UNIVERSITY
of
GLASGOW

Department of Computing Science
and
Department of Electronic and
Electrical Engineering
University of Glasgow
TEAM E PROJECT REPORT
LEVEL 3, 2004/2005

Lego Chess Robot

by

Stewart Gracie, Jonathan Matthey, David Rankin,
Konstantinos Topoglidis

Abstract

As Team E, we aim to build a fully interactive robot that will play chess on a physical board against a human player. This robot's movements will be dictated by a chess engine written in C running under a Windows environment on a desktop computer. The computer and robot will interact via infra red transmissions. The human's moves are detected by a specially built chess board that is connected to the PC via USB using Phidgets technology. The Robot's movement is controlled by motors and sensors which are operated through RCXs which are programmed in NQC. The setup offers a strong game supporting all legal chess moves such as castling, en passant and promoting pieces.

Once complete a human player is able to play against a computer chess program without the distractions of the computer. The game is played on an actual chess board and, after setup, the computer can be ignored altogether. An LCD display, LED lights and sound effects keep the user continually informed. The system will allow individuals to test their chess abilities on a real board without having to move for the computer.

Contents

Chapter 1

Introduction

This document covers the work, difficulties and achievements of team E while undertaking the Level 3 Electronic and Software Engineering (ESE) Team Project.

1.1 Motivations

The project deals with many of the problems faced by modern system designers. The system has to integrate with a foreign piece of code, the open source chess engine that the robot depends on, as well as having to design, prototype and build hardware to capture external events and to integrate with pre-bought technologies. The final system should demonstrate the wide knowledge of technology, both computer science and electrical engineering, that a level 3 ESE student possesses.

The project also has a practical use, allowing chess players to test their abilities on a real chess board while not having to move for their computer opponent. It is a close simulation to a real game of chess and should not distract the human player from their game.

1.2 Background

Currently there are many versions of travel chess available, comparable to the Lego Robot Chess as a computerised chess gaming system but they all provide very little feedback and poor interactivity, requiring the user to tediously move both black and white pieces usually indicated by small LEDs flashing. Also on the market nowadays, many versions of chess software offer players a strong game of chess with a fair representation of graphical pieces, but no matter how realistic they can never compete with an actual physical chess game experience.

1.3 Preliminaries

Knowing the basics of chess is a preliminary of reading this document, as special moves such as en-passant can confuse the reader, all chess technicalities will be explained in a later section.

There are sections which explain the implementation of interfaces that were designed to make this solution possible as well as some electrical engineering detail in the detection of user moves but everything is clearly explained and requires no technical background knowledge.

- All electronics and computation outside the desktop computer should be focused on the Lego Mindstorms RCX or the Phidgets Interface Kit.
- The team has been granted use of the Project Lab on level 7 but this must be shared with other teams from both Level 3 and 4.
- The project had to be completed in a space of six months.
- Our manned resources consisted of the members in our team.
- Sharing of Lego material must be negotiated with the other teams, any extra material should be ordered and reasonable expenses claimed

1.4 Aims

The final product has to meet a certain criteria, the robot must be able to detect the players move, computes its counter move, and moves its piece accordingly. It should support all legal chess moves such as castling, en-passant, and pawns reaching the final rows becoming greater pieces. The robot should offer a very high level of interactivity, communicating to the user through a LCD display which with the help of buttons can incorporate appropriate menus with options to be selected. A series of LEDs around the board as well as speaker system will add to the amount of information provided to the user of the situation. Certain states will require different outcomes, invalid and special moves should be alerted to the user, as well as check and checkmate positions. There are a number of available extras that augment the functionality of the robot, being able to set the difficulty level of the computer increases the skill range spectrum of possible users, having an option to undo the players move, and saving and loading a game state.

1.5 Module Description

The project was easily modularised helping the allocating of resources to maximise productivity. The four modules were Lego Robot Design, Board Design and Construction, Chess Engine Interface Development, RCX Movement Programming

1.5.1 Lego Robot Design

We had to consider different options in building the actual robot to move the pieces, there were three strong suggestions: using magnets to move pieces from below the board, using an arm with a movement axis like an elbow, and the last consists of two parallel struts with another beam across it making a moving frame with an end effect grabber moving on these beams. We found the last suggestion to be most accurate and practical, as the magnets made it difficult and slow for moving in special moves, and the arm was too unbalanced adding many more risks into the equation. We had to study gear to force ratios as well as constructing a strong robot with appropriate sensors and motors.

Stewart Gracie was the team member made responsible for this section of the project.

1.5.2 Board Design and Construction

After researching different detection possibilities, we discarded digital cameras and magnet possibilities and decided to construct a pressure board where each square is to be pushed indicating the start and end squares of a move. This is a much more robust efficient solution and allows us to put our electrical engineering skills into practice as the board consists of 64 switches with a series of resistors and an analogue to digital converter to identify which square was pressed.

David Rankin was the team member made responsible for this section of the project.

1.5.3 Board Software

The board must also have a substantial amount of software running so that the output of the board can be resolved into a players chess move. This move must then be passed to the chess engine, and once a counter has been returned the software must handel the output appropriately so that the user is informed. The LEDs, text display and sounds are all triggered by the physical board but orchestrated by its software running on the computer.

David Rankin was also the made responsible for this section of the project.

1.5.4 Chess Engine Interface Development

We are using Green Light Chess, a free software engine that has many configurable features, we had to learn about opening streams to processes in java as well as writing an algorithm to check for whether player is in check or checkmate as the engine was built to interact with a popular graphical interface called Winboard and it deals with checks. The interface must also deal with all the chess situations and special moves.

Jonathan Matthey was the team member made responsible for this section of the project.

1.5.5 RCX Movement Programming

Deciding on building a core method of move which required detailed measurements of feedback values from rotation and light sensors, every other method relied on a combination of this original move method. A lot of trial and error took place in achieving successful movement functions, as well as finding out what how to send data to the RCX and getting the RCX to communicate between each other to produce fast concurrent movement solutions.

Konstantinos Topoglidis was the team member made responsible for this section of the project.

1.6 Document Outline

The remainder of the report highlights each step taken by the group to create our project solution, beginning with considering possible solutions, specifying quantifiable requirements, producing designs, and creating a working prototype. It details the progress made throughout the project as well as the team structure and risk management techniques used to ensure a successful outcome.

Chapter 2

Project Specification

2.1 Team E - Requirements Specification

We have chosen to tackle our Third year Team Project using Lego Mindstorms, Phidgits and tailored electronics. Our team has decided upon a challenging project which is designing and implementing a Chess Playing Robot, this document will define the specifications and targets for our project as well as the factors that we are expecting to evaluate the final solution against.

The idea behind this project was to create a robot that could play chess against a user. The aim would be to create a robot that could move for itself and distinguishing the moves of its opponent so that the human player can concentrate entirely on their moves and game strategy. Its main purpose is to be able to aid a user in learning chess, and as it will recognise false moves and mistakes. It will also extend to being able to give experts a good game.

Typical chess software nowadays offers players with a fair representation, graphical, of pieces, colours, boards etc But no matter how realistic, it can never compete with an actual game of chess. Our team will be bringing that experience of playing against intelligent software to a real environment. Previously created systems have required the human to move for the computer or to enter in their moves on a computer after they have moved on the board. This process can become very tedious and boring after a few minutes. The aim is that the human will be concentrating on their game solely and have no need to worry about the computer's movements because the robot will work at a realistic level of efficiency and accuracy. This system will be a welcome opponent to chess players, experts or novices alike.

2.2 Essential Goals

The Chess Player Robot is required to be able to:

- Detect its opponent's moves.
- Remember the state of play.

- Move its pieces. As well as take out opponent pieces in the process.
- Play special moves such as castling and en passant.
- Display the move that the user has played, as well as the computers counter move.
- Warn user of an illegal move, and request it played it again.
- Have a procedure to recover from problems such as piece being knocked over.
- Display state of play such as who's move it is, check and checkmate.
- Have several levels of difficulty.

2.3 Quantifiable Factors

The Chess Player Robot will:

- Be able to carry out the computers moves within 2min when not taking another piece with an additional 1.40 min when taking another piece off the board.
- Grab a piece successfully with an accuracy of 90% meaning it will succeed 9 times out of 10.
- Move a piece successfully with an accuracy of 75% meaning it will succeed 15 times out of 20.

2.4 Additional Goals

These are extra goals that should be implemented if time is available, each improving the experience and performance of the Chess Player Robot but yet not essential to its purpose.

The Chess Player Robot could have:

- Be able to play the simpler game of draughts in a similar fashion.
- A clock display to count length of user moves as a way of improving timed games.
- A looped animation to show thinking time for computer.
- LEDs on the side of the board that light up for the according square pressed.
- Different sounds for illegal moves, check, and checkmates.

2.5 Risk Factors

If the robot drops a piece either in travel or badly on a square, it will be detected. It will attempt to rectify it 3 times over and if fails it will ask the user to complete the move for it. This should take place as little as possible.

Another risk is how well centring of pieces, whether it's the user moving the piece not dead centre on a square or the Robot being slightly inaccurate in placing pieces. This will be overcome by a well shaped end piece which should grab a piece from a range of different positions on a square.

Speed of processing, movement, accuracy of detection and placement are all risks identified in the quantifiable factors as they can be measured.

2.6 Timeline

Now - Dec 17th:

- David: Produce board with 64 switches, capable of identifying which square has been pressed. Powered by the Phidgits Kit and also read by a Phidgit's sensor.
- Stewart and Kostas: Design and build an end piece arm prototype in Lego Mindstorms which will grab a piece, raise it, and lower it successfully.
- Jonathan: Researching into an appropriate Chess engine, simulating input and output from it. Programming Phidgit input to detect Chess Board, and output to display required features.
- Documentation is written on current progress.

Jan - Feb

- Lego Mindstorms Prototype is nearly complete, testing and improvements are done on movements, efficiency and accuracy.
- Phidgit and RCX programming is combined to produce output movement from input on board.
- Software is first written to be tested with draughts to test Prototype.
- Documentation is written on progress.

Feb - March (2weeks)

- Final testing takes place with repeated improvements where needed.
- Work on Additional Goals if Essential ones are satisfied.
- Finish Documentation ready for hand in.

Chapter 3

Lego Robot Design

3.1 Design Rationale

The robot had to be able to lift any chess piece from a standard 8x8 chess board, move it to its destination and return to its own start position. This meant that it had to run vertically along side the board while also being able to move the grabber horizontally to any square. This movement had to fit in with the time constraints set by the group at the beginning of the project so that the user wasnt waiting too long for his or her turn.

3.2 Design Options

There were three strong candidates for the implementation of the chess robot; a magnetic robot, robotic arm or girder robot.

3.2.1 Magnetic Robot

One option was to have a robot that would move the pieces around using a magnet. This option would incorporate Lego to handle all moving parts but would require all other parts to be implemented using tailored electronics. The basic idea was to have the robot move underneath the chess board with a magnet in contact with the underside of it, as seen in figure 3.1. Each chess piece would have to be fitted with a metal plate on its base that would allow the robot to move it. This robot would be able to move unseen by the user and not inhibit the users playing area in any way. This idea had already been built by another Lego creator and is fully documented at this site http://www.artilect.co.uk/lego/default.asp.

3.2.2 Robotic Arm

There exists many variations of Lego robotic arms; the key is finding a suitable one for the project. The most popular robotic arm consists of a flat robot with 2 degrees of rotation similar

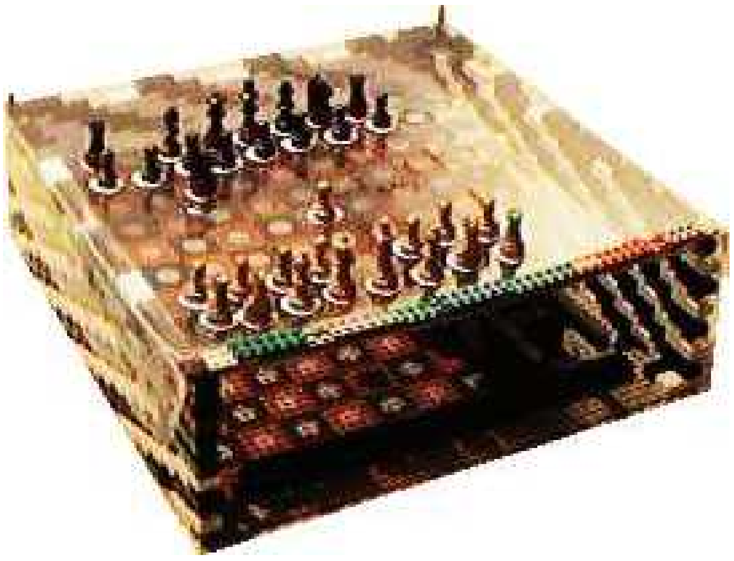


Figure 3.1: Lego Chess Board using Magnets to move pieces

to a shoulder and elbow. One of its many appealing aspects is its resemblance to human movement, enriching the chess playing experience. It most importantly does not importantly doesnt hinder the players movement in any way. An example of the possible robot is shown in figure 3.2. This was taken from the following website http://www.marioferrari.org/lego_mindstorm.html by fellow Lego creator Mario Ferrari.

Since this type of robot has already been created the team felt it would be pointless to try and re-invent it. Its ability to measure the distance the arm had to move and where it was in real time also required more sensors and RCXs than were made available. For both these reasons the team decided to discarded this design.

3.2.3 Girder Robot

The last suggestion for our Lego solution was the Girder Robot which is similar to a theme park grabber. It consists of a top level beam and 2 vertical struts at either end, the end effect grabber runs along the beam. The horizontal and vertical movement axis of the robot are achieved by the grabber moving across the top level. The struts themselves hold the entire structure as it moves alongside the board.

Of the Lego arm designs, this was the most practical and least intricate of the two due to its simple movement and design. It would also require less sensors and RCXs to measure its movement and drive the motors of each part.

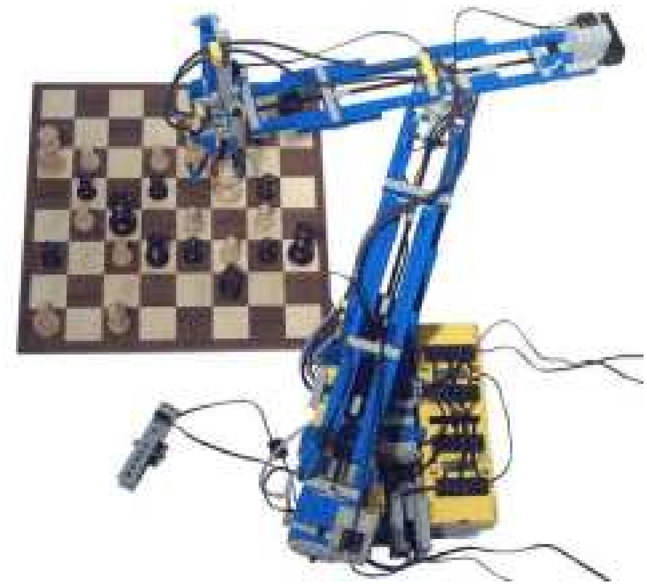


Figure 3.2: Broad Blue Lego design

3.2.4 Comparison Criteria

The following are features that each robot were compared against to give the team a greater understanding of where each of the robot strengths and weaknesses lay.

- **Movement Accuracy**
  Measured by its ability on moving pieces to the centre of squares and the ability to successfully lift pieces. It must also be able to hold the pieces for the length of time to get from its start position to its destination square.
- **Movement Efficiency**
  Assessed on the timeliness and general amount of movement required by the robot to perform standard move operations as well as kill moves.
- **Piece Detection**
  Pieces must be detected by the robot, this is an issue discussed in the implementation of the board but also has a strong impact on the choice of robot.
- **Implementation Achievability**
  Whether the implementation can be achieved depends on certain factors like size, number of pieces, sensors, robots and general balance of the whole robot.
- **Idea Originality**
  This will discuss how original the idea in question is and will be decided based upon other Lego creators finished projects.

3.2.5 Final Decision

The final choice of the group came down to weighing up the advantages of each robot. The magnetic option allowed all the electronics to be hidden from the user within the board and since no lifting of any pieces would be required the chance of dropping any can be discarded. Pieces and moves could be detected using a number of different techniques like having light sensors under the squares to detect if a piece is above it.

It would however be harder to prototype. It has the added complexity of the robot moving a piece from the back row to the front, e.g. a knight moving to a square on the first move. It would have to first move the pawn directly in front of the knight out of the way and then replace it after the knights move was made. This meant that movement for this type of robot could be potentially slow and complex if a piece was surrounded by other pieces. The team felt that this robot would not keep within the time requirements set out and so this idea was discarded.

The final decision lay with the Girder design model, this Lego grabber had a simple design and the ability to split each part of the robot into small, easily prototyped modules. Each prototype could be designed separately and tested before being integrated together to form the fully functional robot. The major drawback of the Lego option was that it had to be visible to the user and would slightly infringe on the players movement area. It did however have the potential to meet all criteria as long as a suitable plan for its construction was decided upon.

With all these arguments in mind the team decided on the girder option based mostly on its easily prototyped components. This would allow the team to work independent of each other on all the different parts of the final system while being able to integrate each part separately. This would allow testing and maintenance tasks to be carried out with less difficulty should any problems arise such as changing broken motors or fixing bugs in design.

3.3 Design Considerations

For each prototype to be made a number of factors had to be taken into account based on limitations set by the system.

First of all the team was only allowed three RCXs of which our team decided we should only use two to control the robot and communicate with outside components.

Second constraint was that each RCX had three inputs and three outputs so any sensors and motors had to be controlled using these. This constraint was most challenging as an effective method of measuring all the distances required hadnt been decided upon at the start of the prototyping stage.

The third constraint set was that the robot wasnt allowed to invade the users playing area when it was their turn. The team wanted the user to have total access to the board when making their move.

The fourth constraint set was that the robot had to be able to move any piece to its destination square within 2 minutes from start of movement to finish. The robot had also to be able to kill any piece and move its own move within 3 minutes and 40 seconds.

The last constraint ensured the grabber had to be accurate enough to successfully pick up a piece 9 times out of 10 and complete any move 15 times out of 20.

3.4 Grabber Prototypes

The design of the grabber mechanism could have been constructed in many different ways. Due to the time limitations set by the project and the group, only two options were prototyped for testing. The prototypes consisted of a pneumatic tripod model and a geared motor model.

3.4.1 Tripod Grabber Model

This model was based on the idea of a tripod with the three axles being pushed in together to clasp the chess piece. The idea behind this was that the three axles, attached with blocks of Lego for friction would close in around the base of the piece all at once and create a firm hold. One of the proposed benefits of this design was that even if the chess piece wasnt centred on its square any axle closing in would push it towards the other axles and it would be secure. This would prove useful to minimize errors when the human player places a piece off centre and the grabber has to pick it up.

This model consisted of three pneumatic pistons forcing the three axles together with enough force to be able to hold it. The force was created by attaching the pistons to a pneumatic air tank which would force air down through a hose and extend the pistons to close the tripod. This model was able to be constructed in such a way that it wasnt very large or heavy which proved useful when it had to be picked up to move a piece. The basic design idea is shown in figure 3.3 below. The problem with this option was getting the air tank to provide a steady

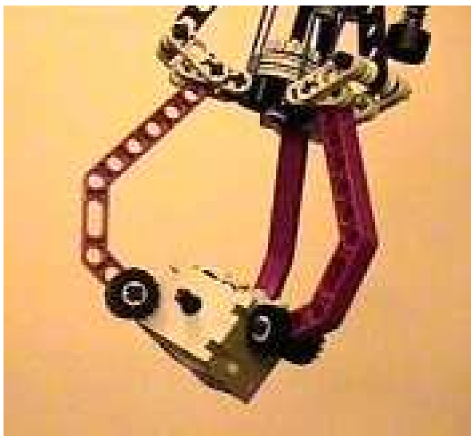


Figure 3.3: Tripod Grabber

pressure to each axle long enough for the piece to be picked up and moved to its destination. When the tank was left on then it would sometimes blow the hose off the tank but when it was turned off the pistons would gradually loosen as the pressure was taken away. The only way to overcome this was to have the tank push air along the hose at regular intervals to keep a steady pressure on the pistons. Even using this method of controlled pressure it was never accurate enough to stop the hose separating from the tank or in some cases the pistons.

3.4.2 Final Grabber Model

The second prototype was based on two parallel bars closing in on the piece at the base. The difference between this model and the tripod was that this had to be constructed with a motor to provide the torque to push the bars together. This model was constructed using a motor and various sizes of gears and axles.

