Automated Car Park System in Lego

Ross Macdonald, John Wallace, Stuart Chalmers

today

Contents

1 Introduction **5**

2 Specification **7**
2.1 Hardware Specification . . . 7
2.2 Software Specification . . . 7

3 Overall Design **9**
3.1 Hardware . . . 10
3.1.1 Base Unit . . . 10
3.1.2 Lifter . . . 10
3.1.3 Extending Arm . . . 10
3.1.4 Barrier . . . 11
3.1.5 LED Display . . . 11
3.1.6 PCs/IR Towers . . . 11
3.2 Software . . . 11
3.2.1 NQC . . . 11
3.2.2 LejOS . . . 12
3.2.3 Start-up Menu (DOS) . . . 12
3.2.4 Java Communications . . . 12
3.2.5 Customer UI (Java) . . . 12
3.2.6 Controller UI (Java) . . . 12
3.2.7 Applet (Java) . . . 13

4 Detailed Design **14**
4.1 Hardware Design . . . 14
4.1.1 Drive . . . 14
4.1.2 Lifter . . . 17
4.1.3 Extending Arm . . . 19
4.1.4 Barrier . . . 21
4.2 Electronics design . . . 25
4.2.1 LED Display . . . 25
4.3 Software Design . . . 28
4.3.1 DOS Start-up menu . . . 28
4.3.2 Robot and NQC Software . . . 29
4.3.3 Communications PC - RCX . . . 31
4.3.4 Barrier and LejOS Software . . . 35
4.3.5 Customer User Interface and CustServer Software . . . 38
4.3.6 LED Display Software . . . 39
4.3.7 Controller User Interface . . . 42
4.3.8 Applet . . . 44

5 Implementation and Integration **45**
5.1 Implementation 45
5.1.1 Hardware 45
5.1.2 Software 45
5.2 Implementation Problems 46
5.2.1 Hardware 46
5.2.2 Software 47
5.3 Testing Strategies 47

6 Status Report **49**

7 Performance and Reliability Testing **51**
7.1 Hardware Testing 51
7.1.1 Structural Reliability Testing 51
7.1.2 Tolerance Testing 52
7.2 Software Testing 53
7.2.1 Software Correctness Testing 53
7.2.2 Communications Reliability Testing 54
7.3 Conclusions of Testing 55

8 User Guide **56**
8.1 Installation 56
8.2 Startup 56
8.2.1 Robot RCX Setup 56
8.2.2 Applet Setup 57
8.2.3 Barrier Control RCX Setup 58
8.2.4 RCX Program Startup 58
8.2.5 RMI Registry Startup 58
8.2.6 LED Display Server Startup 58
8.2.7 Main Server and Controller Startup 58
8.2.8 Customer User Interface Startup 59
8.3 Parking a car 59
8.4 Retrieving a car - Customers 59
8.5 Using the Controller User Interface 59
8.5.1 Checking the status of the car park 59
8.5.2 Retrieving a car 60
8.5.3 Checking the Cost / Time spent in the bay 60
8.5.4 Closing the Bay Control Panel or getting help 61
8.5.5 Updating the LED Display Message 61
8.5.6 Exiting the System and Closing the Car Park 61
8.6 Using the Applet 61

9 The Project as an Educational Experience **63**

10 Project Logs **65**
10.1 Group Log 65
10.2 Stuart Chalmers' Log 66
10.3 Ross Macdonald's Log 67
10.4 John Wallace's Log 68

11 Abbreviations and Acronyms **69**

12 Appendices **70**
12.1 Appendix A - Source code 70
12.2 Appendix B - Project Plan 70
12.3 Appendix C - LED Display Datasheets 70

1 Introduction

Our 3rd year Electronic and Software Engineering Team Project is the design and implementation of an Automated Car Parking System, modelled using the Lego Mindstorms Robotic Invention System.

The flexibility of the Lego Mindstorms kit means that there is a wide range of projects that can be created. Since its launch, many individuals have themselves developed alternative control programs and languages for the kit, the most notable of these being NQC, or Not-Quite-C, a C-like programming language developed by Dave Baum. Also, with the increasing popularity of the Java programming language, many different Java control languages have been developed, each with its own unique characteristics.

The breadth of technology available makes our project interesting. We had at our disposal a large variety of software available to use and adapt to suit our purposes, and as well as the hardware provided by the Mindstorms kit, we were free to add our own devices. As our degree is based jointly in both the Computing Science and Electronics & Electrical Engineering departments, this is an ideal situation. We had the chance to develop both hardware and software, and gain real practice and experience in these areas, and in hardware and software integration.

We felt that the Automated Car Parking System was a suitable vehicle to demonstrate and utilise the facilities offered to us by the Mindstorms kit. From a hardware point of view, we gained experience using the sensors and other Lego components, and also had the opportunity to interface these with any of our own devices. From the software point of view, we had a wide choice of implementation methods and languages. This allowed us to choose those which suited us, and gave us experience in integrating and interfacing different components. We felt that we also gained valuable experience in hardware/software integration, an area that is important in real-life situations.

The Department of Computing Science has had a number of Lego Mindstorms kits for a number of years, and previous Electronic & Software Engineering teams have also used the kits for their 3rd year Team Projects. The kits have also been used by other Computing Science Team Projects, and also for postgraduate projects.

A large number of these previous projects are based on or involve the study of using the Lego Mindstorms kits for educational purposes, particularly aimed at schools. The kits allow school children to learn about programming in a simple and fun way. Lego is familiar to almost everyone, and the fact that the children can construct robots and see the results of their efforts is also a benefit. The Lego kits have also been used extensively for demonstration purposes, particularly on University open-days, where potential students can see the kind

of work they might undertake, and current students can test and demonstrate their systems, providing them with practice in demonstrating and project presentation.

Previous projects have included a maze-navigating robot, a PCB (Printed Circuit Board) driller, a crane system, a distributed warehouse stock system, and using the kit as a tool to teach programming.

Our project team consisted of three members, initially four, all Electronic & Software Engineering students. The project was worth 20 credits of our 3rd year curriculum. We had two supervisors, one from Computing Science and one from Electronics & Electrical Engineering.

We were responsible for organising ourselves, deciding what deliverables we produced as the project progressed, and the general management of the project. We had approximately fifteen term-time weeks in which to complete the project, demonstrate its functionality to our supervisors, submit a project dissertation, and give a short project presentation. We had access to the Computing Science LegoLab room, which contains all the Mindstorms equipment, and 2 networked PC's. We shared this room with postgraduate students and another 3rd year project team, and were also responsible for arranging suitable access times.

The heart of the Lego Mindstorms kit is the RCX brick. This contains a Hitachi microprocessor, and has an LCD display and an infrared communications port. Programs are written and compiled on the PC, and downloaded to the RCX by means of an infrared 'tower', connected to the PC. The RCX allows three inputs and three outputs to be connected to it. The outputs for our kits are simply motors. The inputs are a variety of sensors, such as touch, light, rotation and temperature. Previous Electronic & Software Engineering teams have developed their own sensors. The RCX can gather data from these sensors and react to its surroundings. A line-following robot is a common demonstration application of the Lego Mindstorms robots.

The most commonly used programming language used with the kit is NQC. This is syntactically very similar to C, and is relatively powerful and easy to use. It allows the user to program using some of the features you would expect in C, such as loops, if-statements, and tasks. The RCX has limited RAM (Random Access Memory), and so there is a limit on the size and power of NQC programs.

The remainder of this report covers an overview of the entire system, detailed design descriptions of the hardware and software components, summary of the construction and integration, status report, user guide, a section on performance and reliability testing and a section which details how valuable the project was as a learning experience.

2 Specification

2.1 Hardware Specification

Some form of robot structure will be required to be able to park and retrieve various cars. Such cars will be model toy cars, and are not required to be built from Lego. On the other hand, building the robot from Lego will enable us to build an impressive and challenging structure and give a suitable example of the Lego RIS Kit's capabilities. Initially, it has been specified that the time taken to retrieve or park a car should be no longer than thirty seconds, which seems an acceptable period of time for the customer to wait on arrival or exit.

Furthermore, it is required to incorporate a form of entry system, to make the project as realistic as possible. Again, this will be built from Lego to give another example of the Lego RIS kit's capabilities. On entrance to the car park, a 4 digit ticket number will be given to the customer, preferably being displayed on an LCD, if possible. To provide a sensible and impressive demonstration, it was decided to provide 10 parking bays, with 2 levels having 5 bays each.

Finally, an LED display sign will be built allowing us to utilise our electronics design skills, which will either display how many bays are available in the car park, or display a message when the car park is full.

2.2 Software Specification

The software will control the robot and entry system via the RCX bricks accompanying the Lego RIS kit. The bricks will have to be able to communicate with a PC, sending and receiving messages to be acted upon.

The software that controls the barrier module should be intelligent enough to detect when a new customer should be admitted to the car park. It should lift to allow customer entry when the robot is not in the middle of another process (i.e parking or retrieving another car), and also when the robot is back at its starting point with an available tray to hold the car.

In addition, the software should provide a mechanism for a customer to retrieve his/her car by entering the allocated ticket number into a user interface.

In the event of error or failure of the automated car park system, software will be produced to enable manual control of the car park. This system will be used by an employee of the car park when there are problems with either the customer user interface, or with communications within the system.

Finally, to make the project additionally challenging, it is desired that information about the car park be retrieved over the Internet. Details such as

parking fees so far, and time in a bay should be presented on receipt of a current valid ticket number.

Figure 1: Overall system

3 Overall Design

The final project deliverable consisted of a number of integrated hardware and software components. A picture of the final system is shown in Figure 1.

The hardware is a combination of Lego Mindstorms and PC components, while the final software deliverable is a mixture of NQC, Java and LejOS programs. The NQC, LejOS and some of the Java programs control the hardware, while the remainder of the Java programs are for the User Interfaces and Applet.

The overall system can be broken down into a number of parts:

Hardware

- Base unit
- Lifter
- Extending Arm
- Barrier

- LED Display
- PCs/IR Towers

Software

- NQC
- LejOS
- Start-up Menu (DOS)
- Java Communications
- Customer UI (Java)
- Controller UI (Java)
- Applet (Java)

3.1 Hardware

3.1.1 Base Unit

The Base Unit, constructed using standard Lego Technic bricks, provides movement along a fixed track. Connected to an RCX programmable brick, it achieves movement through the use of two standard Lego motors and position control through a light sensor.

3.1.2 Lifter

The Lifter unit is also constructed using Lego Technic and is connected to the same RCX as the rest of the parking mechanism components. The Lifter unit provides horizontal lifting movement along fixed guide rails, using worm gears for greater stability. Again, it uses two standard Lego motors and a light sensor for position control.

3.1.3 Extending Arm

The Extending Arm is attached to the Lifter unit and exists to provide a 'fork-lift' like component to actually park and retrieve the cars. It is constructed with standard Lego Technic components, with a motor and two touch sensors providing control and position feedback to the RCX.

3.1.4 Barrier

The Barrier is controlled by a second RCX programmable brick, and is used to simulate the entrance to the car park. It uses a motor to drive the barrier, a combination of rotation and touch sensors to control the barrier position, and two light sensors to detect passing or queuing cars. All of these components are standard Lego Technic parts.

3.1.5 LED Display

The LED Display is not a standard Lego Mindstorms component and was specially constructed for the project. It connects to the serial port of a controller PC and displays alphanumeric text via two 4-segment LED modules. The display requires a separate 9V regulated power supply to operate.

3.1.6 PCs/IR Towers

At least two PCs are required to operate the system. One will act as a client, and the other as the server. The PCs are required to have the NQC and LejOS environments installed, to allow the compilation and downloading of NQC and LejOS programs to the RCX bricks. Also, the PCs must have the Java Developer's Kit (JDK) version 1.x, or higher installed, to allow the User Interfaces and the Applet to function. The RMI (Remote Method Invocation) and Swing components will only function with recent versions of Java. A final requirement is that the PCs are networked to allow correct functionality of RMI and the User Interfaces.

The IR transmitter towers should be connected to both PCs via the serial ports, to allow two-way infrared communication between the RCX bricks and the controller programs on the PCs.

3.2 Software

3.2.1 NQC

The NQC (Not-Quite-C) software is used to control the parking mechanism robot. NQC is syntactically similar to C, with similar functions, such as loops, if-statements, and tasks. There is a single NQC program downloaded into the RCX which controls the robot.

