A Project Report on

“Implementation and Evaluation of Modbus Protocol”

Under the guidance of

Prof. Mrs. S. V. Kedar

By:
Deshmukh Nishikant Pramod (B2374205)
Rathi Anand Kamalkishor (B2374222)
Raskar Tushar Vijay (B2374221)
Raut Digvijay Dattatray (B2374224)

In Partial Fulfillment of
BE (Computer Engineering)
Degree Course of University of Pune

2004 – 2005

Department of Computer Engineering
ACKNOWLEDGEMENT

We express our deepest thanks to our project guide Prof S. V. Kedar for her Technical support, guidance, encouragement and moral support without which it would have been difficult to complete this project work.

We sincerely wish to express our grateful thanks to our Project Guide for his valuable suggestions.

It is our earnest endeavor to express sincere thanks to Prof Mr. S. T. Singh (HOD) for their kind co-operation, help and delighted support.

We are thankful to our entire staff of Computer Engineering Department specially for providing time to time help and their valuable guidance to make this work possible.
ABSTRACT

Industrial Automation has been an important aspect of Industrial Revolution. Microcontrollers were integrated with machines for controlling them. Soon the concept of controlling devices remotely came into picture. Thanks to network interfaces supported by microcontrollers, all these things were possible. With the advent of advanced computer technology it became desirable to connect those devices with PC. Different Vendors started implementing their own protocols. One of these protocols which have now become industrial software was developed by Modicon. Modbus is a Master Slave Protocol in which PC acts like a Master and other devices act as Slaves.

In our project, we have implemented this protocol in Java (Master) and embedded C (slave). On PC side we have designed a device manager “Modbus Browser” which detects serial ports on your machine and allows adding of devices with unique device address per port. Commands can be later added for each device. Communication is half duplex and when proper response is received from device it is stored in database.

As an extension, we have tried to connect two Browsers together so that devices can be controlled remotely from Browser to Browser.
Implementation and Evaluation of Modbus Protocol
# INDEX

<table>
<thead>
<tr>
<th>Chapter No.</th>
<th>TITLE</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td></td>
</tr>
<tr>
<td>1.1 System Description</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1.1.1 Modbus Browser</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1.1.2 Database</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>1.2 Further Extension</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>1.3 Software Requirement Specification</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1.3.1 Introduction</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1.3.1.1 Purpose</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1.3.1.2 Scope of Product</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1.3.2 General Description</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>1.3.2.1 Product Perspective</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>1.3.2.2 User Characteristics</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>1.3.3 General Constraints</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1.3.4 Specific Requirements</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>1.3.4.1 Functional Requirements</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>Literature</td>
<td></td>
</tr>
<tr>
<td>2.1 Protocol Description</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>2.2 Serial Transmission Format</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2.3 Modbus Message Framing</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>2.4 ASCII Transmission Mode</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>2.4.1 MODBUS Message ASCII Framing</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>2.5 RTU Transmission Mode</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>2.5.1 MODBUS Message RTU Framing</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>2.6 Error Checking Methods</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>2.6.1 CRC checking</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>2.6.2 LRC checking</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>2.6.3 Parity checking</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Design</td>
<td></td>
</tr>
<tr>
<td>3.1 Data Flow Diagrams</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3.1.1 Level 0 DFD</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3.1.2 Level 1 DFD</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>3.1.3 Level 2A DFD</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>3.1.4 Level 2B DFD</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>3.1.5 Level 2C DFD</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>3.1.6 Level 2D DFD</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3.1.7 Level 2E DFD</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>3.1.8 Level 2F DFD</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>3.1.9 Level 2G DFD</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>3.2 UML Diagrams</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>3.2.1 Use Case Diagram</td>
<td>41</td>
<td></td>
</tr>
</tbody>
</table>
4. Implementation

4.1 System Implementation

4.1.1 Basic .......................................................... 54
4.1.2 Mytree ...................................................... 54
4.1.3 Scheduler .................................................. 54
4.1.4 Server ...................................................... 54
4.1.5 Queue ..................................................... 55
4.1.6 Send ....................................................... 55
4.1.7 Databas ................................................... 55

4.2 Data Dictionary

4.2.1 File: basic.java ........................................... 56
4.2.2 device.java ............................................... 57
4.2.3 deviceData.java ........................................ 57
4.2.4 errorCode.java .......................................... 57
4.2.5 lineparser.java ........................................... 57
4.2.6 databas.java ............................................ 58
4.2.7 mytree.java ............................................. 58
4.2.8 SerialParameters.java ................................ 58

4.3 Algorithm

4.3.1 lineparser ............................................... 59
4.3.2 comportIdentification ................................ 60
4.3.3 removeDevice .......................................... 61
4.3.4 removeNetwork ........................................ 62
4.3.5 deleteCommand ....................................... 62
4.3.6 scheduler .............................................. 63
4.3.7 showIcons .............................................. 64
4.3.8 Master to Master Connection ..................... 65
4.3.9 Errorcode ............................................. 68

5. Testing

5.1 Test Plan ...................................................... 72
5.1.1 Unit-Testing ........................................... 72
5.1.1.1 Mytree ............................................. 73
5.1.1.2 Error Code ........................................ 74
5.1.1.3 lineParser......................................................... 75
5.1.1.4 fileUtil............................................................. 77
5.1.2 Integration Testing.................................................. 79
  5.1.2.1 Bottom-up Testing.............................................. 80
5.1.3 Validation Testing.................................................... 82
  5.1.3.1 Validation Testing for User Interface....................... 83
  5.1.3.2 Validation Testing for DMS............................... 83

6. Result
  6.1 Modbus Browser..................................................... 85
  6.2 Default Network..................................................... 86
  6.3 Adding of device command........................................ 88
  6.4 Communication with device....................................... 90
  6.5 Device Information................................................ 92
  6.6 Configure Device................................................... 93
  6.7 Remove Device..................................................... 97

7. Conclusion
  7.1 Conclusions......................................................... 100
  7.2 Applications of the system....................................... 100
  7.3 Limitations......................................................... 100
  7.4 Future Scopes......................................................... 100

8. Bibliography
  8.1 Books........................................................................ 102
  8.2 Websites.................................................................... 102
Chapter 1

Introduction
1. Introduction

1.1 System Description

1.1.1 Modbus Browser:

Modbus Browser is a device manager which helps in maintaining all device information and allows adding and deleting commands supported by a modbus device. Modbus protocol can be implemented over serial line or TCP/IP. Our project deals with implementing it over serial line. Modbus Browser is implemented in Java and has well defined Graphics User Interface in Java Swing.

![System Block Diagram](image)

**Fig 1.1: System Block Diagram**

Modbus Protocol is a Master Slave Protocol in which a PC acts as a Master and Device acts as a slave. In our case Modbus Browser acts as a Master and Devices act as a slave.

We have written a small slave side software in embedded C for Microcontroller to simulate actual interaction with hardware.
1.1.2 Database

Database being used is MS-ACCESS. Every time device is added and removed, changes will be made in database. All device info is stored in database. All command information will also be stored in database.

Two tables are required
1. Store Device Info
2. Store Command Info

Instead of accessing database each time when user expands a node in a tree or while displaying command information we have maintained a separate string which is stored in file and contains only name of serial ports, device and commands. Following is a sample string

```plaintext
network0{COM1(PLC[temp#press])/COM2()}
```

Above string depicts:-
Name of network : network0
Number of Serial Ports : 2
Name of Serial Ports: COM1 and COM2
Devices Under Ports:
   COM1 : PLC
   COM2 : -----
Commands supported by COM1:
   temp
   press
1.2 Further Extension:
Master-to-Master Connection

As a further Extension we have tried to connect two Masters remotely and make their devices visible to each other. Further they can start and stop their devices remotely.

![Subsystem Block Diagram](image.png)

**Fig 1.2: Subsystem Block Diagram**

The network string depicted in previous section is transferred to server and on request by another master this string will be transferred which will be added to configuration file with new network name.
1.3 Software Requirement Specification:

1.3.1 Introduction:

1.3.1.1 Purpose:

The purpose of requirement specification is to specify both functional and non-functional requirements to the user. This is an initial phase of development of project in which detailed requirements has been collected from company and provided in details all negotiated functions to be added in project.

