To the Graduate Council:

I am submitting herewith a thesis written by Naresh Karnam entitled “MODBUS Implementation Using A Serial Link.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Paul B. Crilly, Major Professor

We have read this thesis
and recommend its acceptance:

Michael J. Roberts

Seddik M. Djouadi

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original Signatures are on file with official student records.)

MODBUS IMPLEMENTATION USING A SERIAL LINK

Thesis Presented For

Master of Science Degree

University of Tennessee-Knoxville

Naresh Karnam

December 2007

Dedicated to

My father Krishna Rao

'anna'-for your wonderful memories

ABSTRACT

Serial Communication is a process of sending one bit at a time, sequentially over a communication channel or a computer bus. RS-232 and RS-484 are two such examples of the architecture of Serial Communication. This work primarily aims at establishing a communication channel between three entities namely the Controller also called the Console, the Relay Unit and a PC. This work aims at designing the communication model, using the Client-Server architecture and using the MODBUS protocol, as the standard for transmitting the bits in a Remote Terminal Unit (RTU) mode and for framing the characters. The MODBUS protocol also specifies the Cyclic Redundancy Check (CRC) algorithm for error detection which is used as a part of the design for framing of the characters.

ACKNOWLEDGEMENTS

I would like to take this opportunity to sincerely thank my major advisor Dr. Paul Crilly, for guiding me throughout my thesis work. I would like to thank him for accepting my work and for his immense support and constant advice. I would also like to thank Dr. Michael Roberts and Dr. Seddik Djouadi for accepting to be a part of my thesis committee and for taking time to read and correct my thesis.

I am hugely indebted to Dr. Alexeff, for his outstanding support and encouragement during my Master's study. He is undeniably the best person in the world that I have met and will remain the same forever. I express my extreme gratitude to him for financially supporting my Master's education and also for the vast amount of knowledge he shared with me. It was an honor to work under him.

I would also like to express my gratitude to Robertshaw Industrial Products- Maryville, for accepting me as an intern and allowing me to present my work done there, as my final thesis. I would like to have a special mention about Brien Evans, my manager who mentored and guided me throughout my internship from January to May 2007. I want to thank Brien for his time and commendable patience in reading my entire thesis document and providing feedback on it. I would say that he stood out as a role model to me for my future in the corporate world. Here, I would also like to especially thank Sampath, Steve Berry and Wojtek Miller for the great support they provided. The interactions with Wojtek and Brien served to a great extent in getting the task completed, although

I must agree that a majority of those interactions with Wojtek were completely humorous and witty.

I would like to express my heartfelt thanks to my friends at UT- Archana, Bharadwaj aka ‘raj’, Chakri aka ‘mama’, Shashank aka ‘paidi’ for their outstanding support and encouragement. The banter and jokes we shared, the discussions wherein me and ‘paidi’ would change the entire set-up of the Indian cricket team with ‘mama’ mocking at us, the number of eatery restaurants Archana introduced to us, the way we mocked at ‘mama’ spilling over his food, ‘raj’s’ cooking disasters and the captivating drives we had during the 2 years would be part of my most pleasant memories with them.

I also want to thank my friends Eric, Nanditha, Swetha and Shankar for their enjoyable company at UT. I would like to thank Swetha for her inputs as my colleague during my internship and Eric for his valuable suggestions and insightful discussions. I would like to thank Nanditha for being such a wonderful friend to me. This is one gang that would stay in my memory forever.

I would also like to put across my love and appreciation to my childhood friends- Vatsan, Hari, Kranthi, Pardhu, Swetha, Shanthi and my undergrad friends-‘the emptyheads’ as we fondly call ourselves, for their constant love and friendship, that has been a major source of support and inspiration to me. My thanks to my friends would be incomplete without mentioning my best friend forever- Gayatri. She has been a pillar of support and strength to me and has

stood by me in every tough situation. She has been a great source of inspiration to me and has helped me make many important decisions in my life without ever expecting anything back from me.

I want to thank my grandmother, Mrs. Suseelamma, whose persistence and infinite encouragement to take up higher studies brought me to USA. I am indebted to her for imbibing so many values and principles in me which have helped me come so far in life and for compromising on her health to bring me up. Without her blessings and sacrifices, nothing of this would have been possible. I want to thank my brother-'chinna' for his constant support to me and for his many sacrifices for me. He always has shown me a different way to look at life. I want to express my immense gratitude to my uncle-'kaka' and my 'amma' for the amount of sacrifices and hardships they have gone through in their lives to provide good education to me. I want to thank them for their unmatchable efforts in bringing me up in dire financial circumstances. I want to thank 'kaka' for being everything for me in life-a father, a friend, a guide and 'GOD' to me. I also want to thank 'amma' for giving so much to me. She has been a great pillar of strength and has been my major source of inspiration for the amount for struggles she has undergone to bring me up. Finally, I want to thank everyone who have directly or indirectly helped me finish this work.