The code listens for IR messages from the PCs and responds appropriately, parking or retrieving a car. It can also send IR messages back to the PCs. The NQC program has no user intervention and communicates only with its con-

troller program on the PC.

3.2.2 LejOS

LejOS is an alternative RCX programming language, and allows special Java programs to be downloaded into the RCX. It provides all the functions of NQC, but allows greater control of the RCX, as it uses its own LejOS firmware.

The LejOS code controls the barrier through IR communications with the PCs.

3.2.3 Start-up Menu (DOS)

A simple start-up menu was written as a script for a freeware DOS-based menu system. It allows the user to start all the other programs, and download firmware and programs to the RCX bricks. A special option allows the user to enter the light values for the light sensors on the robot. These values are then turned into an NQC header file, which is used in the main NQC program.

3.2.4 Java Communications

The Java Communications software is at the heart of the system and co-ordinates all the functions of the system. It controls IR messaging between the PCs and the hardware, and allows the User Interfaces and Applet remote access to the hardware functions. The software implements a specially defined IR protocol for messaging.

3.2.5 Customer UI (Java)

The Customer User Interface is implemented in Java Swing, and provides a graphical interface from which users can activate the car retrieval hardware. The interface achieves this through RMI calls to the Java Communications software.

3.2.6 Controller UI (Java)

The Controller User Interface is also implemented in Java Swing, and is designed to simulate the functionality that might be available to a controller of the system. It provides special operations not available to normal (customer) users, and shows graphically the status of the system. This again is achieved through RMI calls to the Java Communication programs.

3.2.7 Applet (Java)

The Applet is again implemented using Java. Through RMI, it provides a simple graphical interface through which statistics about the car park can be viewed. The Applet is designed to be accessible from any PC on the same network as the client/server PCs.

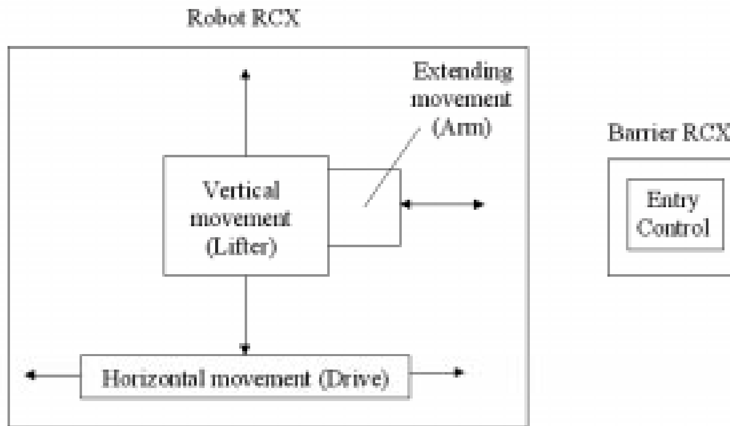

Figure 2: Hardware structure diagram

4 Detailed Design

4.1 Hardware Design

An abstract description of the hardware is shown in Figure 2.

4.1.1 Drive

- Rationale

  We needed a way to move the robot horizontally to the parking bays, and to the pickup point for the cars. The drive has to be reasonably fast, as we do not want to keep customers waiting too long for their cars. We decided that the maximum time it should take to travel from the pickup point to the furthest away bay should be ten seconds. It was also decided that the system should be able to stop within 5mm of the designated stopping point, every time. This way, we were able to leave around 1cm of 'free space' around each bay to allow for these tolerances.

- Design Description

  The drive system of the robot is relatively simple. We decided early on that we would use a track of some sort, so that the robot would not need to be steered. This was done because of fears that there would be insufficient memory in the robot's RCX to implement steering, as the program would already be relatively large. We also wanted the drive to be quick and reliable. Implementing steering would have slowed the system down, and possibly made it unreliable, as extra sensors and code would be required, so was quickly discarded as a bad idea.

Figure 3: Configuration of drive gears

The track is constructed from a number of Lego narrow beams joined together into a long row. The base of the robot incorporates a 'slot' made up of the same type of beams, which fits the track exactly. The robot slots onto the track, and is able to move only up and down the length of it. This can be seen in Figure 4.

The drive uses two Lego Mindstorms motors, which are connected to a single input on the RCX. These are geared together so they turn at the same rate, and connected using more gears to the robot's axles. The gearing ratio is not too low, so the robot is able to pick up some speed. Both front and rear axles are driven. This was done to reduce the load on the motors, and to make the drive smoother and more powerful. The configuration of the gears is shown in Figure 3.

The robot's position is detected using a light sensor, which is pointed at a line of black tape, which runs parallel to the track. Each bay has a corresponding blue marker on the tape, and when the robot reaches the correct marker it slows down, and constantly checks the sensor. When the end of the marker is reached, the robot is in the required stopping position. The car pickup point is found by looking for a white marker, which is at the end of the tape. The light sensor is shielded from external light, but is still sensitive to ambient light values, and even more so to weak battery levels. We have found, however, that as long as the sensor is calibrated fairly frequently, and the batteries are changed regularly, this system retains its accuracy, and is able to stop correctly, at the required

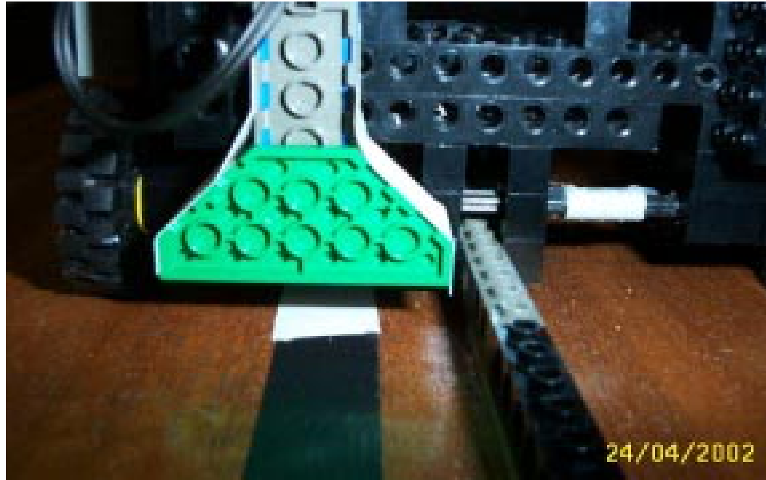

Figure 4: Light sensor for position control

bay, within five millimeters of the end of the marker.

- Changes

  Initially, the chassis was built, and the drive fitted with only one motor. This was sufficient at the time, but we found that as the lifter mechanism was built, the drive became to weak to move the robot fast enough (in fact at times, the robot could hardly move!), due to the added weight. We experimented with different gearing arrangements, but these were found to be too slow. It was decided that we would add a second motor to the robot's drive, and gear it together with the first. This worked reasonably well, but the motor still struggled to turn the axle when the batteries started to fade, and the axle was bending because of the strain on it. This resulted in gears slipping. Finally, the gearing was redesigned so that the motors would drive both axles at once, thus reducing the load on the mechanism. This design worked very well, and we have not had any subsequent problems.
  The only change made to the position sensing system was to add a 'shield' to the sensor to block external light. This was built from two Lego 'angled' plates, and some insulating tape, and has worked very well.

- Other Requirements

  The drive system was to use one single RCX input and one single RCX output only, because the other ports were reserved for the height and extending arm control systems.

- Status

The drive mechanism is built and works very well, as long as the RCX batteries are changed regularly. It is able to reach its target bay within six seconds, and stops between two and three millimetres from the designated stopping position. Occasional re-calibration of the sensors is required, due to ambient light levels changing.

4.1.2 Lifter

- Rationale

  The lifter of the robot is needed since there will be two levels of bays in the parking area. It will be used when a bay on the upper or lower level is accessed.

- Design Description

  A rectangular platform, 16 dots by 10 dots wide, sits directly on top of the driver part of the robot. It climbs, vertically, 4 pillars which are mounted vertically at the corners of the robot, as shown in Figure 6. Two motors are mounted on the platform which directly drive two worm gears. The worm gears are crucial to this part of the structure as they can turn cogs with less speed, thus producing greater power and stability. This is known as "gearing down" in Mechanical Engineering. Each worm gear sits on top of a 16 teeth cog, both of which are attached by a metal axle. On the outsides of the metal axle, there is also another 2 16 teeth cogs which face the outside pillars. In the row of gears facing the pillars, as seen in Figure 5., There are 2 24 teeth cogs, touching each pillar with 2 16 teeth cogs in between. To add yet more balance to the platform, keeping it parallel to the ground, we added another 24 teeth cog under each 24 teeth cog in the gear row. The pillars are 2 by 12 dot, black poles high to which we attached 6 teeth bricks to each pole, enabling the 2 gear rows to climb vertically.

  The coloured tape layout is shown in Figure 14. The height of each colour is as follows:
Black-4cm
Green(upper)-2.5cm
Orange-1cm
Green(lower)-1.5cm
Blue-2cm
White-3cm

  The tolerance for stopping at the required colour of tape is +/- 5mm for each colour.

Figure 5: Lifter gears

- Changes

  There have been major design changes to the lifter before arriving at the above final design. Firstly, instead of trying a climber solution, we attempted a pulley mechanism. The idea behind this design was a motor would turn a plastic axle, to which a thin piece of rope would be attached. On turning of the motor, this would wind the rope around the axle, thus lifting the platform to which was tied to the rope at the bottom of the system. However, this was a very naive, quick first design. It was immediately discounted as it did not lift the platform up parallel to the ground - the platform would collapse lopsided at one end just before it lifted. As a result, we thought of implementing the climber design. Similarly, our first attempt at the climber was unsuccessful. The system made it lift and drop too quickly, which resulted in the lifter breaking off the driver base when it stopped at the bottom of its lift. As stated above, the final design remedied both problems of our initial design attempts.

- Other Requirements

  As we were developing the robot parker in three stages simultaneously, the lifter had to be designed with the consideration that it somehow had to to attach to the driver, and also be flexible enough to allow the parking arm to be attached to it. Furthermore, it was also further required that it be implemented using only one input and one output.. Despite using two motors in our lifter design, these motors are only ever used at the same time so they occupied the single allocated output port. The input was used to control the height of the lifter - specifically when to stop the

Figure 6: Lifter gears climbing pillar teeth

motors when it had reached the top or bottom of its lift. A light sensor served this purpose perfectly - white tape was placed at the bottom of a pillar with black tape placed at the top - when the light sensor detected white or black, the motors would be turned off.

- Status

  The group are extremely satisfied with this design, especially in terms of its strength and reliability. During a test which repeatedly lifted the platform up and down for 30 minutes, the lifter worked to specification continuously without any adverse affect on the strength of Lego structure of the robot.

4.1.3 Extending Arm

- Rationale

  A hardware component was required that was able to 'pick-up' a car from the drop-off point and place it securely into a parking bay. Vertical movement was being provided by the lifting mechanism, and the motorised base allowed us to move along the ground. The Bay Parker offers us horizontal extending movement. When integrated with the lifting mechanism, we have a unit capable of variable-height extension.

- Design Description

The Bay Parker consists of a main unit and an extending arm. The main unit contains a motor and two touch sensors.

The extending arm is simply constructed from standard Lego black beams, with 'teeth' plates on top. At the end of the arm are some small blocks which are placed to trigger the touch sensors, indicating either full extension or full retraction of the arm. At full extension, the arm extends approximately 6cm from the body of the main unit. At full retraction, the arm protrudes approximately 5cm from the back of the unit. There are sufficient free connectors on the front of the bar to allow the fixing of the 'prongs', which will actually lift the car, in a fork-lift like manner.

The main unit is constructed on a standard Lego baseplate. The motor sits at one side, raised slightly, and simply turns the extending arm directly via worm gears. This gives smooth, controlled movement of the arm. The base of the unit has been fitted with small black rods. These allow the bar to slide easily in and out of the unit. Two touch sensors are incorporated into the main unit. One, situated just beside the motor, detects full extension, while the other, situated at the back of the unit, detects full retraction. These touch sensors are raised so they can only be activated by the built-up blocks on the arm, rather than the edge of the arm itself. The sides of the unit were built with standard Lego black beams, and another baseplate was used on top of the unit, to provide extra stability.