1.3.1.2 Scope of Product:

The purpose of this software development project is to implement MODBUS protocol. MODBUS protocol is an Industrial standard protocol to connect remote Modbus devices to a PC. By doing so, we can control devices remotely by a PC. Following are the capabilities that will be provided by software.

1. Auto Detection of serial ports.
2. Facility to add devices on given comports.
3. Add commands for a specific device.
4. Start a device
5. Stop a device
6. Connect to other Modbus browsers.
7. Add other networks with your current network.
8. Start and Stop devices of other network.
9. Easy to use GUI.
10. Generic for serial devices (i.e. supports any device)
1.3.2 General Description:

1.3.2.1 Product Perspective:

The product will be used by Device administrators, analyzers (to study incoming data). It will be used to start devices and put them in polling modes and get data stored on hard disk.

![Modbus network model's perspective](image)

1.3.2.2 User Characteristics:

Types of user for the Library Book System are:

- Device Administrator
- Technical Expert

The following table describes general user characteristics that will affect the functionality of the software product.

<table>
<thead>
<tr>
<th>Type of User</th>
<th>User Characteristics</th>
<th>User Technical Expertise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Administrator</td>
<td>- Good Understanding of device operations.</td>
<td>- Good Computer basics.</td>
</tr>
<tr>
<td></td>
<td>- Responsible for device operations as a whole</td>
<td>- Knowledge of network.</td>
</tr>
<tr>
<td>Technical Expert</td>
<td>- Setting up of device</td>
<td>- Expert in networking</td>
</tr>
<tr>
<td></td>
<td>- Setting device commands and required database</td>
<td>- Database knowledge</td>
</tr>
</tbody>
</table>

Fig 1.3: Modbus network model’s perspective
1.3.3 General Constraints:

The current hardware requirement for the Modbus browser is Pentium III Processor or more with 128 MB RAM and at least one serial port.

Software requirements are Windows 98/2000/XP with JRE 1.4.2 or more and CommAPI libraries.

The limit of Modbus device connected to a PC can be 255.

Security is important. Only user on a PC can change device configuration. It cannot be changed remotely.

1.3.4 Specific Requirements:

This section contains the detailed requirements.

1.3.4.1 Functional Requirements:

User interface:

The user interface requirements are concerned with the user interface and how information is presented to the user.

Device Management System:

1. The System shall display device information including device name and address and serial parameter settings.

2. GUI will be used which will allow users to add and remove devices, adding and removing commands.

Network Management System:

1. System shall connect to the modbus server in order to get device lists.

2. User will be free to add other Modbus networks.

3. Network will be displayed in a tree structure.

4. User is free to start and stop devices of other networks.
Chapter 2

Literature
2. MODBUS PROTOCOL:[2]

“The MODBUS protocol defines a message structure that controllers will recognize and use, regardless of the type of network over which they communicate.”[2]

“It describes:[2]

• The process a controller uses to request access to other devices,
• How it will respond to requests from the other devices, and
• How errors will be detected and reported.”[2]

“MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model that provides client/server communication between devices connected on different types of buses or networks.”[2]

“MODBUS is a request/reply protocol and offers services specified by function codes. MODBUS function codes are elements of MODBUS request/reply PDUs.[2] The objective of this document is to describe the function codes used within the framework of MODBUS transactions.”[2]
2.1 Protocol description

“The MODBUS protocol defined a simple protocol data unit (PDU) independent of the underlying communication layers. The mapping of MODBUS protocol on specific buses or network can introduce some additional fields on the application data unit (ADU).”[2]

Fig 1.1: Modbus ADU

“The MODBUS application data unit is built by the client that initiates a MODBUS transaction.[2] The function indicates to the server what kind of action to perform.[2] The MODBUS application protocol establishes the format of a request initiated by a client. [2] The function code field of a MODBUS data unit is coded in one byte. Valid codes are in the range of 1 … 255 decimal (128 – 255 reserved for exception responses). [2] When a message is sent from a Client to a Server device the function code field tells the server what kind of action to perform. Sub-function codes are added to some function codes to define multiple actions.[2] The data field of messages sent from a client to server devices contains additional information that the server uses to take the action defined by the function code. [2] This can include items like discrete and register addresses, the quantity of items to be handled, and the count of actual data bytes in the field.[2] The data field may be nonexistent (of zero length) in certain kind of request, in this case the server does not require any additional information.[2] The function code alone specifies the action [2]. If no error occurs related to the MODBUS function requested in a properly received MODBUS ADU the data field of a response from a server to a client contains the data requested[2]. If an error related to the MODBUS function requested occurs, the field contains an exception code that the server application can use to determine the next action to be taken[2]. When the server responds to the client, it uses the function code field to indicate either
a normal (error-free) response or that some kind of error occurred (called an exception response)[2]. For a normal response, the server simply echoes the original function code.”[2]

Fig 1.2: MODBUS transaction (error free)

“For an exception response, the server returns a code that is equivalent to the original function code with its most significant bit set to logic 1.”[2]

Fig 1.3: MODBUS transaction (exception response)

“The MODBUS request consists of:[2]

• An address,
• A function code defining the requested action,
• Data (if necessary for the requested function), and
• Error check for testing the integrity of the message.”

“The slave’s response contains:
• The slave address,
• Data conform the request type, and
• Error check.

If the data integrity test fails, no response is sent back. If a request cannot be processed an exception message is returned.”

“The size of the MODBUS PDU is limited by the size constraint inherited from the first MODBUS implementation on Serial Line network (max. RS485 ADU = 256 bytes).”

“Therefore, **MODBUS PDU for serial line communication** = 256 - Server address (1 byte) - CRC (2 bytes) = **253 bytes.**

“Consequently:

RS232 / RS485 **ADU** = 253 bytes + Server address (1 byte) + CRC (2 bytes) = **256 bytes.**

TCP MODBUS **ADU** = 249 bytes + MBAP (7 bytes) = **256 bytes.**

The MODBUS protocol defines three PDUs.

They are:

• MODBUS Request PDU, mb_req_pdu
• MODBUS Response PDU, mb_rsp_pdu
• MODBUS Exception Response PDU, mb_excep_rsp_pdu”
2.2 Serial Transmission Format [2]

“The two transmission modes used are called: [2]
1. ASCII, and
2. RTU.

The user has to select the desired mode along with the serial communication parameters (baud rate, parity type). [2]
Note that all these parameters must be the same for all controllers in the network.” [2]

2.2.1 ASCII-mode [2]

• Each byte of the message is sent as two ASCII characters. [2]
  This means only the ASCII characters 0-9, A-F are transmitted. [2]

• Serial communication parameters: [2]
  1 Start bit, 7 data bits, even/odd/no parity, 1 stop bit if parity is used and two stop bits if no parity is used.

• Error check field:[2]
  Longitudinal Redundancy Check (LRC).
The advantage of ASCII mode is that it allows for a time interval up to 1 second between characters without causing a timeout. [2]
A disadvantage of ASCII mode is the larger message length.” [2]

2.2.2 RTU-mode [2]

• Each byte of the message is sent as 8 bits. [2]
• Serial communication parameters: [2]
  1 start bit, 8 data bits, even/odd/no parity, 1 stop bit if parity is used, and two stop bits if no parity is used. [2]
• Error check field: [2]
  Cyclic Redundancy Check (CRC).”
2.3 Modbus Message Framing [2]

2.3.1 ASCII-MODE [2]

“In ASCII-mode a message starts with a colon character (:) and ends with a carriage return–linefeed. Intervals up to one second can elapse between characters within the message. [2] If the interval is longer, a timeout error occurs and the message is rejected.” [2]

2.3.2 RTU mode [2]

“In RTU-mode a message starts with a silent interval of at least 3.5 character times. [2] The entire message frame must be transmitted as a continuous stream. [2] If a silent interval of more than 3.5 character times occurs before completion of the frame, the receiving device flushes the incoming message and assumes that the next byte will be the address field for the new message.” [2]