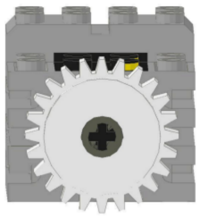


Figure 3.4: Clutch gear attached to motor

The benefits of this design was that the grabber could be geared up to have a greater torque and with it a greater hold on the piece. It also meant that the motor could be left on while the piece is being picked up so that it had a continuous hold of the piece throughout the move. This feature was made available by the addition of a clutch gear to the motor to allow the motor to continue turning its axle without turning the gear. A picture of this set-up is shown in figure 3.4. This is due to the centre of the gear containing a smaller inner gear. This inner gear has a specific torque ratio which if met will continue to turn the inner gear without rotating the larger outer gear. This constant torque applied to the gear allows the grabber to keep a firm hold of the chess piece for the duration of its move. Without these properties of the clutch gear the motor could burn out due to the current being too great when the axle is unable to turn.

The major drawback of the motor model was its increased size and weight due to all the gears needed to create the appropriate torque and all the Lego bricks needed to hold it all in place. The gear ratio is shown in figure 3.5 below. This gear train has a 5:3 gear ratio meaning that five full rotations of the clutch gear corresponds to three full turns of the bottom two gears. This meant that the bottom gears rotated slower than the clutch gear but has increased its torque value by a 5/3. This extra torque is then passed onto the grabbers axles to hold the piece. The main problem lay with its ability to pick up chess pieces accurately without dropping them. Due to the round base of all chess pieces the bars werent shaped appropriately to grab them from the base. Although the torque applied to the grabber was set to be more than enough to hold a piece, it became apparent that the chess piece sometimes flip around in mid-air to become upside-down. This problem was easily fixed though by using chess pieces that had a small ridge just above its base for the grabber to hold. The chess set is shown in

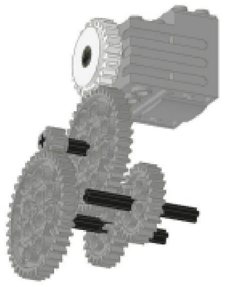


Figure 3.5: Picture of gear ratio

the figure 3.6 below. A number of different chess sets were considered as a possible solution to this problem. These possibilities are documented in Appendix D, section D.1, along with all websites visited. The finished grabber model is shown in figure 3.7. This is actually the second

Figure 3.6: Picture of Chess pieces

of two models made for this type of grabber. The first prototype was deemed too large by the team and then had to be optimized to make the smallest solution possible. It was also found through testing that the longest axles the leg had werent long enough to pick up the pieces. This was fixed by extending the legs using additional Lego pieces fitted at the bottom of the first model. This addition has been added to figure

3.4.3 Lifting Mechanisms for Grabber

The lifting mechanism for the grabber was the easiest to prototype using Lego due to the large choice of parts available. It could have been lifted using pulleys, gears or even a combination of the two to form a lift type architecture. The lift idea meant that the grabber would be attached to poles or wire and lowered down to pick up a piece. This was deemed too unstable as the grabber could potentially swing from side to side and knock over adjacent chess pieces. This method was therefore discarded before prototyping leaving two options open for modelling.

The first model meant that the grabber had to be fitted with two tracks at either side so

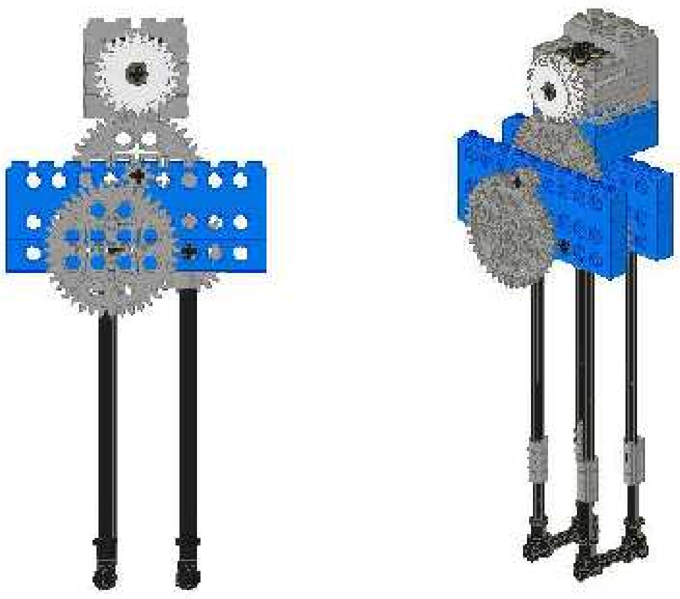


Figure 3.7: Picture of finished grabber with and without extensions

that two motors with gears could drive them up and down. The second model used only 1 set of tracks but was fitted with poles at either side to help balance the whole structure.

3.4.3.1 Two Track Model

The two track model as mentioned above used a track at either side of the grabber and was moved by having two motors fitted with a gear to push it up and down. Having two motors lift the grabber proved a useful solution as it meant each motor had to bare only half the total weight of which was now a very heavy grabber. Also having each motor running off a single RCX output port meant that they both responded at the same time keeping the grabber as straight as possible. This design is shown in figure 3.8. The problems with this arrangement came to light as soon as the testing stage started. Although the motors worked perfectly and were both perfectly matched (Section 3.4.4) at the same speeds the trouble came when trying to stop the grabber at the top of its movement. Any attempt to halt it at its peak height usually ended in a mass of Lego coming apart or a very unhealthy sound coming from the motors. Knowing that the motors could get damaged due to this frequent forced stoppage meant this method had to be discarded. An attempt was also made to stop and start the motors to keep it at its peak height but was too inaccurate to succeed and was subsequently abandoned.

3.4.3.2 Final Lifting Model

The model that we opted for was to attach a single runner for lifting the grabber with a pole either side to steady it. Using a single motor meant that it had to be able to lift the whole

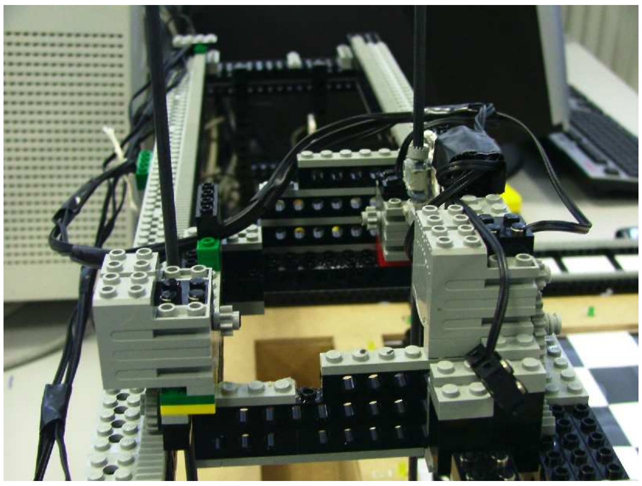


Figure 3.8: Picture of lifter with two motors and two tracks

grabber by itself which could have required it to be geared to get the required power to lift it. This however was not required as the motor was perfectly capable of lifting it on its own. Therefore an 8 tooth gear was attached to the motor and made close contact with the 10 inch track modelled below. The single motor also made it easier for the upwards motion to be halted by placing Lego bricks at certain points to stop the movement where we wanted it. The second

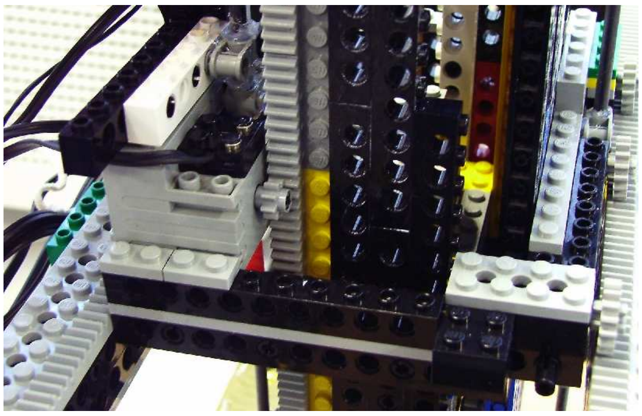


Figure 3.9: Final lifting model

part of constructing this part was to find poles that would hold the whole mechanism steady. They had to be cut to measure from a single length of carbon fibre measuring 4mm by 850 mm. This was cut in half and each length was then glued to each side of the frame. Now the grabber could freely run up and down these poles when the motor was on without having to worry about any instability.

3.4.4 Motor Matching

This process was used to find motors that ran at the same speed so they could be used when parallel movement was needed in the system. If one motor was faster than the other, during a move it could overtake the opposite motor and create a skewing effect which could cause damage to the robot. The method of measuring the actual speeds of each motor was quite simple. It involved attaching the motor to a rotation sensor as shown in figure 3.10 and running it for a given time. During this time the sensor would output the results to the RCX which counted the number of rotations in what is known as clicks. Clicks are the measure of a single rotation divided by 16 meaning there are 16 clicks in 1 rotation. This allows for more precise measurement of each motor. The code used to implement this test is shown below in.

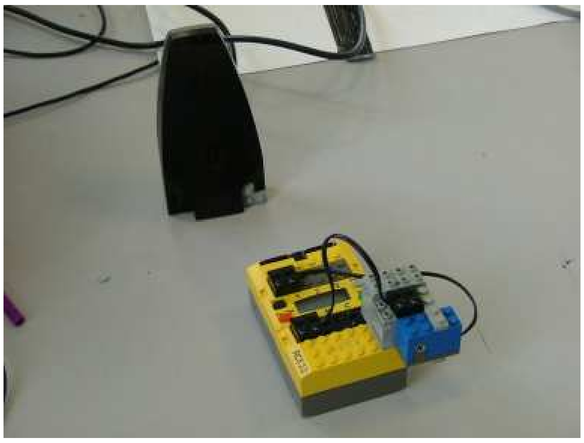


Figure 3.10: Testing of motors

This code is written in NQC and basically loads the program into the RCX, runs the motor for ten seconds and outputs the data back to the computer.

```
Task main()
{
   //Declare sensor 1 as rotation sensor
   SetSensorType( SENSOR_1,SENSOR_TYPE_ROTATION );
   //Clear the sensor before starting program
   ClearSensor( SENSOR_1 );
   OnFwd( OUT_A ); //Start motor running
   Wait( 1000 ); //allow motor to run for ten seconds
   Off( OUT_A ); //Output data to computer
}
```

Using this data I recorded a table of values for all the motors available in the lab. This table can be found in section D.2 of Appendix D. By comparing this data I could pick the two closest matched motors to drive the robots legs running parallel to the board at the same rate.

3.4.5 Platform for Grabber

Before considering the different mechanisms involved in moving the lifting mechanism to any square, a platform for it had to be decided upon. It was decided that the lifting mechanism should run along two tracks as this was the simplest method available. The track is constructed from Lego beams joined together in a row which is 27 inches long with tracks running 19 inches along it. To hold the full weight of the grabber securely it was made three Lego beams wide and two Lego beams thick. The full size of one of the tracks is shown below in figure 3.11. With the structure chosen the next stage was to design the drive that would move the lifting

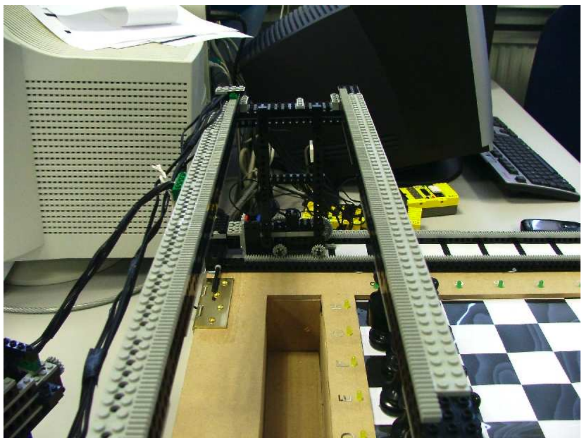


Figure 3.11: Picture of beams

mechanism.

3.4.6 Movement of Grabber

With the whole grabber module now finished the next task was prototype a drive that would move it horizontally to any column on the board. For this task there was only one design that was both simple and effective to use. A single motor could run the full length of the track carrying the lifting mechanism to its destination square.

3.4.6.1 Horizontal Movement

As mentioned above this drive would run along two tracks spanning the breadth of the board allowing access to any column. This drive uses only one motor to move along the two tracks and one rotation sensor to measure the distance travelled. This drive was required to move relatively slowly so that it could stop promptly when the rotation sensor signalled it had reached it required distance. For this reason it had a 5:1 gear ratio which was slow enough and strong enough to move the whole weight of the grabbing unit. This design is shown below in Figure 3.12. The last problem needed to be addressed was how the drive knew where it was at the beginning of a move. It was quickly realised that the easiest way to control this was to make

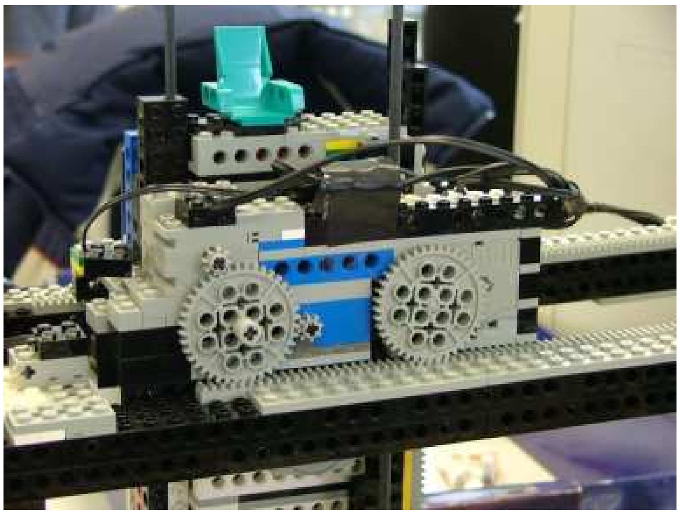


Figure 3.12: Picture of gear train

the drive return to some designated spot after each move. This way the calculations involved in moving the drive would be the same for every column.

To implement this design a touch sensor was placed at one side of the track. The drive had a protruding Lego block that would make contact with the touch sensor to signal it was at its home position. Using this method meant that the distance to any square would be constant and could be calculated using the rotation sensor.

3.4.7 Structure

The next step was to build a platform for these beams to be attached to and that would move them to any row of the chess board. There were two basic designs for this platform; first there was the idea of building a stationary track running nearly 12 inches above the ground which the grabber would move across. This structure would run the length of the board and two motor drives would run along the track allowing the grabber to get to any row.

The second idea was to have the motor drives running along a set of tracks that were alongside the board. This structure would hold the horizontal running tracks above the ground where the lifting model would run across them.

3.4.7.1 Stationary Model

The main benefits of the stationary model were its stable structure and simple design. On each track a single motored drive would move along to whichever row was needed by the grabber. Using this method the drive had only to bare the combined weight of the horizontal tracks and the grabber. Compared to the other design which had to move the whole weight of the total structure this was an immense improvement.

The reason that this model wasnt prototyped entirely was due to its impedance of the players area of movement. A basic model was built to give the team members a general insight to its design before a more solid effort was made. The team agreed its giant structure that ran up by the end of the chess board was a big distraction for the player. Its looming presence and unsightly design was deemed unacceptable by the team and was discarded.

3.4.7.2 Vertical Movement with Tracks

This design was to move the full structure along the length of the board so that the grabbing unit could reach any of the columns. It runs along two tracks set at the side of the board using a motor drive which the struts, described above, are attached to. Each side was attached to its own motor and gear train. This set of gears has a gear ratio of 5:1 as can be seen in figure 3.13. The reason for this is so that the drive moves slowly enough as to not shake the whole structure too much. To measure the distance traveled by this drive a light sensor is attached

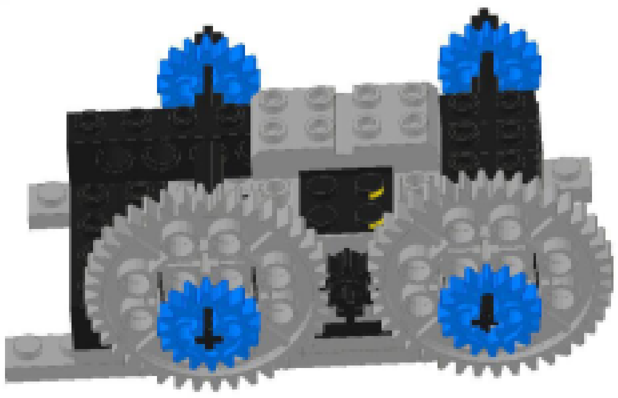


Figure 3.13: Picture of drive

at the base. This light sensor is pointed at the ground between the tracks where it reads black lines off a strip of white paper. This strip of paper runs the entire length of the board with a horizontal black strip every 2 inches. The light sensor reads the transition from white to black and can tell which row its at by counting the number of transitions. When the required count is reached the robot knows it has reached its destination.

Through testing it was found that certain factors reduced the sensors ability to read transitions. On a bright day the sun light can interfere with the values that the sensor picks up and it can fail to read a transition. It was also found that if the RCXs battery was low the sensor lost some of its accuracy to distinguish light values. To overcome this problem the light sensor had to be boxed in and positioned 3mm from the ground so that no background light could interfere with the readings.

3.5 Stopping the Structure

Both motors to be used in the above design were matched using the technique described in Section 3.4.4. This was due to the need for both motors to work at the same speed to stop skewing between each drive. Each drive had a rotation sensor to measure the movement of each square along the track but each had to have a designated starting point. All movements had to be measured from this point and a sensor was required to detect when it had returned after its move. A single touch sensor was then attached at one end of the track to resolve this.

In theory this single sensor method should have worked as both legs would start and finish at the same time due to the motors being matched to the same rotation speed. This meant that the single touch sensor would have detected when both legs had reached their starting point again. In practice it showed that with the added weight of the grabber always resting on one end meant that at the beginning of the move the opposite side moved faster. This meant a little skewing as the robot commenced its move but this effect decreased as the grabber moved to its desired column.

To correct this problem another touch sensor had to be stationed at the starting point of the opposite leg. The problem with this was that the RCX had used its maximum amount of inputs so there was none left to connect it to. This was fixed when Konstantinos designed and built some circuitry to act as a multiplexer for both touch sensors. The circuitry and design can be found within chapter 7.

3.6 Final Design

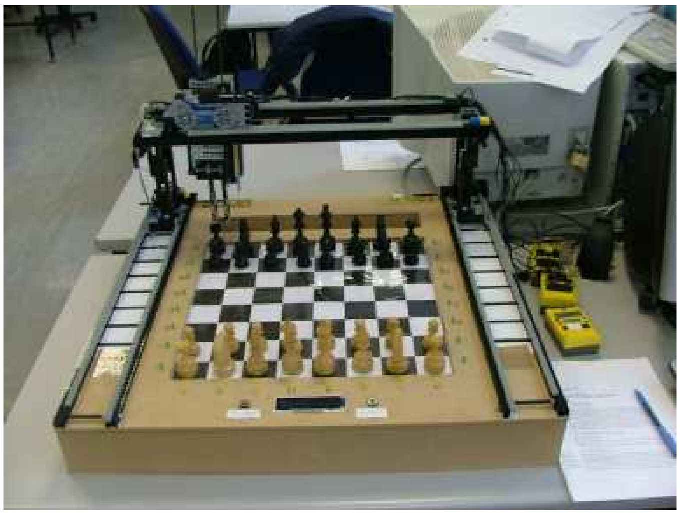


Figure 3.14: Final System

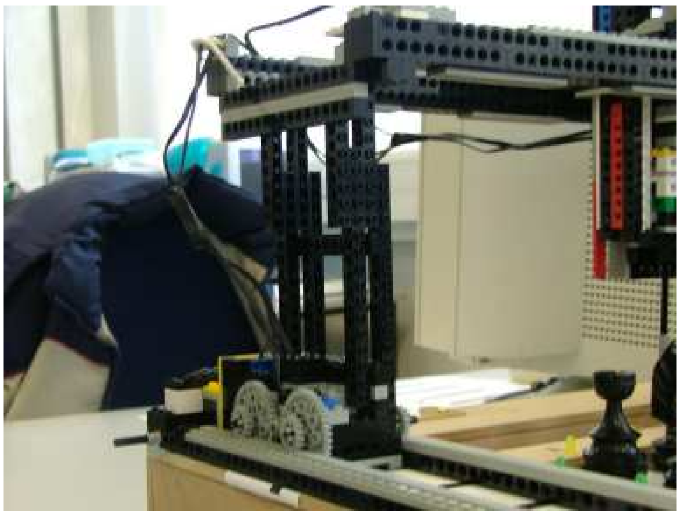


Figure 3.15: Left supporting strut of robot

3.7 Future Improvements

Areas that could be improved from the prototype include introducing a second motor to the grabber. At the moment the grabber can fail to lift properly as the battery power gets low. This is due to the friction applied by the sides which keep it stable after constantly being raised through all the testing that the team used. This was not picked up by the testing team until the two weeks before the final deadline which seemed too close to change the design.

