data. The stop bits are then sent (a series of 1s). If some data remains to be sent, the serialization process restarts and this is done until all data has been sent.

When everything has been sent, the transmitter goes back to its initial state (waiting for CTS).

The hardware representation of the module is shown below (VHDL symbol):

Figure 16: Transmitter Hardware Representation

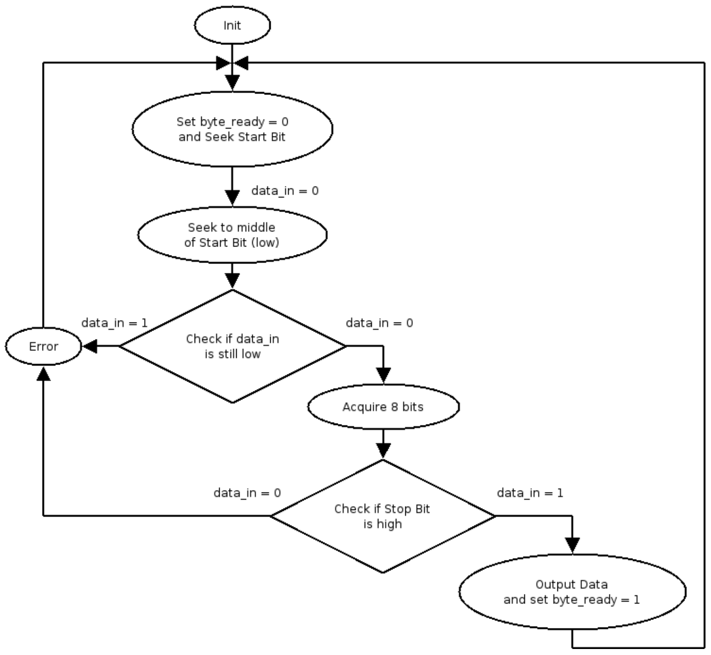


The complete code for this module can be found in Appendix A.

rf_serialdecoder

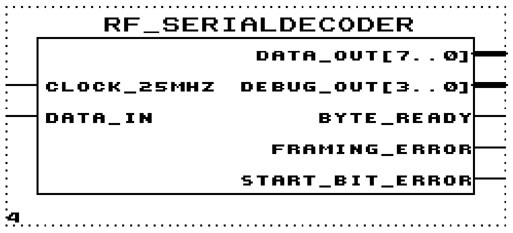
This module converts each serial packet into parallel format. Here is a state diagram followed by a summary of the module's functionality.

Figure 17: Serial Decoder State Diagram



The module takes a series of two inputs. One input is a clock and the other is the incoming serial data in form of serial packets. This module waits for an incoming signal and converts each serial packet into parallel format. It then indicates to the RF receiver module when a byte is ready. It also indicates if there is a framing or start bit error. The debug_out output is used to monitor each state of the module.
The hardware representation of that module is shown below (VHDL symbol):

Figure 18: Serial Decoder Hardware Representation

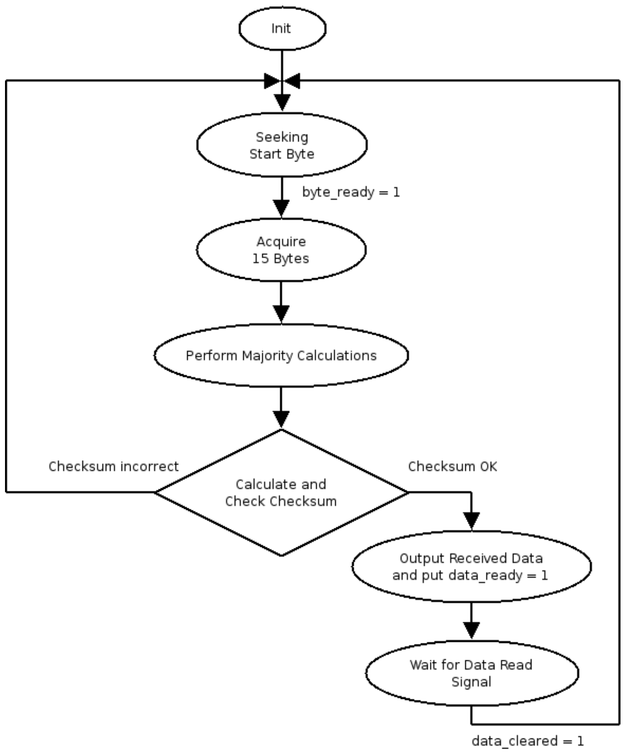


The complete code for this module can be found in Appendix A.

rf_rx

This module simulates the hardware implementation of the receiver. Here is a state diagram followed by a summary of the module's functionality.

Figure 19: Receiver State Diagram


Init
Seeking Start Byte
byte_ready = 1
Acquire 15 Bytes
Perform Majority Calculations
Checksum incorrect
Calculate and Check Checksum
Checksum OK
Output Received Data and put data_ready = 1
Wait for Data Read Signal
data_cleared = 1

The module takes all the useful outputs from the serial decoder along with a clock input and a data_cleared input. The output is sent directly to ai_core module and when correctly acknowledged, the data_cleared input is set to high (set by the ai_core module).

From the incoming data, the receiver module begins by seeking for the start byte. When the start byte is found, it will record the next 15 bytes. The majority algorithm is then performed and the data block of 8-bit as well as the 5-bit checksum is reconstructed. From there, the checksum is calculated on the data block and is verified to see if it matches with the checksum sent.

If the checksum is correct, the receiver module puts data_ready to 1 and waits until the data has been read by ai_core (data_cleared will be set to 1 when data has been read).

After the data has been read successfully, the receiver will go back to seeking state and await the start byte. If the checksum is incorrect, data will never be outputted and the receiver module will go back to the seeking state.

Data analysis system

The implementation of the data analysis system is the base station’s software. This software was written using the Java programming language. The following is a brief description of each designed class. For further details, such as variables and methods implemented by each class, the reader is encouraged to consult our full software documentation which can be found on the enclosed CD-ROM.

<u>AStarElem</u>

Description: The AStarElem class acts as a container for points that are going to be checked when we are trying to find the optimal path. An AStarElem object contains the F, G, and H values for a specific (x,y) point on the path as well as it's parent (which is another AStarElem). The first point that will be added to the closed list has a parent that is null.

<u>CommandManager</u>

Description: Class used to queue commands to the robot, send them and ensure that they are received.

<u>MainGUI</u>

Description: Main class used to display the YARE GUI and bind together the other classes.

<u>Map</u>

Description: Performs maze-solving functions.