<table>
<thead>
<tr>
<th></th>
<th>START</th>
<th>ADDRESS</th>
<th>FUNCTION</th>
<th>DATA</th>
<th>DATA CHECK</th>
<th>END</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII</td>
<td>:</td>
<td>2 chars</td>
<td>2 chars</td>
<td>N*2</td>
<td>LRC</td>
<td>CR-LF</td>
</tr>
<tr>
<td>Mode</td>
<td></td>
<td></td>
<td></td>
<td>chars</td>
<td>2 chars</td>
<td></td>
</tr>
<tr>
<td>RTU</td>
<td>3.5</td>
<td>8 bits</td>
<td>8 bits</td>
<td>N*8</td>
<td>CRC</td>
<td>3.5 character</td>
</tr>
<tr>
<td>Mode</td>
<td>characters silent interval</td>
<td></td>
<td></td>
<td>bits</td>
<td>16 bits</td>
<td>silent interval</td>
</tr>
</tbody>
</table>

“Messages start with a silent interval of at least 3.5 character times. [2] This is easily implemented as a multiple of character times at the baud rate used on the network (shown as T1-T2-T3-T4 in the table below). [2] The first field then transmitted is the device address. [2] The allowed characters transmitted for all fields are hexadecimal 0-9, A-F. Network devices monitor the network bus continuously, including during the ‘silent’ intervals. [2] When the first field (the address field) is received, each device decodes it to find out if it is the addressed device. [2] Following the last transmitted character, a similar interval of at least 3.5 character times marks the end of the message. [2] A new message can begin after this interval. [2] The entire message frame must be transmitted as a continuous stream. [2] If a silent interval of more than 3.5 character times occurs before completion of the frame, the receiving device flushes the incomplete message and assumes that the next byte will be the address field of a new message. [2] Similarly, if a new message begins earlier than 3.5 character times following a previous
message, the receiving device will consider it a continuation of the previous message. [2] This will set an error, as the value in the final CRC field will not be valid for the combined messages. [2] A typical message frame is shown below.”[2]

<table>
<thead>
<tr>
<th>Header</th>
<th>START</th>
<th>T1-T2-T3-T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS</td>
<td>8 bits</td>
<td></td>
</tr>
<tr>
<td>FUNCTION</td>
<td>8 bits</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>DATA</td>
<td>n x 8 bits</td>
</tr>
<tr>
<td>Trailer</td>
<td>CRC CHECK</td>
<td>16 bits</td>
</tr>
<tr>
<td>END</td>
<td>T1-T2-T3-T4</td>
<td></td>
</tr>
</tbody>
</table>

**The Address Field [2]**

“The address field of a message frame contains:

- 2 characters (ASCII-mode) or
- 8 bits (RTU-mode).

Valid slave addresses are 1 to 247.
Address 0 is used for a broadcast to address all slaves.”

**The Function Field [2]**

“The function field of a message frame contains:

- 2 characters (ASCII-mode) or
- 8 bits (RTU-mode).

Valid codes lie in a range of 1 to 127. [2]

The function code tells the slave which kind of action to perform. A slave response always contains the function code of the request. If a function is not applicable, the slave sends an exception response. An exception is indicated by a returned function code with bit 8 (most significant byte) set.”
The Data Field [2]

“The data field contains 8 bit values in the range of 0 to FF hexadecimal.[2] In ASCII mode this byte is made of 2 ASCII characters.[2] The data field of messages contains information which both master and slave use to perform an action.[2] This includes the register address, quantity of registers, and the necessary data.” [2]

The Error Checking Field [2]

“The error checking field contents depend on the transmission mode.[2] Two kinds of error methods are used.” [2]

Error check with ASCII-mode [2]

“When the ASCII mode is used, the error-checking field contains two ASCII characters. [2] The error check characters are the result of a Longitudinal Redundancy Check calculation. [2] This is performed on the message contents with exception of the beginning colon, the carriage return and line feed characters. [2] The LRC characters are appended to the message as the last field preceding the CR-LF characters.” [2]

Error check with RTU-mode [2]

“When RTU mode is used, the error-checking field contains a 16-bit value implemented as two bytes.[2] The error check value is the result of a Cyclic Redundancy Check calculation performed on the message contents.[2] The CRC field is appended to the message as the last field.” [2]

Errors, exception codes [2]

“Two kinds of errors are possible: [2]
- Transmission errors. [2]
- Operation errors.” [2]
2.4 ASCII Transmission Mode [2]

“When devices are setup to communicate on a MODBUS serial line using ASCII (American Standard Code for Information Interchange) mode, each 8–bit byte in a message is sent as two ASCII characters. [2] This mode is used when the physical communication link or the capabilities of the device does not allow the conformance with RTU mode requirements regarding timers management.” [2]

“Remark: this mode is less efficient than RTU since each byte needs two characters.[2]

Example: The byte 0X5B is encoded as two characters: 0x35 and 0x42 (0x35 ="5", and 0x42 ="B" in ASCII).” [2]

The format for each byte (10 bits) in ASCII mode is:[2]

Coding System: “Hexadecimal, ASCII characters 0–9, A–F [2]

One hexadecimal character contains 4-bits of data within each ASCII character of the message [2]”

Bits per Byte: “1 start bit
7 data bits, least significant bit sent first
1 bit for parity completion;
1 stop bit”

Even parity is required;[2] “other modes (odd parity, no parity) may also be used. In order to ensure a maximum compatibility with other products, it is recommended to support also No parity mode. The default parity mode must be even parity. Remark: the use of no parity requires 2 stop bits.”

How Characters are transmitted serially:[2]

“Each character or byte is sent in this order (left to right):
Least Significant Bit (LSB) . . . Most Significant Bit (MSB)”
**Bit Sequence in ASCII mode** [2]

“Devices may accept by configuration either Even, Odd, or No Parity checking. If No Parity is implemented, an additional stop bit is transmitted to fill out the character frame [2].”

![Without Parity Checking Diagram](image)

**Bit Sequence in ASCII mode (specific case of No Parity)** [2]

**Frame Checking Field**: “Longitudinal Redundancy Checking (LRC)” [2]

**2.4.1 MODBUS Message ASCII Framing** [2]

“A MODBUS message is placed by the transmitting device into a frame that has a known beginning and ending point. [2] This allows devices that receive a new frame to begin at the start of the message, and to know when the message is completed. [2] Partial messages must be detected and errors must be set as a result. [2] The address field of a message frame contains two characters.” [2]

“In ASCII mode, a message is delimited by specific characters as Start-of-frames and End-of-frames. A message must start with a ‘colon’ ( : ) character (ASCII 3A hex), and end with a ‘carriage return – line feed’ (CRLF) pair (ASCII 0D and 0A hex).” [2]

**Remark**: The LF character can be changed using a specific MODBUS application command (see MODBUS application protocol specification). The allowable characters transmitted for all other fields are hexadecimal 0–9, A–F (ASCII coded). The devices monitor the bus continuously for the ‘colon’ character. When this character is received, each device decodes the next character until it detects the End-Of-Frame. Intervals of up to one second may elapse between characters within the message. Unless the user has configured a longer timeout, an interval greater than 1 second means an error has occurred.
Some Wide-Area-Network application may require a timeout in the 4 to 5 second range. A typical message frame is shown below.” [2]

<table>
<thead>
<tr>
<th>Start</th>
<th>Address</th>
<th>Function</th>
<th>Data</th>
<th>LRC</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 char</td>
<td>2 chars</td>
<td>2 chars</td>
<td>0 up to 2x252 char(s)</td>
<td>2 chars</td>
<td>2 chars CRLF</td>
</tr>
</tbody>
</table>

**ASCII Message Frame [2]**

“Remark: Each data byte needs two characters for encoding. [2] Thus, to ensure compatibility at MODBUS application level between ASCII mode and RTU mode, the maximum data size for ASCII data field (2x252) is the double the maximum data size for RTU data field (252). [2] Consequently, the maximum size of a MODBUS ASCII frame is 513 characters”. [2]

**2.5 RTU Transmission Mode [2]**

“When devices communicate on a MODBUS serial line using the RTU (Remote Terminal Unit) mode, each 8–bit byte in a message contains two 4–bit hexadecimal characters. [2] The main advantage of this mode is that its greater character density allows better data Throughput than ASCII mode for the same baud rate.[2] Each message must be transmitted in a continuous stream of characters.”[2]