Table of Contents

Contents

Chapter 1: INTRODUCTION 1

Chapter 2: THEORY & DESCRIPTION 3

System Description 3

Architecture 5

Chapter 3: MODBUS 10

MODBUS Function Codes 16

MODBUS Over Serial Line 17

MODBUS Data Link Layer 18

Master/Slave State Diagrams 20

MODBUS Addressing 26

MODBUS Frame 26

Transmission Modes 28

RTU Transmission Mode 29

ASCII Transmission Mode 35

Chapter 4: EXPERIMENTS & RESULTS 42

Previous Tasks Accomplished 42

Procedure & Experimental Results: 46

Chapter 5: CONCLUSIONS 70

References 72

Vita 74

Table of Figures

Figure 2.1: Block Diagram of the System 4
Figure 2.2: Client Server Architecture 6
Figure 3.1: MODBUS frame 11
Figure 3.2: Client Server Transaction for a normal response 14
Figure 3.3: Client Server Transaction for an exception response 15
Figure 3.4: MODBUS protocol and the ISO/OSI Model 18
Figure 3.5: Unicast Mode of Operation 19
Figure 3.6: Broadcast Mode of Operation 20
Figure 3.7: Master State Diagram 21
Figure 3.8: Slave State Diagram 24
Figure 3.9: MODBUS Addressing Model 27
Figure 3.10: MODBUS Serial Line PDU 27
Figure 3.11: Bit sequence in RTU mode 29
Figure 3.12: Bit Sequence in RTU mode for NO parity 29
Figure 3.13: RTU MODBUS Frame 30
Figure 3.14: Message Framing in RTU mode and Timing constraints 32
Figure 3.15: State Diagram of RTU Transmission Mode 33
Figure 3.16: Bit Sequence in ASCII mode 36
Figure 3.17: Bit Sequence in ASCII mode for NO parity 36
Figure 3.18: ASCII MODBUS Frame 37
Figure 3.19: State Diagram of ASCII Transmission Mode 39
Figure 4.1: Previous Console Action Flow 43
Figure 4.2: Previous Relay Unit Action Flow 45
Figure 4.3: Hyper-Terminal window receiving characters every second 49
Figure 4.4: Hyper-terminal window after one minute 49
Figure 4.5: Hyper-terminal window after 2 minutes 50
Figure 4.6: Hyper-terminal window after 5 minutes 50
Figure 4.7: Hyper-terminal window after 10 minutes 51
Figure 4.8: Hyper-terminal window receiving the packet after 1 minute 53
Figure 4.9: Hyper-terminal window receiving the packet after 2 minutes 54
Figure 4.10: Hyper-terminal window receiving the packet after 5 minutes 54
Figure 4.11: MODBUS read packet 56
Figure 4.12: Hyper-terminal receiving the MODBUS read packet after 1 minute 58
Figure 4.13: Hyper-terminal receiving the MODBUS read packet after 2 minutes 58
Figure 4.14: MODBUS read response packet 60
Figure 4.15: MODBUS write packet 61

Figure 4.16: Hyper-terminal window receiving the MODBUS write packet after 1 minute 63
Figure 4.17: Hyper-terminal window receiving the MODBUS write packet after 2 minutes 63
Figure 4.18: MODBUS write response packet........ 66
Figure 4.19: Console Final Action Flow........ 68
Figure 4.20: Relay Unit Final Action Flow........ 69

Chapter 1: INTRODUCTION

The outcome of this work contributes to a system that monitors fluid levels in tanks. Industries such as petroleum, propane, chemicals, agriculture, pharmaceuticals, food and beverages, waste and water management, need to have maximum control over their products distribution, collection and overall vessel management. Thus they need to remotely monitor their fuel tanks in order to have an uninterrupted supply and refills at the appropriate time. These fuel tanks are situated in remote areas/sites and would require a fair amount of travel to be reached.

The system to be designed aims to remotely monitor the fluid levels using float switches and to also monitor various other parameters of the tanks. Currently the system employs two relays while monitoring three tanks. The proposed system would instruct the user for a refill as and when required, thereby reducing the number of trips to the site for the user. Additionally the system to be designed is aimed to extend the existing capabilities as per the customer requests. The system to be designed now would employ 10 relays to monitor the tanks. Also the system to be designed aims to be more complex in nature when compared to the previous designs since it would interface an existing product that provided point and continuous level measurement to a separate unit that could potentially be reused for other applications.

This work involves three principle entities viz. Controller Unit, Relay Unit and a Personal Computer. The Controller is typically installed either outdoors near the filling units or near the office locations. The Relay Unit has a horn located near its enclosure and additional horns and lights could be connected to the individual relays as well. The two units are connected using an RS-232 [1] cable between their respective serial ports. The two units are programmed using the AVRISP-II programmer. The programmer burns the bit file of the coded communication and logic model compiled using the AVR studio, onto the EPROM of each of the units.

The Controller and the Relay Unit have their own set of logic and functions to be performed in accordance to the application. This communication model along with the logic model and other set of functions are implemented in C/C++ using the IAR embedded workbench tool. The RS-232 [2] communication between the two units is thereby established with the bits being framed and coded in accordance to the MODBUS protocol.

The document further focuses on the description of the system, the requirements of the communication system and the architecture model used in Chapter 2. Chapter 3 describes the MODBUS protocol implementation and the logic design of the packets. The Experiments and Results are discussed in the Chapter 4, followed by the Conclusions and suggestions for future work in Chapter 5. The references are provided at the end.

Chapter 2: THEORY & DESCRIPTION

System Description

The system to be designed constitutes of the following entities

1. Switch Monitors
2. Multi-Alarm Tank Console/Controller Unit
3. Relay Unit
4. PC.

This work mainly deals with the Tank Console, the Relay Unit and the PC. In the application, each propane tank whose fluid levels are being monitored is connected to a Switch Monitor. Each propane tank has a float switch which is used to measure the level of the fluid in the tank at regular intervals. Based on the recordings, the fluid levels are calculated and relayed back to the corresponding Switch Monitor. The Switch Monitor is encompassed with two switches denoting ON and OFF. The system designed allows for 3 Switch Monitors to be connected to the Tank Console. The block diagram of the system is shown in the Figure 2.1. The Tank Console is connected to the Relay Unit using an RS-232 cable through its serial ports. The Tank Console and the Switch Monitors are connected through a licensed wireless communication network. The Tank Console alternates its communication between the radio from the Switch Monitor and the Relay Unit using its serial port. The Tank Console actuates the relays in the Relay Unit based on the information provided by the Switch Monitor.