The basic structure for holding the whole robot can also be reinforced to make it a little stronger. At the moment there is a negligible shake as the structure starts its movement from a resting position. The poles used in the lifting mechanism for the grabber could also be changed for stronger ones which allow no movement. This would mean the grabber would lift up straight and have no shake as it climbs to its peak position. It has also been discovered that the poles are one centimetre too long at the bottom and can sometimes touch large pieces like the king. It has never knocked it over in testing although this could happen.

Chapter 4

Board Design and Construction

4.1 Introduction

This chapter documents the design, implementation and testing of all aspects of the chess board required for the human player to use when playing against their robot opponent. It discusses the major design decisions taken throughout all stages of the development including board prototyping, LED circuitry and board construction.

4.2 Board Requirements

The requirements of the board are large and varied and extend far beyond the construction of a simple wooded box, though that is needed. The primary purpose of the board is to detect the human's move of a physical chess piece and convert this into a form that the chess software can understand, while not hindering normal play nor forcing the human to use the computer in any way. The board is the only facility the human player has to interact with the chess engine and so methods for delivering all the required information to the user has be developed.

4.3 Lego Vs Phidgets

As earlier discussed the computational unit of a Lego Mindstorm kit is a yellow block called an RCX. It was this unit that the board design initially utilised, but early on in the project another solution became available. This new kit, called Phidgets, aimed to supply physical representations of software widgets, hence the name, and supplied an alternative to using the RCX .

Similar to the RCX, the Phidgets Interface kit comes with a number of different sensors and outputs such as light sensors and motors, but as well as these the Phidgets has a two row LCD that can be run from a USB port. Also included in the Phidgets kit is a voltage sensor that can be attached to one of the analogue inputs and operates in the range of -30V to +30V. The Phidgets kit has far more analogue inputs, eight, as opposed to the RCX's 3, as well as having eight digital inputs and eight digital outputs neither of which are offered in the RCX. These additions allowed the creation of different, and in the end more sophisticated, designs than would have been possible with the Lego RCX.

From early on in the design process of the board the Phidgets kit was identified as the hardware best suited to the application. As well as the improvements mentioned above the Phidgets also used USB to connect with the computer, as oppose to infra red, and facilitated high level software handlers using languages such as Java.

4.4 Detection Methods

With the primary requirement of the board being the detection of the user's move, the first design decision that had to be made was how this move would be detected. A number of possibilities are discussed below:

4.4.1 Visual Detection

This method transfers the detection of the user's actions to outside the board, but is discussed here so it may be compared against competing methods. Visual detection utilises a satellite camera above the playing area to continually monitor the state of the board. It works by analysing the current and previous pictures of the playing area for changes in the position of pieces. It would be able to detect that a white piece was occupying a black square and, knowing the initial state of the board and the moves that have gone before, work out what type of piece it was. It would also be able to detect that a square is no longer occupied and thus track the movement of pieces throughout the board.

This method has a number of attractive features. The first is that a game of chess would continue entirely as normal. The human player would not need to press any buttons or signal to the computer in any way that they have finished their move. Another key advantage is that a standard board and pieces could be used. Other methods require specialised pieces or playing areas that call on skills not covered in the course, such as wood work.

However, this method is far from perfect and there are a number of key points that made this design not a good choice for satisfying the requirements. The first is that a satellite camera would have to be firmly attached high enough above the board to capture all the playing area. This would require some form of scaffolding and this was deemed unsatisfactory because of the risk it would distract or intimidate a normal human player. The method is also susceptible to changes in light conditions as well as posing great difficulty in resolving the pictures into the state of the board. Because of this other solutions where sought.

4.4.2 Light Sensors

This detection method, like all other methods detailed below, employs a number of sensor inlaid or underneath the playing area. In this variation, light sensors would be placed under every square of the chess board and would detect when a piece is occupying a square because the piece would cover the sensor and thus alter the sensors output. This method would not require any detection equipment outside the board case itself, unlike the visual method, though the detection equipment would still be visible as the surface would have to allow light to pass so readings could be taken. This also would mean that the performance of the system could be altered by external light conditions and may need recalibrated every day. Due to these weaknesses this method is also unsatisfactory.

4.4.3 Magnets

A magnetic solution is an interesting detection method that could be expanded to not just detect the state of the board but also to move the computers pieces in response. A magnet could be used to move the pieces around the board playing out the computers move, though this chapter is only concerned with the detection method. This would require chess pieces with magnetic bases, possibly even using different magnetic poles to represent the two different colours. These pieces, when placed on a square, would alter the output of that squares sensor and comparing the current state of the board with previous boards it would be possible to work out the players move. Such a method would be less susceptible to external conditions than the previous methods and would also allow for more of the detection mechanism to be hidden within a solid board. This approach does have a number of problems in practice. The first is the difficulty in finding suitable magnetic sensors. A magnetic sensor in its simplest form could be an open switch which is closed when a magnet is brought near, a Reed switch, but this device has no way of detecting different poles and it is a rather crude device. Electronic Magnetic sensors are also available but they are very expensive, starting at around 1 individually, and 64 would be needed. There is also a problem in containing the magnetic field of each piece to within the confines of a square. Without careful consideration, surrounding pieces could trigger the sensors under squares that are not occupied. The magnetic solution was an improvement on any technique considered previously but was still lacking.

4.4.4 Switches

The sensors that detect a piece or a move need not be complex; simple switches or touch sensors could be placed under each square and from their output the state of the board could be calculated. Switches are cheap and easy to come by, making them good candidates for detecting the state of an individual square on the board. Switches could be used to implement two different detection methods. The first uses sensitive switches that are depressed when the piece occupies the square. Taking the example of a game that has just started there would be 32 switches depressed, one for each piece on the board. This is a large amount of data to handle yet during a move only a few squares will be affected making most of the data redundant.

The other method uses the same principal as travel chess where the player presses down on the piece they are about to move, triggering the switch, and then moves the piece and presses it down again at its destination. Using this method every move can be constructed from two button press, the source and destination squares, limiting the amount of data that must be interpreted during a single move. This method does slightly affect how the human plays the game, in that they must press down on pieces when moving, but there are many positive points. The switches are hidden from view and are unaffected by external changes in heat, light or small magnetic variations. They are also cheap and readily available. For these reasons we adopted this method for detecting the human's move.

4.5 Analogue Vs Digital

Having decided what detection method was going to be used in the design the next step was to decide how to use the switches because the Phidgets Interface Kit has both digital and analogues inputs.

4.5.1 Digital

To individually number every switch using binary would require six digits which is possible using the Phidgets Interface kit digital inputs. This would leave only two free digital inputs for implementing other features such as user buttons on the board. During the early stages of design two free inputs would have been sufficient to implement all proposed features, though the finally accepted design used three digital inputs. For more details on why an extra digital input was required please refer to section 5.12 of this report.

A digital solution would involve feeding the digital output of all 64 switches into an encoder. Because you can not easily or cheaply buy a 62-to-6 bit encoder a number of smaller encoders would have to be cascaded together to create the functionality of the desired large encoder. Although a digital solution would be more resilient to interference this solution was abandoned because of the difficulty in the digital circuit design. It was difficult to find any suitable encoders to use in this circuit. There would also be a problem if the switches where numbered from 0 to 63 because there would then be an uncertainty as to what the output from the circuit would be if no switches were being pressed because zero output represents the first switch. A solution to this problem would certainly use at least one more of the digital inputs leaving only one input for adding user buttons.

4.5.2 Analogue

An analogue circuit was used to express the switch output to the Phidgets kit. This solution works on the principle that when no switch is pressed the analogue input would be zero, but when a switch was pressed the voltage seen at the input changed corresponding to the exact switch that was depressed. This could be done by using the Phidgets voltage sensor or by using the analogue inputs directly. Using the voltage sensor required an external power source ranging from +30V to -30V to exploit the full range of possible outputs available while still having it grounded in relation to the Phidgets kit. Because of the need to use an external power supply with such a large voltage difference the analogue inputs were investigated with the aim of using their power, ground and input lines directly.

4.6 Investigating the Phidgets Analogue Inputs

Before designing the array of switches that will have to detect a user's move, the internal workings of the Phidgets analogue inputs had to be determined. This was first done by consulting the data sheets and technical specifications found on the phidgets website, www.phidgets.com. The specifications of the light and force sensors were analysed because these are the simplest types of sensors used by the phidgets interface kit, 2-pin Resistive sensors. It was discovered that their resistance varied proportionally to what they were designed to measure and that both sensors had a minimum resistance of 500Ω.

This data was used to construct a simple "sensor" built using a 46KΩ trimmer variable resistor in series with a 1KΩ resistor. Before this test sensor could be connected to the Phidgets kit the three input pins of the analogue inputs had to be examined to determine what pins carried the power, ground and input signals. This was done by connecting the Phidgets Force sensor to the interface kit and analysing the signal on each line using an oscilloscope and multimeter. It was worked out that the lines from left to right (when looking at the connectors face on) are ground (0V), power (5V), input (voltage over variable resistor 0-5V) as seen in figure 4.1.

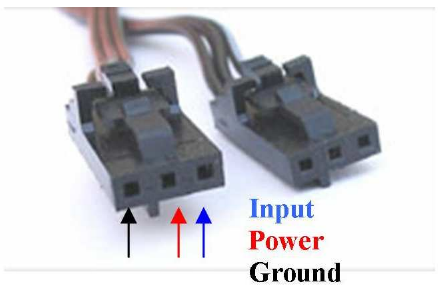


Figure 4.1: Analogue sensor connector for the Phidgets Interface Kit

When wiring up the "sensor", coloured wire was used to ensure the lines were not mixed up; black and red were adopted for ground and power, as is convention, and blue was used for the input line. This colouring scheme is consistent throughout all the circuitry in this project.

Now, with the sensor wired up, the next task was to investigate the accuracy of the Phidgets Interface Kit and the software that accompanies it. The output of the Phidgets kit can be handled by a number of high level languages including VB and Java. Although Java is the chosen implementation language for this project an already created VB program that came with the software was used to take readings from the input. The analogue signal was represented as a value between 0 and 1000 implying it has a resolution of 5mV, presumably a 10 bit Digital-to-Analogue Converter (DAC). This accuracy is far finer than the necessities of the board.

4.7 Switch “Sensor” Prototyping

To test the switch method of detection a five switch sensor was created. In series between the power and ground lines, were 5 resistors with different values to simulate different squares on the chess board. In light of the findings described above 610Ω resistors were used to create a potential divider. When no switch was pressed the output of the sensor, the input to the Phidgets, was unconnected but when a switch was pressed the voltage at that node of the circuit was seen at the output. Because of the input’s high impedance the voltage at the node was not affected by the switching so all that changed in the circuit was that the output line was raised to the nodes voltage. The first sensor when pressed connected the output to the supply voltage and so the output seen by the computer was 1000. The lower switches simulated lower squares with each node having a potential difference of 1/64V from its adjacent switch. This triggered readings from the computer with approximately a difference of 15 between switches.

4.8 Bouncing

Extensive testing was done on this simple circuit to make sure the design was sound. The accuracy of the computer readings and switching method proved to be satisfactory with a reading varying only one or two units from its theoretical output. What was also noticed was that the switches bounced a great deal. A switch bounces when it makes a poor contact with the connecting metal. This can happen when it is pressed down or when it is released and the output response of a bounce is that surrounding the peak, correct, value there can be intermediate values. These values often differed by only 5 units from the theoretical value but in extreme cases these values could be as little as half the desired output. As mentioned, bouncing only happens during a state transition and because it has clearly defined characteristics it is possible to de-bounce a switch within software. The software de-bouncing approach was adopted. More details of the software construct for switch de-bouncing along with a discussion on other software components of the board can be found in section 5.7.

4.9 3x3 Prototype

The simple 5 switch sensor demonstrated that the analogue switched detection method worked in principle but to fully test the design and to realistically simulate the intended setup a 3 by 3 switch circuit was created on a bread board.

From previous discussions with the members of the team in charge of the robot it was decided that each square on the chess board was to be 2 inches wide to allow for the inaccuracies of the lego kit. The new prototype tried to accurately represent what the final circuit would look like. Each of the buttons where spaced 2” apart, with each row of the prototype representing three consecutive switches in a different part of the full sized chess board. During this prototype a numbering system was adopted to label the switches in the full board. In this numbering system the top left switch was numbered 0 and the numbers were incremented from left to right

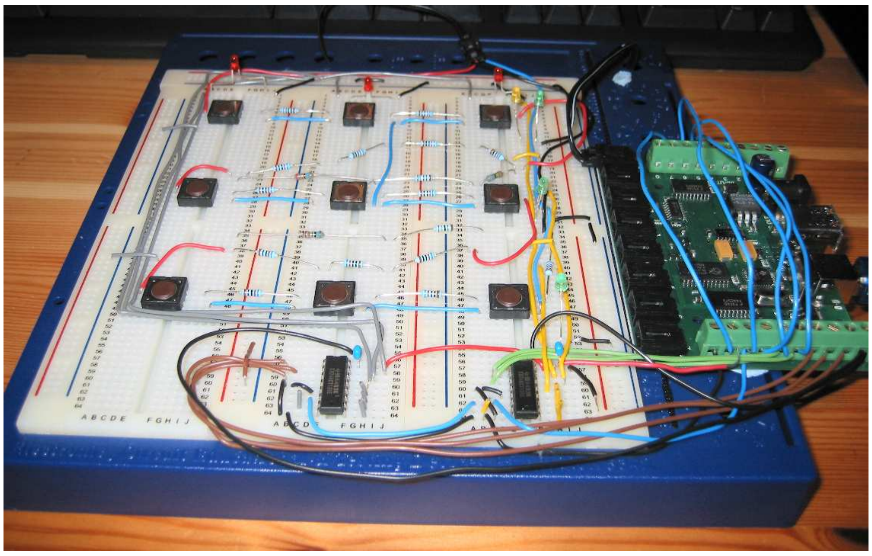


Figure 4.2: The 3x3 Prototype used to prototype the board circuit

across a row and rolling over to the next row. The bottom right switch was thus numbered 63. When switch 0 was depressed it connected the output to the supply, thus simulating the top left square on a full size chess board. The middle row of the prototype simulated the middle switches, numbers 31, 32, 33 on the full size board, where the bottom row of switches on the prototype simulated the bottom right three buttons on the full size board.

To keep this prototype as accurate to the final plans as possible the test software used was written in Java. More details concerning the development and implementation of the board's software can be found in chapter 5.

4.10 Interference

For many weeks the 3x3 prototype stayed in the bedroom of the team member in charge of board design and construction. During these weeks the design was improved, LEDs were added, the software developed and the board responded as expected. The team regularly met with their advisor to discuss the progress of the project and to demonstrate any completed or operational components. It was before one of these demonstrations that a serious problem emerged with the board's circuit design. The prototype was operating normally but to check that the code could be easily run on a different computer all the files where transferred to another computer in a different room. When the prototype was connected and the software run in these new surroundings it did not respond as expected. Instead the prototype's output responded as if someone was repeatedly hitting the lowest two buttons. This response flooded the Phidgets Interface Kit with data making it unresponsive to a humans occasional button press. This was a major problem that threatened the entire design of the board meaning weeks of work and research would be wasted.

4.10.1 The Cause

As mentioned above the prototype consisted of nine buttons spaced 2” apart, each buttons having a common output line. If the length of wire used in this output line was calculated the result would be approximately 24” of copper. This, it was discovered, was a sufficient length to act as an aerial for a large number of transmitted signals. The reason that this problem emerged when it did was due to the change of surroundings when testing the prototype. Within the room it was developed, the number of transmitted signal is at a normal level, but when moved to the new room it sat close to a wireless LAN router. It was the strong signal produced by this wireless device that caused the symptoms witnessed. The wireless signal was being picked up by the output line of the prototype and this induced a voltage of great enough magnitude to register as a low button press, when in fact no buttons where pressed. This was a great problem as the final board would be a 16” square needing an output line of at least 126”, much greater than the nine button prototype that was detecting this error. After consulting Prof. John Weaver in the department of Electronic and Electrical Engineering it was worked out that the board would pick up FM radio and TV transmissions among other interference.

4.10.2 The Solution

One solution to this problem would have been to abandon the design and redesign the circuit mindful of the problems witnessed but the team was reluctant to totally change the design many weeks into the project. It was thus necessary to find a solution to this problem for the existing design.

One obvious solution would be to shield the circuit from the external interference. This would definitely cut out the systems sensitivity to interference but, as was pointed out by Prof. John Weaver, if you are not careful when installing the shielding you can make matters worse not better. An example of this is if you connect multiple shields together, as would be needed to cover the entire board, then you must take care when grounding the shields. If you ground the shields in more than one place you can create a potential difference resulting in current flow through your shield from one ”ground” to another ”ground”, causing problems with other grounded circuitry. In the end the circuit was not shielded because of the way it was going to be installed in the final case. The switches need to be near the surface of a flexible playing area in order that a player pressing down on a piece can easily trigger the appropriate switch. As a result to fully shield the circuit a very thin and flexible shielding material would have to be used between the flexible playing area and the switches. A grounded piece of kitchen foil would supply a limited shielding effect but it is very fragile and would not stand up to continual play. Even if the foil or some other shield was installed that could stand up to the demands of the chess board, adding an extra layer of material between the switch and the playing area would reduces the haptic and audio feedback the user receives, that is to say there would be no satisfying click when pressing down on a button.

A solution which was adopted was to reduce the size of the circuit. Instead of having one circuit with 64 switches and a common output line the board was split into quadrants each made up of an independent circuit with only 16 buttons. This had an effect in two ways. Firstly it quartered the output lines and thus reducing the size of the effective aerial. This did made a difference but the four smaller output lines were still 28” in length (2 foot 4 inches). This is still sufficient to pick up a whole host of external signals because this length is approximately 2.3 light nanoseconds, or a quarter wave at about 100MHz. With the extra zig-zags of the output line it will pick up even higher frequencies such as commercial FM radio that transmits at 100MHz. It should also be noted that computers themselves radiate at all frequencies. A more useful effect this change had on the design was to increase the voltage division between each node. This meant that between each node, and before the first one, there was a larger margin where interference could affect the signal with no overall effect on the circuit.

As well as physically changing the circuit to combat the effects of interference the software was also changed in light of the problem. The software is discussed in greater detail in chapter 5.

4.10.3 Problems in the Future

The actions discussed above do limit the effect that interference has on the circuit but it has not solved the problem totally. It should be noted that this problem can still occur in areas where interference is particularly bad, an example of such a situation would be in the same room as a wireless router. With the aid of hindsight, if another board circuit was to be designed, a digital design may be a more resilient design. An improvement that could be made to the existing setup would be to shield all sides of the case excluding the playing area. This would not supply total shielding but could reduce the likelihood of interference becoming a problem in the future. This has not been implemented in the current board because the actions described above were sufficient to stop the problem, for now, while still keeping the cost of creation as low as possible.

4.11 LEDs

As stated in the board requirements, the board is the only contact the user has between the chess engine and associated software. Because of this the board has to be used to communicate messages from the chess engine to the user. From early on in the design, the suggestion of using lights or LEDs to highlight the squares on the board had been adopted as the best way of communicating to the user their opponent’s move. It should be noted that in the final system a text LCD and appropriate sounds supplement the LEDs but these are seen as extra requirements and in the initial design and prototypes these were not implemented and all communications to the user came through the LEDs.

4.11.1 Row and Column LEDs

It was with the 3x3 prototype that LEDs were first tested in conjunction with the circuit for the playing area. Before this prototype, LEDs had been run directly from the Phidgets Interface Kit's digital outputs. This was not an acceptable solution because the Interface Kit only had eight digital outputs, far fewer than the number of LEDs required to highlight a square. A square was to be highlighted by individual row and column LEDs being turned on to indicate what square attention should be drawn to.