**The format for each byte (11 bits) in RTU mode is: [2]**

**Coding System:** 8–bit binary [2]

**Bits per Byte:** 1 start bit [2]

8 data bits, least significant bit sent first [2]

1 bit for parity completion

1 stop bit

**Even parity is required;** other modes (odd parity, no parity) may also be used. [2] In order to ensure a maximum compatibility with other products, it is recommended to support also No parity mode. [2] The default parity mode must be even parity. [2]

Remark: the use of no parity requires 2 stop bits. [2]”
**How Characters are transmitted serially:** [2]

“Each character or byte is sent in this order (left to right): [2] Least Significant Bit (LSB) . . . Most Significant Bit (MSB) [2]”

![Diagram of character transmission](image1)

**Bit Sequence in RTU Mode** [2]

“Devices may accept by configuration either Even, Odd, or No Parity checking. [2] If No Parity is implemented, an additional stop bit is transmitted to fill out the character frame to a full 11-bit asynchronous character. [2]”

![Diagram of RTU mode bit sequence](image2)

**Bit Sequence in RTU Mode** *(specific case of No Parity)* [2]

**Frame Checking Field:** Cyclical Redundancy Checking (CRC) [2]

![Diagram of CRC](image3)

**RTU Message Frame** [2]

“The maximum size of a MODBUS RTU frame is 256 bytes. [2]”
2.5.1 MODBUS Message RTU Framing [2]

“A MODBUS message is placed by the transmitting device into a frame that has a known beginning and ending point. [2] This allows devices that receive a new frame to begin at the start of the message, and to know when the message is completed. [2] Partial messages must be detected and errors must be set as a result. [2] In RTU mode, message frames are separated by a silent interval of at least 3.5 character times. [2]”

![RTU Message Frame Diagram]

**RTU Message Frame [2]**

“The entire message frame must be transmitted as a continuous stream of characters. [2] If a silent interval of more than 1.5 character times occurs between two characters, the message frame is declared incomplete and should be discarded by the receiver. [2]”

“Remark: The implementation of RTU reception driver may imply the management of a lot of interruptions due to the t1.5 and t3.5 timers. [2] With high communication baud rates, this leads to a heavy CPU load. [2] Consequently these two timers must be strictly respected when the baud rate is equal or lower than 19200 Bps. [2] For baud rates greater than 19200 Bps, fixed values for the 2 timers should be used: it is recommended to use a value of 750µs for the inter-character time-out (t1.5) and a value of 1.750ms for inter-frame delay (t3.5). [2]”
2.6 Error Checking Methods [2]

2.6.1 CRC Checking [2]

“The RTU mode includes an error–checking field that is based on a Cyclical Redundancy Checking (CRC) method performed on the message contents.[2] The CRC field checks the contents of the entire message. [2] It is applied regardless of any parity checking method used for the individual characters of the message. [2] The CRC field contains a 16–bit value implemented as two 8–bit bytes. [2] The CRC field is appended to the message as the last field in the message.[2] When this is done, the low–order byte of the field is appended first, followed by the high–order byte.[2] The CRC high–order byte is the last byte to be sent in the message.[2] The CRC value is calculated by the sending device, which appends the CRC to the message. [2] The receiving device recalculates a CRC during receipt of the message, and compares the calculated value to the actual value it received in the CRC field. [2] If the two values are not equal, an error results. [2] The CRC calculation is started by first pre-loading a 16–bit register to all 1’s. [2] Then a process begins of applying successive 8–bit bytes of the message to the current contents of the register. [2] Only the eight bits of data in each character are used for generating the CRC. [2] Start and stop bits and the parity bit, do not apply to the CRC. [2] During generation of the CRC, each 8–bit character is exclusive ORed with the register contents. [2] Then the result is shifted in the direction of the least significant bit (LSB), with a zero filled into the most significant bit (MSB) position. [2] The LSB is extracted and examined. [2] If the LSB was a 1, the register is then exclusive ORed with a preset, fixed value. [2] If the LSB was a 0, no exclusive OR takes place. [2] This process is repeated until eight shifts have been performed. [2] After the last (eight) shift, the next 8–bit byte is exclusive ORed with the register’s current value, and the process repeats for eight more shifts as described above. [2] The final content of the register, after all the bytes of the message have been applied, is the CRC value. [2] When the CRC is appended to the message, the low–order byte is appended first, followed by the high–order byte. [2]”

2.6.2 LRC Checking

“In ASCII mode, messages include an error–checking field that is based on a Longitudinal Redundancy Checking (LRC) calculation that is performed on the message contents, exclusive of the beginning ‘colon’ and terminating CRLF pair characters. [2] It is
applied regardless of any parity checking method used for the individual characters of the message. [2] The LRC field is one byte, containing an 8–bit binary value. [2] The LRC value is calculated by the device that emits, which appends the LRC to the message. [2] The device that receives calculates an LRC during receipt of the message, and compares the calculated value to the actual value it received in the LRC field. [2] If the two values are not equal, an error results. [2] The LRC is calculated by adding together successive 8–bit bytes of the message, discarding any carries, and then two’s complementing the result. [2] It is performed on the ASCII message field contents excluding the ‘colon’ character that begins the message, and excluding the CRLF pair at the end of the message. [2] In ASCII mode, the resulting LRC is ASCII encoded into two bytes and placed at the end of ASCII mode frame prior to the CRLF. [2]"

**2.6.3 Parity Checking [2]**

“Users may configure devices for Even (required) or Odd Parity checking, or for No Parity checking (recommended). This will determine how the parity bit will be set in each character. If either Even or Odd Parity is specified, the quantity of 1 bit will be counted in the data portion of each character (seven data bits for ASCII mode, or eight for RTU). The parity bit will then be set to a 0 or 1 to result in an Even or Odd total of 1 bit. [2]"

“For example, these eight data bits are contained in an RTU character frame: 1100 0101 The total quantity of 1 bit in the frame is four. If Even Parity is used, the frame’s parity bit will be a 0, making the total quantity of 1 bits still an even number (four). If Odd Parity is used, the parity bit will be a 1, making an odd quantity (five). When the message is transmitted, the parity bit is calculated and applied to the frame of each character. The device that receives counts the quantity of 1 bit and sets an error if they are not the same as configured for that device (all devices on the MODBUS Serial Line must be configured to use the same parity checking method). [2]”

“Note that parity checking can only detect an error if an odd number of bits are picked up or dropped in a character frame during transmission. [2]”For example, if Odd Parity checking is employed, and two 1 bits are dropped from a character containing three 1 bits, the result is still an odd count of 1 bits. [2]”If No Parity checking is specified, no parity bit is transmitted and no parity checking can be made. [2]”An additional stop bit is transmitted to fill out the character frame. [2]”
Chapter 3

Design
3. Design

3.1 Data Flow Diagrams

3.1.1 Level 0 DFD

Overall System
3.1.2 LEVEL 1 DFD

User

Command interpreter

Device manager

Network manager

Scheduler

Storage manager

Queue manager

Device management

Modbus Browser
3.1.3 LEVEL 2A DFD

User command event

Switch

Call appropriate function

Pass action event

Various commands

Command Interpretation
### 3.1.4 LEVEL 2B DFD

**Scheduler**
3.1.5 LEVEL 2C DFD

Command Manager
3.1.6 LEVEL 2D DFD

Device Manager
3.1.7 LEVEL 2E DFD

Network Manager
3.1.8 LEVEL 2F DFD

Queue Manager

Request to
Start / Stop

Queue - handle

Command dispatcher

Start

Started

Stop

Stopped

Send command

Receive response

Save
to file

Save

Start

Request to

Start / Stop

Queue Manager
3.1.9 LEVEL 2 G DFD

Storage Manager
3.2 UML DIAGRAMS

3.2.1 Use Case Diagram

Basic User Interaction
3.2.2 CLASS DIAGRAM

Modbus Browser
3.2.3 SEQUENCE DIAGRAM
3.2.3.1 Sequence Diagram for Device

Device

User

Start

Detect comports

Initialised

Add device

Give device info

Device info

Device added

Process device data

Add commands

Give command info

Command info

Command added

Start device

Load in message queue

Send commands

Process commands

Stop device

Stop sending message to device

Browser

Device
3.2.3.2 Sequence Diagram for Network

Network

user config. Server Browser2

start browser

register on startup

register browser

register on startup

register browser

register browser

give network list

give config. file

send config file

store and add network

start remotely

start selection

start string

start specified device

start specified device

stop remotely

stop selection

stop string

stop specified device

stop selection
3.2.4 ACTIVITY DIAGRAM

Activity Diagram for Modbus Browser
3.2.5 STATE DIAGRAM

3.2.5.1 State Diagram for Master

Diagram showing the state transition of a master node in a network system, with states including Idle, Processing reply, Waiting for reply, and Processing error.
3.2.5.2 State Diagram for Slave
3.2.6 COMPONENT DIAGRAM

Modbus Browser
3.2.7 DEPLOYMENT DIAGRAM

![Deployment Diagram]

Modbus Browser

PLC1, PLC2, PLC3, Server, Modbus Browser
3.3 Database Design:

MSACCESS has been used for storing database information. Following is design view of the Database.