The extending arm will be responsible for holding a cardboard tray, which carries the car to be parked. However, as a result of the horizontal movement when the drive system is in operation, we have to beware that the car will not fall from this cardboard tray. Obviously in a real life situation, a car being carried on a moving platform would not fall off as its handbrake would be enabled. As a result, we will stick the car down on to the tray to mimic this 'handbrake' action, on entrance to the car park.

- Changes

  The most major change to the design of this module was a redesign so that the two touch sensors can share one input on the RCX.

  The initial design had the sensors located in different areas of the unit, and the extending arm had different parts added to it to trigger the sensors. While the sensors still detected full extension or retraction, there were times when both sensors were triggered. This meant that there was no way for the inputs to the RCX to be combined. The redesign meant that the blocks on the arm were moved, and the location of the two sensors was changed. There was now no situation in which both sensors could

Figure 7: Extending arm mechanism showing worm gear

be triggered at the same time. Also, the redesign meant that the overall footprint of the unit was decreased, which made the unit slightly smaller and lighter, which should make it easier for the lifting mechanism to handle the unit.

The only other change to the initial design was using worm gears to turn the arm. This can be seen in Figure 7. Initially, the motor simply turned a cog directly onto the arm. This was approach was jerky and too fast, causing the cars to fall off the parker.

- Other Requirements

  As there are only three sensor inputs available on the RCX, and two are already taken up for height and position sensing, both touch sensors need to be on the same input.

- Status

  The Bay Parker is in working order and has successfully been integrated onto the lifting mechanism.

4.1.4 Barrier

- Rationale

A system was required to control access to the car park. We needed a system that would only let cars into the car park when the car park was ready to accept them, and would also refuse access to the car park when it is full. We also required a system which would issue ticket numbers to cars as they enter the car park, so that the customers could retrieve their cars later.

- Design Description

  The barrier system consists of one RCX brick connected to a barrier which is made out of Lego. The barrier is constructed from standard Lego Mindstorms parts, and is driven by a single motor, via a worm gear. The motor shaft is connected to a rotation sensor, which counts as the motor turns. When the motor has turned a set number of times, the barrier is known to be in the lowered position, and the motor stops. There is also a touch sensor which is used to detect when the barrier is fully raised. As the barrier reaches the raised position, a worm gear which is thread onto it presses down on the touch sensor, and the motor is switched off. The rotation sensor is then reset to zero. This means that the rotation sensor is re-calibrated every time the barrier is raised, and this increases the accuracy of the system.

  The system also incorporates a dual light sensor arrangement, to detect if there is a car waiting to enter the car park, and if the car is blocking the barrier, once it is raised. This is shown in Figure 8. The two sensors are connected to the same input, and are pointed directly at each other, diagonally across the path of the 'road' beneath the barrier. Once this beam of light is broken by a car, the light value read by the sensor drops by a a large amount, and the barrier is raised. The barrier is then only lowered once the beam is again clear, and the car has passed.

  The ticket issue system makes use of the RCX unit's LCD display and buttons. Once the barrier is raised and a car is in the car park, a four digit ticket number is displayed on the LCD display. This can be seen in Figure 9. The system then waits for the customer to press the RUN button on the RCX, indicating that they have taken the ticket number, and the car is ready to be taken away. As soon as this button is pressed, the car is taken away and parked.

- Changes

  The barrier system, like all other parts of the car park, has undergone significant evolution. Initially the barrier was programmed in NQC, and the ticket issuing was going to be a separate part of the system. We discovered, however, that it would be very easy to have the ticket number displayed on the LCD display of the RCX, by using the LejOS Java firmware for the RCX. Once the barrier had been reprogrammed using Java, this feature

Figure 8: Barrier system with entry light sensors

Figure 9: RCX unit showing ticket number

was added.
The physical structure of the barrier has also changed somewhat. Initially the barrier used standard Lego gears (rather than worm gears), and the rotation sensor was connected to the shaft of the barrier itself. This resulted in very fast jerky movement of the barrier, and the position sensing was far too inaccurate. This was due to the fact that the rotation sensor only counts every 12th of a full rotation, i.e. every 30 degrees. By incorporating a worm gear, and moving the sensor onto the motor shaft, we were able to both slow the movement down, and increase the accuracy of the rotation sensor, since the sensor now turns several times for every rotation of the barrier shaft.
Many changes have been made to the 'car sensing' system. Initially this used one light sensor to detect when a car had actually climbed the ramp, and was ready on the platform, indicating that we can close the barrier. It also used a light sensor in front of the barrier, to detect any cars waiting. These sensors were on separate inputs (one of the sensors was connected to the same input as the barrier touch sensor). This design had severe limitations, since the light values detected varied greatly between different colours of cars, and when lighting conditions changed. For example, a red car would give a very small rise in light value, or no change at all, and a blue car would give a drop in light value (since the light used by the sensor is from a red LED). The end result was that it was impossible to detect whether a car was present or not. The solution was to connect two light sensors to each input, and point them directly at each other. This resulted in a very high value (close to 100) when there was no car, but a significant drop (down to around 60, depending on the colour of the car) when a car entered the beam.

The final change we made was to remove the sensor which was connected in parallel with the touch sensor of the barrier (which detected whether a car was present). Due to the input being used in two different modes(Boolean touch and percentage light), the sensor would sometimes detect the value incorrectly for an unknown reason. This made the system unreliable, so we decided to remove this sensor. The other remaining sensor was then placed diagonally across the road, and could then detect both a car waiting, and when the car had cleared and the barrier could close.

- Other Requirements

  The system was to be implemented using one single RCX, in order to cut down on infra-red communication. We were therefore restricted to three inputs. Once it had been decided that the barrier system should control ticket issuing, we knew that the barrier RCX had to be programmed in Java, rather than NQC.

- Status

Figure 10: LED Display circuit board

The barrier system is now in fully working order, and works very reliably with no problems. The 'car sensing' also works well because of the dual sensor arrangement, even in vastly changing light conditions, and with any colour of car.

4.2 Electronics design

4.2.1 LED Display

- Rationale

  Although not an 'essential' feature, a scrolling LED display provides a very useful description of the status of the car park to potential customers, and was thought to be a worthy addition to the project. It also gave us a chance to try out some of our electronics skills. The final circuit is shown in Figure 10.

- Design Description

  The LED display is made up of two Siemens SLR2016 alphanumeric dot-matrix LED display units, coupled to some 74LS164 shift registers. A detailed circuit diagram is shown in Figure 11. The circuit also contains a 7805 5V voltage regulator, which maintains a constant 5V. It can then be powered by an AC-DC adapter, which produces a voltage in the range

of around 9-16V. The display is interfaced to the serial port of a PC, and data is written to it using custom software, as described in the Software Design section.

The design works as follows: The shift registers are cascaded together to produce a 24-bit shift register, of which 18 bits are used. The serial input (to the left-most shift register in the circuit diagram) is connected to the DTR line of the PC's serial port. The clock line of the shift registers are connected to the RTS line of the serial port. Both of these lines have Zener diodes to ground, which prevents voltages greater than the supply voltage from appearing across these lines. The TXD line of the serial port is connected to the active low 'write' input of the display modules, via transistor T1. If we now pulse the TXD line, any data on the shift register's parallel output will be written to the display modules. All that is required to put data onto the shift register's parallel outputs is to set the DTR line high or low as required, and then pulse the RTS (clock) line to shift the data into the register. Once we have done this 18 times, we have the required data word on the register's outputs. As can be seen from the circuit diagram, the data word on the shift register's output is connected to the display units as follows, from left to right: display 1 - D0...D6 - A1...A0 - display 2 - D0...D6 - A1...A0. D0...D6 represents the ASCII code for the character which we want to send to the display, and A1...A0 is an address representing the display section we want to display the character in (e.g. position 4 ... position 0). The data word we need to send is then DATA + ADDRESS + DATA + ADDRESS (from left to right), followed by a pulse on TXD to latch the data into the displays. The required characters are then automatically displayed by the LED display units, in the required positions. Since one character is written to both displays at the same time, it takes 4 of these cycles to fill the display with text. More information on the Siemens LED display unit is provided in the data sheet in the appendix.

The circuit was prototyped on strip-board, and we originally planned to build a PCB. However, it was decided that since we had other tasks to complete, and little time to develop a PCB, the strip-board circuit would suffice.

- Changes

  No significant changes were made to this design, other than those needed to correct mistakes in the circuit, due to soldering errors.

- Other Requirements

  We require a 'scrolling' display, so that we can display variable length strings. This means that data will have to be continuously written to the display, by the PC, in order to produce the scrolling effect. A design which holds a string of text and automatically scrolls it was deemed too complex, and probably too expensive to build for this project.

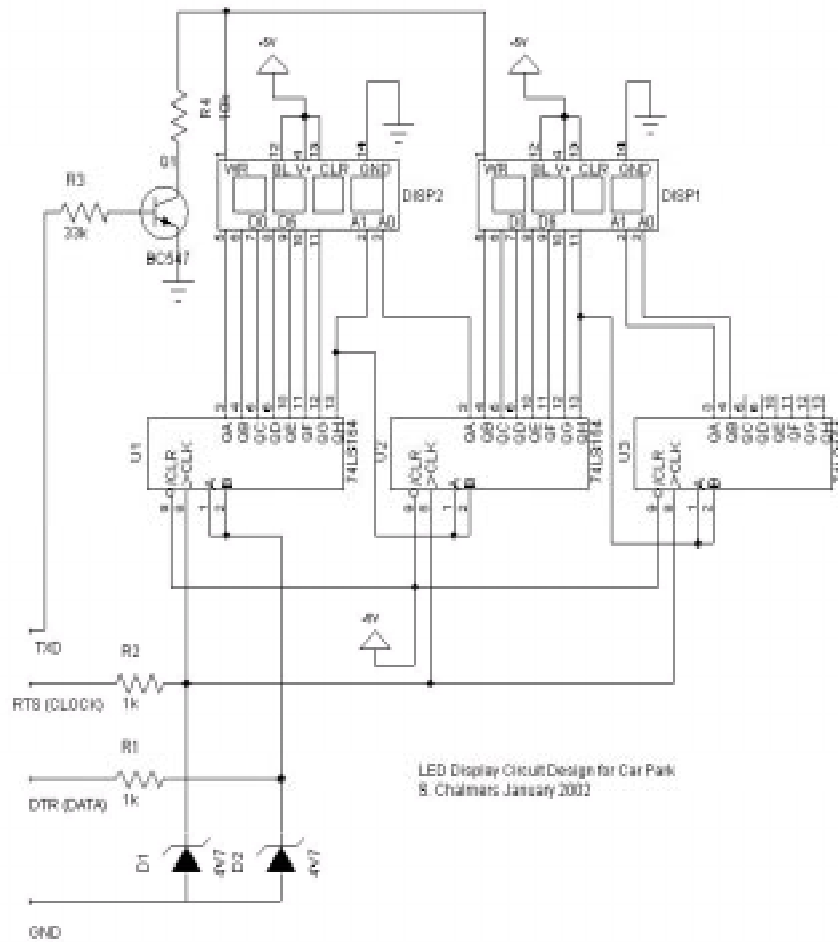

Figure 11: Circuit diagram of LED Display

- Status

  The LED display is fully built and working, albeit in its 'prototype' form.

4.3 Software Design

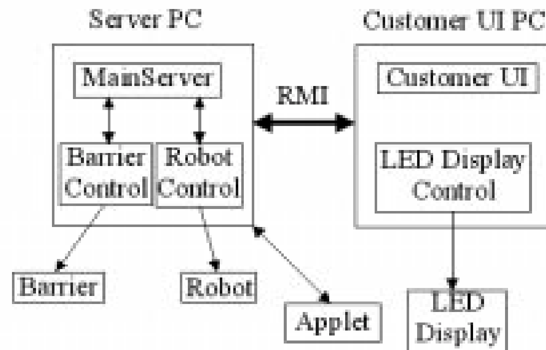

Figure 12: Software structure diagram

4.3.1 DOS Start-up menu

Figure 13: Startup menu

The startup menu is a simple script for a freeware menuing system called DougMenu. The script enables the user to enter light values for the robot, which are then stored in an NQC header file. This can then be compiled and downloaded with the main NQC program from another menu option.