<u>MapPoint</u>

Description: The MapPoint Class encapsulates data that was received from the Robot. It provides methods to access the x,y coordinates for what was seen on the left, right, and middle IR sensor. Methods are also provided to access the Robot state, as well as the Robot direction in radians (with respect to the right side of the map).

<u>MathFunctions</u>

Description: The MathFunctions Class contains all of the mathematical functions used througout our code. All the functions are static, this means that you don't have create an instance of MathFunctions to use these functions.

<u>SerialDecoder</u>

Description: Takes raw byte steam received over serial link and checks it's integrity. Creates a serial packet and passes it the actual data stream.

SerialPacket

Description: Container class used to store data received over RF link.

SerialReceiver

Description: Class used to perform all communications over the serial link.

XYDecoder

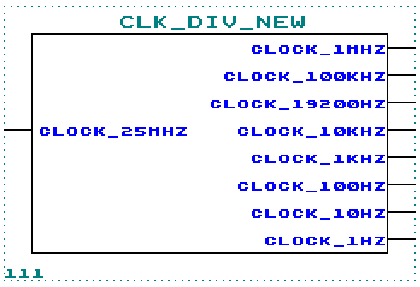
Description: Processes SerialPackets in order to obtain x,y positions of the robot and its IR sensor readings.

Miscellaneous hardware modules

Clock divider

This module is used to create slower clocks from the 25MHz clock that powers the Altera board. Since such code has been written countless times already, a module that was freely available was used. However, the original code was slightly modified to output, in addition to the already existing clocks, a clock of 19.2 kHz. This clock is used by our RF_TX module. This is the hardware representation of the Clock Divider module (VHDL):

Figure 20: Clock Divider Hardware Representation

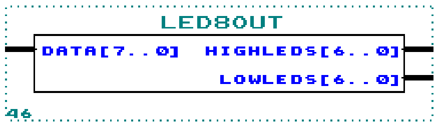


The original code was taken from [5].
The complete code for this module can be found in Appendix A.

Seven segment decoder

Perhaps the most useful debugging module was the 7-segment decoder: This decoder would take 8 bits of data and output them as two hexadecimal characters on two 7-segment LEDs. This, combined with a debugging signal being outputted from many of our modules, allowed us to quickly observe which state our robot was currently in. The hardware representation of the 7-segment decoder, in VHDL, was:

Figure 21: Seven Segment Decoder Hardware Representation



The complete code for this module can be found in Appendix A.

RF serial decoder

This module was used to debug the RF receiver, and is still used to quickly determine how much interference is received by the robot. This module interfaces with the RF SerialDecoder module and counts the number of correctly received serial packets and the number of packets containing a framing or start bit error. Those numbers are then made available on 4 bit signals which can easily be displayed on the 7-segment LEDs. This is the hardware representation of that test module, in VHDL:

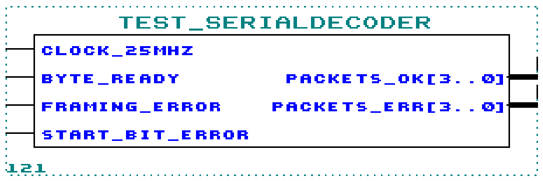


Figure 22: Debugging Decoder Hardware Representation

Other modules have also been created, but since they were of extreme simplicity, they will only be mentioned briefly: a 4-to-8 merger, allowing two 4 bit buses to join into a single 8 bit bus, a 16-to-8 splitter and an 8-to-4 splitter, to easily split buses, and a zeros-4 generator, to easily generate a bus containing nothing but zeros (useful to 'fill' empty inputs when debugging). The code for these modules can be found in Appendix A.

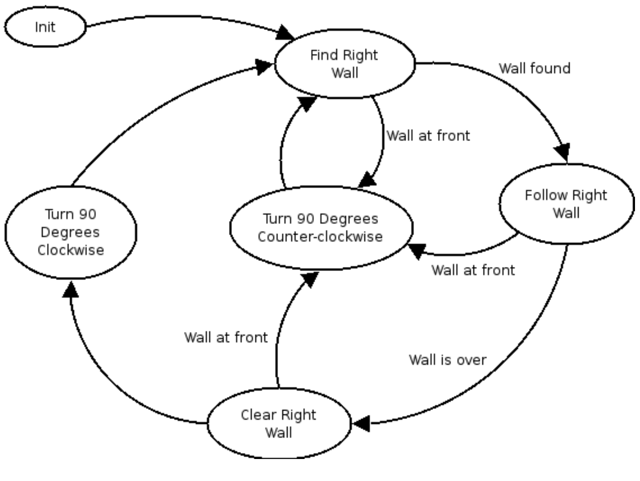
The reader will want to note that a key component of our implementation is missing: the artificial intelligence core. This component is part of the data acquisition system. This component is discussed in the following section because it directly affects the mobile robot's behaviour and makes decisions based on various control algorithms.

Control Algorithms

The robot has to be able to solve the maze by itself and it has to receive commands from a base station and execute them. Since the same commands can be used to solve the maze and to control the robot remotely, those two functionalities were combined within the AI Core module. After executing a command, the AI Core module goes in a Waiting for Command state. In this state, the robot takes one of two actions depending on the mode it is in. This method was used, instead of using two separate modules, in order to avoid repeating the same code twice. This means that less code needs to be synthesized on the FPGA, and, eventually, smaller chips could be built to house the hardware implementation. Smaller chips mean lower costs, both in fabrication expenses and in power requirements.

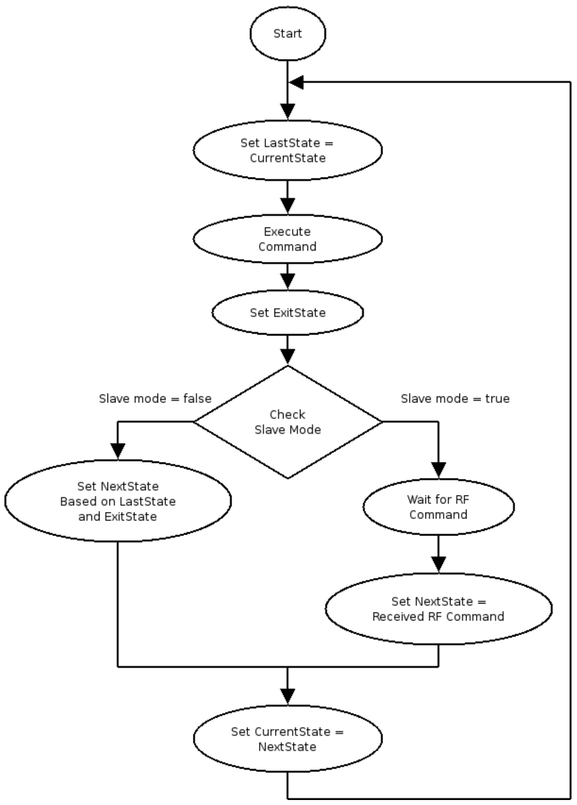
If the robot is in autonomous mode, meaning the robot is attempting to solve the maze by itself, it will examine what state the robot was previously in as well as the reason for the state change. From that information, the robot will decide what to do next. The following diagram illustrates how the robot goes about solving the maze:

Figure 23: Autonomous Mode State Diagram



As you can see, the robot simply follows the right wall. When exiting one of the command states, the robot will go to a decision state in which it decides which state to switch to depending on the exit conditions. When the robot is running in Slave Mode, the next state is chosen by awaiting instructions from the base station and blindly executing them. This diagram illustrates the decision-making processes that the robot uses:

Figure 24: Descision Process

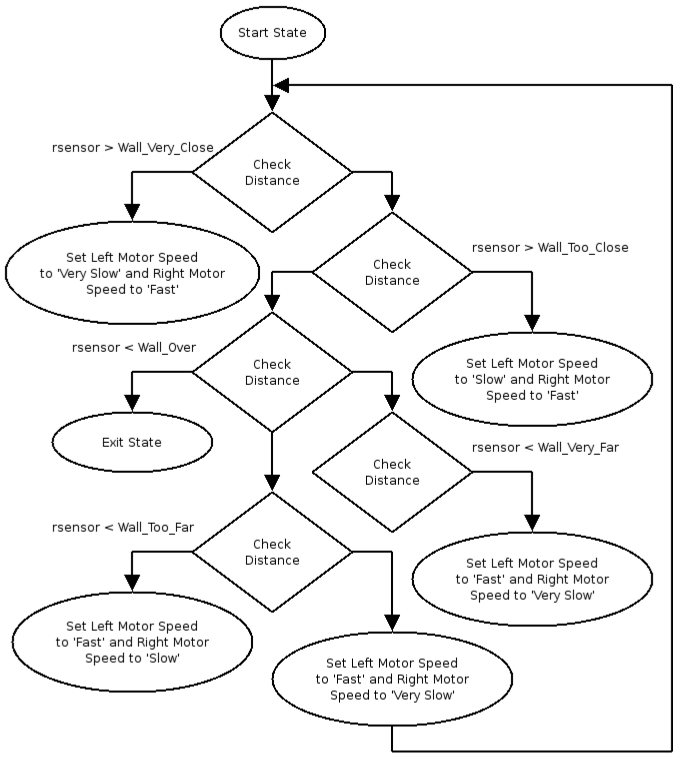

Start
Set LastState = CurrentState
Execute Command
Set ExitState
Slave mode = false
Check Slave Mode
Slave mode = true
Set NextState Based on LastState and ExitState
Wait for RF Command
Set NextState = Received RF Command
Set CurrentState = NextState

This technique allows the reuse of existing commands, such as 'Follow Right Wall', for both autonomous navigation and slave navigation.

Most commands executed by the robot are straightforward: 'Turn 90 Degrees Clockwise' corresponds to making the left motor go forward and the right motor go backwards for a certain amount of time; 'Find Right Wall' simply makes the robot move forward until the right IR sensor reads a distance below an acceptable value. The only commands which really require an explanation are the 'Follow Right Wall' and 'Follow Left Wall' commands.

What makes those two commands more complex is the compensation algorithm that was included in them. When the robot is following a wall, it cannot simply be told to go forward: if the wall curves slightly or if the robot starts to deviate away from the wall (for example, if one wheel turns slightly faster than it should), the robot can lose track of the wall. To prevent this, the following algorithm is used to keep the robot at a relatively constant distance from the wall:

Figure 25: Compensation Algorithm


Start State
rsensor > Wall_Very_Close
Check Distance
Set Left Motor Speed to 'Very Slow' and Right Motor Speed to 'Fast'
Check Distance
rsensor > Wall_Too_Close
Set Left Motor Speed to 'Slow' and Right Motor Speed to 'Fast'
rsensor < Wall_Over
Check Distance
Exit State
Check Distance
rsensor < Wall_Very_Far
Set Left Motor Speed to 'Fast' and Right Motor Speed to 'Very Slow'
rsensor < Wall_Too_Far
Check Distance
Set Left Motor Speed to 'Fast' and Right Motor Speed to 'Slow'
Set Left Motor Speed to 'Fast' and Right Motor Speed to 'Very Slow'

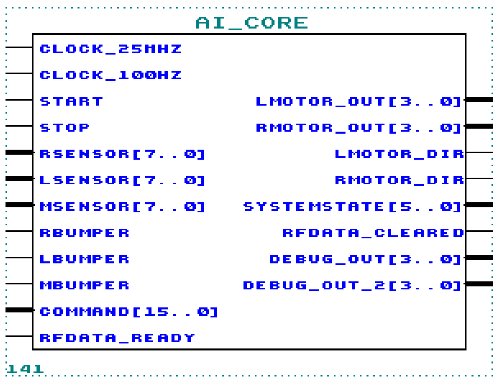
The algorithm is simple: if the robot is too far from the wall, the speed of the right motor is reduced in order to make the robot converge towards the wall. If the robot is too close to the wall, the speed of the left motor is increased in order to put some distance between the robot and the wall. The algorithm includes provisions for both slight adjustments and large adjustments: this is because when the robot is relatively parallel to the wall, there is little need to adjust the trajectory. Only slight adjustments are required as the robot starts to deviate. However, when the robot is noticeably off its course, large adjustments are made to ensure that the robot does not collide with the wall or that the robot loses track of the wall.

Furthermore, while the robot will exit the command state when the execution of the command is completed (e.g. wall is over, or robot has finished turning), the robot will also exit when either bumper is triggered. This is because the bumpers can only be triggered by unexpected events and the robot will need to be instructed as to what to do next. When the bumpers are triggered, the robot will automatically go into Slave Mode.

Artificial Intelligence Core description

This is the VHDL representation of the AI_Core module:

Figure 26: Artificial Intelligence Core Hardware Representation



As can be seen, the AI_Core module has a lot of inputs. This is because this module needs to receive data from all sensors in order to make intelligent decisions. The core also controls the movements of the robot. Four outputs are used to control the speed and direction of both motors. The current state of the robot is given as an output: this is because that data will be sent over the wireless link by the RF_TX module. Since the AI_Core needs to be able to receive commands, it has two input signals, Command and RFData_Ready, as well as an output signal, RFData_Cleared, to handle received commands. The Start and Stop signals are also interfaced to the push buttons and this allows the user to manually toggle the robot's mode between autonomous and slave mode. This is useful in case of RF failure.