Table name: First

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>Number</td>
<td>Device address</td>
</tr>
<tr>
<td>device_name</td>
<td>Text</td>
<td>Name of device</td>
</tr>
<tr>
<td>id</td>
<td>Number</td>
<td>Internal device id</td>
</tr>
<tr>
<td>nid</td>
<td>Number</td>
<td>Internal network id</td>
</tr>
<tr>
<td>noc</td>
<td>Number</td>
<td>Number of commands</td>
</tr>
<tr>
<td>tid</td>
<td>Number</td>
<td>Transmission id</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. ASCII</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. RTU</td>
</tr>
<tr>
<td>portname</td>
<td>Text</td>
<td>Name of port</td>
</tr>
<tr>
<td>baudRate</td>
<td>Number</td>
<td>Baud rate of port</td>
</tr>
<tr>
<td>flowControlIn</td>
<td>Number</td>
<td>----</td>
</tr>
<tr>
<td>flowControlOut</td>
<td>Number</td>
<td>----</td>
</tr>
<tr>
<td>databits</td>
<td>Number</td>
<td>Number of databits to be sent</td>
</tr>
<tr>
<td>stopbits</td>
<td>Number</td>
<td>Stopbits to be sent</td>
</tr>
<tr>
<td>parity</td>
<td>Number</td>
<td>Parity (even or odd)</td>
</tr>
</tbody>
</table>

Table 3.1: The above table depicts information that will be stored by Modbus browser after adding a new device.
### Table name: Second

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Number</td>
<td>Address of device</td>
</tr>
<tr>
<td>Name</td>
<td>Text</td>
<td>Name of device</td>
</tr>
<tr>
<td>commandID</td>
<td>Number</td>
<td>Internal command id</td>
</tr>
<tr>
<td>id</td>
<td>Number</td>
<td>Internal device id</td>
</tr>
<tr>
<td>nid</td>
<td>Number</td>
<td>Internal network id</td>
</tr>
<tr>
<td>portName</td>
<td>Text</td>
<td>Name of port</td>
</tr>
<tr>
<td>function_code</td>
<td>Number</td>
<td>Function code supported by device</td>
</tr>
<tr>
<td>data1</td>
<td>Text</td>
<td>Command data</td>
</tr>
<tr>
<td>data2</td>
<td>Text</td>
<td>Command data</td>
</tr>
<tr>
<td>LRC</td>
<td>Number</td>
<td>Error code for ASCII mode</td>
</tr>
<tr>
<td>CRC1</td>
<td>Number</td>
<td>Error code for RTU mode</td>
</tr>
<tr>
<td>CRC2</td>
<td>Number</td>
<td>Error code for RTU mode</td>
</tr>
<tr>
<td>length</td>
<td>Number</td>
<td>Length of command</td>
</tr>
<tr>
<td>expected_pdu</td>
<td>Number</td>
<td>Return length</td>
</tr>
<tr>
<td>Filename</td>
<td>Text</td>
<td>Filename to store response</td>
</tr>
</tbody>
</table>

**Table 3.2:** The above table depicts the information to be stored about each command.
Chapter 4

Implementation
4. Implementation

4.1 System Implementation

We have divided project into various parts. Following are some of the major modules in our project.

1. Basic
2. Mytree
3. Scheduler
4. Server
5. Queue
6. Send
7. Databas
4.1.1 Basic:

This module contains basic classes for implementation of Modbus protocol. They are ADU and PDU. ADU stands for Application Data Unit and PDU stands for Protocol Data Unit. PDU is basically a command supported by each device and ADU tells to which device to send a specific PDU. One ADU may have more than one PDU. User has to enter ADU only once. PDU will be attached to ADU automatically when it is entered.

4.1.2 Mytree:

This is central ActionListener, which listens to all user commands through GUI, then process those request and handle them properly. It also connects to modserver to connect to other modbus client. It handles all database requests and maintains correct configuration file status. On startup it tries to connect to server, registers itself and sends configuration string.

4.1.3 Scheduler:

This is a command dispatcher, which adds and removes commands from the commands queue. When user starts or stops device, scheduler adds and removes those commands from the queue.

4.1.4 Server:

The main purpose of server is to connect two or more Modbus browsers so that they can control each other’s device remotely. Server collects all configuration string of the browsers and dispatches it to browsers who request other browser’s configuration file.
4.1.5 **Queue:**

The purpose for queue is to hold commands to be sent to the device(s). Queue essentially works as FIFO and sends commands first which were added first. Commands are added and removed by scheduler.

4.1.6 **Send:**

Sends and receives commands from queue to the device. Error control is done and if response is positive and with current checksum it is stored.

4.1.7 **Databas:**

Databas provides connectivity to MS-ACCESS so that device info and command info can be stored on hard disk. Databas uses standard ODBC-JDBC Bridge to connect java to MS-ACCESS.
4.2 Data Dictionary

A data is an organized listing of all elements that are pertinent to the system with precise rigorous definitions so that both the user and system analyst have common understandings of inputs, outputs, components of stores and intermediate calculations. The data dictionary compiles the characteristics of each data object and control item in a standard format.

4.2.1 File: basic.java

<table>
<thead>
<tr>
<th>Class: PDU</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of command</td>
</tr>
<tr>
<td>Function_code</td>
<td>short</td>
<td>Corresponding function code for a device</td>
</tr>
<tr>
<td>Data</td>
<td>int [ ]</td>
<td>Data area for a command</td>
</tr>
<tr>
<td>LRC</td>
<td>short</td>
<td>Calculate LRC for ASCII mode</td>
</tr>
<tr>
<td>CRC</td>
<td>short [ ]</td>
<td>Calculate CRC for RTU mode</td>
</tr>
<tr>
<td>Length</td>
<td>int</td>
<td>Length of PDU</td>
</tr>
<tr>
<td>Expected_pdu</td>
<td>int</td>
<td>Expected length of response string</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class: ADU</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CommandId</td>
<td>Int</td>
<td>Internal command ID for a command</td>
</tr>
<tr>
<td>Address</td>
<td>short</td>
<td>Device Address</td>
</tr>
<tr>
<td>X</td>
<td>PDU</td>
<td>Stores command information</td>
</tr>
<tr>
<td>Temp</td>
<td>int [ ]</td>
<td>Calculate CRC and LRC</td>
</tr>
</tbody>
</table>
### 4.2.2 device.java

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>adu</td>
<td>ADU</td>
<td>Store device information</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of device</td>
</tr>
<tr>
<td>ID</td>
<td>int</td>
<td>Device ID</td>
</tr>
<tr>
<td>transID</td>
<td>int</td>
<td>Transmission ID</td>
</tr>
</tbody>
</table>

### 4.2.3 deviceData.java

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>String</td>
<td>Stores name of node</td>
</tr>
</tbody>
</table>

### 4.2.4 errorcode.java

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>short []</td>
<td>Stores ADU</td>
</tr>
<tr>
<td>Length</td>
<td>int</td>
<td>Stores length of ADU</td>
</tr>
<tr>
<td>auchCRCHi</td>
<td>private final static short []</td>
<td>Lookup table</td>
</tr>
<tr>
<td>AuchCRCLo</td>
<td>private final static short []</td>
<td>Lookup table</td>
</tr>
</tbody>
</table>