It is also possible to send firmware to the RCXs and upload the Barrier program. The menu also allows the user to start the system's Java software. A screenshot of the menu is shown in Figure 13.

4.3.2 Robot and NQC Software

There is one single program that runs on the main robot RCX. This program receives messages from the PC, decodes the message and carries out the required function, and then signals to the PC that it has completed its task.

The main() function of the program is a simple message-listening loop. The program continuously loops, waiting until it receives a message specifically for it and that it understands. It then executes a number of internal functions, before returning to this messaging loop. The robot will continue to listen for messages until it is switched off, or the program is explicitly stopped.

Upon receiving a valid message, the program will call one of the following functions:

- system_reset()

  This function is designed to be called on start-up, or if the system is in an unknown state and requires to be reset. If this function is called, the robot will retract the extending arm and lower the platform, if necessary, and reverse along the track until its starting point is reached.

- goto_bay()

  This function parks a car. The robot will slightly raise the platform, to avoid hitting the ramps which the cars drive up. To do this, it raises the platform until the height sensor first reads green. (See Figure 14) Then, it calls the move_to_bay() function, which is described below. If the bay required is on the top row, move_to_bay() will already have raised the platform to the black marker. The arm is then extended, and the car will be lowered gently onto its supports. The platform lowers until the orange marker is reached. If the bay is on the bottom row, then the platform will be lowered until the white marker is reached, after extending the arm. The arm is then retracted, and the platform is lowered to its starting position if not already there. A message is then sent to the PC, indicating that parking is complete. The robot remains at the bay, until it is instructed to move by the PC.

- return_from_bay()

  This function works in a similar manner to goto_bay(), except that it collects a car from a bay and returns it to the starting point. This time, upon arrival at a bay, it extends the arm and gently 'scoops up' a car.

The robot then uses the system_reset() call to return it to the starting point, and then signals that the retrieval is complete

- stop_all()

  This function is designed to be called in emergencies. If the RCX receives this message, regardless of whether it is meant for this RCX, all the motor outputs are switched off, and the robot will not move any further.

- move_to_bay()

  This function is very important as it controls the position and height of the robot. It also holds the current position of the robot, so that when it is next required to move, it knows where it is.

  After being called, this function listens for another message. This is because after the Controller class sends a park or retrieve message, the next message always contains the bay number. Upon receiving this message, the function calculates what position it requires to move to. This is because bays 1 and 6 share the same horizontal position, as do bays 2 and 7 etc, as the bays are two rows high.

  The robot will then move to the required position. Using the light sensor on the base unit, it travels along the fixed track. Special markers have been placed alongside this track to give position information to the RCX. The layout of these markers is shown in Figure 14.

  The white marker indicates the start position. When system_reset() is called, it reverses until it reaches this white marker. When moving between bays, the robot counts the number of blue markers it passes. Once it reaches the start of the blue marker it is required to stop at, the speed is set to a crawl, and the robot stops when it reaches the end of the marker, as the sensor begins to read black. The robot is set to a crawling speed so that it stops accurately at the end of the marker. The markers have been aligned with the bays, and so accuracy was crucial.

  As the robot continues to hold the current position, when the robot is required to retrieve an empty tray or retrieve a car after another park, it simply moves along to the required bay, instead of having to return to the start position and counting from there. This greatly speeds up the parking and retrieval process.

  After the robot has reached the required horizontal position, it will raise the platform to the black marker if the bay required is on the top row.

Then this function returns control to the goto_bay() or return_from_bay() functions.

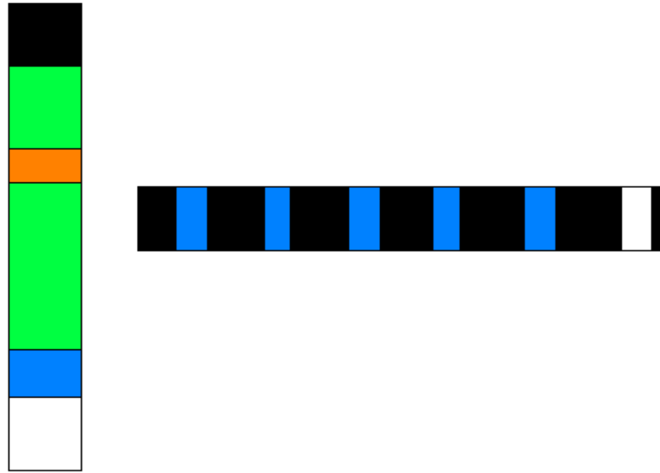

Figure 14: (Layout of markers for height (left) and position control (right). For the height markers, white and black are the end points between which the lifter will move. The blue is used to give a point where the lifter can be raised to for parking on the bottom row of bays, and the orange marker is used to give a point where the lifter can lower to when parking on the top row of bays. On retrieval of a car, the lifter raises from the orange and white markers, to 'scoop-up' the parked car. For the position markers, white is the starting point and the blue markers are the points where the robot should stop, as they are aligned with the bays.)

The NQC program also contains two other functions, arm_extend() and arm_retract(). These are small functions, and are required since a special algorithm controls the arm movement, as both touch sensors are connected to the same input, a situation requiring special control.

The program is quite large, and of the 26 kbytes available to the programmer once the firmware has been downloaded, only about 4 kbyte remained free.

4.3.3 Communications PC - RCX

The Java communications code is based on the Java Tanks Demonstration by Sun Microsystems. The host PC is responsible for sending commands to both RCX bricks, one for both the robot and barrier. It is also responsible for waiting

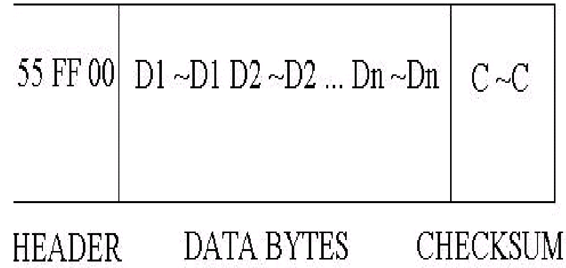

Figure 15: Format of IR message packet

for a response from the RCX bricks before continuing.

Therefore the communications module is split into three parts; firstly, the host PC encapsulates the functionality of the robot and barrier, and is used by the CustServerIntf and ControllerUI classes, as well as the Applet class, to extract information about the car park from the Controller class. Secondly, the robot RCX brick is responsible for interpreting messages sent from the host PC to it, and executing the desired function. This is written in NQC. Similarly, the barrier RCX brick again is responsible for interpreting messages sent to it but this is written in LejOS, not NQC.

The communications module is responsible for the reliable sending and receiving of messages, ensuring that only the intended recipient acts on the desired function. However, although RCX bricks can send and receive messages easily, it is not as simple for PC's to send and receive such messages in accordance with the Lego Mindstorms IR protocol.

The RCX expects infra red messages of the following form, as shown in Figure 15:

The data to be sent is prefixed with an F7 - the opcode to send messages to the RCX.
The RCX serial port operates with the following parameters:

- Stop bits - 1
- Parity - odd
- Data bits - 8
- Baud rate - 2400

This information is important as it is needed by the Java communications API to function properly.

Here is a brief description of the classes which are part of the communications module

- IRProtocol

  This class imports the Java communications API (import javax.comm.*) to give low level control of the serial port, and is converted from the Java Tanks Demonstration.

  The IRProtocol constructor configures the serial port in accordance with the RCX parameters (see above). In addition, it also creates a MessageBuffer instance (described later) before printing out diagnostics to the DOS window, to inform the user that the constructor has been called correctly.

  The serialEvent() method ensures that when there is data available on the serial port, it gets appended to the MessageBuffer instance.

  The addMessageListener() method allows the host PC to receive messages from the MessageBuffer class.

  The send() method is a very important method to this class. It creates an array of size (2 * lengthofdatatosend + 5) i.e the 5 bytes are made up of the 3 byte header 55 FF 00 + the checksum and the checksum's twos complement, and the (2 * lengthofdatatosend) bytes are made up of each data byte followed by its twos complement. This method then stores the desired packet in this format and sends it via the IR tower.

- MessageBuffer

  This class is again converted from the Java Tanks Demonstration by Sun Microsystems. It buffers any messages sent over the IR interface and notifies the host PC upon reception of a complete message, as some messages are broken up into several chunks. It notifies the host PC by calling messageReceived() from MessageListener.

- MessageListener

  This is a simple interface which must be implemented by IR protocol clients that wish to receive asynchronous messages.

- Brick

  This class adds a layer of functionality on top of the IRPrototcol class. Indeed, it still allows sending of messages from PC to RCX but this class

implements the MessageListener interface, allowing it to receive messages from the MessageBuffer class.

The sendMessage() method simply calls IRProtocol's send() method, and catches any exceptions this may throw.
The getAnswer() method returns a value sent by either the PC (as the IR tower picks up messages it sends as well as those it receives), or more importantly, sent by the RCX brick.

- Controller

  The controller class is very important to the communications module. Firstly, by calling high-level, abstract, methods such as takeToBay() or sendTicketNumber(), it allows the host PC to give specific commands to either the robot or barrier.

  Such methods make use of the waitForRCXAnswer() method, which is effectively used to keep communications open between the host end and RCX brick. During initial testing of message sending between host and RCX, it was found that if the host PC sent an RCX an instruction and then waited for a reply that the RCX had finished a task, without continuously sending a dummy message, then the IR tower attached to the host PC would "shut off" for communication after a period of idle time. Therefore, waitForRCXAnswer() continuously loops, sending a dummy message in the code, this is a byte with value 0xF3, and then checks if the answer received is what is specified in the parameter to the call. If so, then the method ends otherwise, the loop continues until the desired reply is received. This period between sending of each dummy message is short enough so that it does not cause the IR tower to "shut off" communication.

The high level command methods sent to the the RCX brick from the host PC are as follows.

Robot

- takeToBay()
- returnFromBay()
- emergencyStop()
- reset()

Barrier

- sendTicketNumber()

In addition, the controller class needs to be able to provide the LED controller, CustServerImpl, ControlUI and Applet application, methods to extract information about the car park.

LED Controller

- getBaysRemaining()

CustServerImpl

- setCarReturn()

Applet

- getBaysRemaining()
- getCost()
- getTime()

ControlUI

- Uses all methods that Robot and Applet uses

4.3.4 Barrier and LejOS Software

- Introducing LejOS and the barrier control software

  The software running on the RCX which controls access to the car park (using the barrier), and issues ticket numbers to customers, is written in Java. A Java Virtual Machine, known as LejOS, has been developed, which can be uploaded to the RCX, and replaces the standard firmware. Once this has been done, Java programs can be written, using the LejOS API (application programmers interface) to access the RCX's sensors and outputs. The LejOS firmware also gives the programmer access to the RCX's LCD display and buttons.
  The LejOS program running on the barrier control RCX has two functions: firstly, it will only allow cars into the car park if it is safe to do so, i.e. if the car park is ready to accept another car. It does this by raising or lowering the barrier as necessary. Secondly, once a car has parked on the platform, the barrier software issues a ticket number to the customer, and tells the server that the car is ready to be taken away.

- Class Barrier

  The Barrier class, defined by Barrier.java, encapsulates all the functionality of the barrier system. In order to control a barrier connected to an RCX, we simply create an instance of Barrier, and call its instance methods.

- Constructor Barrier()

  The Barrier constructor initialises the barrier RCX, by setting all the required sensors to the correct mode, and activating them. It then sets infra-red transmission to long range, and clears the LCD.

- void raise() instance method

  This method attempts to raise the barrier, by turning on the motor, and entering a continuous loop which counts whilst the motor is activated. It waits for sensor 1 (the touch sensor) to be activated, indicating the barrier is raised and the motor can be switched off, and the rotation sensor can be reset to zero (automatic recalibration of the sensor). If this does not happen within a count of 3000, the motor is reversed, and the procedure is repeated to try to find the sensor. If this fails, the motor is switched off, and an exception is raised which halts the program. At this point 'Error' appears on the RCX's screen. This provides protection if the barrier ever gets stuck, and prevents damage to it.

- void lower() instance method

  This method lowers the barrier, and works almost identically to the raise() method. The main difference is that the motor is switched on until the barrier rotation sensor reaches a certain position, as defined by the lowerPos instance variable.