A single state machine is used to operate the AI Core module. Three signals, however, take precedence over the state machine: Start, which toggles between autonomous and slave modes; Stop, which resets the robot; and, Reset, which is only used when the robot is being programmed, and, of course, resets the robot.

The implementation of the state machine is rather standard: one process executes the state machine during each clock cycle, while another changes the CurrentState to NextState every clock cycle. A third process is used to count the time. This time is used, for example, when turning or when clearing corners. The complete code for this module can be found in Appendix A.

Implemented solution

Implementation Variations

This section of the report will discuss how some of the components differ from their theoretical solution.

One component that differs from the theoretical approach is the implementation of the A* algorithm. A modified version of the algorithm was used. Instead of dividing the map into square regions and checking every square when trying to find the optimal path, another approach was selected. The points that were on the robot's path, from its first time in the maze, were examined. Only the points that corresponded to the robot turning or to openings in the wall were kept. This approach made it somewhat simpler to implement the algorithm. However some generality was lost, because two points can only connect together if the robot has passed through those points before. Furthermore, the angles for the lines of the path were restricted (with respect to the right side of the map). They have to be approximately a right angle or zero (ie. close to 0, 90, 180, or 270 degrees). This ensures that the optimal path only consists of lines that are nearly parallel (or perpendicular) to a wall, since the robot either follows the right or left wall.

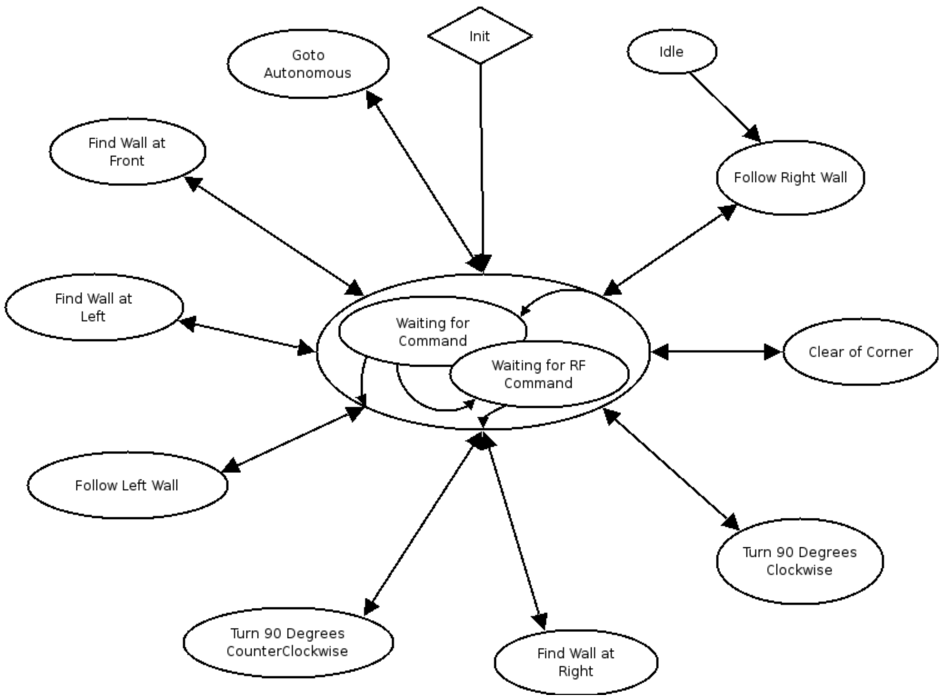
To create the optimal path, the path vector originally contains the set of points (a point being defined as an x,y position in the map) that the robot discovered during its first pass through the maze. The first element of the path vector is added to the ClosedList (also a Vector of points). Then, all of the points that are reachable, from the current point, are added to the OpenList (yet another Vector of points). In order to see if a point is reachable or not, an attempt to form a line between the current point and the desired point is made. If that line intersects a section of a wall the point is not reachable. All the wall segments (each wall segment consists of a Vector of points) are contained in a wall Vector (ie. the wall Vector was a Vector of Vectors), and each wall segment is checked for possible intersections.

Apart from these minor differences, the implementation of the A* algorithm that was implemented follows fairly faithfully the algorithm described in the theoretical section of the report.

Another component that was changed from its theoretical design was the artificial intelligence core. The following paragraphs describe how the implemented core functions.

To implement the AI Core functionalities, a single state machine was used. However, while each state was generally used to execute a single command, two special states were used to take decisions: 'Waiting for Command' and 'Waiting for RF Command'. The first state would check if the robot is in Slave Mode and switch to *Waiting for RF Command* if it was. If it was not, the robot would decide what to do in accordance to the 'Follow Right Wall' algorithm. This diagram illustrates the actual state machine implemented in the AI Core.

Figure 27: State Diagram of Implemented AI Core



As can be seen, there are additional commands that are only used when in Slave Mode, such as 'Follow Left Wall'. Such commands are useless in Autonomous Mode since the robot always follows the right wall. The robot has the ability to execute the following commands:

Command	***Description***
Follow Right Wall	Robot will follow the right wall until it is over or until a wall is detected at the front.
Clear of Corner	Robot will move forward for *x* seconds or until a wall is detected at the front.
Turn 90 Degrees Clockwise	Robot turns clockwise for *x* seconds.
Find Wall at Right	Robot moves forward until a wall is found at the right or at the front.
Turn 90 Degrees Counter-Clockwise	Robot turns counter-clockwise for *x* seconds.
Follow Left Wall	Robot will follow left wall until it is over or until a wall is detected at the front.
Find Wall at Left	Robot moves forward until a wall is found at the left or at the front.
Find Wall at Front	Robot moves forward until a wall is found at the front.
Goto Autonomous	Robot will switch from Slave Mode to Autonomous mode. The initial state used when switching from Slave Mode is *Find Wall at Right*.

Overall System Specifications

This section describes some of the systems overall properties. To gain a better understanding of the robot's implementation, the reader is encouraged to consult Appendices B and C. They contain a photo of Y.A.R.E, and a diagram of the hardware circuitry.