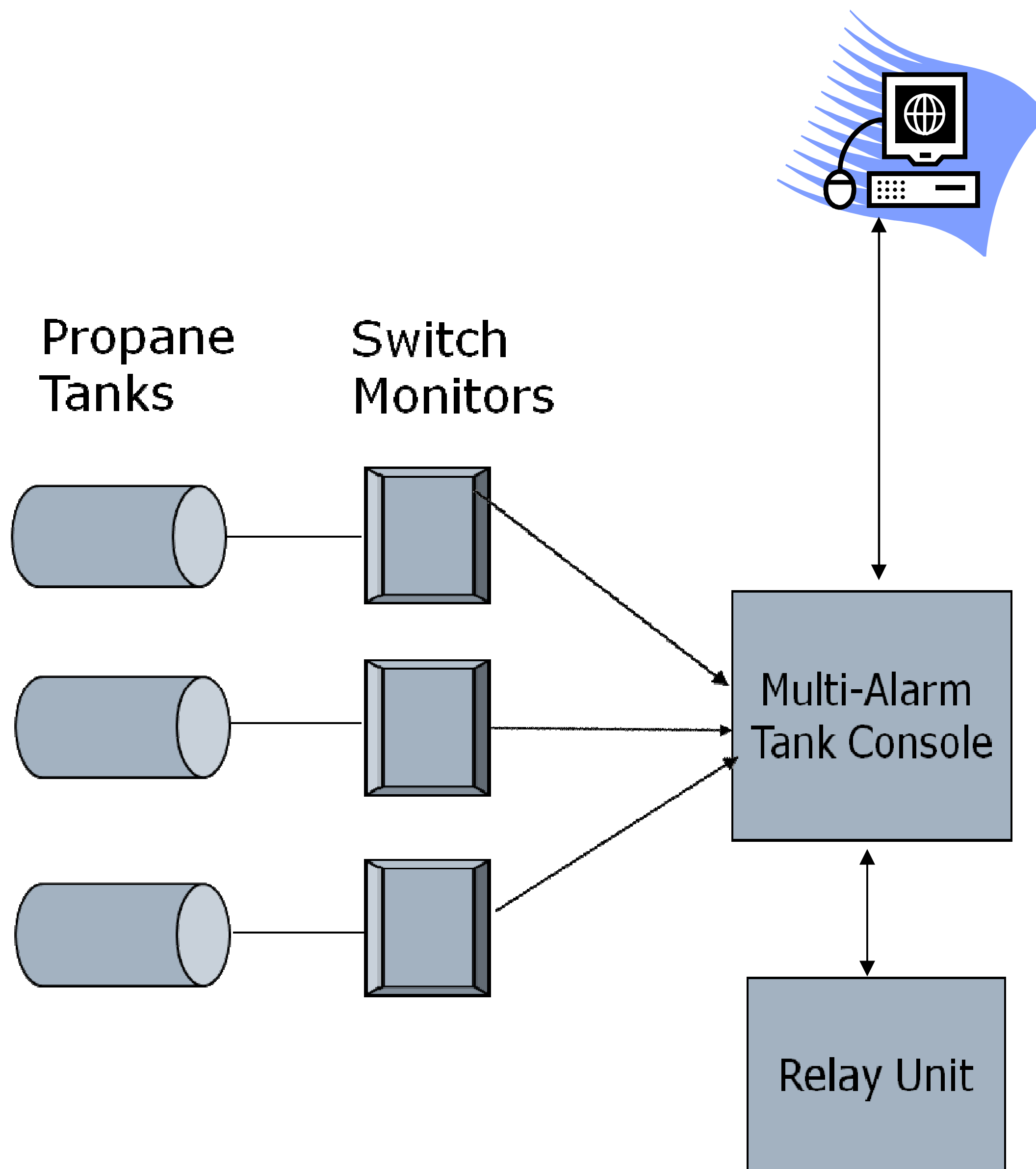


Figure 2.1: Block Diagram of the System

The Relay Unit has a horn circuitry which blows horns when a relay is set, indicating when a refill is needed. The Tank Console is intended to provide additional relay output capability compared to the existing Alarm Console which has two relay outputs while monitoring up to three tanks.

The design module consists of the Tank Console interfacing to the Relay Unit using the Tank Console's EIA\TIA-232 interface. The Relay Unit constitutes a motherboard consisting of the relays, alarm horn circuitry and the EIA\TIA-232 to EIA\TIA-485 conversion circuitry. Also the communication between the Tank Console and the PC had to be modeled for initial set-up and further updates. The Tank Console and the Personal Computer are connected via the RS-232 cable. The communication model implemented on these two design boards was based on the Client-Server architecture and the Modbus protocol was used as the standard for data transfers.

Architecture

The Client Server architecture model [3] is shown below in the Figure 2.2. The Client-Server architecture is implemented in a computer network. As the name suggests, the architecture constitutes of two main entities namely the Client and the Server. A Client is defined as a requester of services and a Server is defined as the provider of services. A single machine can be both a Client and a Server depending on the software configuration. Each Client or Server connected to a network can also be referred to as a 'node'. The most basic type of Client-Server architecture employs only two types of nodes:

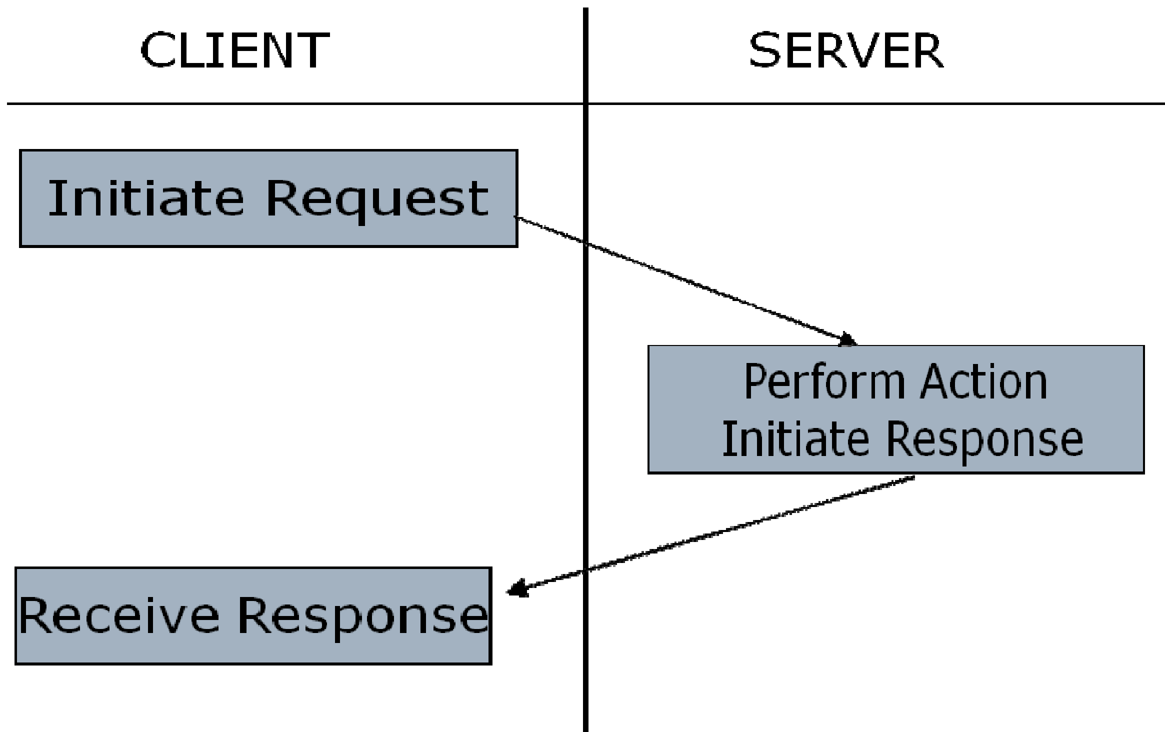


Figure 2.2: Client Server Architecture

Clients and Servers. This type of architecture is sometimes referred to as two-tier. It allows the devices to share information and resources. The Client can send data or other information requests to one or more Servers connected on the network. In turn, the Servers can accept these requests, process them, and return the requested data or information to the client. Although this concept can be applied for a variety of reasons to various kinds of applications, the architecture remains fundamentally the same.

A Client is a device that accesses a service on another device through some kind of network or connection. The term was first applied to devices that were not capable of running their own stand-alone programs, but could interact

with remote devices through a network. A Client is usually referred to as the Master in the network. The characteristics of a Client include

1. Active part of the network.
2. Initiates requests.
3. Waits for replies.
4. Receives the replies.
5. Connects to a small number of servers at any point of time.
6. Usually interacts directly with end-users using a graphical interface or through other means of alerts.

A Server is device that accepts connections in order to accept requests for data and information, service them and to send back responses. A Server can also be defined as a computer appliance or an appliance hardware that provides specific services on a network. Servers can be made using customized hardware for specialized needs of the application and network configuration. A Server is usually referred to as a Slave in the network. The characteristics of a Server include

1. Passive part of the network.
2. Waits for requests from the Clients.
3. Receives the requests.
4. Processes the requests.
5. Replies back to the Clients for their corresponding requests.

6. Accepts connections from a large numbers of Clients at any point of time.
7. Usually does not interact with end-users.

Advantages of Client-Server architecture:

1. The roles and responsibilities of the entire computing system are distributed amongst several independent devices that are known to each other only through the network.
2. There is greater ease of maintenance, for example it is easy to replace, repair, or upgrade a Server without the Clients being affected.
3. Data storage is centralized, thereby making updates to those data easier to be administered.
4. All the data is stored on the Servers, and hence there is a far greater security and access control to resources. Only Clients with appropriate permissions are granted access and are allowed to effect any changes to the data and information.
5. This model allows a system to function with multiple Clients with multiple functionalities and capabilities.

Disadvantages of Client-Server architecture:

1. As the number of Clients and the number of requests to a Server increases, the traffic in the network grows heavily leading to congestion.

2. This model lacks robustness. The failure of a Server will lead to a large number of Client requests not being processed and the Client sitting idle for a long time.

In this work, the Console was modeled on the lines of a Client and the Relay Unit and Personal Computer were modeled on the lines of a Server. This is in contrast to the previous work which modeled the Console as a Server and the Relay Unit as a Client. This was done so because it would lead to better functioning of the system and also the logic involved in the design of the system made it appropriate. The Console would initiate the request for data to be read from the Relay Unit. The data would include a wide range of parameters like the tank number, status, monitor id etc. The Relay Unit which acts as the Server would accept the request, perform the required function and would respond to the Console. The Console would thereafter receive the response from the Relay Unit.

Chapter 3: MODBUS

The MODBUS protocol [4] provided the platform and standards on which the entire communication was modeled. MODBUS is an application layer messaging protocol, positioned at level 7 of the OSI model, which provides Client/Server communication between devices connected on different types of buses or networks. It also standardizes a specific protocol on the serial line for exchanges of MODBUS requests/responses between a master and one or several slaves. MODBUS has been the industry's serial de facto standard since 1979. MODBUS is a request/reply protocol and offers services specified by function codes. MODBUS function codes are elements of MODBUS request/reply Protocol Data Units (PDUs). MODBUS is currently being implemented in

1. TCP/IP over Ethernet
2. Asynchronous Serial Transmission over a variety of media like EIA/TIA-232, EIA-422, EIA/TIA-485, fiber etc.

The MODBUS protocol defines a simple Protocol Data Unit (PDU). The PDU is independent of the underlying communication layers of the design. However, when mapped onto a network or specific buses, the MODBUS protocol incorporates an additional unit known as the Application Data Unit (ADU). The Figure 3.1 shows a MODBUS frame

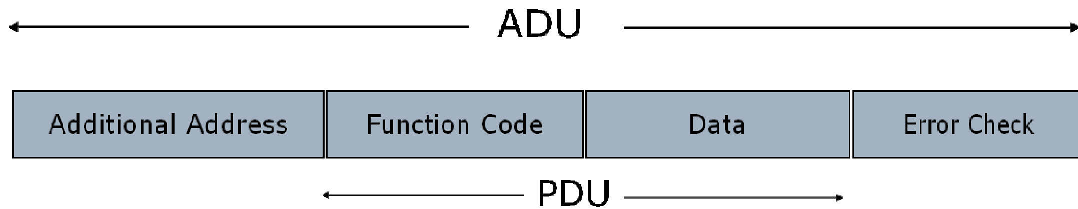


Figure 3.1: MODBUS frame

The entire Application Data Unit is built by the Client which is responsible for initiating a MODBUS transaction with the Server. The initial transaction may include a request or a query to the Server whose format is defined by the MODBUS application protocol. The Additional Address field is filled in by the Client depending on the Server to which it tries to communicate. This Address field is also application dependent. The Function Code field determines what kind of action has to be performed by the Server. This Function Code is coded in bytes and the valid range of codes is from 1 to 255 decimal with '0' being a non-valid code. However, codes in the range 128-255 are reserved by the protocol to handle exception responses. In few cases, sub-function codes are also added to the main Function code in order to handle multiple actions.

The Data field of the Application Data Unit consists of additional information that is intended to be used by the Server for performing the required actions in accordance to the Function Code. The Data field may include various items like the number of bytes to be read/written, the number of byte counts, the addresses of the registers etc. In certain cases, the Server would not require any additional information to perform the requested action defined by the Function Code. In such cases, the Data field of the ADU is non-existent.