- void sendMsg() instance method

  This method uses LejOS's Serial class to access the infra red port of the RCX, and send a message to the server PC. It simply calls sendPacket() twice, to try and ensure the PC gets the message.

- int getTicketNo() instance method

  This method is called when the barrier system is ready to accept another car, and wants to receive a ticket number from the server. It waits for a message containing MSG_BARRIER (0x64) indicating that the message is for the barrier system. Once a valid message has been received, a reply containing MSG_CONFIRM (0x65) is sent. The next message received is the first part of the ticket number (e.g. 0x10 would indicate the ticket number begins with 16), and again we reply with MSG_CONFIRM. Finally we receive the second part of the ticket number, and once more reply with MSG_CONFIRM. The system is then ready to accept a car. A typical dialogue would look as follows:
  PC - RCX : 0x64

RCX - PC : 0x65
PC - RCX : 0x01
RCX - PC : 0x65
PC - RCX : 0x05
RCX - PC : 0x65

This would tell the RCX that the next ticket number is 0105, and this is the integer returned by the method.

- Boolean carWaiting() instance method

  This method simply checks sensor 3 (the light beam), and returns true if the value of the sensor is below cutoffWaiting (75). If so, this indicated that a car is present, and is blocking the beam.

- void showTicketNumber() instance method

  This method displays the int passed into it on the LCD display of the RCX. This uses LejOS's LCD class, and is used to display ticket numbers to customers. Once the number has been displayed, we use the Button class to wait for a press and release of the RUN button, indicating that the car is ready to be taken away.

- public static void main()

  This is the barrier's main program which is run when we start the RCX. The main() method creates a Barrier object, and then uses the instance methods which this object provides to control the barrier, and communicate with the PC.
  Once the program starts, we enter a continuous loop. Firstly, we get a ticket number from the server, using the getTicketNo() method of the Barrier object. Once this has been done, we enter a loop which counts, waiting for the light beam to be broken, indicating that a car is ready to enter the car park. If a car has not entered the beam within 15 seconds, then we send the message MSG_NO_CAR (0x67) back to the server. This indicates to the server that no car is ready, and the robot can go away and return any cars which might be waiting. This was a very important feature, which initially was overlooked. Without this time-out, a car waiting to be returned would only be retrieved if a car had been parked and the robot needed to retrieve a tray. Once MSG_NO_CAR has been sent, we return to the start of the main loop, and wait for the ticket number to be re-sent, indicating that the robot is again ready to accept a car. If the beam is broken whilst we are waiting for a car, the barrier is raised, and the light sensor is checked, whilst waiting for the car to leave the beam. Only once the beam has been clear for several seconds is the barrier lowered. This is a safety feature designed so that we are sure that the car is out of the way of the barrier, before it is lowered. Finally, the ticket

Figure 16: Screenshot of the Customer Interface

number is displayed on the LCD, and we wait for a press of the RUN button. This is accomplished using showTicketNumber(). The program then sends MSG_CAR_READY (0x66) to the server, indicating that the car is ready to be parked.

4.3.5 Customer User Interface and CustServer Software

- Customer User Interface

  The Customer User Interface is a simple GUI with a text field, to enter a ticket number. We had to choose between Java's AWT and Swing libraries to implement the interface. The AWT would have been a good choice as it was taught in the Advanced Programming 3 module, hence we are familiar with it. However, it was decided to use Swing as the interface would look the same on various platforms (i.e Unix, Windows, Mac) since Swing is platform independent whereas AWT is not. Moreover, the group also believe that Swing interfaces are more aesthetically pleasing than AWT interfaces, as it is possible to display icon's on buttons etc.
  As can be seen from the software structure diagram in Figure 12, the CustomerUI class is linked to to the Controller class. This link is via RMI (Remote Method Invocation) since the classes reside on separate machines.

- CustServer

As a result of the CustServer and the CustomerUI classes residing on different machines, there is a need for RMI, as we need the CustServerIntf class containing the methods it needs to call remotely, on the same machine as the CustomerUI class. RMI provides a way for Java classes on one machine to access classes contained on another machine, using TCP/IP. A diagram demonstrating this process is shown in Figure 17. On the other machine housing the Controller and CustServer classes, we need the CustserverImpl class which contains the implementation of the remotely available methods in the CustServerIntf class. The CustServer class then instantiates an instance of this implementation class, and binds it to the RMI registry.

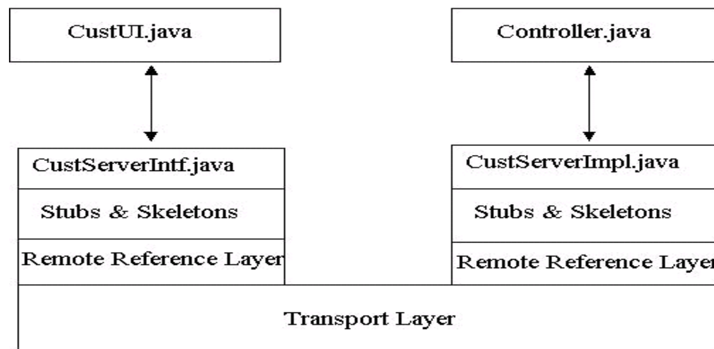

Figure 17: RMI Structure Diagram

4.3.6 LED Display Software

The LED Display software consists of a client and a server (LedServer class). The client runs as part of the Main Server program in its own thread. This thread continuously checks the number of bays remaining (using an RMI call

to CustServer) and the current custom message which has been set in the Controller user interface. This is done via an RMI call to another RMI server called UIServer, which has access to the data set in the Controller user interface. The client then updates the Led Display, using an RMI call to the LedServer. The LedServer is running on the same machine as the Customer User Interface, but as a separate process, and simply displays the required text, via an LedDisplay object, on the display. Originally, the LED display was going to be connected directly to the main server machine. However, we found that this machine ran extremely slowly, as it was continuously sending data to the LED display, and to the robot and barrier control system, over the serial port, as well as running the Controller user interface. This meant that not only was the machine slow to respond, and the running text display slow and jerky, but the server was beginning to miss messages from the robot and barrier, making the system far too unreliable.

It was decided that the LED display should be connected to a server on the Customer User Interface machine, and that the Main Server should send data to it in strings, which would drastically improve performance. Once this change had been made, we found the system much more responsive, and our infra red messaging error rate dropped substantially (from an error approximately every 10 messages to an error approximately every 30 messages). As of yet, we have found no reliability problems with the LED display. However, it can become slightly slower when the CPU's load increases. This effect could be reduced by having the LED display buffer the whole string, and then using a shift register to display the string bit by bit. However, this would have made the display more complex and expensive, and was deemed unnecessary.

- The LedDisplay Class

  The LedDisplay class is a general purpose class, used for writing text to an LED Display of the type we have built. All that is required to scroll text along the display is to create an instance of this class, specifying the correct serial port in the constructor, and then call the provided instance methods to write strings onto the display. This class uses the Java Communications API which can be downloaded from http://java.sun.com/.

- setDelay() instance method

  This method is used to set the delay time between characters scrolling on the display, in milliseconds. It simply updates the delayTime instance variable.

- setupPort() instance method

  This method is used by the LedDisplay() constructor, and sets up the port correctly for the LED Display. It opens the port, and prepares it to receive data from the server.

- closePort() instance method

This method simply closes the port associated with the LED Display.

- writeChar(char char1, char char0, int pos) instance method)

  This method writes two characters to the display. The first (char1) appears in the position (indicated by pos) on the leftmost display module. The second character appears in the same position on the rightmost display module. The method works by producing a binary string made up of the ASCII codes of the characters, and the address of the display position, and sending it to the display as detailed in the Electronic Design section.

- sendToDisplay(String s) instance method

  This sends a binary string to the display, as detailed in the Electronic Design section. The method looks at the string, bit by bit, and then sets or unsets the DTR line, according to the value of the bit. Next, a pulse is sent on RTS (the clock line) to latch the bit into the shift register. When all bits have been sent, we pulse TXD to latch the string into the display modules.

- (writeLine(String s) instance method

  This method writes a line of up to 8 characters onto the display. It works by going through the characters to be displayed on each section of the two displays, one by one, and sending them to the display.

- clear() instance method

  This clears the LED display. It simply calls writeLine(), to send 8 blank spaces to the display.

- displayRunning(String s) instance method

  This is the most important, and most frequently used method in the class. It allows the user to display a string of any length on the display, by scrolling the text across the display. This method works by entering a loop which looks at the substring which should currently be on the display, and displaying it, using writeLine(). After a short delay, the next substring is found, and it is displayed. This continues until the whole string has scrolled on and off the display.

- The LedServerImpl Class and LedServerIntf Interface

  The LedServerImpl class and its associated RMI interface LedServerIntf, are used to allow access to an LedDisplay object, via RMI, remotely over the network. The LedServerImpl class creates an LedDisplay object in its constructor, and provides remote methods for writing text onto the display. These methods mostly have the same name as those in the LedDisplay class, and take in the same parameters. They then simply call the associated method in the LedDisplay class, and return the result to the remote object, via LedServerIntf.

- The LedServer Class

  This class simply has a main method, which creates an instance of LedServerImpl, and binds it to the server name 'LedServer' in the RMI registry. This provides remote access to the LedServerImpl object's methods, via the RMI registry.

4.3.7 Controller User Interface

The Controller Interface is intended to be used by the system supervisor as it provides additional options not available through the Customer Interface. As with the Customer Interface, we implemented the interface in Java Swing, as it provided additional features that we required.

The main function of the interface is to provide a visual representation of the status of the car park. As can be seen in Figure 18, the centre component of the screen represents 10 bays, with their status displayed in 3 colours: gray indicates an empty bay, blue indicates an occupied bay, and yellow indicates that a car is scheduled to be returned. The status component is updated via calls from the Controller class.

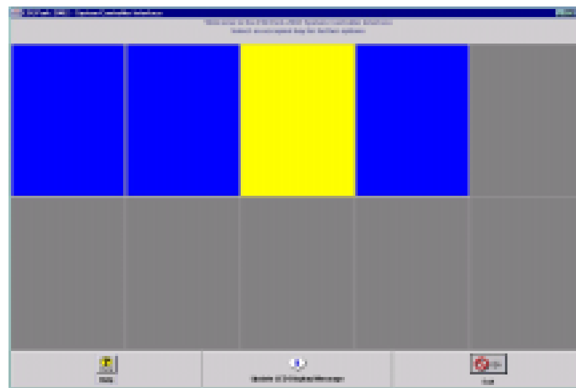

Figure 18: Screenshot of the Controller Interface

Selecting an occupied bay will bring up another window with further options, as can be seen in Figure 19. The supervisor has the choice of manually returning the car, checking the cost so far and the duration of the park, viewing a help dialog, or exiting back to the main screen.

Manually retrieving a car has exactly the same effect as a customer entering their ticket number in the Customer Interface. In this interface, however, no ticket number is required, as it is assumed that the supervisor has enough privileges to access this function. This manual retrieval works through RMI calls to

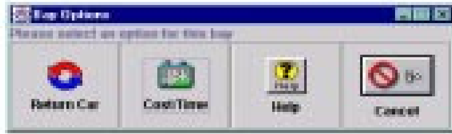

Figure 19: Screenshot of the Bay Options menu

the Controller class.

Selecting 'Cost/Time' from the Bay Options menu will bring up a further window. A sample is shown in Figure 20. This uses RMI calls to Controller to update the cost and time.

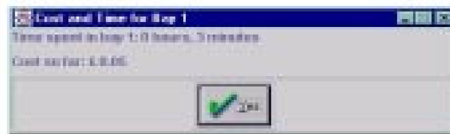

Figure 20: Screenshot of the Cost/Time window

The 'Update LED Display Message' option on the main screen allows the user to choose the scrolling message on the LED display. Selecting this option will bring up a new window with a text field in which users can specify the text to be displayed (See Figure 21) When the user clicks on 'OK', the text specified is immediately sent to the display via RMI calls. It is first concatenated with a message indicating the number of free bays.

Figure 21: Screenshot of Update LED Display dialog

4.3.8 Applet

The Applet combines the functionality of the Customer Interface and parts of the Controller Interface. It exists to provide users with a remote facility with which they can view statistics about the car park, their own car, and general information about the system.