Power consumption

The various components of the implemented robot were tested in order to compute its total power consumption. The results obtained are as follows:

Connected to the voltage regulator, (LM78XX Series from National Semiconductor) which is located on the development board:

Futaba FP-5148 servo motors =	265 mA * 2
Fairchild 74F244 Octal Buffer:	90 mA
Lynx Technologies HP series II Board:	30 mA
Lynx Technologies HP series II Transmitter:	17 mA
Lynx Technologies HP series II Receiver:	20 mA
Total:	687 mA

Connected to the Altera evaluation board:

Altera board:	350 mA
Sharp GP2D02 infrared sensors:	25 mA * 3
Total:	425 mA

Total load on battery:	1112 mA

Since the regulator and the Altera board can both provide up to one Ampere of current, this implementation ensures that all the components will receive enough power to function properly.

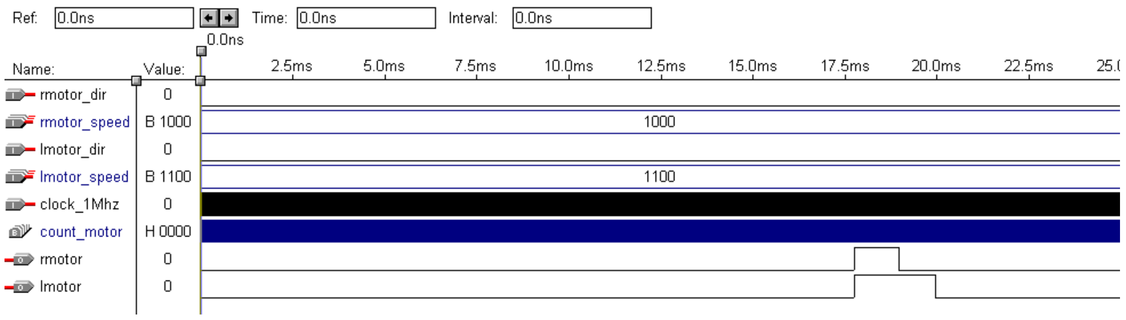
Implementation Results

The following section of the report contains simulations of various hardware components. Tests of the wireless communication link and the overall system are also provided.

Hardware Simulations and Analysis

This is a simulation of the execution of the Servo Control VHDL module:

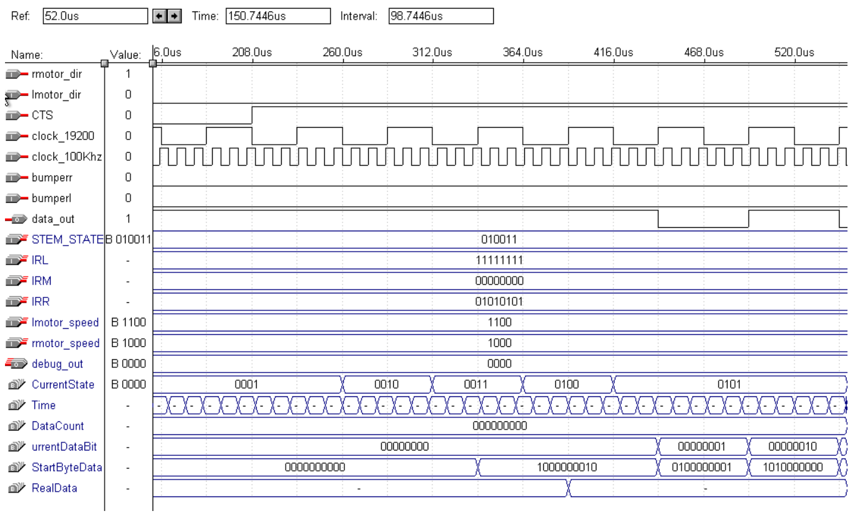
Figure 28: Servo Motor Simulation



As can be seen from the simulation, a request for higher speed creates a pulse with a duty-cycle that is longer than a request for a lower speed.

Here is a simulation of the execution of the RF Transmitter VHDL module:

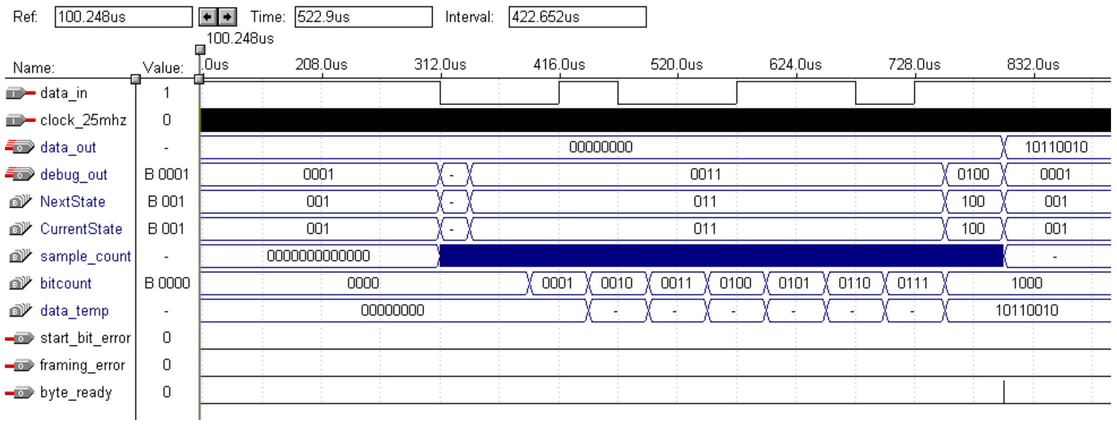
Figure 29: RF Transmitter Simulation



It can be observed that the transmitter waits for the CTS signal to be set to high. Once this condition is satisfied, the machine will compile the data from the sensors (illustrated by CurrentState = 0010). Following this procedure, the checksum is calculated on the data (CurrentState = 0011). The checksum and data are assembled together in a single register (CurrentState = 0100). From this point forward, the transmitter is transmitting. It can be clearly seen in the data_out signal the start_bit (0) followed by the packet start bit (1) and, though it cannot be seen on this diagram, the 7 zeros that follow.