### 4.2.5 lineparser.java

**Class: Netw**

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Name of network</td>
</tr>
<tr>
<td>Numb</td>
<td>int</td>
<td>Number of comports</td>
</tr>
<tr>
<td>Cm</td>
<td>COMPORT []</td>
<td>Stores info about comports</td>
</tr>
</tbody>
</table>

**Class: COMPORT**

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Name of comports</td>
</tr>
<tr>
<td>numb</td>
<td>int</td>
<td>Number of devices</td>
</tr>
<tr>
<td>Cm</td>
<td>ComDevice []</td>
<td>Stores info about devices</td>
</tr>
</tbody>
</table>

**Class: ComDevice**

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeviceName</td>
<td>String</td>
<td>Name of device</td>
</tr>
<tr>
<td>numb</td>
<td>int</td>
<td>Number of commands</td>
</tr>
<tr>
<td>CommandName</td>
<td>String</td>
<td>Stores info about</td>
</tr>
</tbody>
</table>
### 4.2.6 databas.java

### 4.2.7 mytree.java

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jt</td>
<td>JTree</td>
<td>A control that displays a set of hierarchical data on the left of mytree.</td>
</tr>
<tr>
<td>Mb</td>
<td>JMenuBar</td>
<td>Displays menu bar at the top of mytree.</td>
</tr>
<tr>
<td>Jpopupmenu</td>
<td>JPopupMenu</td>
<td>A small window which pops up and displays a series of choices.</td>
</tr>
<tr>
<td>mousclik</td>
<td>ActionListener</td>
<td>Listens popup menu.</td>
</tr>
<tr>
<td>listboxbutton</td>
<td>ActionListener</td>
<td>Listens list box.</td>
</tr>
<tr>
<td>inputListener</td>
<td>ActionListener</td>
<td>Listens input given.</td>
</tr>
<tr>
<td>commandListener</td>
<td>ActionListener</td>
<td>Listens device_nam class.</td>
</tr>
<tr>
<td>actionListener</td>
<td>ActionListener</td>
<td>Listens scroll1 class.</td>
</tr>
<tr>
<td>iconListener</td>
<td>ActionListener</td>
<td>Listens icon panel.</td>
</tr>
<tr>
<td>menuListener</td>
<td>ActionListener</td>
<td>Listens menu items.</td>
</tr>
<tr>
<td>toolbarListener</td>
<td>ActionListener</td>
<td>Listens toolbar items.</td>
</tr>
<tr>
<td>checkListener</td>
<td>ActionListener</td>
<td>Listens for device commands being checked.</td>
</tr>
</tbody>
</table>

### 4.2.8 SerialParameters.java

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Data Structure Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>portName</td>
<td>String</td>
<td>Specifies the port name to be connected to.</td>
</tr>
<tr>
<td>baudRate</td>
<td>int</td>
<td>Specifies baud rate for modbus device.</td>
</tr>
<tr>
<td>flowControlIn</td>
<td>int</td>
<td>Specifies flowControlIn for port.</td>
</tr>
<tr>
<td>flowControlOut</td>
<td>int</td>
<td>Specifies flowControlOut for port.</td>
</tr>
<tr>
<td>----------------</td>
<td>-----</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>databits</td>
<td>int</td>
<td>Specifies no of data bits to be send in a packet.</td>
</tr>
<tr>
<td>stopbits</td>
<td>int</td>
<td>Specifies the no of stop bits for modbus protocol.</td>
</tr>
<tr>
<td>parity</td>
<td>int</td>
<td>Specifies no of parity bits for modbus protocol.</td>
</tr>
</tbody>
</table>

4.3 ALGORITHM

4.3.1 Algorithm: LineParser

Objective:
Separate out network name, com port names, device names and command names in configuration string

Level 1: Separate network name

1. Detect index of “{“. Put it in closeIndex.
2. Substring from start of configuration string to closeIndex is name of network.

Level 2: Separate Comport name

1. Detect index of “{“. Put it in startIndex.
2. if “}” detected, put it in closeIndex
3. if(closeIndex-startIndex) then no comports goto exit
4. Detect index of “{"Put it in closeIndex.
5. Substring (startIndex, closeIndex) is new comport name.
6. Repeat for delimiter “/”.

Level 3: Separate Device name (Repeated for each COM ports)

1. Detect index of “{“Put it in startIndex.
2. if “}” detected, put it in closeIndex
3. if(closeIndex-startIndex) then no devices goto exit
4. Detect index of “[“. Put it in closeIndex.

5. Substring (startIndex, closeIndex) is new device name.

6. Repeat for delimiter “,”.

Level 2: Separate Command name (Repeated for each device)

1. Detect index of “[“. Put it in startIndex.

2. if “[” detected, put it in closeIndex

3. if(closeIndex - startIndex) then no commands goto exit

4. Detect index of “#“. Put it in closeIndex.

5. Substring (startIndex, closeIndex) is new command name.

6. Repeat for delimiter “#”.

4.3.2 Algorithm: comportIdentification

Objective: Detect number of comports on your PC and their names

Number of comports:

1. Enumeration portlist = getPortIdentifiers ()

2. counter = 0.

3. while portlist has more Elements

   3.1 if port type is Serial then counter ++

1. return counter
Name of comports:

**Input:** Index
1. Enumeration portlist = getPortIdentifiers().
2. counter = 0.
3. while portlist has more elements
   3.1 if port type is Serial
      3.1.1 if Input == counter return name of comport
      3.1.2 counter++

4.3.3 **Algorithm: removeDevice**

**Objective:** remove device name from configuration file, database and tree structure

**Level 1:** remove device name from configuration file
1. Search name of device in configuration file.
2. Note startIndex and closeIndex of name.
3. Remove it from configuration string.
4. Restore the string.

**Level 2:** remove device name from database
1. Get deviceId
2. Delete device records using DELETE command of SQL both in first and second table.

**Level 3:** remove device name from tree structure
1. Get Last Selected path component.
2. Remove device node from its parent node using TreeModel.
3. Set Selection path to comport name.

4.3.4 Algorithm: removeNetwork

**Objective:** remove network from configuration.mod and delete Network Configuration file

1. Determine networkId (i.e. child number of root) and store in numb.
2. Read configuration.mod.
3. Get filename from numb (th) string.
4. delete that file.
5. remove that filename from configuration.mod.

4.3.5 Algorithm: deleteCommand

**Objective:** delete command from Network Configuration file and device database

1. Search for command name in Network configuration file.
2. Store startIndex and closeIndex of Command Name.
3. Create new string excluding command name to be deleted.
4. get commandId to be deleted( Index of checkbox in panel )
5. Remove it from database using DELETE command of SQL for deviceId and commandId
6. remove checkbox from panel
4.3.6 Algorithm: scheduler

Objective: Adds and deletes elements in queue according to user starts and stops the devices.

Initial:
1. Determine no. of comports on the PC.
2. Start threads for each comports.
3. threads will share queue with scheduler

Event: User starts a device
1. Determine deviceId to be started.
2. Load following things in new queue element about selected device command from database
   i. Address
   ii. function code
   iii. data field
   iv. error code
   v. serialparameters
3. Add the above element in queue for appropriate Thread.
4. Each thread uses above information to send using Modbus Protocol.
5. Repeat step 2 until all selected commands of devices are added.

Event: User stops a device
1. Determine deviceId to be removed.
2. Remove all elements in queue related to that device.
3. Stop.
4.3.7 Algorithm: showIcons

Objective: Show icons on right side for network, comports, devices

Input: String Array of networks/comports/devices

1. Remove previous icons
2. Add button with a specified gif image
3. give name to button from string array
4.3.8 Algorithm: Master to Master Connection

Initial: Server side

1. Listen for connections.
2. Device registers itself.
3. Add Client Socket in connection Vector.
4. Start a new Thread for that client.
5. Accept configuration string from this client.

Initial: Client side

1. Connect to server using client socket at startup.
2. Send configuration string.
3. Listen for requests.

Event: User requests Network List

Server:

1. Server retrieves device Sockets from connection Vector.
2. Add device name and IP address in a string and sends it.

Client:

1. Receives Network List.
2. Displays it.
Event: User selects a Master out of Network list

Server:
1. Server interprets request String and separates out IP address.
2. Selects corresponding socket thread and sends configuration String.