The Error Check field of the ADU is dependent on the error checking/correcting algorithm employed by the application. The MODBUS application protocol defines various methods for error checking and correction depending on the media on which it is implemented.

During a MODBUS transaction, the Server, on properly receiving a MODBUS Application Data Unit responds to the Client. This response from the Server will include the data requested by the Client in the Data field of the response ADU, if no error relating to the action specified by the MODBUS function request to the Server occurs. If any error relating to the implementation of the MODBUS function request is encountered, the field will include an exception code which the Server uses to determine the next action that needs to be taken.

The Server uses the Function Code field in its response ADU to indicate to the Client about either an error free implementation i.e. normal response or about an error that occurred due to the action requested. The response by the Server for indicating an error is called an exception response.

The Server simply echoes the Function Code in its response PDU to the Client in the case of a normal response. However, in the case of an exception response the Server returns the Function Code equivalent to that obtained from the request PDU from the Client, with its most significant bit (MSB) set to logic 1.

The two types of basic transaction responses i.e. the normal response and the exception response are shown in the Figure 3.2 and Figure 3.3.

The first MODBUS implementation was done on a Serial Link, which determined the size of the PDU to be 256 bytes. In the RS 232 or RS 485 mode of serial communication, the Server address usually takes one byte and the Cyclic Redundancy Checksum (CRC) which is the error checking method employed, takes two bytes. Hence a MODBUS ADU for a RS 232 or RS 485 mode of communication has 253 bytes of the PDU and a byte of the Server address and 2 bytes of CRC which totals to 256 bytes. However, a TCP MODBUS ADU will have 253 bytes and an additional 7 bytes for the MBAP, totaling to 260 bytes.

The MODBUS protocol defines three different types of PDUs:

1. MODBUS Request PDU = { function_code, data_request}

   Where function_code is a 1 byte MODBUS function code corresponding to the action requested by the Client to the Server. data_request is the bytes containing information such as variable counts, data offsets, sub-function codes etc.

2. MODBUS Response PDU = { function_code, data_response}

   Where function_code is a 1 byte MODBUS function code received by the Server from the Client. data_response is the bytes containing