The Applet was implemented in Java Swing, and had two tabbed panes. The first, Statistics, allows users to enter their ticket number via a small text field. (See Figure 22) This window also shows the current number of free bays in the car park. If the user has entered a valid ticket number, they will be presented with the duration of their park, and the cost so far.

The 'About' pane simply gives information about the car park, and has no functionality. (See Figure 23).

The Applet was embedded within a web page, and used RMI to obtain the time and cost data from the Controller class.

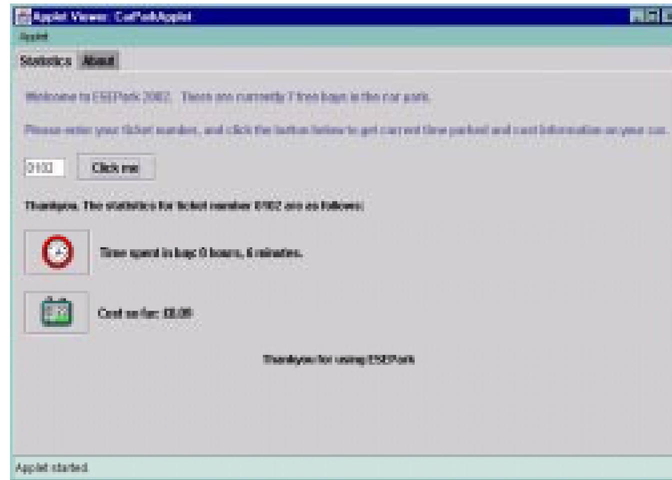

Figure 22: Screenshot of Applet 'Stats' page

5 Implementation and Integration

5.1 Implementation

5.1.1 Hardware

The construction of the different hardware modules was done concurrently. Once the lifter was completed, it was integrated onto the base unit, and the extending arm component was then attached onto the lifter.

The LED display and barrier unit are both independent of the robot and each other, and so were constructed separately.

5.1.2 Software

The NQC control program was implemented alongside the construction of the robot. The code evolved with the hardware, and as modules were added to the robot, the appropriate controlling procedures would be coded in NQC.

The communications software was adapted to suit our needs, and once the hardware reached a reasonable level of completion, integration between the two began.

The User Interfaces and the Applet were the final components to be implemented, since they required the hardware and communications to be fully operational.

Figure 23: Screenshot of Applet 'About' page

The two User Interfaces and the Applet were implemented concurrently.

5.2 Implementation Problems

5.2.1 Hardware

Every part of the final robot design was changed during the project due to implementation problems. The exact changes made to each component are described in Section 4.1. All of these changes were required due to the weakness and instability of the Lego components, either to function on their own or when another module was added to them. Once each module had been redesigned and rebuilt, we encountered no more major hardware difficulties.

We did however encounter intermittent minor problems with the final hardware. Occasionally, due to either decreased RCX battery levels or changes in the ambient light, or both, the light sensors used for position and height control would return different readings to the robot RCX, resulting in positioning errors.

Our solution to this problem was to place a cardboard 'shield' around the height sensor, and construct additional Lego parts around the position sensor. With these modifications in place, the problem with changes in the ambient light levels was greatly decreased, and this problem did not occur as often as it previously had. Keeping the RCX powered with fresh batteries was the only solution to prevent weak batteries causing any problems.

Light levels also caused problems with the barrier entry detectors. If the light level was too high, then the barrier had difficulty detecting an approaching car, and the barrier would not raise to admit it. The barrier was also redesigned during the project, and this, along with careful monitoring of light levels and adjustments to the sensors, meant that the barrier functioned correctly.

The LED display had no implementation issues, but did however have an effect on the PC which ran its control software. As the display constantly required messages from the PC to update it, this meant that the PC ran very slowly and had difficulty performing other tasks while the LED display was in use. In the final system, the only other component using the same PC was the Customer User Interface. As this was just a simple program, there was no detrimental effect to it from the LED display, and the speed of the scrolling message on the display was not adversely affected, as the interface was not a processor intensive application.

5.2.2 Software

There were no major problems with the implementation of the software. The only issue was with the messaging between the RCXs and the PCs. Occasionally, the PC would pickup a random unexpected message, or the controller PC would not correctly receive a message from an RCX.

This random message was not being sent by the bricks, but was causing the controlling program to stop during its run. The origins of these messages are not clear, but it is possible that they are caused by messages from the RCXs interacting with messages from the PC that are sent at the same time. This problem was intermittent, and so did not cause major problems.

Conflicting messages was also thought to be responsible for the PC not correctly receiving RCX messages. We tried to solve this problem by increasing the number of times we sent a message from the RCX, and improving our error-checking protocols.

If we had more time, we would have improved the handshaking and error-checking protocols further for the communications. However, there is still the problem of conflicting messages, as it is difficult to prevent messages being sent from two sources at the same time, and it would still be necessary for dummy messages from the PC to keep the IR tower active.

5.3 Testing Strategies

The robot was tested by writing suitable NQC programs that tested each module thoroughly. The main requirement was that the robot position and height

sensing worked correctly. By writing suitable test procedures, this was checked properly. We also checked that the position sensing was correct by manually viewing the light level values on the RCX display. This showed us that the NQC algorithms were correct.

We were also able to check the function of the extending arm and of all the motors from the RCX display. The RCX display shows when touch sensors are being pressed, and also when a motor is operational, and in which direction it is turning. This function is especially useful as it means a motor does not actually have to be connected to the RCX to see what will happen when a program is run.

The communications was tested using special dummy messages and the RCX built-in sounds. We used dummy messages to send to the PC to ensure it was receiving correctly. By printing out the value it received, we could instantly check if the messaging was working correctly. As the RCX cannot display the value of messages it receives, we programmed it to respond with different sounds, depending on the message it received. This allowed us to check that the RCX was correctly interpreting our messages.

The Applet and the interfaces were tested using stubs and console comments. Until the RMI was successfully implemented, we placed comments in the code that sent messages to the console window indicating the status of the system. From this, we could tell exactly what state our program was in.

These testing strategies uncovered some minor bugs in the system that were successfully fixed.

6 Status Report

The group are satisfied that all requirements, as stated in the specification section, have been implemented successfully, to a level that all were presented with success at the final demonstration.

Firstly, a basic requirement was that the system had to be able to park a toy car. This requirement met specification with no problems, as the park did not ever take longer than the specified thirty seconds, even when the occupied bays were at the furthest point away from the robot (i.e bays 5 and 10). Secondly, another requirement was that the system should be able to return a car. This was possible from two user interfaces: CustomerUI and ControllerUI. CustomerUI was a simple GUI which asked a customer for a ticket number. If the entered ticket number was valid, this request was placed in a queue, and upon reaching the head of the queue, the car would indeed be returned to the customer, after an unavoidable retrieve time. ControllerUI was also specified to return cars. By clicking on an occupied bay on the interface, an option appeared which, in effect, performed the same operation as CustomerUI did. Upon retrieving a car from both interfaces, system information was updated that departed bay was now available again. This requirement again met specification - this was judged on when the robot actually started the retrieve of the car, not on the time taken between entering the ticket number from the interface and the receiving of the car at the system's starting point. This is because the system could have already been in the process of parking a car, or even more time consuming, in the process of returning a number of cars from a queue.

Moreover, the system was specified to incorporate a form of entry system, to be built from Lego. The resulting Barrier module was an effective entry system; the barrier was only ever opened to a waiting customer if it had a cardboard tray on the robot arm at its starting point. This also meant that the barrier could not open to waiting customers if no bays were available. For safety, the barrier gate was only ever lowered when the light sensors detected the car had passed through - this was much more effective than say a timed delay, in case a car had been stuck. A challenging requirement was to display a customer's ticket number upon entry to the car park on an LCD. Our initial design thought was to program the barrier in NQC. However, by programming the barrier RCX brick in LejOS, which gives access to alphanumeric display via the RCX display panel, this was a very effective and cheap solution to what we had first thought may have required a separate hardware module to be built. As a result of having to learn a new programming language for this module, LejOS, this part of the software design took a little longer than previously allocated in the schedule. However, the group believe that the time invested in learning this new programming tool was spent wisely as it provided us with a cheaper solution to what had previously been anticipated, which also managed to meet specification.

Nevertheless, the LED display sign, which was effectively the only means by

which outside customer's knew how many bays were available, did have to be built as a separate hardware module. As was sought in the specification, the sign displayed how many bays were available, when indeed there was space for customers cars, and displayed a message when the car park was full otherwise. The speed of updating how many bays were remaining after a return, i.e increment, or park, i.e decrement, is almost instantaneous, which was possible as the LED controller class is running as a thread, continually requesting the integer baysRemaining variable from the Controller class. Again, this module to a little longer than anticipated to design and construct, which was understandable considering the module's design complexity added to the fact that the majority of the project focus is on software. Despite this, the quality of the visual impression during demonstration is justification for the time spent in designing and constructing this impressive module.

Overall, if the group are critical of the standard achieved of any of the requirements sought in the specification, it would have to be the communications software. This is not saying that communications software did not work, it was more a problem with the reliability of it (More information can be found in 'Performance and Reliability testing' section). Nevertheless, despite this reliability short coming, the communications software was delivered to a standard where it was demonstrated successfully during the final demonstration. If the group was given more time to improve on any aspect of the project, this would probably be the first topic to start on, with the main focus on improving the reliability of it.

Since the group had accomplished all of its earlier and more important goals with reasonable success, it was decided to make an attempt at constructing an Applet which would present information about parking fees incurred thus far by an existing customer, time spent in the car park etc, over the Internet. Again, this requirement was accomplished - the Applet successfully displayed this information upon receipt of a valid ticket number through a web browser, but this was through a .html file local to the machine, not via a URL viewable from outside the machine. Given more time, it would be a very simple task to upgrade the local .html file onto a webpage to be accessed from outside the University.

7 Performance and Reliability Testing

In order to determine that both the software and the hardware functioned correctly, we had to perform rigorous testing, in both of these areas. It was also necessary to determine whether or not our system would perform within the limits and timescales that we had specified.

7.1 Hardware Testing

The Lego hardware was tested in two areas. Firstly, we needed to test that the Lego structure was robust enough to work for a reasonable amount of time. Once we had determined that the structure could hold together, we tested the system's tolerances. This meant that we had to test whether the structure would work correctly to the specification we had set.

7.1.1 Structural Reliability Testing

- Description of Test

  We decided that the robot and barrier should work, under continuous operation, for a period of two hours. This was deemed reasonable, as Lego naturally loosens itself after a period of time, and pieces under stress begin to come apart. In order to test that our system would hold together under maximum load, we wrote a special test program, and sent it to the robot. This program sent the robot backwards and forwards along the track, and at each end, the robot would stop. It would then raise its lifter to maximum height, extend its arm, and complete by lowering the lifter, and retracting the arm. The robot would then proceed to the other end of the track, and the procedure would be repeated. During the whole of the procedure, two toy cars were attached to the end of the arm, to simulate the system operating at maximum load.
  A similar program was sent to the barrier, which simply continuously raised the barrier, and then lowered it. This simulates the barrier operating at the maximum rate, where cars are continuously entering the car park.
  Both programs were started at the same time, and the time taken before a structural failure was measured.

- Test Results

  The results for the testing were as follows:

Run	$Timebeforefailure$	$Itemthatfailed$
1	10 Minutes	Robot drive gear.
2	12 Minutes	Robot drive gear.
3	36 Minutes	Robot wheel fell off.
4	22 Minutes	Robot lifter gearing.
5	1 Hour 20 Minutes	Drive motor came loose.
6	+ 2 Hours 15 Minutes	-

- Evaluation of Test Results

  This test gave us a very good indication of the parts of the robot which were failing. For example, the main problem was with the drive system. We found that after a time, the gears on the drive axles would slide along, and this would cause the drive to fail. The same problem was exhibited by the wheels. These gradually came loose from the axles, and eventually fell off completely. These problems were resolved by placing a small amount of glue, from a hot glue gun, onto the axles, to hold the gears or wheels in place. This solution worked extremely well and solved all of the problems with the axles. In the fourth run, the lifter gearing failed, as the supporting pieces worked loose. This problem was solved with a slight redesign of the supports, and has not reoccurred. During the fifth run, one of the drive motors became loose. This was again because of the Lego supports becoming loose, and the problem was solved by adding more structural supports to the motor. Finally, in the sixth run, we achieved our goal, to have the system running for more than two hours. The test had to be aborted due to a lack of time, but the robot remained structurally sound at the end of the test.