To control 16 LEDs individually, 8 row and 8 column, required at least one decoder. A 3-to-8 line decode was used to control the row and column LEDs. This limited the functionality of the LEDs because only one row and one column LED could be on at any one time, but this was sufficient for the requirements. The first challenge was to find two suitable decoder chips. After some research it was discovered the Electronic and Electrical Engineering Department's Stores has a wide range of integrated circuit logic available for our use. In the end, two 74HCT138E decoder chips were used because they performed the appropriate logical function, 3-to-8 line decoding, could be run off a supply voltage as little as 2V or as large as 7V making it very adaptable to any circuit design, and cost only 18p.

4.11.2 The Decoder

The 74HCT138E decoder chip comes in a 16 pin package with 3 input pins, 8 output pins and 3 enable pins as well as power and ground pins. A pin diagram of the chip is included in figure 4.3. There are three enable pins, two of which are active low, E1 and E2, and one active high

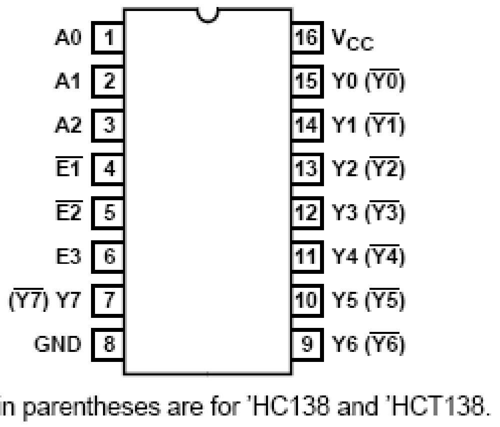


Figure 4.3: Pin Diagram extracted from the Decoder's data sheet

pin, E3. When installed in the circuit E1 and E2 were permanently connected to ground, GND, so that only the E3 pin was used to enable the chip. This pin was connected to digital output 6 of the Phidgets Interface Kit. The first three digital outputs of the Phidget were connected to the input lines of the column decoder and the next three, outputs 3, 4 and 5, were used as inputs to the row decoder.

4.11.3 Prototype Circuit

For the first prototype, three small LEDs where connected to the least significant outputs of each decoder. The decoders were powered by the last digital output of the Phidgets Interface Kit, output 7. It is usual practice to wire the LEDs in series with a resistor to limit the current flowing and protect the LED. This was not necessary because the Phidgets outputs were themselves current limited to make it safe to directly connect an LED to the outputs. Because the decoders outputs are active low, the anode of the LED had to be connected to the supply of the decoder, Vcc, and the cathode was connected to the decoders output. This

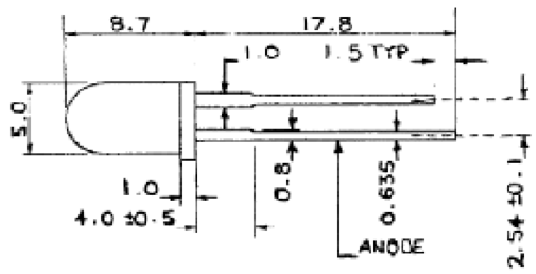


Figure 4.4: Diagram extracted from the LED data sheet

meant that when the LED was not on, both pins of the LED were raised to the supply voltage. When the output line of the decode was selected by the digital output of the Phidgets, the line dropped low creating a potential difference and the LED was illuminated while current flowed into the output pin of the decoder. This was not satisfactory because small fluctuations in the supply meant that the LEDs would flicker from time to time. To fix this problem an inverter was used to make the outputs of the decoder active high. As a result when the LED was not turned on both pins were grounded and when turned on the current flowed out of the decoder.

4.11.4 Power Supply Changes

When the LED control circuit was tested it operated correctly but it was noticed that the LEDs were rather dull. Small LEDs were being used which would not be sufficient in the full board, so they were replaced by larger 5mm LEDs which would be seen easily surrounding a 16" square board. When the LEDs were replaced, the circuit failed to illuminate the larger LEDs. Using a multimeter to analyse the voltages within the circuit it was discovered that the voltage of the digital output being used to power the chips and LEDs dropped to under 2V when the chips were enabled. This was because the outputs could not supply the necessary current to drive the two decoders, inverter chips and the illuminated LEDs. To remedy this problem the circuit was powered directly from the power adapter that supplies the Phidgets Kit. This supply was 6V DC and so resistors were needed to limit the current and protect the LEDs. To calculate the values of the resistors needed key information about the LEDs was extracted from the datasheet. The LEDs operate at a maximum constant current of 30mA and dropped approximately 2.5V when lit. This required a resistor to drop 3.5V at a normal operating current of 20mA and so 180 Ohm resistors were chosen. This new power supply also allowed for two LEDs to be run in parallel off a single output, allowing row and column LEDs to be placed at each end of the board.

4.12 Circuit Layout and Construction

Once the prototypes proved conclusively that all the circuitry would perform correctly the final board had to be designed. The LED circuitry had been placed on vira board during the prototype design and so there was no need to change this circuit as it was operating correctly. It was decided that printed circuit boards (PCBs) would be used for the board quadrants because of their size and the need for the switches to be accurately spaced 2" apart. The circuit design software suite Orcad was used for laying out the quadrant circuit. To reduce the complexity of making the circuit the quadrants were designed identically and space was left between them so wiring could be fed from below.

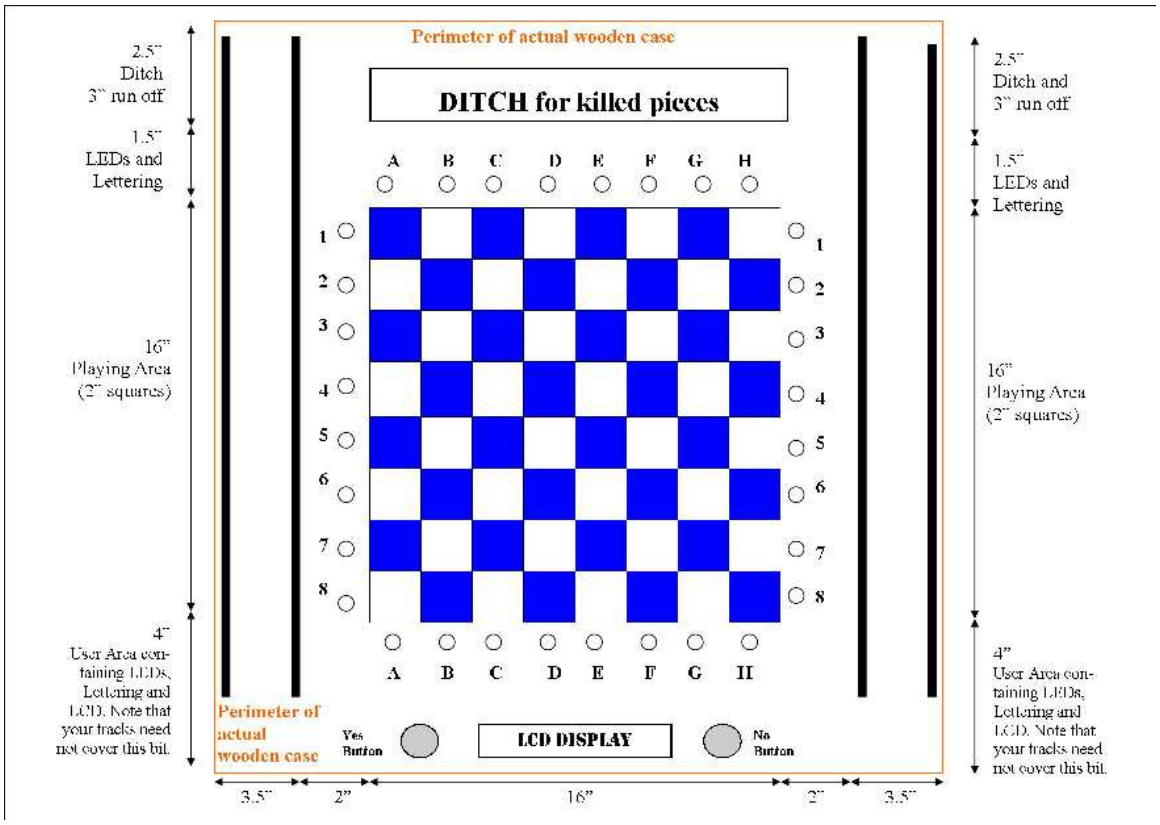
The first step was to create a schematic for the circuit with Orcad Capture, so that it could then be used to produce a PCB layout with Orcad Layout. Using Orcad Capture the schematic was created with 16 switches, 15 resistors and 1 3-pin connector. The 16th resistor was left off the PCB to keep the possibility open of using a variable resistor to calibrate the circuit in high interference situations. In the final version a standard resistor was used as the 16th resistor and was added to the circuit just before the connector used to attach the circuit to the Phidgets' input leads.

Once the schematic was complete a net list was generated and exported to Orcad Layout where the PCB could be laid out. The footprints of all the components were taken from the Project_Footprints library supplied to students by the Electronics Workshop in the Electronic and Electrical Engineering Department so that the PCBs could be adequately created in-house. The "Dimensions" tool was used to maintain the 2" spacing between switches. Initially standard tracks were laid out over the board but the use of wide tracks was advised by the Electronics Workshop. This is because the acid used during the etching process can run over and miss the tracks when there is a large space between them, as is the case in this PCB layout.

After the PCB layout was completed the layout was printed onto tracing paper and handed into the Electronics Workshop for etching. Once the boards were created all that was left was for all the 64 switches and other components to be soldered in to position and an appropriate connector to be created to attach the circuit to the Phidgets Kit.

4.13 Board Construction

Now that all the components were completed it was time to build the case to house the playing area and Phidgets kit. A few dimensions had already been decided such as the size of a square on the board, but other measurements had to be discussed with those responsible for creating the robot. After discussion and analysis of the robot prototype the following schematic was design. In short this required a case that was 27" square, and around 3" high to contain a



Perimeter of actual wooden case
DITCH for killed pieces
A B C D E F G H
1 2 3 4 5 6 7 8
LCD DISPLAY
Yes Button
No Button
Perimeter of actual wooden case
2.5" Ditch 3" run off
1.5" LEDs and Lettering
16" Playing Area (2" squares)
4" User Area containing LEDs, Lettering and LCD. Note that your tracks need not cover this bit.
2.5" Ditch and 3" run off
1.5" LEDs and Lettering
16" Playing Area (2" squares)
4" User Area containing LEDs, Lettering and LCD. Note that your tracks need not cover this bit.
3.5" 2" 16" 2" 3.5"

Figure 4.5: Diagram showing the layout of the top of the board

suitable ditch for the pieces. Such a case was created out of 7mm thick MDF, using pine blocks to support the corners and to raise the PCBs. A pine strut was also inserted under the MDF between the ditch and the playing area to give extra support. The wood was held together using wood glue and wood screws to give a solid join. The 16" square playing area was cut out and replaced with medium thickness clear plastic. Paper with the checked patter was then attached under the plastic. To allow for easy access to the circuitry contained within, the top of the board was hinged. Because no soldering irons are allowed in the Project Lab where this board was going to stay, all components were made so that they could be easily swapped out. The LEDs are held in place using screw connectors so that if one was to blow they could be replaced easily and quickly with just a screwdriver.

While the final board was being constructed a paper replica of the top surface was created so the robot team could continue in their work knowing exactly the dimensions of the board.

4.14 Testing Outcomes

Testing of each individual module was carried out within the team but summative testing was also carried out using testers who had no previous knowledge of the project. The summative testing yielded a number of interesting points about the boards design and construction.

Users found no difficulty pressing the switches under the playing area, though sometimes the software resulted in those presses not being picked up. It was also noted that the performance of the switches decreased with prolonged use. The switches are sensitive mechanical components that operated perfectly during the in team testing but with the battering they were put under when being tested by other people their performance became noticeably poorer. This performance deterioration is not serious enough to threaten the board's usefulness presently though if it was used continually for many more months the switches would need replaced.

Another point raised during testing was that when testers took an opponents piece they did not know where to place them. This was problematic when they placed them in the path of the robot or in the ditch because the ditch is not big enough to hold all the pieces in the board. A solution would be to build in some box or ditch for users to keep those pieces that they have taken.

On the whole, the board fulfilled all its requirements allowing the players a way of playing against the robot and informing them of the computers actions via LEDs and the LCD.

Chapter 5

The Board's Software

5.1 Introduction

The Board is useless without software running on the attached computer to control and interact with it. The Phidgets Interface Kit can interact with programs written in a number of high level languages but for this project the team decided to use Java as this is the object orientated language that is primarily used in level 3 computing. The Phidgets kit supplies an interface between the real word and the computer and so it has one major difference to the RCX; the program that the hardware is interacting with is running on the connected computer because the Phidgets has no microprocessor that can be programmed nor memory that could hold instructions. This chapter documents the development of the software that is used to communicate with the board, from its early design and use in prototypes, to the multiple class structure that is used in the final setup.

5.2 Requirements

The software brings together all hardware components and marries them with the chess engine which is orchestrating the computer's response to the user's moves. The software must capture the information recorded by the Phidgets Interface Kit when a button is pressed down on the playing area and decipher it to represent a square on the chess board. It must also inform the user that it has picked up the button press properly. The software must then build up moves from individual button presses and pass them on to the chess engine. Once the chess engine has returned the computer's counter move the software must highlight that move to inform the user of the computers actions and then wait the user's next move.

5.3 The Phidgets Software Components

The Phidgets Interface Kit comes with a number of software components for utilizing all the functions of the kit. The first task was to discover the function of all the components and how they may be used to greatest effect. The software comes in two parts.

The first part is Phidgets.dll which must be installed on a Windows computer so the operating system can detect and communicate accordingly with the hardware attached.

The second and more extensive part of the software is the Phidgets package. Contained within this package are all the methods and software constructs that can be used by a Java programmer to communicate with the Phidgets components. This project only used the classes concerned with the Phidgets Interface Kit and the Phidgets LCD but there are many other classes for controlling servo motors, RFID tag detectors and Humidity sensors contained within the Phidgets package.

5.3.1 The `PhidgetInterfaceKit` class

The `PhidgetInterfaceKit` class is the simplest of the classes to understand. This class contains a number of standard methods including `Open()` that returns true if the kit was successfully opened and `GetNumSensors()` that returns the number of sensor inputs (analogue inputs) that can be used. These methods are useful for setting up the Phidgets Interface Kit but cannot be used for reading changing data from the kit, such as the value of the sensors input. For this task an event listener must be associated with the Phidgets kit.

5.3.2 The `_IphidgetInterfaceKitEventsAdapter` class

This class gives the programmer the ability to handle events triggered by the Phidgets Interface Kit. When software is written for the Phidgets Interface Kit an event listener is associated with the Phidgets kit. When an input change occurs an event is triggered in the software. It is the job of the event listener to listen for these events and to handle them appropriately. The events are handled by implementing the methods laid out by this interface class. The two methods that are useful for the boards applications are `OnInputChange()` and `OnSensorChange()`. `OnInputChange()` is the method that is called when one of the digital inputs changes state while `OnSensorChange()` is the method that is called when a change occurs at one of the analogue inputs. Both methods are passed the event and from this data can be extracted about the event using the correct methods. `get_Index()` will return the index of the input that triggered the event where as `get_SensorValue()` or `get_NewState()` can be used to find out the new value of the analogue or digital input respectively.

5.4 Early Programs

Early on in the project programs where used to test the response of the prototype hardware and little if any handling of the events were done. Initial programs were only concerned with the sensor inputs and simply outputted to standard output the raw value of the change, obtained using `get_SensorValue()`, and the index of the sensor that had changed. Once the 3x3 switch prototype had been created the next task was to convert the received raw values into button pressed. A large case statement was initially used but this was cumbersome and not easily adaptable so a formula was created to calculate the square number from the raw sensor value. The formula worked by knowing the difference between each switch and any initial calibration. Through a number of divisions and multiplications the formula was able to return the correct box number.

Basic error handling was also added to the software by informing the user when the change did not occur at the right sensor index.

5.5 Simple GUI

Even with the software outputting the correct square number it was not always easy to identify what square the software thought had been pressed. Because of this a simple graphical user interface (GUI) was created to make it obvious to the tester what the software was picking up. The GUI used the interface packages of java swing and awt. The window contained a grid of 9 squares laid out to represent the buttons of the 3x3 prototype. A bright green circle was displayed on the square that was last pressed. This GUI made the testing process far easier and more accurate as it was less likely the tester would miss an error.

Due to the complexities of chess and the demands of other parts of the project the GUI was not expanded and used in the final system. Because the board used LEDs and a LCD to communicate any necessary information a GUI in the final system is not entirely needed but if more time allowed a GUI would be a worthwhile improvement.

5.6 LED Control

Once basic square identification had been implemented in the early software the task of controlling the LEDs had to be tackled. Before the LEDs were incorporated into the program test programs were written to test the workings of the LEDs. The first such test program looped through all the "squares" by setting the correct digital outputs directly using the `SetOutputState()` method contained within the `PhidgetInterfaceKit` class. After the appropriate row and column LED was set the program slept for two seconds so that the illumination could be seen, before moving on to illuminate the next LED.

This test program worked correctly though it was evident that this method of setting LEDs could not be used in the final system because it required the main program to sleep while the LEDs were lit, thus holding up the main program. It also became evident that the program should be split up into a number of clearly defined classes. As a result the LedOutput class was created. This class extended Thread and so could run in parallel with the main program without holding it up. The LedOutput constructor had as parameters the interface kit and the row and column number of the square that was to be lit. When the thread was constructed it set the appropriate digital outputs, enabled the control circuitry and then slept for a set time before awaking and resetting the digital outputs and disabling the circuitry.

5.6.1 LedOutput2

After testing, a number of problems were discovered with the original LedOutput class and a new version, LedOutput2, was released. When using LedOutput the program occasionally crashed with an obscure error message, which after some further testing it was discovered was due to multiple treads trying to access and control the shared resource of the Phidgets Interface Kit. To combat this problem a new method called `settLed()` was created so that threads did not directly change the state of digital outputs but called this method to do that for them. This method was made thread safe by including static guarding objects within the class, one for each digital output used, and creating critical sections of code in the new method. Critical blocks are created in Java by using the `synchronised` keyword.

It also did not make sense to label switches by a number throughout most of the program but to set the LED by passing in the row and column numbers. In LedOutput2 the constructor was passed a box number and the row and column numbers where calculated by dividing or modulo dividing by 8 respectively.

5.6.2 LedOutput3

The improvements implemented in LedOutput2 were sufficient to stop the program from crashing but it was not performing as well as should be expected. It was noticed that if a button was pressed quick enough after a previous button press then the LEDs would light for only a few milliseconds before turning off again. This was obviously another problem concerned with managing the shared resources of the Phidgets. When a button was pressed a new LedOutput2 thread was sporned and only died when the LED had been on for a set length of time. The reason for this was because if two LED threads were running simultaneously the second would turn on its LEDs but the first would then turn off all LEDs when it died. LedOutput3 aimed to fix this problem by giving each new LED thread a unique id number and saving the id of the last thread to alter the LEDs. Using this technique `resetAll()` and `settLed()` were changed so that only the latest thread could alter the state of the LEDs. This meant that when two threads ran simultaneously the first would be unable to alter the outputs once the second thread was alive.

This improvement made LedOutput3 ready to be used fully in the software so it was expanded to work for not only the 9 squares of the prototype but for all 64 squares of the full size board. New methods were also added to allow the programmer to change the time the LEDs would be turned on for. LedOutput3 is used in the final system every time an LED is light.

5.7 Software Debouncing

As discussed earlier in section 4.8 switches are mechanical devices that are far from perfect and often make poor contacts, the result of which is that intermediate values appear during the state change. If not controlled, when these intermediate values create an event the software handler would treat them as separate button presses and highlight the square that the intermediate value represents, not the correct square.

To combat this problem switch debouncing was added into the software, at that stage an earlier version of the event listener called FullBoard1. FullBoard1 debounced the switches by time stamping every event. The time was taken from the system clock using the java Date object and then stored by the event handler. When another event occurred the current time was compared with time stamp of the previous event and if it was less than 200milliseconds after the last event its effects where ignored. This value was eventually dropped to 100milliseconds. The previous state was also stored so that the event with the highest value, the correct event, was recognized and acted upon.

Also to ignore small erroneous values the output of the Phidgets Interface kit was normalized using the `SetSensorNormalizeMinimum()` method contained within the Phidget `PhidgetInterfaceKit` class. This method sets the minimum setting of a sensor's range. This is usually 0, but when set to 30 in FullBoard1 the scale is adjusted so that the real range of 30 - 1000 is normalized to 0 - 1000.