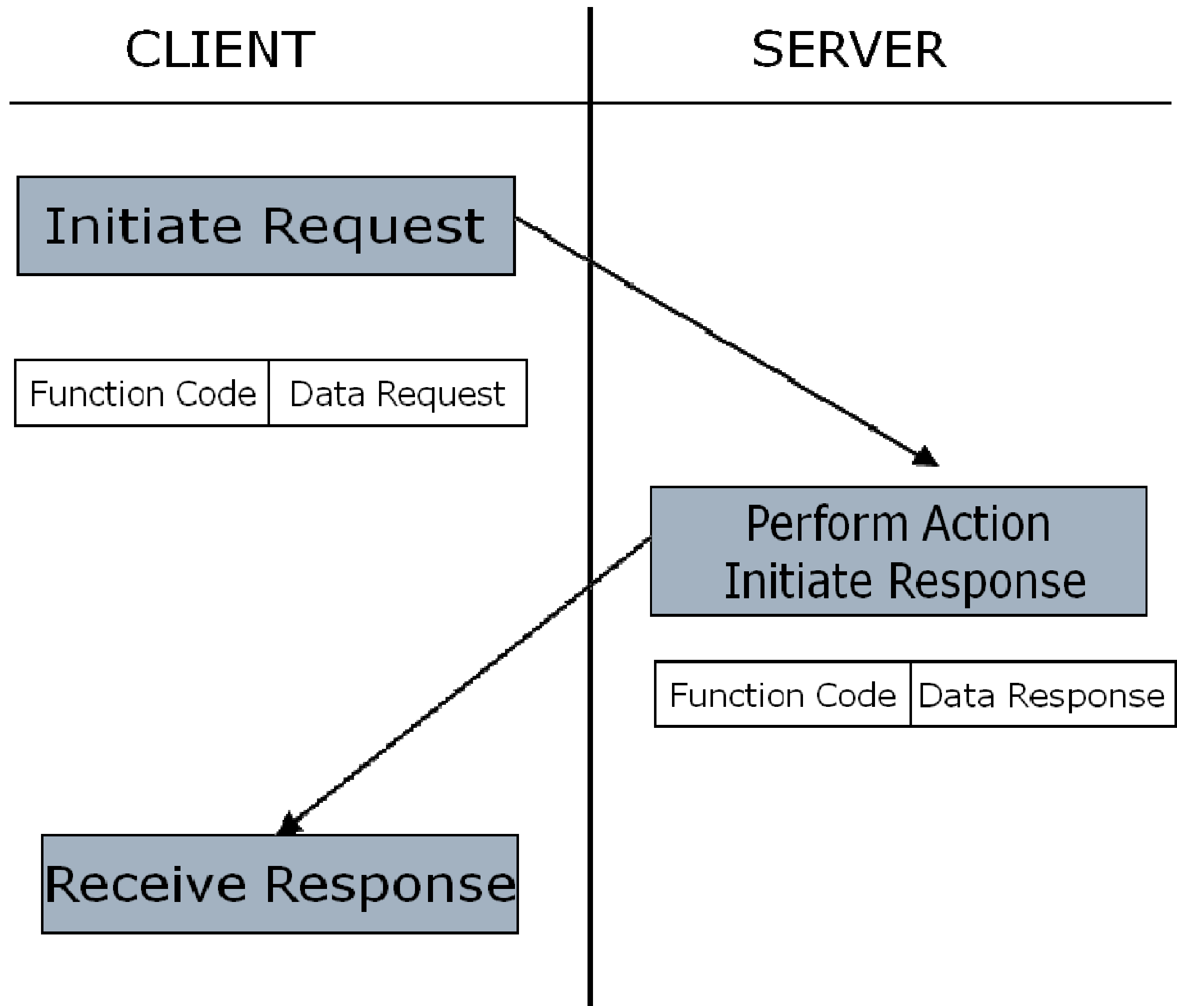


Figure 3.2: Client Server Transaction for a normal response

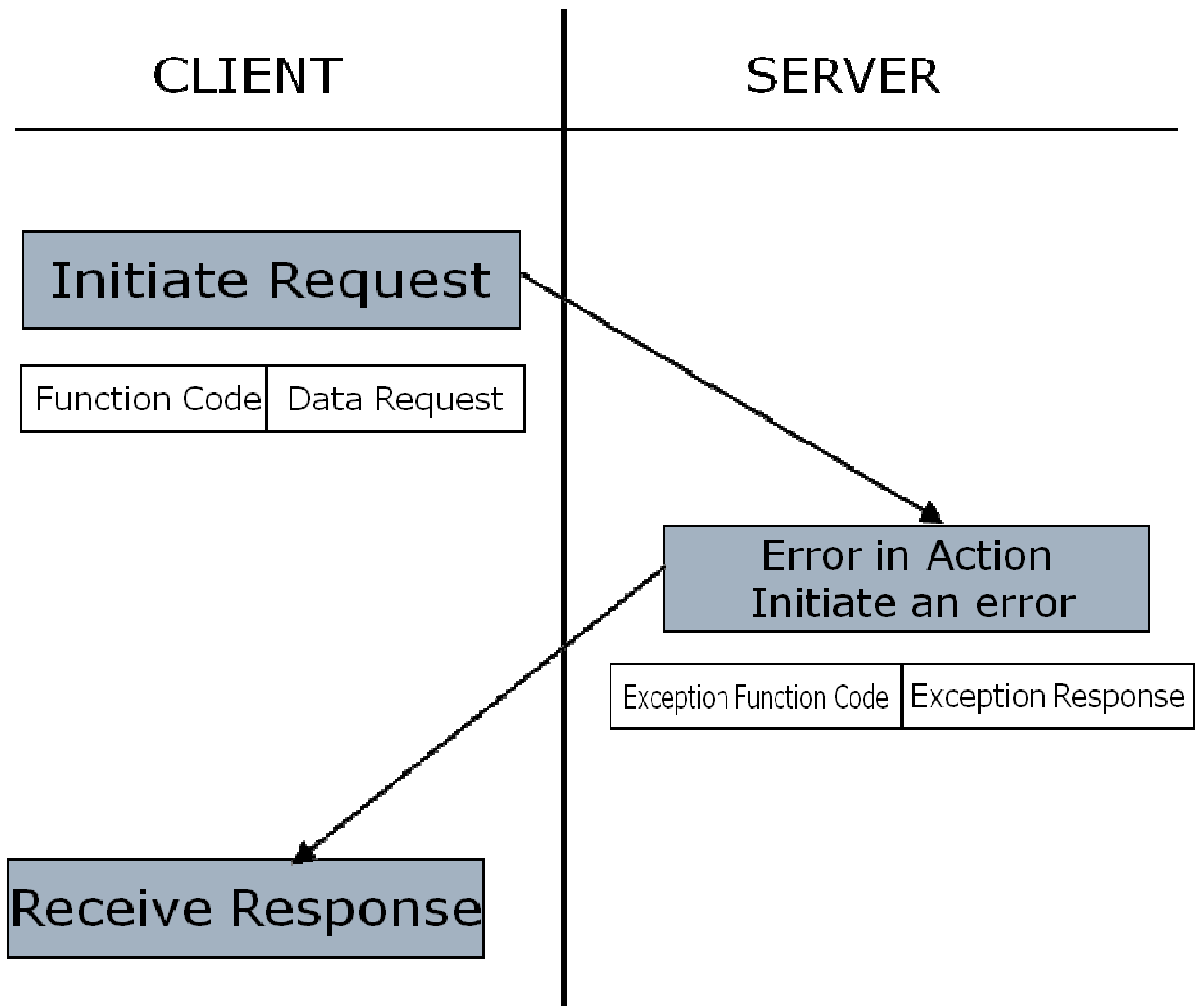


Figure 3.3: Client Server Transaction for an exception response

information such as variable counts, data offsets, sub-function codes etc.

3. MODBUS Exception Response PDU =

{ exception_function_code, exception_code}

Where exception_function_code is the 1 byte MODBUS function code received by the Server from the Client with its most significant bit (MSB) set to logic 1 and exception_code is a 1 byte MODBUS exception code.

The 'big-Endian' representation is used by MODBUS to represent the various addresses and data. Any quantity larger than a single byte to be transmitted has its most significant byte sent first.

MODBUS Function Codes

The MODBUS Function Codes are classified into three different categories:

1. Public Function Codes

   These codes are unique, well defined, approved by the MODBUS-IDA community, publicly documented and have a conformance test. These codes include both well defined, public assigned as well as unassigned codes for future use.

2. User Defined Function Codes

   These codes are available in the range 65 to 72 and 100 to 110 decimal. As the name suggests, these are defined and implemented by users and are not entirely unique. These codes

may not be supported by the specification and for any code and its functionality to be changed into the Public category and to be assigned a new Public Function Code, the pertinent user must initiate a RFC (Reserved Function Code).

3. Reserved Function Codes

   These codes are used by few companies for their exclusive products and are not publicly usable.

MODBUS Over Serial Line

MODBUS standardizes a specific protocol on a Serial line [5] to implement a MODBUS transaction between a Master/Client and a Slave/Server or several Slaves/Servers. A Master-Slave system has one Master node which issues explicit commands to a Slave node or several Slave nodes and processes the responses from the Slaves. The Slave nodes do not transmit any information to the Master unless requested and also do not communicate with other Slave nodes. On a MODBUS serial line the Client role is provided by the Master of the Serial bus and the Slave nodes act as the Servers. Hence, this MODBUS Serial line protocol is referred as a Master-Slave protocol which is placed at level 2 of the OSI model as shown in the Figure 3.4. MODBUS when used on Serial lines uses different interfaces like the RS-232, RS 485 at the physical level with the TIA/EIA-485 (RS 485) 2-wire interface being the most common. A TIA/EIA-232-E (RS 232) serial interface is used when short point to point communication is required.