This is a simulation of the execution of the RF Serial Decoder VHDL module:

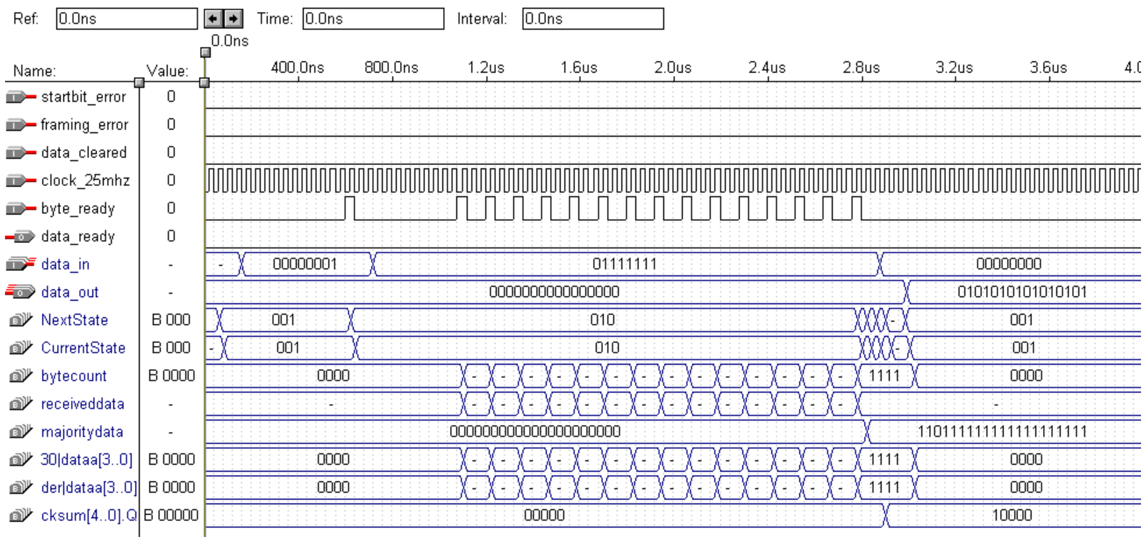
Figure 30: Serial Decoder Simulation



The Serial Receiver seeks a change from high to low in the incoming signal: this marks the beginning of the start bit. From the current state it can be observed that the module will take a reading at the middle of the start bit to ensure that no interference was received. Since the start bit is still low, the program can start sampling the bits. From the *bitcount* signal, it can be seen that the module takes eight samples at the middle of the bits. Once 8 bits have been read, a final check is performed: the stop bit should be set to high. If it is not, a framing error occurs. Since the stop bit is correct, it is safe to output the received serial packet and signal, as shown by the spike in byte_ready, to indicate that the data is ready to be read. The data is only valid for one clock cycle. This is not a problem because the module that processes this data, RF_RX, is perfectly synchronized with this module.

This is a simulation of the execution of the RF RX VHDL module:

Figure 31: RF Receiver Simulation



This simulation illustrates the reception of a start byte packet (data_in = 00000001). The reception of the start byte changes the state of the module from 001 (seeking start byte) to 010 (receiving bytes). Fifteen bytes are sent to the RF_RX module. Those bytes contained 1s only, with the exception of the packet start bits (0s). It can be seen from the byte count that the module reads each data set as it is received. Once 15 bytes are received, the module performs majority calculations on the received data, calculates the checksum and verifies that the calculated checksum matches the received checksum. In the simulation no coherent data was entered. As expected, the calculated checksum does not match the (absent) checksum provided. The module will therefore not signal that it has data ready and returns to the state 001, which corresponds to seeking start byte.

Wireless Communication Link Tests

After completing the design of the communication protocol a few tests were performed:

- Wireless communication without antennae
    - Close range (0 – 0.5 meters)
        - At close range, the wireless communication was working both ways and packet integrity was higher than expected. The packet integrity was nearly 100%.
    - Mid range 0.5 – 4 meters)
        - At mid range the wireless communication was working fairly well both ways. The packet integrity was approximately 80%.
    - Long range (4 - 10 meters: occasional loss of line of sight)
        - At long range, there were significant packet losses and depending on the amount of interfering objects the packet integrity would varies considerably.

- Wireless communication with antennae on both transmitter and receiver modules
    - Close range (0 – 0.5 meters)
        - At close range the wireless communication was not working. The packet integrity was nearly 0%.
    - Mid range (0.5 – 4 meters)
        - At mid range the wireless communication was not working well. It appears that communication from the robot to the base station was working, but not reliably. The packet integrity varied greatly.
    - Long range (4 - 10 meters: occasional loss of line of sight)
        - The wireless communication link was not reliable.

It is quite surprising that the wireless link performs better without antennae. The various objects and equipment in the lab reflect the signal and deteriorate signal quality. The problem seems to be with the transmitter. It was hypothesized that antennae should be kept for the receiver modules only. To verify this hypothesis another test was performed. The results are shown below:

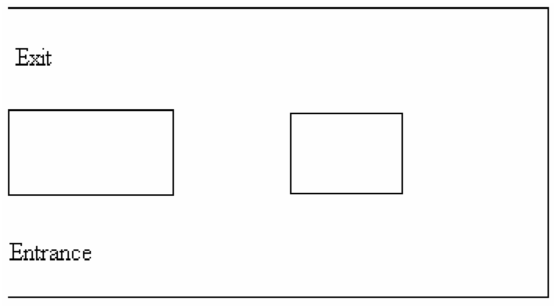
- Wireless communication without antennae on transmitter modules and with antennae on receiver modules
    - Close range (0 – 0.5 meters)
        - At close range the wireless link was working both ways and packet integrity was higher than expected. The packet integrity was nearly 100%.
    - Mid range (0.5 – 4 meters)
        - At mid range the wireless link was still working reliably in both directions. The packet integrity was nearly 95%.
    - Long range (4 - 10 meters: line of sight occasionally absent)
        - At long range, the packet integrity varied depending on interference factors. However, it was generally higher than 75%.

These results were satisfactory and enabled commands to be sent to and from the mobile robot reliably.

Overall System Test

This section of the report will attempt to illustrate the results that were obtained when the robot was asked to solve a maze. Here is a diagram that shows the maze that was chosen for this test.

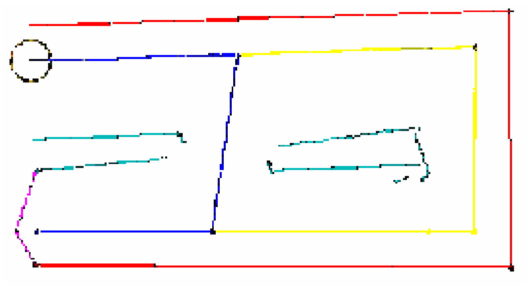
Figure 32: Test Maze



What the robot is supposed to do during its first attempt to solve the maze is simply to follow the right wall. This will result in a square path. During its second attempt, the robot will use the shortest path which corresponds to turning left immediately, and left again to the exit.

During the test the wireless link worked flawlessly. The robot was set in autonomous mode remotely from the base station. It followed the right wall without any problems. The base station computed the shortest path and sent the information to the robot through the wireless link. The robot solved the maze correctly. Here is the resulting shortest path as computed by the base station's software.