  As can be seen from the results, there were no problems with the barrier. This was to be expected, however, as the barrier is a relatively simple component with little to go wrong.

7.1.2 Tolerance Testing

- Description of Test

  We wanted to ensure that our hardware worked to the specifications which we had set, in terms of the allowed tolerances for position control, and for the timescale of the parking operation. This test involved sending the robot to each bay in turn, to park and retrieve cars. We used a pen to mark the stopping position of the RCX, at each bay, and also marked the height which the lifter raised, and lowered to during the operation. We then measured the distance from the intended stopping position to the actual position. The operation was repeated three times for each bay, so that we could measure the variation of the position measurements. We also measured the time taken to travel to each bay, and the time for a complete park and retrieve.

- Results of Test

Run	*Measurement*	*Max*	*Min*	*Avg*
1	Park Time	26s	14s	20s
	Travelling Time	2s	7s	4.5s
	Stopping Position Tolerance	3mm	1mm	2mm
	Height Position Tolerance	2mm	1mm	1.5mm
1	Park Time	27s	12s	19s
	Travelling Time	2s	6s	4s
	Stopping Position Tolerance	3mm	1mm	2mm
	Height Position Tolerance	2mm	1mm	1.5mm
1	Park Time	1	1	1
	Travelling Time	1	1	1
	Stopping Position Tolerance	3.5mm	1mm	2mm
	Height Position Tolerance	2mm	1mm	1.5mm

There were also two cases where the light sensor incorrectly detected that it was at the required bay. This was attributed to changing ambient light values.

There was one case where the height sensor incorrectly detected that it had reached the required position. This was also attributed to changing light levels in the room.

- Evaluation of Results

  These results show that in all cases, the stopping positions were within the tolerances which we had specified (e.g. 30s for a park, 10s to travel to bay, 5mm for position sensing). This indicates that our system was working within specification. However, there were some cases where the sensors incorrectly detected the position. This was due to the changing light values in the room, and possibly also due to the battery voltage of the RCX dropping. In order to minimise these cases, it was necessary to build 'shields' around the sensors. The height sensor was shielded by a piece of cardboard, which blocked as much external light as possible from the coloured tape. The horizontal position sensor was shielded by two angled Lego plates. It is also necessary to change the RCX batteries regularly, and to re-calibrate the system for new light values regularly. We found that it was best to do these things at the same time. Once this is done, the system becomes far more reliable.

7.2 Software Testing

The software was tested thoroughly in order to ensure that it worked as required, and that the communications software worked reliably.

7.2.1 Software Correctness Testing

- Description of Test

  The software was tested at all stages of development, by producing test programs for each class, which would test them in turn. We attempted to

test all parts of the software, including boundary conditions. We also completed testing on the final product. The test schedule for this is detailed below:

- Add car to car park.
- Remove the car from car park via Customer UI
- Try ticket number in Customer UI to ensure it is now invalid.
- Try some invalid ticket numbers.
- Add car to car park.
- Remove the car from car park using Controller.
- Try ticket number to ensure no longer valid in Customer UI.
- Add several cars to car park.
- Check time and costs for these cars.
- Remove the cars - some using Controller, some Customer UI, check queuing.
- Ensure cars are being removed when there are no cars waiting at barrier.
- Ensure that cars are both added and removed when cars are waiting for both.
- Fill the car park. Check LED display as car park fills, and ensure it shows full, when car park full.
- Ensure the barrier will not raise when car park full.
- Ensure we can remove cars, when car park full, and LED display updated.
- Use Applet to check car park statistics - check spaces remaining.
- Use Applet to check valid and invalid ticket numbers.

• Results of Test and Evaluation

The software test schedule enabled us to detect errors in our code, and to fix them. In most cases, the software worked correctly, except that several communications errors occurred, where messages were missed, or arrived incorrectly. We did find a bug, however, which prevented cars from being removed from the car park when it was filled. This bug was fixed, and the software re-tested.

7.2.2 Communications Reliability Testing

• Description of Test

This test simply involved running the system for as long as possible, to determine how often the messaging errors discovered in the previous test occurred. We ran the system for as long as possible, and timed how long it worked for before a communications error occurred.

- Results of Test

$RunNumber$	$Timebeforefailure$	$No.Parks$	$Fault$
1	5 Minutes	6	Message Missed
2	6 Minutes	5	Message Missed
3	12 Minutes	10	Message Incorrect
4	11 Minutes	10	Message Missed
5	20 Minutes	14	Message Incorrect
6	25 Minutes	20	Message Missed

- Evaluation of Results

  The results showed us that our system had problems with communications. This problem occurred when infra red messages were sent between either the barrier or the robot, and the PC. We believe that the problem occurs as the tower must send a message approximately once every second, to keep the communications open, whilst it is waiting for a message. If this message 'collides' with the message from the robot or barrier, then the RCX's message is either missed completely, or is corrupted. In order to solve this problem, we would have to rewrite the communications software completely. This was out of our timescale, so was not an option. If we had had extra time, however, this would have been done, and we would have incorporated some error checking into the messaging. We did manage to improve the situation slightly, as the results in the last part of the test show. This was done by increasing the number of times the RCX sends a message to the server. The messaging reliability is also drastically improved by changing the infra red tower batteries regularly.

7.3 Conclusions of Testing

Our tests enabled us to find and solve problems with our system. We uncovered several small bugs in the code, and were able to ensure that the system works correctly to specification. The only problems which we did not manage to solve were those involving the light sensors (re-calibration problems) and those involving the infra red communications software.

8 User Guide

8.1 Installation

To install EsePark 2002 for Windows, double click on the self-extracting es-epark.exe file, and enter a directory to install EsePark to. Please note that you must have the Java SDK or Java Runtime Environment v1.3 or greater, with the Java Communications API installed. These can be downloaded from http://java.sun.com. You are also required to have NQC, and LejOS installed on your machine and your path Environment variable set up to point to nqc.exe, lejosc.exe and lejos.exe if you want to be able to set up the robots. NQC can be downloaded from http://www.enteract.com/ dbaum/nqc/, and LejOS from http://lejos.sourceforge.net.

8.2 Startup

Once EsePark 2002 has been installed, you can bring up the startup menu by clicking on the menu.bat file in the directory you installed EsePark into. This should bring up a blue menu window, as shown in Figure 24:

Figure 24: Startup menu

Please note that EsePark 2002 requires two PC's. One will act as the main server and controller, and the Lego IR tower should be connected to the COM1 port of this machine. The other machine should have the LED display connected to it, and will act as the Customer User Interface.

8.2.1 Robot RCX Setup

Firstly, you will need to set up the Robot RCX. In order to do this, from the startup menu on the Main Server, press alt-1 to enter the RCX setup menu. This menu is password protected, to prevent unauthorised access, and you should

enter the password 'secret'. You will now see the 'RCX Setup' menu, as shown in Figure 25:

Figure 25: RCX Setup menu

Next select 'Robot Setup' from the menu, using the cursor keys. Firstly you will want to upload the required firmware to the robot RCX. Use the power button on the RCX to switch it on, and ensure that the Barrier Control RCX is switched off. Now select 'Upload Firmware' from the menu, and the firmware will be sent to the RCX.

Once the firmware has been uploaded to the RCX, you must set up the light sensor cutoff values, so that the position sensing mechanism will work correctly. To do this, select 'Setup Light Values' from the menu, and enter the values displayed on the RCX display when completing the following procedure: To find the values for the height sensor, move the sensor to the required colour of tape by pressing the 'View' button on the RCX until output A is selected. Now by holding 'View' and pressing 'Prg' or 'Run' you can move the lifter up or down. Once you are on the required colour of tape, press 'View' to select Sensor 1, and enter the value displayed on the screen into the program. To find the values for the horizontal position sensor, push the robot manually until it is on the required colour of tape, and then use the 'View' button to select Sensor 2. Now you can read the value of the display.

Finally you need to upload the program to the RCX by selecting 'Upload Program' from the menu. Please note that these light values are only sent to the RCX if you upload the program. Otherwise, the internal values of the RCX are not updated, and the RCX will not recognise the change in light values.

8.2.2 Applet Setup

To set up the Applet so that users can connect to your Car Park system remotely and view how long they have been parked, and what their cost of parking so far is, you need to have a web server installed on your Main Server ma-

chine. A freeware web server such as BRS WebWeaver can be downloaded from http://www.zdnet.com/. Once the server is installed, simply double click on applet.exe, and select the root directory of your web server as the destination for the files. This will make the applet and its main page accessible from your web server, by typing in http://hostname/Applet.html, where 'hostname' is the hostname or IP address of your Main Server machine. Once your main server is started, customers will be able to look at this page to get live statistics about the car park.

8.2.3 Barrier Control RCX Setup

To set up the Barrier RCX, select 'Barrier Setup' from the RCX Setup Menu. Firstly, you need to upload the firmware by selecting 'Upload Firmware'. When doing this, please ensure that the Barrier RCX is switched on, and the Robot RCX is switched off.
Once the firmware has successfully been uploaded, select 'Upload Program' to upload the Barrier Control program to the RCX.

8.2.4 RCX Program Startup

The final stage in setting up the RCXs is to start the programs. To do this, simply switch both RCXs on, and press the 'RUN' button. On the Robot RCX, you must ensure that program 1 is selected, by pressing the 'PGM' button repeatedly.

8.2.5 RMI Registry Startup

Both PCs require a Java RMI Registry to be running in order to function correctly. In order to start an RMI Registry, select 'Start RMI Registry' in the Main Startup Menu on each machine.

8.2.6 LED Display Server Startup

To enable the LED display to function correctly, you must start an LED Display Server on the Customer User Interface machine. Select 'Start LED Display Server' in order to do this. You will need to enter the COM port (e.g. COM2) that the LED display is connected to. Once you have done this, you should see the LED display flash a message, and then clear.

8.2.7 Main Server and Controller Startup

Once the LED Display Server is started, start the Main Server on the other (Main Server) machine. In order to do this, select 'Main Server Startup' from the main menu. You will need to enter the IP address or hostname of the LED Server machine. Once you have done this, the Controller user interface will start. Finally, to start up the Robot Controller, and open the car park, select 'Controller Startup' from the main menu. Note that at this stage the Robot's

arm should be fully retracted, and the lifter should be at its lowest position, at the car pickup point. An empty tray should also be present on the arm.

8.2.8 Customer User Interface Startup

The final stage in the startup procedure is to start up the Customer UI on the same machine as the LED Display Server. This will enable customers to retrieve their cars by entering a ticket number. To start the server, select 'Customer UI Startup' from the main menu. You must then enter the IP address or hostname of the Main Server Machine. The Customer User Interface should start in a new window. Finally, you may hit F2 in both Startup menus, to close the menu down. It is no longer required.

8.3 Parking a car

To park a car, drive it up to the barrier. When the system is ready to receive a car, the barrier will raise. Drive the car onto the platform, note the ticket number on the Barrier Control RCX display, and press the green button to park the car. The car will be placed automatically in a bay. Notice that the corresponding bay will now be marked blue (rather than grey) in the Controller software, indicating that the bay is occupied.

8.4 Retrieving a car - Customers

Customers can retrieve a car by entering their ticket number using the keypad connected to the Customer User Interface machine, and pressing Enter. If the ticket number is valid, the total cost of parking will be displayed. The customer should then pay this cost, and their car will be added to a queue of cars waiting to be retrieved. Whilst the car is waiting, the bay will be displayed yellow in the Controller software, and will finally turn grey, once the car is returned, and the bay is unoccupied.

8.5 Using the Controller User Interface

The Controller User Interface provides a means for the system supervisor to monitor the state of the car park, and perform various control actions, as explained below.