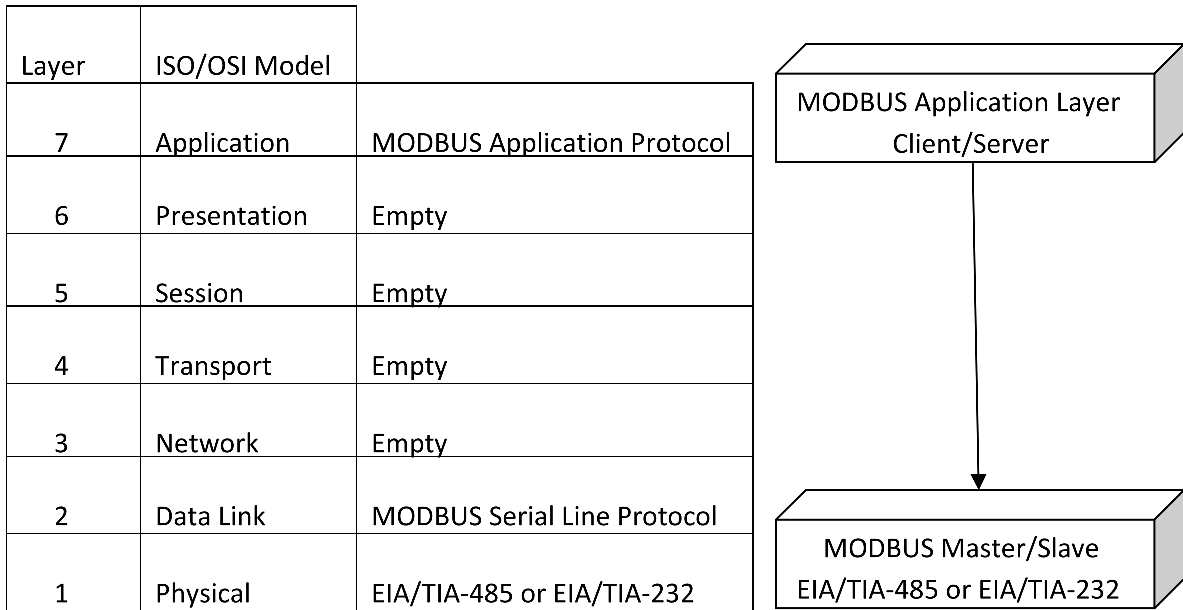


Figure 3.4: MODBUS protocol and the ISO/OSI Model

MODBUS Data Link Layer

The MODBUS data link layer is comprised of two layers:

1. The Master/Slave protocol
2. The Transmission modes-RTU and ASCII

In the MODBUS Serial protocol, only one Master is connected to the serial bus at any point of time and one or more number of Slaves are also connected to the same bus. The maximum number of Slaves that can be connected to a bus at any point of time is specified to be 247. The Master is responsible for initiating a MODBUS transaction and it initiates only one MODBUS transaction at any point of time.

The Master can issue a MODBUS transaction to a Slave in two modes:

1. Unicast mode

   In this mode, the Master addresses only one individual Slave at any point of time. On receiving the request from the Master, the Slave processes the request and issues a reply to the Master. In this mode a MODBUS transaction consists of just two messages viz. the request from the Master and the corresponding response from the Slave. Each of the Slaves connected to the serial bus need to have a distinct address in the range 1 to 247 decimal which facilitates the Master to address a particular Slave independently from the other Slave nodes. Figure 3.5 shows the Unicast mode.

2. Broadcast mode

   In this mode, the Master sends a request to all the Slaves connected onto the serial bus. No response is sent back to the Master from any of the Slaves. Usually the MODBUS message sent by the Master in this mode is more of a command than a request. It includes

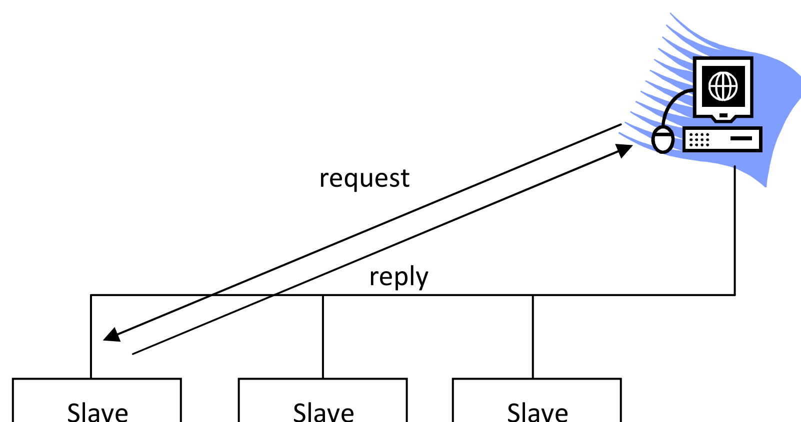


Figure 3.5: Unicast Mode of Operation

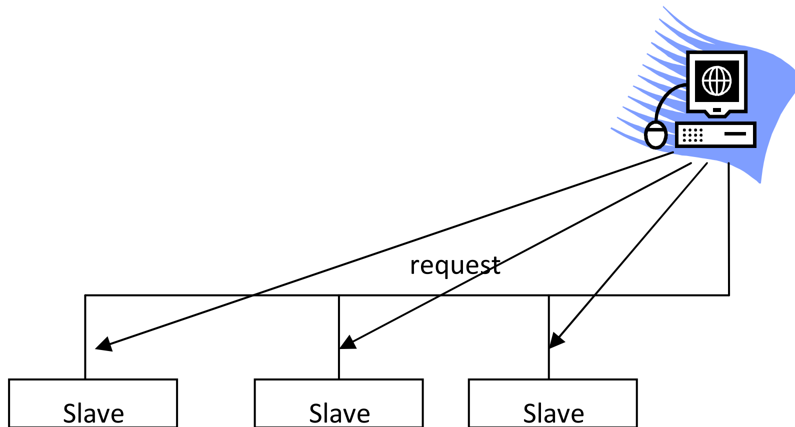


Figure 3.6: Broadcast Mode of Operation

necessary writing functions intended for all the Slaves that need to be accepted and performed by all of the Slaves. A Broadcast MODBUS message is differentiated by having the address bit set to 0. Figure 3.6 shows the Broadcast mode.

Master/Slave State Diagrams

The Master State Diagram is shown in the Figure 3.7. When the Master is powered up initially, it goes into the 'Idle' state which is also used for denoting that there are no requests pending. As and when the Master sends a request, it leaves the 'Idle' state and cannot send a second request at the same time.