Figure 33: Shortest Path



This figure was modified to be printed. The reader is encouraged to run the base station software that is included with this report in order to obtain the actual results from the tests. The software has the ability to load the raw sensor data collected from the robot.

On the figure, the left walls are represented in red and the left walls in cyan. The first attempt of the robot is displayed in yellow while the optimal path, which also corresponds to the path the robot traveled during its second attempt, is displayed in blue.

In theory, using the algorithms described in the previous sections, the robot should be able to solve any maze. However some restrictions were found through numerous tests.

One example of a restriction is that the width of the maze's corridors and openings should be the same, and should not be too small. The robot should be able to pass easily. If the corridors are too large, the robot cannot find the wall when it clears an opening because the walls are further than the acceptable threshold value.

Furthermore, whenever the robot needs to perform a turn in the optimal path, there must be walls on both sides of the robot. This is a restriction due to the algorithm that was developed to send commands to robot to solve the shortest path.

Finally, the walls of maze should be at 90 degrees angles since our robot is not capable of making gradual turns.

Y.A.R.E. finds the shortest path successfully in mazes that satisfy these conditions.

Project organization

Scheduling

In order to design and implement the project within the set deadline, time and resource management played an important role in the success of this project. The following section of this report will describe this component of the project.

Each member of the team, had their strengths and weaknesses evaluated. These qualities incorporated in a list. From this list, each member was assigned tasks. These tasks were recorded in the following table:

Responsibility Assignment Matrix (RAM)

Member / Work Item	**Erick**	**Martin**	**Mathieu**	**Dominic**	**Bruno**	**George**	**Notes**
Assemble Control Board	Prime	Backup					
HW Architecture	Backup		Prime	Backup			
SW Architecture	Backup			Backup	Prime	Backup	
Software Development	Limited	limited	yes	yes	yes	yes	
Integration Test	Backup	Backup	Backup	Backup	Backup	Prime	
RF protocol	Backup	Backup		Prime			
Documentation & Report		Prime	Backup				

Here is a brief description of each item in the table. The first item, assembling the control board, refers to the act of combining the various sensors and ensuring that they communicate together in a coherent manner. The section on hardware architecture refers to the design and implementation (mostly in VHDL) of the hardware code (such as the servo motors, compass and IR sensors). Software architecture refers to the design and implementation of the software application located in the base station. The integration test is to verify if the robot solves the maze correctly. The RF protocol refers to the design and coding of the wireless communication protocol between the base station and the robot. The documentation is self-explanatory.

These tasks were then broken down into smaller more manageable steps and the time required to complete each step was estimated. Using this estimation a Gantt chart was drawn.

Task name		Optimistic duration	Pessimistic duration	Most likely duration
		(weeks)	(weeks)	(weeks)
design and architecture		1	2	1
write VHDL code		2	4	3
test and repair VHDL code		1	4	2
write documentation		1	3	2
become familiar with sensors/actuators		1	2	1
mount sensors/actuators on robot		1	2	1
code maze mapping software		1	4	2
learn to design/use wireless protocol		1	4	2
combine hardware and software		1	3	2
test functions / if robot performs task		1	3	2
make presentation flyers/slides		1	2	1
demonstration		0	0	0
presentation		0	0	0
Totals :		12	33	19

Most Likely Timeline

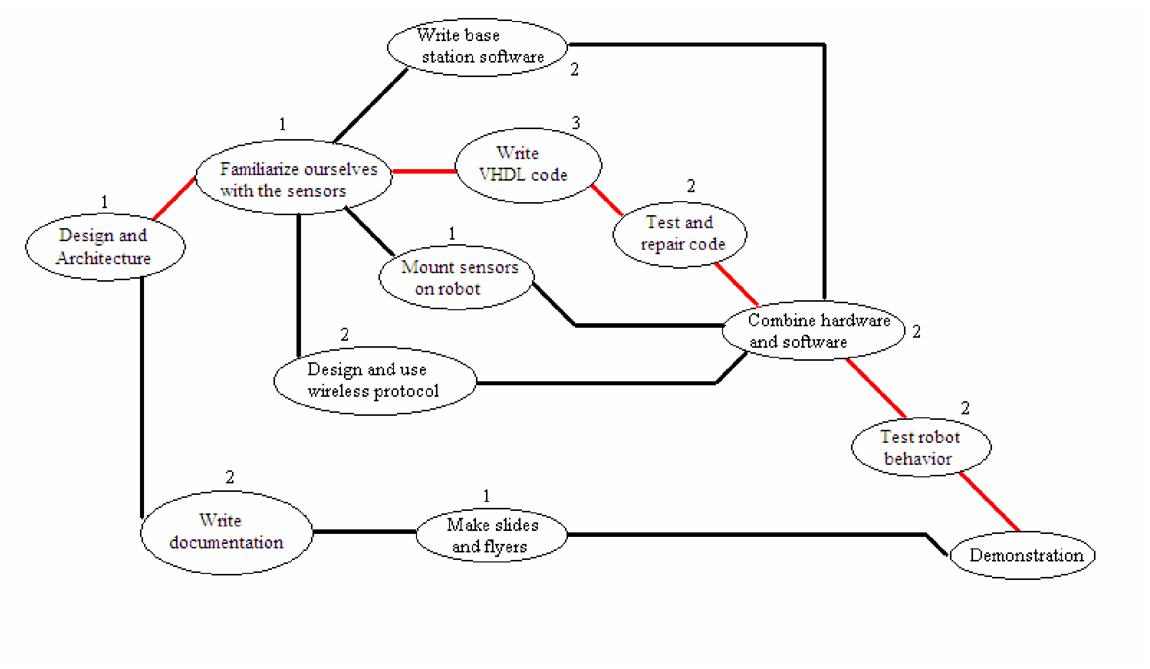
	Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Activity																					
design and architecture		🟦																			
write VHDL code				🟦	🟦	🟦															
test and repair VHDL code						🟦	🟦														
write documentation						🟦	🟦														
become familiar with sensors/actuators			🟦																		
mount sensors/actuators on robot							🟦														
code maze mapping software					🟦	🟦															
learn to design/use wireless protocol				🟦	🟦																
combine hardware and software								🟦	🟦												
test functions / if robot performs task										🟦	🟦										
make presentation flyers/slides											🟦										
Demo																					
Presentation																					

Tasks that are super-imposed can be done in parallel.

As an added method to estimate the time required to design and implement our robot, a pert chart was used.