5.8 GuardedLCD

As the whole system was being developed it was evident that the use of the Phidgets LCD would be of great benefit. Only three self explanatory methods, `Open()`, `SetBacklight()` and `SetDisplayString()`, are needed to operate the LCD. Initially, like the LEDs, classes that wished access to the LCD used these methods directly but with the increasing use of threads within the system these direct methods were exported into a new class called GuardedLCD which contained synchronised, therefore thread safe, versions of all the methods used. Now the classes only needed to call the GuardedLCD version of `SetDisplayString()` to display test to the user without having to worry about the threads currently running.

5.9 MoveGen

5.9.1 Normal Moves

So far all the software has been concerned with single square presses but it is necessary to construct these individual presses into moves so a new class called MoveGen was created. The main method in MoveGen is `generate()` which is called every time a valid event occurs on the playing area. If the button press is the beginning of a move, there is not a previous button press stored, then `generate()` simply stores the number of the square pressed and displays to the user that the press has been picked up as the beginning of a move. When called again `generate()` uses the previous square number along with the newly passed square number to create a move. These square numbers are then converted into the standard chess coordinate system of letters and numbers and then fed into the chess engine interface called Main through its `callengine()` method. This method eventually returns the computers counter move in the form of a string. This string must be parsed and the useful data extracted from it. In a standard counter move the string will be 4 characters long and in the form of “b8c6”. The user can be informed of this move by passing it to the LCD and it can also be split in to square numbers and highlighted using the LEDs. Discussion of the method used by MoveGen to highlight a move using LEDs can be found at section 5.10.2.

5.9.2 Special Moves

Sometimes the output of `callengine()` is not simply a four character string in the format shown above. If the move is an invalid move then the string returned is “Invalid”. When this happens the user is alerted to the invalid move and asked to move again.

If the counter move places the computers opponent in check then a ‘+’ symbol will follow the move notation. In this case the user is informed that they have been placed in check by the computer. A similar setup happens when the player is checkmated. The ‘#’ symbol follows the move and an appropriate message is displayed. After a number of seconds wait to ensure the move has been carried out by the robot the software closes the chess engine and then exits. There is a known problem with some checkmate conditions that is discussed in section 5.14.1.

The symbol ‘$’ is outputted if the user has checkmated the computer. This scenario has never been tested due to the difficulty of winning against the chess engine but it carries out a similar sequence of operations to that of the ‘#’ condition.

5.9.3 Cancel

The original designs of the system allowed a user to cancel a move. This is not strictly allowed in the chess rules but it was included in the design in case the board did not respond correctly and formed the wrong move from the users actions. As a result a cancel method is included in MoveGen. This method allows only limited canceling as a user is only allowed to cancel half a move, once the move has been passed to the chess engine it can not be undone and the method returns false. The reason the cancel method is not as powerful as the original design had intended is because of the complexity raised if the robot has already begun to move the computer’s pieces. To avoid this complexity a simple half cancel was implemented so that the user can cancel the first half of a move but not once the two squares have been pressed.

5.10 Extra Output from MoveGen

5.10.1 ThinkingOutput

When MoveGen was complete a new version of the event listener, FullBoard2, was created to communicate appropriately with MoveGen. When this new event listener was run, after being fully rested, there was a user orientated problem with the output. After the user had completed their move the chess engine spent some time thinking before returning its counter move. During this thinking time it was difficult to tell if the software was still calculating the response or if the software had crashed. To combat this problem a new output thread called ThinkingOutput was created. The job of this thread was to continually update the LCD when the software was calculating the response to show the user that the software had not crashed or hung up. This was done by creating a scroll bar which moved continually along the lower line of the LCD while the top line of the LCD displayed the text "Calculating Response". The scrolling bar was produced by displaying a character array the same length as the LCD. Three of the characters in this array where the symbols ' |' while the rest of the array was filled with blanks. Every time slice, a predetermined short interval of time, the characters within the array would be shifted along one place, the end characters being looped round to the beginning. The thread continued outputting the changing scroll bar until the `reset()` method killed it.

The result of this thread was that the user was never in doubt as to what the computer was doing at any time throughout the game.

5.10.2 LEDhighlightMove

The LEDs are not just for highlighting the squares that are pressed by the user but also for informing the user as to what squares the computers move involves. Original versions of MoveGen called LedOutput3 to highlight the individual boxes involved in the counter move but this required the program to be halted until the first LED was finished so that the both squares where lit up for an equal length of time. It was also best for the LEDs to stay on for a longer time when highlighting the computers move than when highlighting the users presses because the user does not know when to expect the computers response. For this reason a new thread called LEDhighlightMove was created to illuminate the moves of the computer. This thread in turn used LedOutput3 to illuminate individual squares while the main program was not required to wait for the LEDs to finish. When a new instance of LEDhighlightMove is created the on time for LedOutput3 is set to a longer duration and then the first square is highlighted using LedOutput3 to ensure all the actions are thread safe. LEDhighlightMove then waits for the first LED to turn off before setting the second LED. Once both LEDs have been illuminated the 'on time' of LedOutput3 is reset to its original value and the thread dies.

5.11 User Buttons

To allow the user to cancel a move or quit the game user buttons where added in to the board. Two buttons were added, a Yes/Begin button and a No/Cancel button. These buttons where wired to the digital inputs of the Phidgets Interface Kit and thus the event listener had to be expanded to include the method `OnInputChange()`to cope with events triggered by the changing digital inputs. Programming the response of the buttons was the most complex programming that was required when writing software. This was because the actions of the buttons change depending on the state of the system. An example is when pressing the Yes button, most of the time the yes button is ignored but if the user had been given the option to quit then pressing the Yes button will quit the game. Also adding more complexity is operations such as cancel that act on information contained within other classes, in this case MoveGen. Below is a simple state transition diagram for the two user buttons.

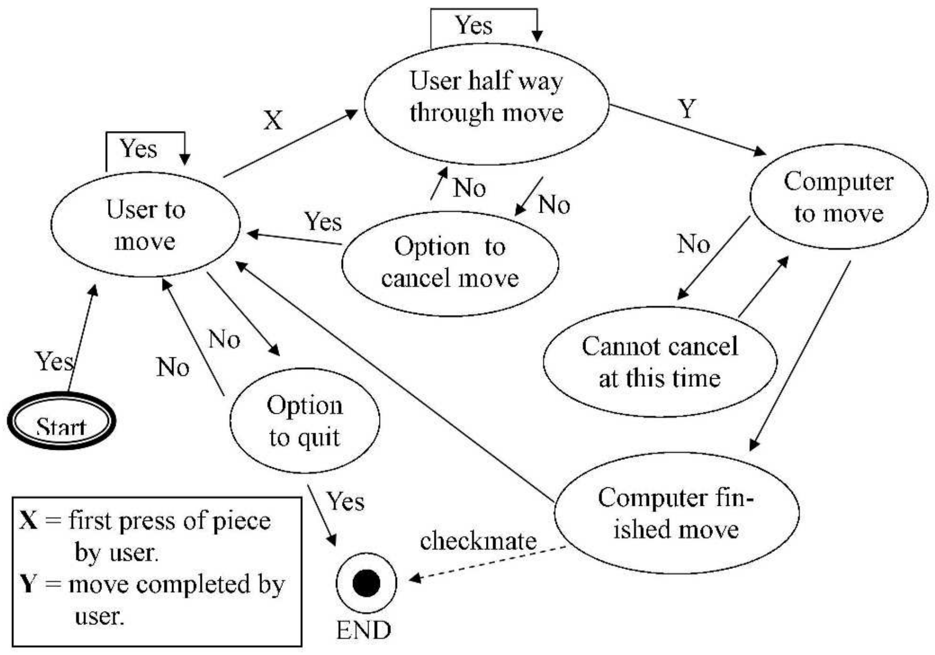


Figure 5.1: State transition diagram for the user buttons

5.11.1 The Yes/Begin Button

The Yes button is only used in two scenarios, to begin the game and to confirm the quitting from a game. The first scenario is signaled by the running of the WelcomeLEDs startup thread that will be discussed below. If this thread is alive then pressing the yes button will begin the game. To ensure a move cannot begin before this the playing area is blocked, all events are ignored, and the Yes button unblocks the playing area when starting the game.

The state variable quitOption is used to signal if the user is responding to the option of quitting the game. If this variable is set and the Yes button is pressed then the game is stopped.

5.11.2 The No/Cancel Button

The software handling the cancel button is more complex than that of the Yes button. If a move is in progress, when the user has entered the first square of the move but has not yet completed the move, then the state variable moveInProgress is set. When this state variable is set the handling software calls `cancel()` from MoveGen. If the cancel is successful then the method returns true and the user is prompted to move. If the cancel is not successful for whatever reason the user is told that they cannot cancel at this time.

If the variable quitOption is set then the user is responding to the option of quitting and so the variable must be reset and the user prompted to move.

Under any other scenario not mentioned above when the cancel button is pressed the user is given the option to quit which they can respond to using these two buttons. In this case quitOption is set by the software handling the cancel buttons press.

5.12 RobotMove

When the board, chess engine and robot's control software were integrated together an inconsistency was found between the robot's control software and the chess engine as to what constituted an individual move. For the chess engine a kill move was one where the destination square had to be cleared of an opponent's piece and then the computers piece moved to occupy the square. The robot's control software considered a kill to only be the removal of an opponent's piece from the playing area. As a result an intermediately class had to be used to break down the chess engines commands into commands for the robot. The problem was that the second half of the kill move, the moving of the computer's piece to the destination square, could only be sent to the RCX once the first half was completed. There was difficulty transmitting data from the RCX to the infra red tower long after a command had been sent so a solution involving a new button attached to the Phidgets Interface kit was devised. A new class, a thread called RobotMove, was added to boards software along with the new button added to the board. The new button was situated at the computer's side of the board, the robot's home. When a normal move is passed to the robot a new instance of RobotMove is created within Main, the chess engine's interface, and passes the information on to the robot's control method, Test3. During a kill move the first half of the move, the robots kill, is passed directly to Test3 and the state variable MultiMove is set by the FullBoard2 method `setMultiMove()`. The RobotMove thread then sleeps until the robot is home again, signaled by the clearing of MultiMove. When the robot is home the thread sleeps for a time longer to ensure the robots grabber has also finished. The time RobotMove sleeps after the robot has reached home is proportional to the column number of the square were the opponents piece was removed. The formula to calculate this time is

```
                    (destination%8) > 4
        ? (500 * (destination%8)) + 1000 : 500 * (destination%8)
```

and was derived from experimental analysis of the robot (times are in milliseconds).

5.13 Extras

5.13.1 WelcomeLEDs

The board's software can be run by typing the command "run" into command prompt but to allow the user time to set up the pieces and prepare themselves for a game the class WelcomeLEDs was created. This class extends Thread and makes a number of calls to LedOutput3 to create recurring sequences of light using the board's LEDs. This thread is terminated by pressing the yes button to begin the game.

5.13.2 Sounds

Audio feedback was added to enhance the players experience and to make the task of playing the robot simpler and more enjoyable. Sounds are played during key points in the game such as when a button is pressed on the playing area or when the user is placed into check or has entered an illegal move. A sound is played by firstly creating an AudioClip object and then using the `play()` method on that object. An audio clip is created by feeding the constructor a path to the selected clip. These paths were written out in full and can be found in constants at the top of the classes that use this function. Writing the paths in full made the program very difficult to move to a different directory but was done so that the program can be run from Development Environments such as NetBeans which changes the current directory.

5.14 Known Problems

5.14.1 Checkmate Kills

When the computer checkmates the user and wins the game the move is not always executed properly by the robot. The error occurs when a kill move is used to checkmate the user. The controlling thread RobotMove sends the robot the first half of the move, the removal of the opponent's piece, correctly. On some occasions the robot does not execute this final half of the kill move. It is thought this is due to the main thread dying before RobotMove can finish its job and so a large delay has been inserted after the sending of the first half of the move. This action has helped but not fully solved the problem. Due to time constraints and the limited scope of the problem a full fix was not developed

5.14.2 Normal Kill Moves

The infra red link between the RCX and the computer is into the most reliable of data communications and because of this it is susceptible to errors. A problem occasionally occurs when the robot only executes the second half of a kill move. This is due to the RCXs not being ready to receive another move command when it is sent. To remedy this problem an algorithm to calculate the necessary time between transmissions was developed but this is not always sufficient. If this problem occurs the user may finish the move for the robot and then trick the system in thinking the robot has moved by moving the robot out from home by a few centimeters and then returning it to home.

5.14.3 Pawn Promote

The ability for the user to promote a pawn to a queen is implemented and working correctly. The code for allowing the computer to promote a pawn is not fully tested. The move can be tested to ascertain whether a promotion has occurred but the handling of this, and the possibility that this may place the user now in check has not been implemented. This is only occasionally occurs in games of chess and so it has been left as an acceptable weakness in the cod. If time was available this problem would be remedied by changing the code that analyses the feedback retrieved from the Main function `generate()`.

5.15 Possible Improvements to the Software

- **Fixing the Known Problems**
  This would be the first improvement that would be implemented if time and resources where available.

- **Adding a GUI**
  As discussed earlier in section 5.5, a simple GUI was used in the testing of the 3x3 Prototype but not rolled out to the entire system. A GUI for the entire system would keep the user always informed displaying the current state of the board as well as the move and other useful information.

- **Sounds Only Play After Debounce Time**
  This is the single biggest problem that emerged from testing. The programmers were aware of this problem but felt that changing the design significantly, as would be required to fix this problem, would involve too great an effort invested in an extra feature. The problem is that the sound that plays when a button is pressed on the playing area plays when the button is pressed not when the software picks up the move after the debouncing time delay. Only the LEDs and the LCD are good representations of what the software has picked up. To fix this problem would require polling the interface kit's outputs after the 'down time' to find out if the button is still pressed and if so play the sound. Just now all actions occur at input transitions and to change this to incorporate this improvement is foreseeable but not with the current time line and deadlines.

- **Save and Load Games**
  The option to save and load games would add great functionality to the current system. This would require changes at both the board software level and the chess engine interface level. At the board level a user would have to be able to access a load/save menu probably via a new button and then communicate with the chess engine interface to correctly load and save.

- **Pawn Promote Option**
  Another improvement that could be added to the system is instead of assuming a player wishes to promote a pawn to a queen allowing the user to choose. This may require the addition of scroll buttons on the board as well as a far more complex implementation of the event handler `OnInputChange()`.

Chapter 6

Chess Engine Interface

6.1 Requirements

The core of the system is the chess engine, it being the brain behind all the hardware and user interaction. It must receive the users move that has previously been detected from the board, calculate a counter move and feed it back with a description of the move to the board software that also deals with user interaction in the form of the LCD display, speaker and LEDs. It is also in charge of feeding the same move to the RCX software that will physically move the computers pieces on the board accordingly. Without such a tight time constraint on the project, a Chess Engine could have been developed from scratch providing a greater challenge and understanding of the game, but instead a suitable open source engine (the term open source is used for free software) was chosen.

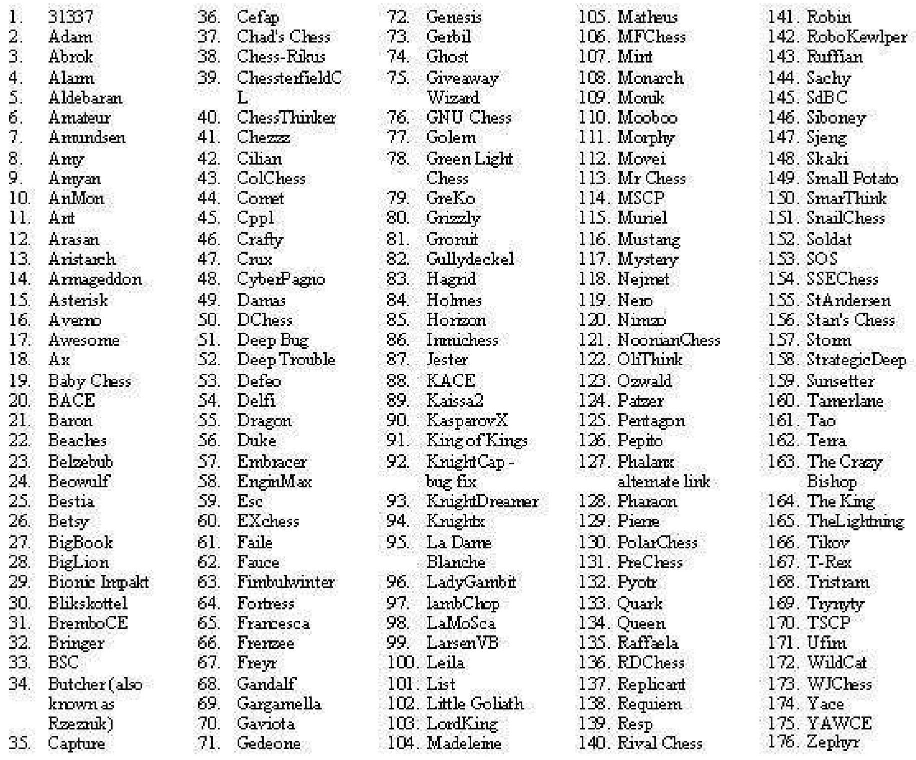
6.2 Choosing the Chess Engine

6.2.1 Engine List

Hundreds of engines are freely available; Figure 6.1 is a list to illustrate the large number of candidates:

6.2.2 Winboard/Xboard

Winboard and Xboard are both very popular graphical user interfaces for chess which all the engines are designed to work with. Winboard is the windows version and Xboard the Linux version. They display a chessboard on the screen, accept moves made with the mouse, and load and save games in Portable Game Notation (PGN), a standard designed for the representation of chess game data using ASCII text files. PGN is structured for easy reading and writing by human users and for easy parsing and generation by computer programs. One of its most useful features is ability to load 2 chess engines and have them play against each other while we log and observe their progress through Winboards graphical interface.



1. 31337
2. Adam
3. Abrok
4. Alarm
5. Aldebaran
6. Amateur
7. Amundsen
8. Amy
9. Amyan
10. AnMon
11. Ant
12. Arasan
13. Aristarch
14. Armageddon
15. Asterisk
16. Averno
17. Awesome
18. Ax
19. Baby Chess
20. BACE
21. Baron
22. Beaches
23. Belzebub
24. Beowulf
25. Bestia
26. Betsy
27. BigBook
28. BigLion
29. Bionic Impakt
30. Blikskottel
31. BremboCE
32. Bringer
33. BSC
34. Butcher (also known as Rzeznik)
35. Capture
36. Cefap
37. Chad's Chess
38. Chess-Rikus
39. ChessterfieldCL
40. ChessThinker
41. Chezzz
42. Cilian
43. ColChess
44. Comet
45. Cppl
46. Crafty
47. Crux
48. CyberPagno
49. Damas
50. DChess
51. Deep Bug
52. Deep Trouble
53. Defeo
54. Delfi
55. Dragon
56. Duke
57. Embracer
58. EnginMax
59. Esc
60. EXchess
61. Faile
62. Fauce
63. Fimbulwinter
64. Fortress
65. Francesca
66. Frenzee
67. Freyr
68. Gandalf
69. Gargamella
70. Gaviota
71. Gedeone
72. Genesis
73. Gerbil
74. Ghost
75. Giveaway Wizard
76. GNU Chess
77. Golem
78. Green Light Chess
79. GreKo
80. Grizzly
81. Gromit
82. Gullydeckel
83. Hagrid
84. Holmes
85. Horizon
86. Inmichess
87. Jester
88. KACE
89. Kaissa2
90. KasparovX
91. King of Kings
92. KnightCap - bug fix
93. KnightDreamer
94. Knightx
95. La Dame Blanche
96. LadyGambit
97. lambChop
98. LaMoSca
99. LarsenVB
100. Leila
101. List
102. Little Goliath
103. LordKing
104. Madeleine
105. Matheus
106. MFChess
107. Mint
108. Monarch
109. Monik
110. Mooboo
111. Morphy
112. Movei
113. Mr Chess
114. MSCP
115. Muriel
116. Mustang
117. Mystery
118. Nejmet
119. Nero
120. Nimzo
121. NoonianChess
122. OliThink
123. Ozwald
124. Patzer
125. Pentagon
126. Pepito
127. Phalanx alternate link
128. Pharaon
129. Pierre
130. PolarChess
131. PreChess
132. Pyotr
133. Quark
134. Queen
135. Raffaela
136. RDChess
137. Replicant
138. Requiem
139. Resp
140. Rival Chess
141. Robin
142. RoboKewlper
143. Ruffian
144. Sachy
145. SdBC
146. Siboney
147. Sjeng
148. Skaki
149. Small Potato
150. SmarThink
151. SnailChess
152. Soldat
153. SOS
154. SSEChess
155. StAndersen
156. Stan's Chess
157. Storm
158. StrategicDeep
159. Sunsetter
160. Tamerlane
161. Tao
162. Terra
163. The Crazy Bishop
164. The King
165. TheLightning
166. Tikov
167. T-Rex
168. Tristram
169. Trynyty
170. TSCP
171. Ufim
172. WildCat
173. WJChess
174. Yace
175. YAWCE
176. Zephyr

Figure 6.1: A list of freely available chess engines

6.2.3 FEN

6.2.3.1 Description

FEN is "Forsyth-Edwards Notation"; it is a standard for describing chess positions using the ASCII character set. A single FEN record uses one text line of variable length composed of six data fields. The first four fields of the FEN specification are the same as the first four fields of the EPD specification. A text file composed exclusively of FEN data records should have a file name with the suffix ".fen".