Client:
1. Client receives configuration string.
2. Stores this configuration string in a file.
3. Client makes changes in configuration.mod.

Event: Master1 starts commands of Master2

Client1:
1. Prepare a string with IP address of Master2 plus name of comport selected, Device ID and commands selected.
2. Send this start string to server.

Server:
1. Accept the above string from Client 1.
2. Extract IP address of Client2.
3. Select Socket of corresponding Client2 and send this String.

Client2:
1. Client2 accepts the above start string.
2. Checks if commands already started.
3. If “yes” skip else add that command in queue.
4. Repeat step 3 until command list ends.
Event: Master1 stops commands of Master2

Client1:
1. Prepare a string with IP address of Master2 plus name of comport selected, Device ID and commands selected.
2. Send this stop string to server.

Server:
1. Accept the above string from Client 1.
2. Extract IP address of Client2.
3. Select Socket of corresponding Client2 and send this String.

Client2:
1. Client2 accepts the above stop string.
2. Checks if commands already started.
3. Remove that command in queue.
4. Repeat step 3 until command lists ends.

Event: Master exits or Disconnects

Client:
1. Client sends “Remove me” string.

Server:
2. Server determines source of “Remove me”.
3. Removes Socket from connection Vector.
4. Thread for Client terminates.
4.3.9 Algorithm: errorcode

Objective: Generate LRC and CRC codes

LRC: Input ADU string

1. LRC = 0
2. I = 0
3. while ( I < lengthOf(ADU string))
   a. LRC += charAt ( I ) of ADU string
   b. I++
4. return LRC

CRC: Input ADU string

1. CRC[] = {0xFF, 0xFF} // stores CRC
2. nextShort = 0
3. I = 0
4. while( I < length )
   a. nextShort = 0xFF & (charAt(I) of ADU string)
   b. uIndex = CRC [0] XOR nextshort
   c. CRC [0] = CRC [1] XOR auchCRCHi [uIndex]
   d. CRC [1] = auchCRCLo [uIndex]
5. CRC [0] = CRC [0] & 0xff
7. Return CRC
Reference Tables:

*auchCRCHi*

short [] auchCRCHi = {
0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC0, 0x80, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x04, 0x01, 0xC

*auchCRCLo*

short[] auchCRCLo = {
0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04, 0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09, 0x08, 0xC8, 0xB8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC, 0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3, 0x11, 0xD1, 0xD0, 0x10, 0xF0, 0x30, 0x31, 0xFD, 0xFB, 0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A, 0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38, 0x28, 0xE8, 0xE9, 0x29, 0x22, 0xE2, 0x3E, 0x23, 0xE1, 0x21, 0x20, 0xE0, 0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4, 0x6C, 0x6B, 0xA8, 0x68, 0x6F, 0xA9, 0x6D, 0xAF, 0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0x6E, 0x26, 0xA2, 0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B, 0x4A, 0x4F, 0x8D, 0x4D, 0x4C, 0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80, 0x40}
Chapter 5

Testing
5 Testing

5.1 Test Plan:

Software Testing is the procedure that is conducted to uncover the errors that were made inadvertently as it was designed and constructed. Testing is the set of activity that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can specify test case design techniques and testing methods—should be defined for the software process.

“Testing is the unavoidable part of any responsible effort to develop a software system“.

A strategy for software testing may also be viewed in the context of the spiral. Unit testing begins at the vortex of the spiral and concentrates on each unit of the software as implemented in the source code. Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter validation testing, where requirements are established as part of software requirement analysis are validated against the software that has been constructed. Finally, we arrive at the system testing, where the software and other system elements are tested as whole. To test computer software, we spiral out along streamlines that broaden the scope of testing each turn.
5.1.1 Unit Testing:

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of the tests and the errors those tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

The module interface is tested to ensure that information is properly flows into and out of program unit under test. Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm’s execution. All independent paths (basic paths) through the control structure are exercised to ensure that all statements in a module operate properly at boundaries established to limit or restrict processing. And finally, all errors handling paths are tested. Following are test cases performed by us.

5.1.1.1 Mytree:

Objective:

This class acts as the action listener for all the action that takes place in the main browser such as adding and deleting the network from LAN, adding new devices (soft PLC’s) or removing the devices (soft PLC’s) from the COM ports, which are detected by the com port identifier, adding new commands to the device and removing the command for the device of the default network.
### 5.1.1.2 Error code:

**Objective:**

The function of this class is to calculate the CRC or LRC for the frame generated by the protocol that is calculate the CRC or LRC for the PDU. In unit testing the error code module is tested separately for calculating the error code for different length PDUs and for different values of PDUs and it found that error code module is working properly.

**Case 1:**

**Input:**

- Transmission mode: ASCII
- Device Address: 12
- Function Code: 43
- Data: 3433

**Output:**

- LRC: 182

**Case 2:**

**Input:**

- Transmission mode: ASCII
- Device Address: 12
- Function Code: 43
- Data: 3443

**Output:**

- LRC: 198

**Test Result:** The above class is working properly
**5.1.1.3 Line parser:**

**Objective:**

The input to line parser is the configuration file which contains name of network, comports, devices it supports and commands supported by it. It smartly separates out these components and stores them in proper variables.

**Input:**

```
network0{COM1()/COM2()}
```

**Output:**

- Name of network: network0
- Number of Serial ports: 2
- Name of Serial ports: COM1, COM2
- Number of devices under COM1: 0
- Number of devices under COM1: 0

**Input:**

```
network0{COM1(PLC[])/COM2()}
```

**Output:**

- Name of network: network0
- Number of Serial ports: 2
- Name of Serial ports: COM1, COM2
- Number of devices under COM1: 1
- Number of devices under COM1: nil
- Name of device under COM1: PLC
- Number of commands under PLC: 0

**Input:**

```
network0{COM1(PLC[])/COM2()}
```
Output:

Name of network: network0
Number of Serial ports: 2
Name of Serial ports: COM1, COM2
Number of devices under COM1: 1
Number of devices under COM1: nil
Name of device under COM1: PLC
Number of commands under PLC: 0

Test Result:  The above class has been found to be working properly.
5.1.1.4 fileUtil:

Objective:
This class provides functionality to save configuration files and configuration strings. Following is testing done on it.

Name of Function:
Write:
Objective: writes to network configuration file.
Input: filename, data
Type of file: Random Access File

Test Input:
Filename: network0.mod
Data: network0{COM1(PLC[])/COM2()}

Test Result: network0{COM1(PLC[])/COM2()} written successfully in file with filename network0.mod.

Name Of Function:
Read:
Objective: writes to network configuration file.
Input: filename
Return: Data
Type of file: Random Access File

Test Input:
Filename: network0.mod
Data in file: network0{COM1(PLC[])/COM2()}

Test Result: network0{COM1(PLC[])/COM2()} returned successfully from file with filename network0.mod.
Name of Function: writeConfig

Objective: writes to configuration file configuration.mod.

Input: filenames[]

Type of file: Random Access File

Test Input:

Filename: network0.mod, network1.mod

Test Result: filenames network0.mod, network1.mod written successfully in file named configuration.mod.

Name of Function: readConfig

Objective: reads from configuration file configuration.mod.

Input: filenames []

Type of file: Random Access File

Test Input:

Filename: null

Contents of configuration.mod: network0.mod

Test Result: filenames network0.mod returned successfully from file named configuration.mod.

Test Result for fileUtil:

It is been found that class fileUtil is working properly.
5.1.2 Integration Testing:

*Integration testing* is the systematic technique for constructing the software architecture while at the same time conducting tests to uncover the errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been dedicated design.

There are different integration method are presents as follows:

**Top-down integration:** Top-down integration testing is an incremental approach to construction of the software architecture. Modules are integrated by moving downward through the control hierarchy, begins with the main program.

**Bottom-up integration:** Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules.

**Regression testing:** Each time a new module is added as a part of integration testing, the software changes.

**Smoke testing:** Smoke testing is an integration testing approach that is commonly used when software products are being developed.