Activity	Duration (week)	Earliest Start (week)	Earliest Completion (week)	Latest Start (week)	Latest Completion (week)	Slack (weeks)
design and architecture	1	0	1	0	1	0
write VHDL code	3	2	5	2	5	0
test and repair VHDL code	2	5	7	5	7	0
write documentation	2	1	3	8	10	7
become familiar with sensors/actuators	1	1	2	1	2	0
mount sensors/actuators on robot	1	2	3	6	7	4
code base station software	2	2	4	5	7	3
learn to design/use wireless protocol	2	2	4	5	7	3
combine hardware and software	2	7	9	7	9	0
test functions / if robot performs task	2	9	11	9	11	0
make presentation flyers/slides	1	3	4	10	11	7
Demonstration	0	11	11	11	11	0

This shows that the minimal time required for the entire project is eleven weeks. It also demonstrates when each task must be completed in order for the project to be on schedule. If any of the tasks on the critical path are delayed, the entire project will suffer.



Although multiple time estimations were computed, few of them were accurate. Many unpredictable factors influenced the progress of the project.

Some of these factors manifested themselves in the form of implementation problems. It took much longer than expected to implement the wireless transmitters and receivers. The main reason is that the base station's receiver needed to communicate with the base station itself using the serial port. The software used to create this interface required a great degree of research. A solution to the problem was eventually found. A particular package of the Java programming language offers some classes that perform such functions.

In general, the software's complexity was greatly underestimated. The reader will note that the estimate to create the entire maze mapping software (this was the original name for the base station software) was two weeks. In reality, over a month was required to complete this task.

Another area which was underestimated was the testing phase of the project. Many errors were discovered while testing and the project could have benefited from a longer testing period.

In contrast, other areas of the project required fewer resources than expected. The VHDL code was written fairly easily once its algorithms were completed. The design of the control algorithms and the overall hardware structure took longer to design than the code itself. This was a pleasant surprise.

In order to fulfill the various needs of the project, some of the team members performed additional tasks or were reassigned from their original tasks. The final breakdown of the tasks can be found in the following section.

<u>Member Participation</u>

The following table is based on the original list of tasks. It demonstrates how the tasks were actually divided between the various members of the engineering team.

Members	Erick	Martin	Mathieu	Dominic	Bruno	George
Item						
Assemble Control Board	Backup Designer	Primary Designer		Backup Designer		
HW Architecture Design		Backup Designer	Primary Designer	Backup Designer		
Servos Code	Debugger		Main Programmer			
Wireless Tx Code	Debugger		Main Programmer	Debugger		
Wireless Rx Code	Debugger		Main Programmer	Debugger		
Serial Decoder Code	Debugger		Main Programmer	Debugger		
IR sensors Code		Debugger	Debugger	Debugger		
AI core Code	Debugger		Main Programmer	Debugger		
SW Architecture Design	Backup Designer		Backup Designer		Primary Designer	
Wireless Tx Code	Debugger		Main Programmer			
Wireless Rx Code	Debugger		Main Programmer			
XY Decoder Code	Main Programmer		Debugger			
Serial Decoder Code	Debugger		Main Programmer			
Shortest Path Algorithm Code	Debugger	Debugger			Main Programmer	
Maze Wall finder Code	Main Programmer	Debugger			Debugger	
Maze Path finder Code	Main Programmer	Debugger			Debugger	
GUI Code			Main Programmer		Secondary Programmer	
Maze Display Code			Programmer		Programmer	
RS 232 research	Backup Researcher	Backup Researcher	Backup Researcher	Backup Researcher	Main Researcher	

RF protocol	Backup Designer		Backup Designer	Primary Designer		
Software Development	Yes	No	Yes	No	Yes	No
Integration Test	Yes	Yes	Yes	Yes	Yes	No
Documentation & Report	Writer	Primary Writer	Writer	Writer	Writer	Writer
Presentation		Secondary Designer		Primary Designer		
Building Maze						Primary builder
Maze Design	Designer				Designer	

Conclusion

Y.A.R.E. was developed by six engineers. It is capable of solving a maze using an optimal path. The robot collects data from its sensors and sends it through a wireless communication link to a base station. The base station's software analyzes the data and computes the shortest path. The computed path is transmitted to the mobile robot through the wireless link and the robot is capable of solving the maze. This approached worked well and the task was completed successfully.

Possible Improvements and Further Studies

Even though the overall solution adopted by the project was fairly good, there are still some improvements that could be implemented in future versions.

Currently in the maze solving algorithm, the only points that can be added to the OpenList are the ones that were on the path that the robot traveled during its first attempt. What could be done in a future version is to examine a broader area by splitting up the map into small regions (squares). Also, when information is gathered from the sensors, if a wall is out of range it would be safer to assume that this area is "unexplored" until its content is known. The algorithm works for a static environment (where the map doesn't change), but it might be interesting if the robot was capable of handling a dynamic environment. For this, it would be interesting to consider making use of the D* algorithm (Dynamic A*).

Another possible improvement in the base station software would be to perform a verification to see if the robot can fit in an opening. This feature was part of the original design but other features seemed more important and it was abandoned.

It would also be interesting to modify the robot so that it could turn corners at any angle. This would significantly increase our robot's ability to solve mazes.

References

[1] Emil Petriu, “CEG 4392 Project Assignments”
http://
www.site.uottawa.ca/%7Epetriu/CEG4392-2003-ProjectAssigned-Sep17.pdf

[2] William Stallings, *Data & Computer Communications Sixth Edition*, Prentice Hall, 2000, pages 342 and 343

[3] Rensselaer Polytechnic Institute, “Graph Theory: Networking”
http://links.math.rpi.edu/devmodules/graph_networking/compat/page1.html

[4] Linx Technologies Inc., “Application Note AN-00160: Considerations for Sending Data with the HP-II series” http://www.linxtechnologies.com/ldocs/pdfs/AN00160.pdf

[5] James O. Hamblen and Michael D. Furman, *Rapid Prototyping of Digital Systems*, Kluwer Academic publication, 1999

P. Lester, “A* Pathfinding for Beginners”
http://
www.policyalmanac.org/games/aStarTutorial.htmhttp://www.policyalmanac.org/games/aStarTutorial.htm

A. J. Patel, “Amit’s A* Pages”
http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

J. Macgill, “A* Demonstration”, http://www.ccg.leeds.ac.uk/james/aStar/

T. Stentz, “Real_Time Replanning in Dynamic and Unknown Environments”,
http://www.frc.ri.cmu.edu/~axs/dynamic_plan.html

IEEE Computer Society, “Style Guide version January 2003”,
http://www.computer.org/author/style/refer.htm?SMSESSION=NO