8.5.1 Checking the status of the car park

The main screen of the Controller User Interface provides a graphical representation of the car park bays, as shown in Figure 26:

Here a grey box indicates that the bay is empty, and a yellow box indicates that a car is waiting to be retrieved. In these cases, no actions are available for the bay. If the bay shows blue, then the bay is occupied. In this case it is possible to click on the bay, and a control panel will pop up, providing various options, as shown in Figure **??**:

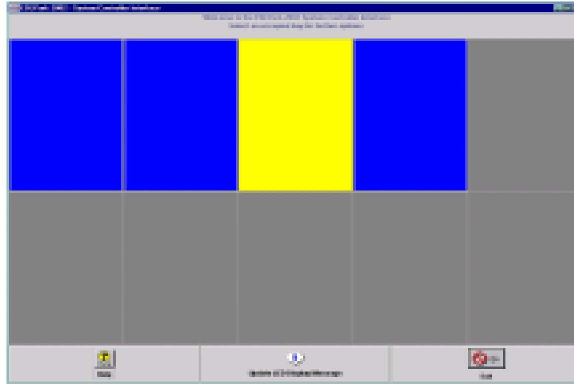

Figure 26: Controller User Interface

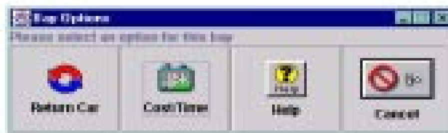

Figure 27: Bay Options control panel

8.5.2 Retrieving a car

To retrieve a car from a bay, simply click on the 'Return Car' button on the bay control panel. The bay should turn yellow, and the car will be retrieved.

8.5.3 Checking the Cost / Time spent in the bay

To see how long a car has been in the bay, and the total cost of parking so far, click on the 'Cost/Time' button on the bay control panel. This will bring up a panel, as shown in Figure 28, displaying the total time and cost of parking so far.

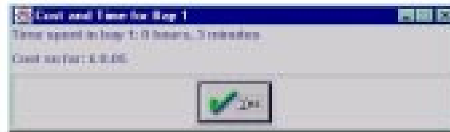

Figure 28: Cost/Time display

8.5.4 Closing the Bay Control Panel or getting help

To close the bay control panel or get help on the panel, simply click on the 'Cancel' or 'Help' buttons respectively.

8.5.5 Updating the LED Display Message

The LED Display can display custom messages, as well as the number of bays remaining. To add a custom message, click on the 'Update LED Display' button on the Controller User Interface. A box will pop up where you can enter a message, and press the 'Yes' button to update the display. To cancel the update, simply click on 'No'.

8.5.6 Exiting the System and Closing the Car Park

To exit the system, simply click on the 'Exit' button on the Controller User Interface. This will close the server down. You should then close any other server/diagnostic windows which are open, and close all windows on the Customer User Interface/LED Display Server machine.

8.6 Using the Applet

As long as you have your web server correctly installed and running, and the Applet files are in its root directory, the applet should work whenever the Car Park is open. To connect to the applet from a web browser, enter 'http://hostname/Applet.html' in the URL bar, and press enter. The applet should load (and the Java plugin will be downloaded if required). Once the applet has loaded, you can click on the 'About' tab to view information about EsePark, or the 'Statistics' tab to view live car park statistics, such as the number of bays remaining. An example is shown in Figure 29. To view information

about your car, simply enter a valid ticket number in the text box, and click on the 'Click Me' button. A page will appear giving information about how long you have been parked, and how much it has cost so far.

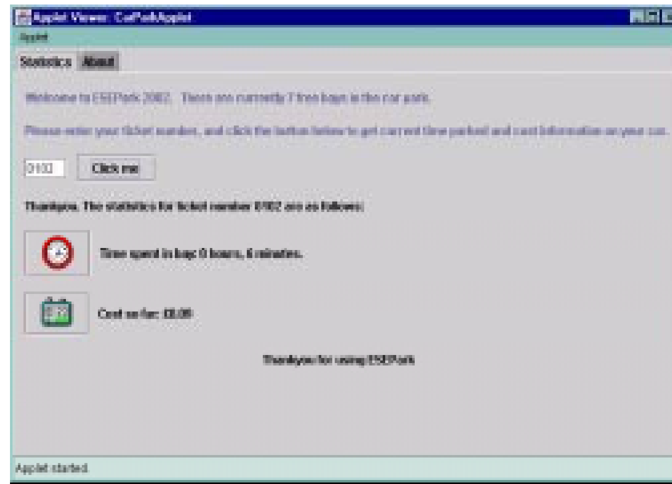

Figure 29: Applet Statistics Example

9 The Project as an Educational Experience

This section details what the group learned from the project, and what modules of the ESE3H curriculum aided us in achieving our goals for the project.

The project greatly consolidated much of the ESE3H curriculum knowledge, as well as courses from previous years at university. For example, the work with Lego's various sensors had given us the chance to practice theory learned in Control 3A and Control 3B modules. In particular, the control work involved with the precision stopping mechanism highlights this. This mechanism forms a closed loop system, as the robot is constantly switching between turning motors on and then checking whether the given light value was the desired value for which to stop the robot at. The barrier system's work with sensors also consolidate control knowledge, interpreting when a car has completely passed through the light sensors before lowering the barrier.

Furthermore, a fundamental aspect of the Real Time Computer Systems 3 module is proving how work over input and output can significantly degrade PC and processor performance, as opposed to speeds achieved within the PC such as register transfer and quick assembly language instruction execution. This is highlighted in our project work, as we needed 2 PC;s to handle the workload of all tasks; user interface and LED display input and output was not able to be completed in on 1 PC as it would operate so slowly that it resulted in unacceptable performance which could not meet specification.

The hardware design and construction of the LED display consolidated knowledge from all years in the electronics department, possibly even more difficult than previous Electronic Dice and Electronic Weighing Machine projects.

The Advanced Programming 3 module aided us in much of our software written in Java. On the other hand, the project also built on our Java knowledge in many areas which were not taught in Advanced Programming 3.

- Remote Method Invocation- for networking
- Java Swing - for User Interface design
- Applet programming
- Interfacing with Hardware and Software via serial port - for communications

The project also reflects material taught in the Networks Systems Architecture 3 module. The knowledge received on client/server applications was crucial to our understanding of the software architecture of the project. This is justified as there were 3 servers used in the project - one for each of Customer

User Interface, Controller User Interface and LED display controller. Remote Method Invocation was the perfect tool to implement a client/server as objects and methods from one machine can be invoked from another machine using this technique.

Moreover, the Professional Software Development 3 module prepared us very well for our overall third year project work.The coursework for this module was distributed over 2 medium size projects throughout the year which gave us excellent experience in requirements gathering, project management, teamwork, scheduling, software design, software implementation and testing; all essential parts of this project. The fact that the group for this project were together for the 2 projects in the Professional Software Development 3 coursework only improved team spirit and togetherness which are essential components for maximising communication at team meetings and overall group utilisation of skills.

Invaluable experience has been learned in hardware and software integration. Despite this being an academic aim of the Electronics and Software degree program, this has been the first real practical experience of it, which demonstrates various problems associated with it. Such problems included were testing strategies of the integrated hardware and software system, updating of software when hardware changes and vice-versa and how to incorporate intelligence in the system - i.e hardware letting software know that hardware is not functioning properly. Perhaps the project has actually helped discover how little hardware/software integration knowledge is covered before 4th year studies.

More generally, the project has given the group great pleasure in knowing that the delivered system would have been outwith our capabilities at the end of second year, only ten months ago. The project allowed us to incorporate various aspects of a demanding 3rd year curriculum into a final deliverable system. As a result, the project has stimulated n interest in robotics study for all members of the group, and we would maybe like to pursue this area as a career choice after our University studies.

10 Project Logs

10.1 Group Log

Month	*Activties*
December	Initial meeting with project supervisor Shown Legolab facilities Began background research Began to think about project ideas
January	Project plan written up First experiments with Lego structures Submitted project proposal and project plan to supervisors Construction begins
February	Progress report submitted to supervisors Construction continues Begin development of NQC and Java programs Individual introductions completed Design descriptions written up
March/April	Construction completed NQC and Java Comms continue to be developed Work begins on Interfaces and Applet Integration of system components Project demonstration to supervisors Preparation of report and presentation

10.2 Stuart Chalmers' Log

Month	*Activties*
December	Researched Lego Mindstorms and previous projects Learned about NQC through books in Legolab Researched Dev. tools such as LegOS, LejOS and NQC.
January	Suggested ideas such as LED display and web applet Finished off project proposal. Team built a basic Lego Pathfinder robot. Completed parts of Project plan: Configuration Management Researched building an LED display unit. Tried to work out how to build one using 7 seg. displays and decoder. Too difficult. Found out about Siemens SLR2016 - seemed easy to use. Ordered 2 Siemens SLR2016 displays. Acquired rest of parts from E&EE Built the base for the robot, using a track and slot mechanism. Added drive to base. Worked really well. Designed and built an LED display using Siemens SLR2016. Tested display using basic software. After some resoldering it worked! Wrote Java class for scrolling text on display. Works a treat.
February	Built a basic barrier system. Didn't work very well. It kept losing its position as rotation sensor inaccurate. Came in on a Saturday to rebuild lifter. Used worm gears. After several hours, and complete redesign, got it working ok. Worked out how to combine arm and lifter and put together. Redesigned barrier - worm gears. Much more accurate. Wrote barrier control program in LejOS. Wrote first draft of project introduction. Wrote RMI server software for LED Display. Redesigned arm to use worm gears. Now much smoother. Detailed hardware deign for barrier & Led Display and Base/drive.
March/April	Coded comms part of barrier software Spent all Easter building main server Had to redesign drive - not powerful enough. Robot is now 4x4! Improved Controller code to queue waiting cars. Improved car detection system for barrier. Performed many tests on all parts of system. Tried to improve reliability of messaging. Fixed some bugs in Controller. Didn't retrieve cars when car park full. Helped John with RMI code to update LED display message and change the colour of the bays in the GUI according to status. Produced first prototype of applet. Finished final draft of project intro. Demonstrated system to Supervisors. Much work on final report. Testing, User guide done by me.

10.3 Ross Macdonald's Log

Month	*Activties*
December	Familiarisation with Legolab equipment Read sections of following books; Nagata, Baum, Constructopedia Thought of topic for project - Automated Car park Read previous years reports
January	Helped finalise proposal document to be sent to Rod Helped build first Lego robot - Pathfinder - completed all training missions Drew a diagram of system Did Resources, budgets, schedules, organisation part of project plan document Researched books/Internet for lifting mechanisms E-mailed Dave Baum (NQC author) for suggestions. Replied investigate pulleys Designed and constructed pulley. Not strong enough. Designed climbing system
February	Constructed Climbing design. Much Stronger. Reinforcing work on climber Integration of robotic arm with lifter Lifter could not hold weight. Needed re-design Allocated Java - RCX Comms module. Identified classes needed for comms S/W Wrote first draft of project introduction Wrote lifter detailed hardware design
March/April	Came in for first 3 weeks of Easter holidays coded all comms software for PC - robot communication Finished final draft of project introduction Built the bays for the system Coded all comms software for PC - barrier communication Wrote CustomerUI, CustServer classes Wrote RMI classes to allow CustomerUI class to access Controller methods Took part in extensive overall integration and testing of complete system Wrote various sections of dissertation Helped prepare the group's demonstration and presentation

10.4 John Wallace's Log

Month	*Activties*
December	Familiarisation with Legolab equipment Read over previous ESE Project Reports Investigated Lego Mindstorms on the web Borrowed various book from the Legolab for background reading
January	Helped finalise proposal document to be sent to Rod Wrote Risk Management part of Project Plan Began construction of extending arm unit Created project website
February	Wrote first draft of project introduction Began coding NQC Control programs Developed height/position sensing systems Wrote hardware description for arm module Continued to update website
March/April	Finished coding NQC Researched Java Swing Implemented Controller UI and Applet using Swing Helped with construction of bays Helped with integration of hardware/software Demonstrated system to supervisors Prepared report and presentation

11 Abbreviations and Acronyms

- RIS - Robotic Invention System
- LCD - Liquid Crystal Display
- LED - Light Emitting Diode
- RCX - Robotic Command Explorer
- PC - Personal Computer
- NQC - Not Quite C
- DOS - Disk Operating System
- IR - Infra Red
- UI - User Interface
- GUI - Graphical User Interface
- JDK - Java Developer's Kit
- RMI - Remote Method Invocation
- LejOS - Lego Java Operating System
- API - Application Programming Interface
- AWT - Abstract Window Toolkit

12 Appendices

12.1 Appendix A - Source code

12.2 Appendix B - Project Plan

12.3 Appendix C - LED Display Datasheets