We are using bottom-up integration testing. The modules are incorporate together and called from main routine. The inputs of each module are kept track of and are available till application ends. The purpose of this testing is to check the software for bugs during its navigation through various levels built in the form of routines. The input from every routine should reach the higher routines, also graph should not be corrupted on the way through the routines even when previous processes executing. Hence, the software should perform for several applications irrespective of any previous application in execution.
Care was taken for the following things:

1. The capability of the modules in their input parameters and return values.
2. The correctness of the overall system architecture being realized
3. Any manipulation of the data in the interfaces
4. The effectiveness of the combined modules
5. Interface integrity
6. Functional validity
7. Information content
8. Performance

5.1.2.1 Bottom-up Testing:

As stated earlier in Bottom-up testing lower level modules in hierarchy are tested first until final module is tested.

Bottom-Up integration strategy may be implemented with the following steps:

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software sub function.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.
**Fig. 5.1** Bottom-up Integration testing for sending commands to devices

**Explanation:**

**Send:**

Initially the send module is tested which works closely at device level to send data serially using java comm Api. It gets input from Queueimplementer. For test conditions data can be manually sent. Data can be tested on receiver computer using serial port testing softwares.

**Queue implementer:**

Queue implementer consist of a queue in which device commands are added and which are sent to **Send** module which safely transmits it to the device.
Scheduler:

Scheduler adds and removes from queue in command implementer. Java’s inbuilt semaphore technique is used to allow synchronized access to queue.

Mytree:

Mytree Listens to user inputs and on pressing of start button uploads the command from database and sends it to the scheduler.

Test Result: The above modules are found to be firmly integrated with each other.

5.1.3 Validation Testing

After all the errors are removed and integration testing is completed, the validation testing begins. It can be defined as validation succeeds when software functions in a manner that can be reasonably expected by the customer.

- Does project matches the problem statement defined?

The project is exactly as was specified in the problem statement submitted at the beginning of the project.

- Does the functionally matches the SRS?

The project works exactly as in defined in the SRS. The behavior of each of the modules is, as was given in the SRS.

After each validation test has been conducted, the function or performance characteristic conforms to specification and is accepted.
5.1.3.1 Validation testing for User Interface:

User Interface was built in Java swing. All swing components were found to be working correctly.

5.1.3.2 Validation testing for Device Management System:

All validation has been performed on this system. Device addition, deletion, modification has been performed successfully. Commands when added, deleted works fine with system. The database system works smoothly and corresponding changes to configuration string are performed correctly.
Chapter 6

Result
6. RESULT:

6.1 Modbus Browser:

Fig 6.1: Front end.
6.2 Default Network:

Fig 6.2: Default Network: network0.

Configuration file: network0{COM1()/COM2()}

Configuration file being created for the default network (network0).

Adding of device:

Fig 6.3: Device to be added to COM1 port of default network.
**Fig 6.4:** Add device popup window.

**Fig 6.5:** Updated Database: Table first is updated with every device added.

**Configuration file:** network0{COM1(PLC[])/COM2()}
Contents of configuration file being updated after adding the device (PLC) to COM1 port of network0.

6.3 Adding of device command:

Fig 6.6: Add command popup window.
**Fig 6.7:** Updated Database: Table second is updated with every device command added.

**Configuration file:** network0{COM1(PLC[temperature])/COM2()}

Contents of configuration file being updated after adding device command (temperature) to PLC.

**Fig 6.8:** Device Added (PLC) to COM1 port of network0 and temperature is the device command for PLC.
6.4 Communication with device:

**Fig 6.9:** After selecting the temperature command and clicking the start button the following data given below is send to PLC at the respective address.
Baud Rate 19200
Flow control in None
Flow control out None
data bits 8
stop bits 1
Parity Bits Even
device Index 0
ADU 110111133
Port names COM1
Baud Rate 19200
Flow control in None
Flow control out None
data bits 8
stop bits 1
Parity Bits Even
device Index 0
ADU 110111133
Port names COM1
Baud Rate 19200
Flow control in None
Flow control out None
data bits 8
stop bits 1
Parity Bits Even

**Fig 6.10:** Data send to PLC.
6.5 Device Information:

![Device Info Diagram]

**Fig 6.11:** View device information.
**Fig 6.12:** Window displaying device information.

### 6.6 Configure device:

**Fig 6.13:** Configure device.
**Fig 6.14:** Popup window to configure device.
Fig 6.15: Popup window fields being enabled after clicking the update button, allowing us to configure the device.

Fig 6.16: Updated device being added to the COM1 port of network0.
Fig 6.17: Updated Database: Table first is updated with every device configured.

<table>
<thead>
<tr>
<th>address</th>
<th>device_name</th>
<th>id</th>
<th>nid</th>
<th>noc</th>
<th>tid</th>
<th>portName</th>
<th>baudRate</th>
<th>flowControlIn</th>
<th>flowControlOut</th>
<th>databits</th>
<th>stopbits</th>
<th>parity</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>BOILER</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>COM1</td>
<td>19200</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Configuration file: network0{COM1(BOILER[temperature])/COM2()}

Contents of configuration file being updated after configuring the device.

Fig 6.18: Updated Database: Table second is updated with every device configured.

<table>
<thead>
<tr>
<th>address</th>
<th>name</th>
<th>commandID</th>
<th>id</th>
<th>nid</th>
<th>portName</th>
<th>function_code</th>
<th>data1</th>
<th>data2</th>
<th>LRC</th>
<th>CRC1</th>
<th>CRC2</th>
<th>length</th>
<th>expected_pdu</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>temperature</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>COM1</td>
<td>10</td>
<td>1111</td>
<td>51</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>25</td>
<td>temp</td>
</tr>
</tbody>
</table>
6.7 Remove Device:

**Fig 6.19:** Removal of device connected to a COM port.

**Fig 6.20:** Network0 been updated after device (BOILER) has been removed from COM1 port.
**Fig 6.21**: Updated Database: Table first is updated with every device being removed.

<table>
<thead>
<tr>
<th>first</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>device name</td>
<td>id</td>
<td>nid</td>
<td>noc</td>
<td>tid</td>
<td>portName</td>
<td>baudRate</td>
<td>flowControlIn</td>
<td>flowControlOut</td>
<td>databits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig 6.22**: Updated Database: Table second is updated with every device being removed.

<table>
<thead>
<tr>
<th>Second</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>name</td>
<td>commandID</td>
<td>id</td>
<td>nid</td>
<td>portName</td>
<td>function_code</td>
<td>data1</td>
<td>data2</td>
<td>LRC</td>
<td>CRC1</td>
<td>CRC2</td>
<td>length</td>
<td>expected_pdu</td>
<td>filename</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Configuration file**: network0{COM1()/COM2()}

Contents of configuration file being updated after removing the device(BOILER) from COM1 port of network0.
Chapter 7

Conclusion
7.1 Conclusions

Modbus Browser has been implemented successfully which fulfills all requirements. Thus we have developed a Device Manager which can manage Modbus devices correctly. Modbus protocol has been implemented which is utilized by this browser to communicate with this device.

7.2 Applications of the System

- Graphical Device Manager helps to manage device properly.
- Any serial communication Modbus device can be connected to the system.
- This uses polling method and separate thread for each port is used to communicate with devices.

7.3 Limitations

- System being generic can’t support specific functions of devices like sending a single command only once.
- A real time graph cannot be created because of varying nature of incoming data from devices.
- No network security for master to master connection.

7.4 Future Scopes

- Efforts can be put in to develop a real time graph which takes varying length of input and displays information real time.
- Master to Master can be made more secured and put in many more features like transferring real time data remotely.
Chapter 8

Bibliography
8. BIBLIOGRAPHY

8.1 Books:

1. THE UNIFIED MODELING LANGUAGE – USER GUIDE
   - Grady Booch
   - James Rumbaugh
   - Ivar Jacobson

2. Core Java 2 Volume I-Fundamentals
   - Cay S. Horstmann, Gary Cornell

3. JAVA 2 – BLACK BOOK
   - Steven Holzner

4. JAVA 2 – THE COMPLETE REFERENCE
   - Herbert Schildt

8.2 Web Sites:

1. www.modbus.org
3. www.modicon.com
4. www.java.sun.com