6.2.3.2 Data Fields

FEN specifies the piece placement, the active color, the castling availability, the en passant target square, the halfmove clock, and the fullmove number. These can all fit on a single text line in an easily read format. The length of a FEN position description varies somewhat according to the position. In some cases, the description could be eighty or more characters in length and so may not fit conveniently on some displays. However, these positions aren't too common.

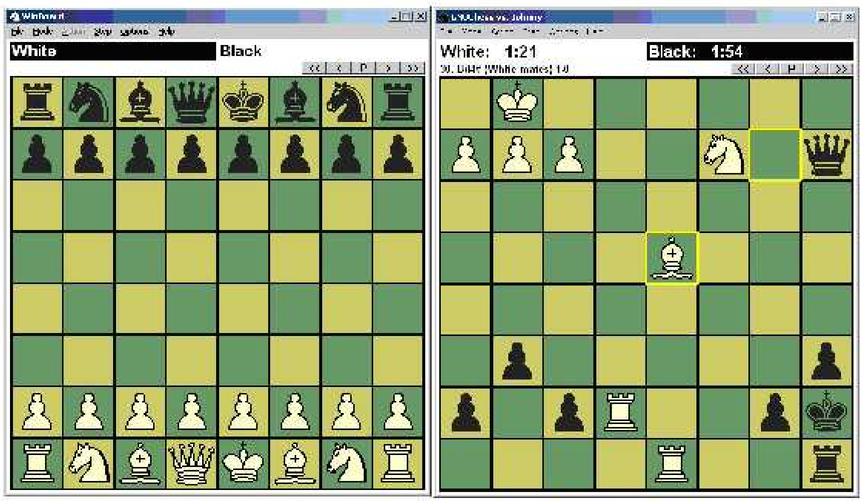


Figure 6.2: Screen printout of Winboard

A FEN description has six fields. Each field is composed only of non-blank printing ASCII characters. Adjacent fields are separated by a single ASCII space character.

```
Examples
Here's the FEN for the starting position:
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
And after the move 1.  e4:
rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1
And then after 1.  ...  c5:
rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2
```

6.2.4 Chess technicalities explained

Castle: To move your unmoved King 2 squares toward an unmoved Rook and to move the Rook on the other side of the King.
En Passant: Capturing a pawn that moved 2 spaces with a pawn that could have captured it if it had only moved 1 space, on the next turn only.
Fifty-Move Rule: A type of draw where both players make 50 moves consecutively without either player advancing a pawn or making a capture.
Promote: What a pawn does that reaches the other side of the board, and assuming the move is legal, then under any circumstances it can promote to a Queen, Rook, Bishop, or Knight on the promoting square. So you can have nine Queens, possibly.
Touch Move: The rule that says: 1. If you touch a piece you have to move it, 2. If you let go of a piece you have to leave it there 3. If you displace an opponents piece, you have to take it.

6.2.5 Evaluation Process

Having such a vast choice of chess engines required an efficient evaluating scheme; each of these could have been compared and tested against each other to filter out poor performances and find the best version to use in our system. The Lego Chess Robot required a competitive engine and originally was to provide extra features, such as undo and load/save board state. The main priority was being able to find a robust, stable, and easily interfaced engine.

6.2.6 Comparing Chess Engines

To simplify and shorten the time required to find a suitable engine, four main engines were chosen for having different strengths and features. They were GNU, GreenLightChess, Nero and Horizon.

6.2.6.1 GNU Chess

http://www.gnu.org/software/chess/chess.html
The GNU engine GNU Chess is a free chess-playing program developed as part of the GNU project of the Free Software Foundation (FSF).
GNU Chess is a communal chess program. Contributors donate their time and effort in order to make it a stronger, better, sleeker program. Contributions take many forms: interfaces to high-resolution displays, opening book treatises, speedups of the underlying algorithms, additions of extra heuristics. These contributions are then distributed to the large user-base so that all may enjoy the fruits of our labor. Unlike dedicated chess machines, or PC chess programs that run on only a few different models of Intel processors, GNU Chess runs on many different kinds of CPU at many different speeds. Thus its strength depends on how fast a machine you run it on and how much optimization your C compiler does.

6.2.6.2 Green Light Chess

http://www.7sun.com/chess/
GreenLightChess focuses on outputting plenty of feedback to the user, displaying board states after every change as well as its thinking process when calculating a move. Its documentation is very thorough and useful.

6.2.6.3 Nero

http://www.mit.jyu.fi/∼huikari/
The Nero engines author Jari Huikari has supplied the source code written in Pascal of his 5.1 version of Nero. It consists of 3000 lines and offers very quick responses to user input, definitely the most basic of the 4.

6.2.6.4 Horizon

http://www.horizonchess.com/
The Horizon engine offers standard features for a chess engine but does show when the user is put in check by adding a + to the end of the move, something that the other engines dont provide, the check feature was one of the challenges faced in interfacing a chess engine and will be discussed in further detail.

6.2.7 Engines playing against each other

6.2.7.1 GreenLightChess 1 - 0 GNUChess in 56 moves

```
[Event "Computer chess game"]
[Date "2004.12.4"]
[Round "-"]
[White "GNUChess"]
[Black "Green Light Chess v3.00"]
[Result "0-1"]
[TimeControl "40/300"]
1. e4 c5 2. Nf3 d6 3. d4 cxd4 4. Nxd4 Nf6 5. Nc3 a6 6. Bg5 e6
7. Qf3 Be7 8. O-O-O Nbd7 9. Be2 O-O 10. h4 Qc7 11. Qg3 Kh8 12.
Nb3 b5 13. a3 Bb7 14. f3 Rab8 15. Nd4 h6 16. Be3 Ne5 17. Rhe1
d5 18. Bf4 Bd6 19. Qh3 dxe4 20. fxe4 Nxe4 21. Nxe4 Bxe4 22. Bf1
Nd3+ 23. Bxd3 Bxf4+ 24. Kb1 Bd5 25. Be4 Rfd8 26. Bxd5 Rxd5 27.
c3 Rbd8 28. Re4 e5 29. g3 exd4 30. gxf4 dxc3 31. Rxd5 c2+ 32.
Kc1 Rxd5 33. Qc3 Rc5 34. Re8+ Kh7 35. Qd3+ f5 36. Re5 Rxe5 37.
fxe5 Qxe5 38. h5 Qf4+ 39. Qd2 Qg4 40. b3 Qxh5 41. Kxc2 Qf7 42.
Qd6 f4 43. Qd3+ Qg6 44. Kd2 Qxd3+ 45. Kxd3 g5 46. a4 h5 47. Ke4
h4 48. Kd4 bxa4 49. Ke4 a3 50. Kf3 h3 51. Kg4 h2 52. Kf5 h1=Q
53. Kxg5 a2 54. Kg4 a1=Q 55. b4 Qe5 56. b5 Qhh5#
Black mates 0-1
```

6.2.7.2 GreenLightChess 1 - 0 Nero in 59 moves

```
[Event "Computer chess game"]
[Date "2004.12.4"]
[Round "-"]
[White "nerowb51"]
[Black "Green Light Chess v3.00"]
[Result "0-1"]
[TimeControl "40/300"]
1. d4 d5 2. c4 c6 3. Nf3 e6 4. Nbd2 Nf6 5. g3 Bb4 6. Qa4 Na6
7. a3 Bd6 8. Bg2 O-O 9. O-O Qe7 10. c5 Bc7 11. b3 e5 12. b4 e4
13. Ng5 e3 14. Ndf3 exf2+ 15. Kxf2 Ng4+ 16. Kg1 Qxe2 17. Bd2
```

```
Bf5 18. Rad1 Ne3 19. Bxe3 Bc2 20. Rfe1 Qxd1 21. Rxd1 Bxa4 22.
Rd2 Rae8 23. Kf2 f6 24. Nh3 g5 25. Ne1 Re6 26. Bf3 Rfe8 27. Ng2
h6 28. Bg4 f5 29. Bxf5 Rf6 30. g4 Bxh2 31. Kf3 Bb5 32. Rf2 Nc7
33. Rd2 Ne6 34. Ne1 Bc7 35. Ng2 Ng7 36. Kf2 Nxf5 37. gxf5 Rxf5+
38. Bf4 gxf4 39. Kf3 Re4 40. Kg4 Rf6 41. Kf3 Rg6 42. Ngxf4 Rf6
43. Rg2+ Kh7 44. a4 Bxf4 45. Nxf4 Rfxf4+ 46. Kg3 Rg4+ 47. Kh2
Rxg2+ 48. Kxg2 Bxa4 49. Kf2 Rxd4 50. Ke3 Re4+ 51. Kf3 Bb5 52.
Kg3 Re2 53. Kg4 d4 54. Kf5 d3 55. Kf4 Kg6 56. Kg4 d2 57. Kf4
d1=Q 58. Kg3 Qd3+ 59. Kf4 Re4#
Black mates 0-1
```

6.2.7.3 GreenLightChess 1 - 0 Horizon in 40 moves

```
[Event "Computer chess game"]
[Date "2004.12.4"]
[Round "-"]
[White "Green Light Chess v3.00"]
[Black "Horizon"]
[Result "1-0"]
[TimeControl "40/300"]
1. e4 e6 2. d4 d5 3. Nc3 Nc6 4. exd5 exd5 5. Bb5 Be6 6. Nge2
Bd6 7. Nf4 Nf6 8. Nxe6 fxe6 9. Bxc6+ bxc6 10. O-O O-O 11. Qe2
Qd7 12. Be3 e5 13. dxe5 Bxe5 14. Bc5 Bd6 15. Bxd6 Qxd6 16. f3
Qc5+ 17. Kh1 Rfe8 18. Qd2 Rab8 19. b3 Qd6 20. Rfe1 a6 21. Ne2
Re5 22. Nf4 Rf5 23. Nd3 d4 24. Qe2 Re8 25. Qd2 Rb8 26. a3 c5
27. Nf2 Rf8 28. Qd3 Rh5 29. Qc4+ Kh8 30. h3 Qg3 31. Re2 a5 32.
Rae1 Rg5 33. Nd3 Nd7 34. Qb5 c4 35. Qxc4 c5 36. Qe6 Qc7 37. Qe7
Rd5 38. Re5 Rd6 39. Nxc5 Kg8 40. Ne6
Black resigns 1-0
```

6.2.7.4 GNUChess 0 - 1 Horizon in 73 moves

```
[Event "Computer chess game"]
[Date "2004.12.4"]
[Round "-"]
[White "GNUChess"]
[Black "Horizon"]
[Result "0-1"]
[TimeControl "40/300"]
1. d4 e6 2. c4 c5 3. d5 f5 4. dxe6 d6 5. Qb3 Nc6 6. Nh3 Bxe6
7. Qxb7 Nb4 8. Na3 Bxc4 9. Bg5 Ne7 10. e4 Bxf1 11. Kxf1 Rb8 12.
Qxa7 fxe4 13. Rd1 Ra8 14. Qb7 d5 15. Qb5+ Qd7 16. Qxc5 Nec6 17.
Qe3 h6 18. Bf4 g5 19. Bg3 Ra5 20. Qb3 Bg7 21. f3 O-O 22. Nf2
```

```
exf3 23.  gxf3 Qe7 24.  Re1 Qf6 25.  Rb1 Qxf3 26.  Qxf3 Rxf3 27.
Kg2 Re3 28.  Nd1 Re2+ 29.  Kf1 Re8 30.  h4 g4 31.  Nf2 h5 32.  Bc7
Ra7 33.  Bg3 Nxa2 34.  Nb5 Ra4 35.  b3 Rb4 36.  Nc7 Nc3 37.  Nxe8
Nxb1 38.  Nxg7 Kxg7 39.  Kg2 Rxb3 40.  Re1 Kf7 41.  Bf4 Na3 42.  Rd1
Nb4 43.  Bg5 Nac2 44.  Rf1 Ke6 45.  Nd1 d4 46.  Rf6+ Ke5 47.  Rb6
Rb1 48.  Rb5+ Ke6 49.  Bd2 Rxd1 50.  Bxb4 Rb1 51.  Rb6+ Kd5 52.  Ba5
Rxb6 53.  Bxb6 d3 54.  Ba5 Ne3+ 55.  Kf2 Ke4 56.  Be1 Kf4 57.  Ba5
g3+ 58.  Ke1 Kf3 59.  Bc7 g2 60.  Bh2 d2+ 61.  Kxd2 Nf1+ 62.  Kd3
Nxh2 63.  Kd4 g1=Q+ 64.  Ke5 Qg4 65.  Kd6 Qc4 66.  Ke5 Qc5+ 67.  Ke6
Ng4 68.  Kf7 Qc7+ 69.  Ke6 Qc6+ 70.  Ke7 Qf6+ 71.  Kd7 Ne5+ 72.  Ke8
Qf7+ 73.  Kd8 Qd7#
Black Mates 0-1
```

6.2.7.5 GNUChess 1 - 0 Nero in 33 moves

```
[Event "Computer chess game"]
[Date "2004.12.4"]
[Round "-"]
[White "GNUChess"]
[Black "nerowb51"]
[Result "1-0"]
[TimeControl "40/300"]
1.  e4 c5 2.  Nf3 e6 3.  d4 cxd4 4.  Nxd4 a6 5.  Bd3 Nf6 6.  Nf3 Bc5
7.  O-O O-O 8.  e5 Nd5 9.  Bxh7+ Kxh7 10.  Ng5+ Kg8 11.  Qh5 Qxg5
12.  Qxg5 Nc6 13.  c4 Nb6 14.  Nd2 Bd4 15.  Nf3 f6 16.  exf6 Bxf6
17.  Qc5 Bd8 18.  b3 d5 19.  cxd5 exd5 20.  Bg5 Bc7 21.  Nh4 Bd7 22.
Ng6 Rfc8 23.  Bf4 Bd8 24.  h4 Kf7 25.  h5 Ne7 26.  Qd6 Bc6 27.  Rae1
Bb5 28.  Qe6+ Ke8 29.  Ne5 Bc7 30.  Qf7+ Kd8 31.  Nc6+ Kd7 32.  Qe6+
Ke8 33.  Qxe7#
White mates 1-0
```

6.2.7.6 Results

GreenLightChess won against all 3 other chess engines. Nero lost against GNU chess showing that the Pascal engine cannot compete at the same level as engines written and optimized in C. Horizon played fairly well but its unusual and unpredictable move output cannot be handled as easily as the other 3 engines, also on two occasions it "Resigned" against GreenLightChess, therefore it has to be discarded. GNUChess shown the same functionality as GLC but poorer results in competition therefore GLC was chosen over it. GLC and Nero were chosen as the final two possible candidates; GLC offers competitive interactive play with very good documentation and Nero is very quick and its source code is freely available offering possibilities of expandability. Neither display check situations, this is a major setback which had to be resolved and is discussed below.

6.3 Checking For Check

6.3.1 Problem Description

All of the engines listed in the section above are designed to work with Winboard which works out for itself when the game is in check or checkmate therefore requires no extra information from the engines. As our Chess Robot is to inform the user of check situations through the LCD display it has to be computed and signalled by the interface.

6.3.2 Possible Solutions

There are two possible solutions to informing the user of check, one is to recode the actual engine and have it feed an extra character, such as a + in the returned move f5d2+ when the move puts the opponent in check. Only Nero has its source code available and it is written in Pascal. The other solution to coincide with the usage of the GLC engine is to store board representations of the current and previous move, which with methods that use these, can check whether a colour is in check. This feature would be implemented in the java interface. The algorithm that would check for this is a complicated one and involves thorough testing but would work for all Winboard compliant engines.

6.3.3 Editing the Nero Engine

The Nero engine consists of around 3000 lines of Pascal code, with no comments, it was studied and global variables were singled out to provide extra output when the move computed placed the opposing colour in check. Below is the code which was used to provide this feature:

```
computers_move;
checkcheck := true;
searchlegmvs(whitesturn, 1);
if(wkingincheck) then write('+');
writeln('');
```

This proves very successful and with the Nero engine being the quickest it was a good solution to the problem.

This solution had one major drawback, using Nero which has no extra feature such as setting the engines difficult, its set time to perform a move, undo a previous move and update its game according to a FEN notation. Therefore the second solution is now adopted

6.3.4 Creating the Java "isColourInCheck" Method

The java check method runs through the current state of the board, represented by a two dimensional character array. The method takes in a colour which defines which pieces are being inspected, white pieces are uppercase and black are lowercase. It also identifies the opponents king as the one that is to be found. Every piece on the board is inspected and all the squares that they can attack are looked at, returning a check result if the opponents king is present.

```
Method Pseudo code:

If colour is white
  Make char max and min range limits the ASCII uppercase
  Set the opponenets king ASCII uppercase
Else
  Make char max and min range limits the ASCII lowercase
  Set the opponents king ASCII uppercase
Endif
Loop through every square on board
  (
  If piece on square is a rook
    Loop through every square on its vertical and horizontal while it is blank
      If the square contains opponents king return check
  If piece on square is a knight
    Loop through each of its 8 attacking squares while it is blank
      If the square contains opponents king return check
  If piece on square is a bishop
    Loop through every square on all its diagonals while it is blank
      If the square contains opponents king return check
  If piece on square is a queen
    Loop through every square on its vertical and horizontal and all diagonals
    while it is blank
      If the square contains opponents king return check
  If piece on square is a pawn
    Loop through each of its 2 attacking squares
      If the square contains opponents king return check
  )
Return not in check
```

This method along with the board representations made it possible to use any of the chess engines and have a successful way of telling when the user was put in check by the computer.

6.4 Engine Interface Design

The engine interface provides two main public methods that the board interface calls, as well as two static multi-dimensional array to represent the current and previous boards, a number of private methods are included to modularise the problem.

6.4.1 Using Data Streams

Data input and output stream lets an application read and write primitive Java data types from an underlying input and output stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream. The two public methods glcsetup and callengine use DataInputStreams and DataOutputStreams to interact with the chess engine. Originally setup in this class, they required to be declared by the main thread which is the board interface which instigates engine actions. The streams are passed as parameters into the methods.

```
Standard methods used include:
flush(): Flushes the stream.
writeChars(String): Writes a String as a sequence of chars.
readChars(String): Reads a String as a sequence of chars.
close(): closes the stream.
```

6.4.2 Method and Algorithm Descriptions

Public Methods

6.4.2.1 function `glcsetup()`

A basic method that reads and prints the initial text that the glc engine prints declaring its status and details which include version number, copyright information, its hash table size and the books it refers to for opening moves.

6.4.2.2 function `callengine()`

The main method of this class which takes in both the input and output stream to the engine process as well as the move that is to be sent to it.
Tasks

1. Sends the move to the engine
2. Reads its response
3. Check whether response was “invalid” if so return the message
4. Compute the type of the user’s move

5. Update the current board representation with the user's move
6. Compute the type of the computer's move
7. Update the current board representation with the user's move
8. Translate destination and source square and movetype to numbers
9. Call the moveHandler class with the 3 numbers
10. Return the computer's move back to the board interface for user interaction, this also includes extra information such as whether its Check or Checkmate.

Private Methods

6.4.2.3 function `updateboard()`

Updates the current and previous board representation with the relevant move given, supports special cases such as kill, castling, en-passant and piece promote.

6.4.2.4 function `printBoard()`

Displays the current board representation, very useful for debugging purposes and progress display in the testing phases.

6.4.2.5 function `fenUpdateBoard()`

Updates the current board from a FEN notation which can be read from an engine at any point in a game. It could be used for undoing a move as well as the possible enhancement of allowing a user to play against the robot from a set chess situation.

6.4.2.6 function `isColourInCheck()`

Given a colour, checks to see if its pieces are in a check situation. It achieves this by running through the current board representation inspecting whether the king of the colour given is attacked by any of the opposing pieces from any of the possible direction.

6.4.2.7 function `getMovetype()`

An extensive set of comparisons that are based on the move value compared to the current board, it returns the type of move such as: move, kill, en-passant, shortcastle, longcastle and piecepromote.

6.4.2.8 function `readMove()`

Reads a move as a String from the DataInputStream that has been passed in and should have been opened to the engine. Also returns invalid and checkmate if the engine outputs it.

6.4.2.9 function `sendMove()`

Sends a move as a String and flushes the DataOutputStream it is given

6.5 Pawn Promotion

6.5.1 Problem Description

The user has to be able to make a choice for a pawn promotion when he or she moves a pawn to the last row of the board. As we have a limited input of two buttons and an LCD display, the simplest way to offer this choice is to have a menu which lets the user choose either a queen, bishop, knight or rook.

6.5.2 Solution

To solve this problem the Board Interface needs to know when such a move occurs so it can offer the choice of a new piece. This is achieved by a new method called piecepromote() which is called before sending the move to the engine. It checks whether the move is a promotion type of move and if so returns a signal to the board interface. Once it has received the users choice it adds the extra character representation of the piece onto the end of the move that is sent to the engine. When the engine promotes one of its pieces the Board Interface is able to check for this extra character and inform the user of this new piece.

6.6 Testing

All the errors in the chess interface are systematic. Therefore extensive case testing was used to iron out any bugs in the code. It was easily achieved by removing the chess engines input and output, the introduction of a new method called readKeyboardMove() made it possible for the user to imitate the chess engines output. With no move validation there was a lot of freedom to play out every special scenario which might have difficult to test when playing the engine. As the interface took the move from keyboard just like a chess engine move it was possible for it to also pass it on to the Robot movement section and the board interface to test the movement and display procedures in all cases considered.

6.6.1 Results

Long castling was found to be buggy as the internal board representation was not updated correctly; it interpreted it as a normal move. After running through the testing it was found that the wrong castling move was being detected, a basic interpretation of the chess move made it difficult to find the problem as the Java syntax was correct.

Enpassant was not functioning properly originally, as a character in the move fed from and to the engine was being compared to an integer. This was quickly fixed and tests also found the Robot movement to move as expected in en-passant.

Piece promotion was tested to find that when the engine had a piece promoted the user was simply told of the new piece and its location. When the user promoted their pawns they were automatically made into queens. Future improvement would have the board interface offering a menu with a choice of pieces.

6.7 Future Improvements

There are a number of future improvements available for expanding the usability of the Lego Robot Chess as far as the chess interface is concerned. Most of these features would only be developed as an extra as they were not targeted as major goals in our original specification unlike Robot piece movement efficiency and timing.

6.7.1 Undo

The undo feature was discussed by the team, although a useful feature it brought much more complexity into overall robot movement. The main problem faced in implementing the undo feature is undoing a kill move where the robot has dumped one of the pieces in its previous move, for this reason it was abandoned. The current robot does however allow the cancellation of first square the user presses that is the square that the piece to be moved resides.

6.7.2 Difficulty Setting

A range of difficulty settings should be available for the user to choose before beginning his game, the amount of user feedback did restrict this option a little as the system only contains two buttons but a simple menu could be shown on the LCD display.

6.7.3 Piece Promote leading to Check

Due to restrictions on time and the rarity of this situation it was not implemented. Piece promotion and check is dealt by the chess interface but it also has to interact and be understood by the board interface as new characters are added to the end of the move, with the two interfaces being written by different members the differences in coding styles meant the changes would have to be discussed thoroughly.

6.7.4 The 50 move and the 3 repetitive move draw

Some chess players play with the 50 move draw rule where the game is a draw if 50 moves have been played without a pawn capture. The 3 repetitive move draw occurs when an identical move is repeated by each side 3 times. In this Lego Chess Robot they were ignored for the prototype but could be implemented in the final product.

6.7.5 Portability

Having the chess engine running on a different system is a possible enhancement. In the final product the Robot should be an independent system therefore would require the software to be programmed into a microprocessor with memory placed in the board.

6.8 Conclusion

The main goals of the chess engine interface were successfully achieved; it provided a robust and efficient chess game which instructed Robot RCX interface on the next move as well as pass the move to the board interface to inform the user. The time spent on researching a suitable chess engine was well spent, although undergoing editing of an engine in a new language to the team did lead to a dead end it was a possible solution which was thoroughly explored. A lot of time was spent going through java documents as well as chess engine documentation to find the available features and tailor them to the need of our Lego Chess Robot. The difficult algorithms used in the chess interface were all carefully designed to make it easier for them to be tested in a black box fashion. Many of the unimportant extra features could have been introduced to the interface but could have jeopardised the main goals being achieved.
Testing quickly revealed the bugs in the rare chess situations which the chess engine did not often produce. The chess engine being the core of the Lego Chess Robot, simulating its process output made it possible to test other parts of the Robot system easily.

Chapter 7

Programming RCXs

7.1 Available Programming Languages

Over a number of years a vast number of alternative programming languages have been developed since the Lego Mindstorms kit became popular. Most of them are freely available and some of them open source. Several alternatives are described below.

7.1.1 Ada/Mindstorms 2.0

This programming language allows the user to program the RCX using Ada 95. Ada 95 might be easier or even faster to program in but it is not efficient because the programs are translated into Not Quite C (NQC), and then to byte-code. The produced NQC code would probably be more complicated and bigger than the original Ada code.

7.1.2 pbForth

This is an incarnation of the Forth programming language for use with the Lego RCX. It can be used as an alternative firmware. pbForth's strong point is that it has a small footprint, thus it leaves more space for the user. The programs are being interpreted and compiled on the RCX, thus it does not have any additional compilers. Some big drawbacks are that pbForth does not have a notion of multi-tasking and a new programming language that has been nearly dead for some years now would have to be learned.

7.1.3 BrickOS

This is another alternative firmware for the Lego RCX, implemented in C . It provides C and C++ Applications Programmable Interfaces (APIs) for an RCX programmer, using the GNU C/C++ cross compilation tool chain. BrickOS also provides the necessary tools to download the compiled programs to the RCX. It allows more low-level access to the hardware such as memory locations, variables, the LCD and the IR transceiver. It also gives programmers the ability to use threads and POSIX semaphores for process synchronization as well as being a priority-based preemptive multitasking operating system. The disadvantages of BrickOS are that it is unstable, according to the projects documentation, it requires the gcc tool chain and a H8 cross compiler and linker. It also restricts the programming environment to Unix based operating systems mostly.

7.1.4 leJOS

leJOS is another firmware replacement for the Lego RCX. It's implemented and can be programmed in Java. It is multi-threaded, it uses floating point values, multi-dimensional arrays and exception handling. Extension packages can be added to increase the abilities of the operating system. The biggest disadvantage was that the Java virtual machine would take up too much of the precious memory space on the RCX.

7.1.5 tinyVM

TinyVM is an open source Java based replacement firmware for the Lego Mindstorms RCX micro controller. tinyVM is the predecessor of leJOS. It has the same basic features as leJOS. It also has more limitations than leJOS. For example tinyVM does not include a garbage collector, it does not support floating point arithmetic and constant values of type string are ignored completely.

7.1.6 Lego RIS graphical environment

This is the programming environment that comes with the Lego Mindstorms kit. It is a graphical way to program the RCX. It does not require extensive programming knowledge, but it gives far less control to the user than any previous explained language. It is more useful for people with no programming experience or the development of quite small programs and so not suitable for this project.

7.1.7 Gordon's Brick Programmer

This is an alternative graphical programming environment which is more powerful than the Lego RIS environment. It allows the user to access more variables, tasks, subroutines, while also giving access to the data-log and simple symbolic debugging facilities.

7.1.8 RoboLab

RoboLab is a robust piece of software powered by National Instrument's LabView. RoboLab is a graphical programming environment with tiered levels of programming that build skills and results without painful tutorials. The draw back with RoboLab is that it is not free and it is not as capable as other programming languages previously mentioned.

7.1.9 Spirit.ocx

There is a windows module that allows the user to program the RCX in a few programming languages such as Delphi, Visual Basic and Visual C++. Unfortunately it is not a great deal of information concerning this module and also restricts the scope of the project to be Microsoft Windows.

7.1.10 NotQuiteC (NQC)

NQC is a C like programming language for the RCX micro controller. It uses the already installed firmware of the RCX, thus it makes it easier to install. It is platform independent, since the same compiler is ported on Microsoft Windows and Unix operating systems such as Linux. NQC is relative simpler to use compared to other programming languages and, being a textual language, is more powerful than the graphical programming environments. On the minus side, NQC must live within the constraints of the standard firmware. For example, since the firmware does not provide floating point support, NQC does not support the format unlike other languages such as pbForth or BrickOS. NQC is very well documented and can be used with any programming editor like emacs, ConText and others. It creates a smaller footprint, so it can make more efficient use of the RCX's memory.

7.1.11 Conclusion

The choice of the programming language used in the project was based on efficiency, portability, extendability and on the programming languages that the team was comfortable with. Because of this all the graphical based languages, such as RoboLab, the Lego RIS programming environment, were rejected. Spirit.ocx and pbForth was rejected because it might be too time consuming to learn the Forth language or Visual Basic and the second was also not portable. tinyVM is just an older implementation of leJOS, so it was discarded. This narrowed the possibilities to Ada, NQC, leJOS and BrickOS. The last criteria used to decide upon a programming environment was efficiency and ease of use. NQC seemed to be the most appropriate tool due to ease of use; we didn't have to install new firmware, was well documented, was as efficient as required, it is platform independent and contains all the essential tools to compile and download the programs to the RCX micro controller.

7.2 Sensors

The RCX micro controller has a large set of sensors in order to sense it's environment and respond appropriately. Lego provides a wide range of sensors like touch sensors, light sensors, rotation sensors, temperature sensors and others. For example a temperature sensor gives different values that correspond to different temperatures, as expected. Similar hypothesis can be made for the other type of sensors. Also, because the sensors are analogue devices many people have developed their own sensors such as ultrasonic sensors, sound sensors, torque sensors, pressure sensors and many others. Unfortunately the RCX micro controller has only three inputs and three outputs. Many people have also developed multiplexers to increase the number of inputs or outputs to overcome this limitation. The use of multiplexers to increase the number of inputs and outputs makes the program more complicated to handle.

Originally the team decided to use one RCX because the task appeared to be simple enough. While prototyping and the constraints of the RCX were more obvious. The lack of enough inputs and outputs for the task resulted in the decision to use two RCXs communicating with each other. This helped to reduce the complexity of the code used, because the tasks were divided in two parts. We also had more inputs and more outputs to utilize and of course more memory space available.

After it was agreed on which hardware architecture the team will follow, the RCX were assigned to their specific tasks. Part of the team was building the robot, the rest were researching and testing the communications between the two RCXs using NQC.

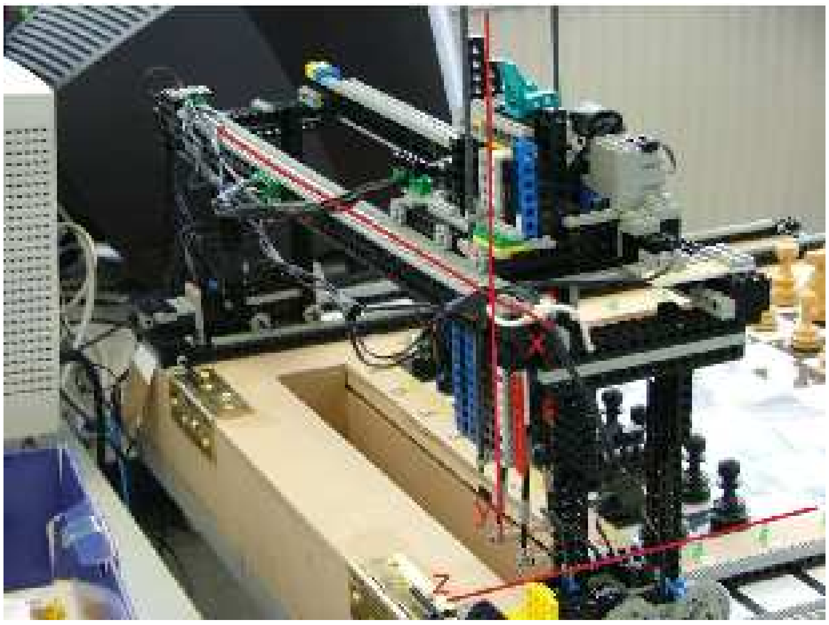


Figure 7.1: Illustrating the different axii the robot must move in

The master RCX was configured as the grabber controller, which is responsible of moving the grabber across the horizontal axis and the vertical axis, in order to grab the piece and move horizontally along the board. A rotation sensor was used to calculate the position of each square on the actual board. The need to move in two axis, x for moving the graber horizontally and y for moving it up and down, as showen in figure 7.1, imposed the use of two motors on the outputs. The third output of the RCX was used to control the method of grabbing the pieces. Under development an additional input was used to sense the home position of the robot, using a touch sensor. The only disadvantage of using the rotation sensor was that the position of each square in a row must be predefined. There was no other way to sense above which square the robot was at any time, due to its height. Measurements were carefully taken to ensure as much accuracy as possible.

The slave RCX was configured to control the mover. It's job was only to move the robot on the lateral axis, with respect to the other RCX. There was a problem though. Initially two rotation sensors were used, but the available resources were limited because other groups were using the same type of sensor and consuming two of them, was a problem. Ergo, two light sensors were used instead. Specially printed strips of paper were also used so the sensors could detect which square they are over. During development and testing of this part of the system, it was discovered that the robot was not moving both sides at the same time, due to excessive weight on the side of the robot where the grabber was situated. The problem was fixed in the software by allowing the system to move each side independently, stop on the intended line and wait for both sides to synchronize before it continues. The need of two buttons, for returning to the home position to ensure the robot was re-calibrated each time was discovered. The difficulty in implementing these two buttons was that there was only one spare input on the RCX. Hence, a multiplexer had to be used. After researching, the simplest possible solution was chosen to multiplex two touch sensors on one input.

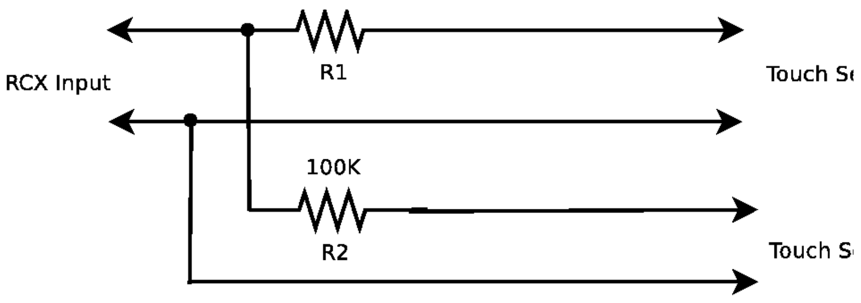


Figure 7.2: Schematic of multiplexer circuit

The design of the multiplexer is very simple and relays on the fact that the RCX uses analogue inputs. The idea was that the analogue input can be exploited by using a potential divider, two different resistors effectively, in series with each button. This design did not work when we declared the input as a standard touch sensor type, thereby we masked it as a light sensor. This subject will be revisited in more detail in the next section.

7.3 The Code

The algorithm for the two RCXs is relatively simple. The basic idea is to wait for a message. After receiving a message, it is evaluated and the appropriate tasks were performed e.g. making a move. Since the overall architecture is master/slave, the master is responsible for communicating with the slave and it instructing as to what task to perform. The algorithm for the master RCX is as follows

```
Wait for message
evaluate message and begin task
calculate where to move
send message to slave RCX to tell it where to go
wait for acknowledgment, when you arrive at position
get piece from board
send signal to slave RCX to move to new square
wait for acknowledgment
put piece to board
send signal to slave RCX to go home
return home and wait acknowledgment
```

This algorithm changes slightly according to the performed task, but the overall higher level pseudo code is the same. The algorithm for the slave RCX is simpler because the RCX has less actions to perform. The pseudo code is as follows

```
Wait for message
evaluate message and begin task
go to given square
send acknowledgment
```

Both RCXs try to achieve the main goal concurrently. Each task is executed independently with one RCX synchronizing the overall task. Concurrency is important when timings are quite critical, but still there must be some kind of synchronization in order to complete the goal without one task executing prematurely.

A great challenge was implementing the mover so it was able to detect transitions and to correct itself from skewing. To make the robot detect a transition was not very difficult but the problem was that the one side of the robot was moving faster than the other. During investigation and careful study of the problem, the error was not consistent. It was related to the fact that the weight was not equally shared on the robot. Hence, the problem had to be fixed in the software. The easiest way to do this, without making big changes to the code, was by stopping individually each side of the robot once it reached the desired square.

Another problem that the mover had, due to the skewing error, was the fact that it did not calibrate sufficiently each time it reached the home position. Ergo, a small error was being added each time to the movement and it became considerable after moving a few times. The solution to this problem was to add two buttons on each side of the robot so it could detect when both sides were at the home position. There were not sufficient free inputs on the RCX but construction of a small multiplexer solved the problem. Masking the input as a light sensor gave a larger resolution than defining the input as a touch sensor. This trick allowed the RCX to pickup different values for each button pressed and even different value when both buttons were pressed. This proved to be helpful later on when the code to control the robot actions was developed. The same idea of home sensing was also adopted on the grabber, in order to minimize any errors from the imperfection of the hardware.

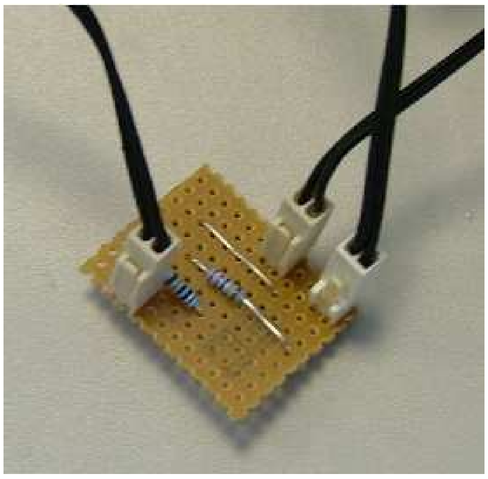


Figure 7.3: Picture of mulitplexer

Code was also developed to allow the computer to communicate with the RCXs. After some researching on the Internet, a Java API was found. This API was simple to use and included a few examples to demonstrate it's usage. The API satisfied our needs, since it supported sending commands and messages from the computer to the RCX. After playing around to familiarize with the API, a basic and simple method was constructed. The method takes three values as parameters; the start square, end square and move type. The start and end squares are the square numbers on the board, using the numbering scheme described in section 4.9. The type of the move is an integer from zero to five associated with the five possible move types, a regular move, kill move, short castling move, long castling move and en passant move. All members involved in programming agreed on this numbering scheme.

7.4 Communications

Communications between the RCXs and the computer was essential. Since, the RCXs were in a master/slave architecture, the communication protocol between the RCX and the computer was not as complex as it was expected. The only official way to communicate with the RCX was through messages. But this method was not sufficient because, from the possible 255 messages we could not use the first 63, due to the numbering of the squares on the board. Also, the method of waiting for three messages to be received and then start executing the task would add to the overall performance time of the robot.

After careful study and a lot researching on the Internet to find a solution that would be faster, the team came across an opcode reference for the RCX. The opcode reference included commands that allowed control of the RCX from a computer. Also this reference included a few special commands, like setting variables on the RCX, locking and unlocking the firmware among others. The most interesting for the purpose of this project was the opcode that allowed you

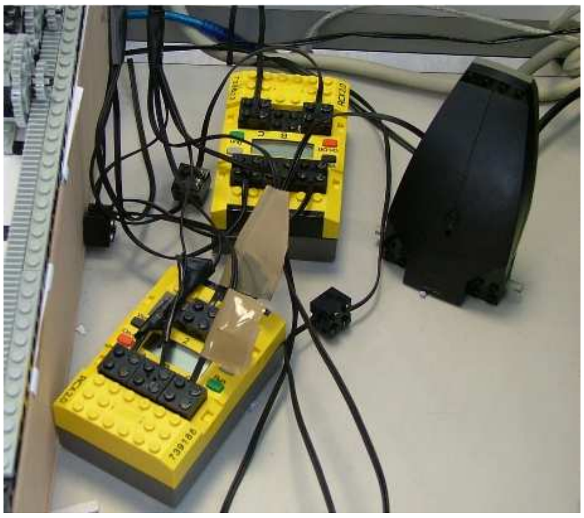


Figure 7.4: The two RCX blocks used, lined up for communication

to set a value to a specific memory location on the RCX. Immediately testing on the opcodes started. The opcode reference was confusing, with some descriptions for only some types of opcodes. There was a short hint at the end of the page, in the "Known Issues", that clarified the issue somewhat.

Thankfully, NQC can reserve memory locations for specific use. So reserving some memory locations for storing the transmitted values of the start and end squares was a simple task. This idea can also be used to identify different RCXs in the same area.

```
void msg(int sqr)
{ /* Special opcode message to the other RCX to set where it will go
        - opcode 14 - see opcode reference */
    ClearMessage();
    SetSerialComm(SERIAL_COMM_DEFAULT);
    SetSerialPacket(SERIAL_PACKET_RCX);
    SetSerialData(0, 0x14);
    SetSerialData(1, 0x17);
    SetSerialData(2, 0x02);
    SetSerialData(3, sqr);
    SetSerialData(4, 0x00);
    SendSerial(0,5);
}
```

Sending information back from the RCX to the computer was needed in order to notify the rest of the system to continue the game. This would make the response time smaller, because the system would be notified sooner for small moves or failures. The problem was that the tower connected to the computer only stayed open for a few seconds. A solution to this problem was that a ping could be sent every few seconds keeping the tower open and ready to receive the response from the RCX. The idea was abandoned quite early on because if the RCX would try to send data when the tower was off it would miss it and it would not know if the message was received successfully. There was also a great number of errors on the computer side and that made the message handling slower and more unreliable. A simpler and efficient way had to be found. After consulting the project supervisor for ideas and after giving some thought on the problem, an idea to connect a second button to the one end of the robot's home position was suggested. The idea was to handle the feedback of the robot with the Phidgets kit since it sounded simpler, easier and neater to do it in Java. Ergo, we could detect when the robot finished it's move by sensing when it was "home". This is discussed in section 5.12.

7.5 Testing the system

Testing the system was essential since hardware and different timings from subsystems were involved. Testing the code often decreased the coding time since bugs were obvious. Once a subsystem or part of the code was developed it was immediately tested. After the final version of the code for the RCXs was complete, integrating the subsystems revealed some previously unconsidered timing problems, which were fixed immediately.

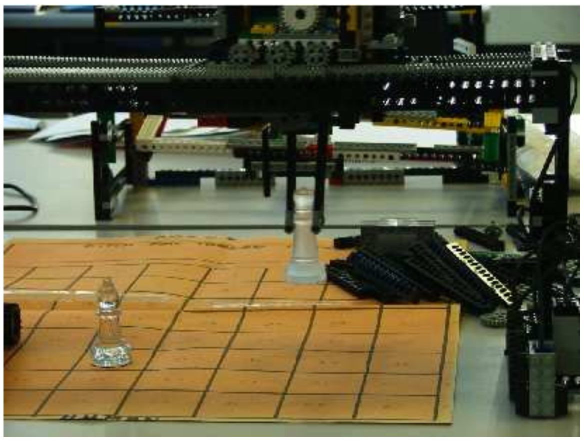


Figure 7.5: The robot moving a glass piece during early testing

The basic strategy for testing the system was to test each individual move as soon as possible and as often as possible. As a result the same tests were repeated each time a new subsystem was tested, in order to ensure that it did not affect the previous developed code.

The basic problem that was faced during testing was timings. Many times things appeared to work perfectly but the timings were not synchronized so it ended up doing the wrong thing. It was having timing problems when it should place the piece on the board. Through performing many different tests and designing special tests that were more likely to make it fail, the final code is relatively bug free.

7.6 Improvements / Extensions

There is room for improvement in the code. An option of adding an identity on the RCXs in order to identify them among others in the same area, is also taken into account. Error detection and error handling from the computer can be added, in order to ensure that the commands are always received correctly by the RCXs. Finally, detection when the batteries are running low could be added, in order to notify the user to take appropriate actions.

Chapter 8

Conclusion

8.1 Project Status

The project has accomplished all goals set by our requirements specification; the robot detects the players move, computes its counter move, and moves its piece accordingly. As the quantifiable factors are concerned standard moves are achieved in under 2 minutes and the kill moves in 3min 40. Accuracy and efficiency has also been met. The team also managed to include extra features on top of the finished model like speech output of moves.

8.1.0.1 Vehicle Design

The finished model for the robot is fully functional and performs all tasks for the computers moves. It was based on the Theme Park Grabber model which was prototyped into 5 different modules before being integrated together. There are 3 different gearing systems to provide a slow accurate movement of each component of the robot. Although the vertical movement of the grabber is sufficient to pick up and drop pieces in the prototype, the addition of an extra motor would ensure smoother movement.

8.1.1 Positioning

The robots movement is accurate to within 2mm from position on starting square to destination square. The skewing between either sides drive mechanism is handled by the light sensors stopping each motor at the line corresponding with the destination line.

8.1.2 Chess Interface

The interface functions successfully with the GLC chess engine running on a Windows environment. The maximum thinking time has been set to 20 seconds although never takes that long as it has been allocated a 16MB hash size. It provides a hard game to the user and to date has not been beaten. It detects all invalid moves and supports castling, enpassant and basic piecepromotion.

8.2 Conclusions

8.2.1 Achievements

During the course of this project all team members have gained experience in team work and communication skills. Coordinating all our efforts together to meet deadlines improved our planning and time management. Verbal reasoning also allowed team members to voice there opinions in a way that didnt force others to agree with them. Each team member can also boast specialised knowledge in their own area of the project. Good time keeping was required of all team members as some work on some sections such as RCX program implementation depended on others being completed. This knowledge had to be passed to other members when integrating separate components of the project and this improved each members communication skills. Skills were demonstrated throughout the project; the team was able to research a possible project idea and prove its use and need for development to a third party. The project was properly broken into modules for each member to tackle and deadlines were set, with weekly meetings to assess possible risks and progress the overall project was run very smoothly. Electrical and electronic engineering skills were shown in the creation of the board circuit and a multiplexer for the RCXs. Innovative engineering frames of mind helped construct Lego models possible of achieving general theoretical ideas. Software engineering skills were relied upon when developing interfaces for the board, chess engine and the RCX NotQuiteC programming. Strong object orientated ideas helped produce reusable software that could be easily integrated together to form a fully functioning system.

8.3 Further Improvements

With more time the team would have liked to implement more functionality to the project including:

- The system could easily be made possible for the user to play as black instead of white. Due to time restrictions, efficiency aspects of the robots were concentrated on.
- The system could be made machine independent, even portable with the chess engine and all three board, engine and RCX interfaces programmed onto a microprocessor with memory alongside it. The RCX would require to be fixed as well as the IR tower which sends the instructions.
- The addition of more buttons to the final model to allow the user to undo a move as well as being able to offer a draw. The user being able to set the difficult it wished to be playing against. The LCD display could also be made larger to increase the feedback provided to the user.
- The games could be logged and possibly printed for experts to analyse a certain game pattern that they had played.

Appendix A

Summary of Project Logs

A.1 Stewart Gracie

October	• Introduced to Lego Midstorms Kit in the Lego lab • Meeting arrangements decided - location, day and time each week • Im allocated job of keeping meeting minutes • Communication between members sorted- email and phone • Possible project proposals including Rubik Cube solver, GPS robot and 3d scanner • Experimented with Lego design
November	• Chosen project as Chess playing robot and explained idea to supervisor • Make choice between robot with magnets or arm, arm is decide upon • Started to create a requirements specification to be accepted by supervisor • Began to create a Gantt chart to show deadlines • My main task is to design and build the Lego robot • Started to prototype models for the grabbing unit

December	• Finished design for grabber, started on building mechanism to lift it • Allocate different sensors to be used with robot prototypes • Discover there are constraints on the number of sensors and motors • Ordered two 16 inch poles for lifting mechanism • Started designing geared motors to create greater torque and slower speed
January	• Started prototyping structures to move grabbing unit • Ordered more Lego components to complete structure and tracks • Prototyped drive train to move structure • Drive is too fast and needs to be geared down • Testing shows light sensor needs to be boxed in to block out ambient light • Test motors to find two that run at the same speed
February	• All robot mechanisms are integrated together for testing • Testing shows changes need to be made to lifting mechanism for grabber, grabs pieces too high • Grabbers axle doesnt hold pieces well, added material to create friction on axles • Start on documentation for report
March	• Start MLCAD drawings for report • Start taking photographs of robot for report • Proof read other team members reports • Finished drawing and cropping all pictures

A.2 Jonathan Matthey

October	• Introduced to Level 3 ESE Lab and Lego Mindstorms • Researched intensively for a week on possible project ideas • Shared with team resourceful websites found with previous lego projects • Presented Magnetic Chess Robot, Arm Chess Robot and Rubix Cube ideas to the team with drawings and explanations • Met with supervisor to discuss shortlist of ideas, he didn't take to the Lego Chess Robot initially • Produced an early project description with goals and a project plan to convince Supervisor of the Project's depth • Supervisor introduced us to new Phidgit technology, we found it useful for the board and displaying information to user through LCD display.
November	• Wrote Final Requirements Specification with Risk Management Scheme • Set up Team FTP webspace for documents and illustrations • Meeting with team to delegate tasks across our four team members • I was in charge of the Chess Engine, its Interface and also helping Stewart developing useful Lego prototypes • Found Mario Ferrari Lego Mindstorms book at Library for Stewart • Helped Stewart prototyping the Lego Grabber to the final model • Found five suitable chess engines to compare • Learnt about Winboard and using it to play engines against each other • Read through Winboard and GNUChess documentation to highlight how to communicate to the engines as well as where functions were implemented (eg. the Check function)

December	• Installed Java Netbeans 4.0 to team machine • Studied using Java DataStreams and Runtime Environments on ways to interface a running executable. • Hunted for appropriate Chess pieces online, compared them to find a suitable set • I chose a weighed set with very good ridges to ease the grabbers task • Built the two vertical struts and beam for Lego Robot to work on a small scale • Took pictures of early prototype for documentation purposes, showed to supervisor • Learnt Pascal basics through books and tutorials • Installed FreePascal Development Environment
January	• I found the grabber to have a limited clearance due to length of plastic Lego bar • Bought carbon fibre rods rather than metal to extend its vertical reach • Designed the algorithms and board representations needed in the Chess interface for the various extra features required to support the GLC Chess Engine • Implemented the Chess Interface in Java and publicised all method calls to David • Discussed with Kostas on how to translate the board squares and the move types to integer values

February	• Integrate Chess Interface with working board interface producing a playable chess game without any robot movement • Demonstrated the game playing aspect of the system to supervisor with good feedback • Created extra functions to chess interface; undo and loading board representation from FEN notation • Identified vertical movement problem on robot grabber, solved by strengthening the contact and smoothing the movement.
March	• Ran extensive simulations of the chess engine by feeding it set patterns to test with a wide range of systematic test cases • Fixed code according to results from tests of castling, en-passant and piece promote. • Created illustrations and pictures for team to use in Documentation sections • Wrote personal Introduction to Team Project • Wrote Chapter 6 of Documentation on Chess Engine Interface • Wrote Abstract and merged Introductions for final document • Proof read Stewart and David's Document Sections

A.3 David Rankin

October	• Introduction to the project lab, lego kits and our advisor Roderick Murray-Smith. • Possible projects are discussed and researched. • Researched into the difference between the Lego mindstorm RCX and the Phidgets, eventually advising the group to adopt the Phidgets for use in the board. • Carried out experiments and research into the Phidgets hardware. • Created test programs to interact with Phidgets hardware • Decided on Lego Chess Robot project. • Assigned responsibilities for the Board and all its software.
November	• Development of 5 switch prototype and other Phidget 'sensors'. • Created an early prototype for chess robot which influenced the final design. • Designed and developed 3x3 Prototype. • Development of basic software for use with the 3x3 Prototype. • Testing of all hardware and software developed thus far.
December	• Expansion of software for use in 3x3 Prototype, GUI added. • Design of PCBs for board circuit. PCBs etched over Christmas holiday period. • Researched into logic for LED control circuit. • Testing of all hardware and software developed thus far.
January	• Construction of board circuit. • Construction of LED Control Circuit. • Extensive testing of board circuit and software. • Design of board's wooden case.

February	• Interfacing board software with chess engine interface software. • Creation of wooden case and the installation of all circuitry with in it. • Programming of user buttons on board. • Creation of RobotMove to integrate robot and chess engine. • Addition of sounds in the software.
March	• Development of User Guide which was used when I ran formal summative testing (appropriate documentation attached). • Final Testing and improvements. Assisted Kostas in making the computer talk. • The “Board Design and Construction” and “The Board’s Software” chapters were written. • Compilation of final report into LaTeX form and carried out proof reading of document. • Creation of source and media CD.

A.4 Konstantinos Topoglidis

October	• Introduced to Lego Midstorms Kit in the Lego lab by Rod • Arrangements made for weekly meetings • I was the Configuration Manager for the team • Exchanged emails and phone numbers • A small research was made to find out about the RCX • Project proposals for a GPS robot • Sorted the Lego in the lab and counted the RCXs • Divided the RCXs into the three teams
November	• Presented ideas to supervisor and made a decision making a chess robot • Plan for the project and the constructive way that the team will follow, was made • Requirements specification was made and presented to the supervisor • The task I took was to make the programs for the robot and help with construction when needed • Researched to learn more about the RCX • Started to play around with the RCX and NQC
December	• Researched more about the RCX and it's limitations • Played around with the rotation sensor and a motor • Studing other peoples code and the NQC documentations for communication • First program to do a very simple movement of the prototype robot was developed

January	• Started coding and testing the RCX and light sensors for transitions and accuracy • Splitting the code into small basic and reusable functions • Found out more about the Java API for the Lego tower and played around with it • Decided upon Master/slave communications architecture • Found out more about the opcodes and tested them • The robot can move on all three axis
February	• Robot subsystems are integrated together for testing • Skewing problem appeared • Decided to put touch sensors for sensing home position • A small multiplexer needed for the two touch sensors • Idea for the multiplexer designed, constructed and tested • Different timing problems occured • All code was ready and testing started • Start on documentation for report
March	• Coding finished completely and played a game • Assigned as the teams photographer • Start taking photographs of robot for report • The “Programming RCXs” chapter was written • Proof read other team members reports and made suggestions

Appendix B

Installation Instructions and Stepwise Guide

B.1 Installation

B.1.1 The Chess engine

The default chess engine for the Lego Chess Robot is called Green Light Chess (glc). This chess engine runs on the computer and so it is necessary to install all the appropriate files in the right place. The files for the chess engine can be found on the cd in the 'Chess Engine' folder. Within this folder is a zip folder called glc300 and this should be extracted to a glc directory in the C drive. Once complete the path

```
 C:\glc
```

should lead to the contents of the folder.

Once done that copy the "Board Software" folder to a destination of your choice.

Navigate to this directory within Command Prompt (accessed by trying

```
cmd
```

in the run menu). Once within the Board Software folder on the computer type in the command

```
runTalk
```

to run the software. The LEDs will begin to flash. Once a welcome sound is played press yes to begin the game.

Ensure the Phidgets is plugged in and the RCXs are on and running and facing the tower.

B.2 Starting a Game

When the software is launched (typing "run" within the appropriate directory) the lights will flash in a recurring pattern. When the pieces are set up and you are ready to play, press the

YES button as prompted by the display.

B.3 Making a Move

- First press down on the piece you want to move at the square it is currently in. A noise will sound and if the press was detected the lights will highlight the correct row and column. The display will then be updated to show this first half of the move.
- Once the first press has been detected, move the piece to the new square and press down on it again. If the move is detected, the move will be displayed on the screen and the chess engine will begin calculating the response. Once a counter move has been calculated you will be informed via the lights and the screen. The robot will move for the computer.
- If a press is not detected by the board then the lights will not illuminate and the text screen will not update. In this scenario please repeat the press of the piece but holding down for a longer duration.
- If the move entered is an illegal chess move, the chess engine will stop calculating the counter move. The display will then be updated to indicate an illegal move has been made and will also prompt the user to make another move.

B.4 Sounds and Lights

When making a move, a sound will be heard as the square is pressed but the lights on the board will only update after the piece is lifted from its square. If the move has not been detected properly then the lights and the screen will not update (note the sound will still occur even if the press has not been detected). If the move has not been detected repeat the pressing of the piece but hold it down for longer.

Different sounds can be heard when an invalid move is entered or if you are put in check by the computer. If an unfamiliar sound is heard please refer to the text display for more detail.

B.5 Cancelling Moves

The cancel button can only be used to cancel the first half of a move. If you have only pressed down on the piece once and not moved it, then you may cancel by simply pressing the button. If you have completed the move then you cannot cancel.

B.6 Quitting

Pressing the Cancel button when you have not started a move will give you the option to quit. To exit a game press Cancel and then press Yes to confirm. Pressing Cancel when you are given the option to quit returns you to normal play.

B.7 Castling

If you wish to castle, press down on the King when moving him but do not press down on the Rook (Castle). Simply move the rook without pressing on any squares.

B.8 Taken Pieces

Please do not place taken pieces in the path of the robot or in the ditch at the back of the board.

Appendix C

Glossary of Terms

RCX : Robot Command eXplorer, the Lego microcontroller

Phidgets : “Physical Widgets” are USB interface modules

PCB : Printed Circuit Board

LED : Light Emitting Diode

MDF : Medium Density Fiberboard

NQC : A C-like programming language for the Lego Robotic Inventions kit

API : Application Programming Interface

LCD : Liquid Crystal Display

IR : InfraRed

GNU : Gnu’s Not Unix, a free Unix style Operating System

JAVA : A high-level object oriented programming language developed by Sun Microsystems

C : A widely used, general-purpose programming language developed by Dennis Ritchie in the late 1960s

RIS : Robotic Inventions System

Lego : Danish “leg godt”, which means to “play well.”. A well known toy company

ESE : Electronic Software Engineering

DAC : Digital to Analog Converter

FM : Frequency Modulation

GUI : Graphical User Interface

PGN : Portable Game Notation, used for saving games of chess

FEN : Forsyth-Edwards Notation. It is a standard for describing chess positions using a text character set

WinBoard	A graphical user interface for chess
GLC	Green Light Chess, a WinBoard engine that plays chess
NetBeans	A cross-platform Java development enviroment
ASCII	American Standard Code for Information Interchange
FSF	Free Software Foundation

Appendix D

Chapter 3 Extra Information

D.1 Matching Motor Data

Motor Number	**Test 1**	**Test 2**
1	678	680
2	598	608
3	676	678
4	764	745
5	701	709
6	597	611
7	567	579
8	679	701
9	592	579
10	743	752
11	682	700
12	632	650

By examining this data motors 1 and 3 were chosen as suitably matched to drive the robot with little skewing.

Motors 5 and 8 were also possible candidates but the range of values between the motors was considered too large.

D.2 Chess Piece Selection

http://www.chess-sets-uk.co.uk

This chess set was considered as it had slight ridges around the base of each piece although the pawns had a slightly smaller ridge than other pieces. For this reason this chess set could not be used.

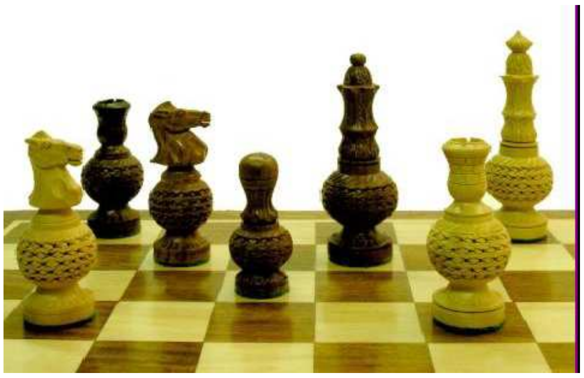
http://www.wooden-chesssets.co.uk

This slightly different wooden chess set was also considered as all pieces had a ridge at the base. On closer inspection though the rooks ridge isn't large enough to be held

http://www.chessbaron.co.uk

This chess set was chosen to be used for the project as its large ridge and round body was ideal for the grabber to pick up. This set was only substituted for the current set due to its 59.99 price tag compared with the 29.99 of the set used.

The chess set used by our project can be seen at the following web address



http://www.chessbaron.co.uk\CARTgallery1.htm. This set is described as a Concave Staunton Chess Set and has good quality pictures to show all pieces.

Appendix E

Tester's Consent Forms