In the Unicast mode, the Master goes into the 'Waiting for Reply' state once it sends a request to a Slave. Simultaneously the 'Response Time-out' is started. The Response Time-out is entirely application dependent and is used to prevent the Master from staying indefinitely in the 'Waiting for Reply' state.

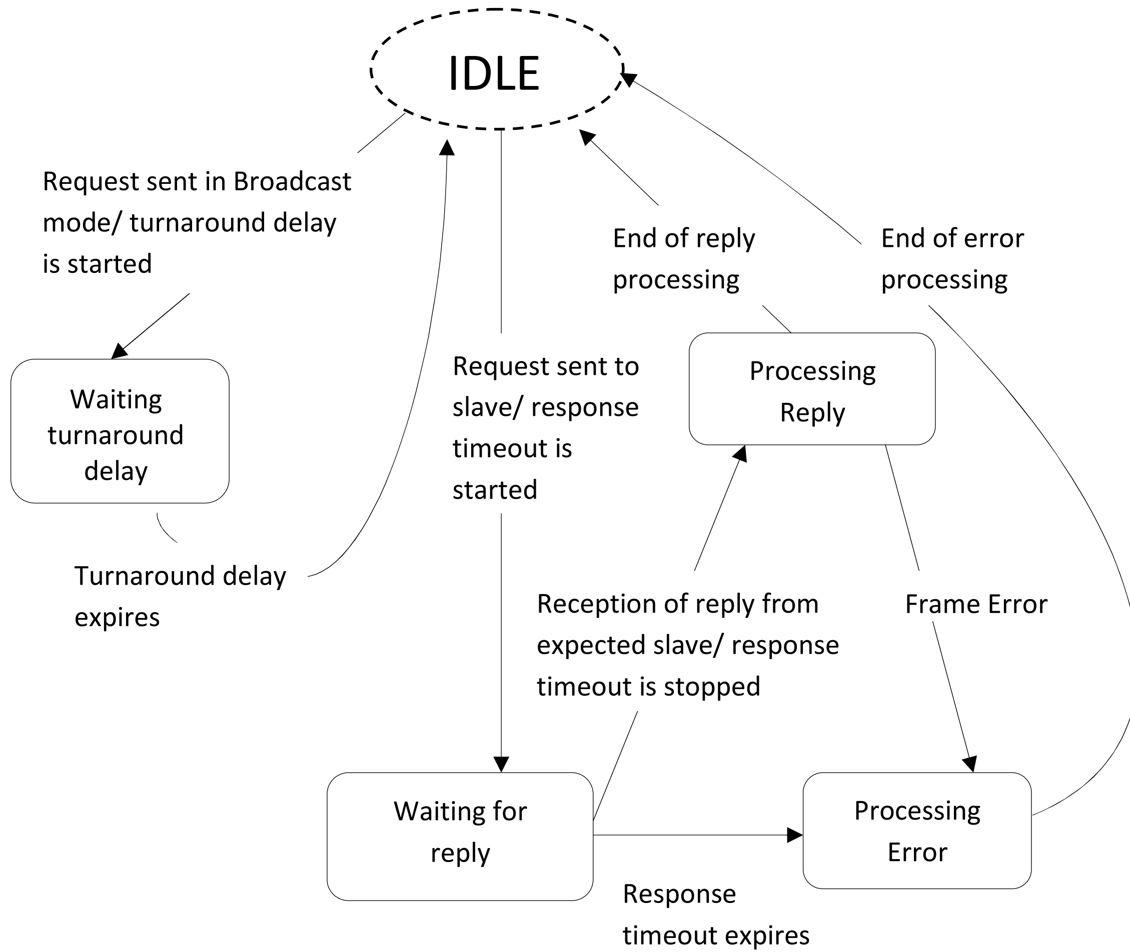


Figure 3.7: Master State Diagram

When the Master receives a reply, it checks the reply for any errors in transmission or for errors in the frame. There may also arise a case wherein the reply is received from a Slave to which the initial request was not intended for. In such cases the reply is rejected and the Response Time-out is kept running. However if an error is detected in a frame from the intended Slave, then a ‘Retry’ may be initiated. When the Master does not receive a reply after a time stipulated by the ‘Response Time-out’, an error is issued and the Master goes back into the ‘Idle’ state enabling it send requests to new Slaves or also to send a retry. The permissible number of retries is dependent on the Master set-up.

The ‘Waiting Turnaround Delay’ state comes into prominence when the Master operates in the Broadcast mode. When the Master issues a Broadcast message to all the Slaves, it delays its next request for some amount of time. This delay is called the ‘Turnaround Delay’. This delay facilitates the Slaves to process the Broadcast request sent by the Master. The Master stays in the ‘Waiting Turnaround Delay’ state for the amount of time stipulated by the ‘Turnaround Delay’ and goes back to the ‘Idle’ state in order to issue new requests.

Logic would determine that the ‘Turnaround Delay’ time should be shorter than the ‘Response Time-out’. This is because the Slave would need time to process the request and respond to the Master in the Unicast mode. However, in the Broadcast mode, the Slave would just have to process the request from the Master without having to respond to it.

The Slave State diagram is shown in the Figure 3.8. When the Slave is initially powered up, it goes into the ‘Idle’ state which also is used to indicate that there are no requests from the Master pending for processing. When the Slave leaves this state, it would thereafter be able to receive any new requests from the Master.

When the Slave receives a request from the Master, irrespective of its mode of operation, it checks the received packet for various kinds of errors before performing the action requested. The errors may include framing errors, errors in frame checking or the frame being not addressed to that particular slave. In such cases, the Slave does not respond to the Master and goes back into the ‘Idle’ state and waits for further requests.

When no errors occur in checking the packet, the Slave goes ahead and processes the request. Prior to this it also checks if there is any error in the data requested by the Master. If such an error occurs the Slave responds to the Master with an error reply, indicating the error type using an exception code. When there is no error in the data requested, the Slave performs the requested action determined by the Function Code.

When performing the required action, if the Slave encounters any problem or error in the processing, it formats an error reply and sends it to the Master, with the exception code indicating to the Master the kind of error that occurred while processing. Once the Slave successfully processes the required

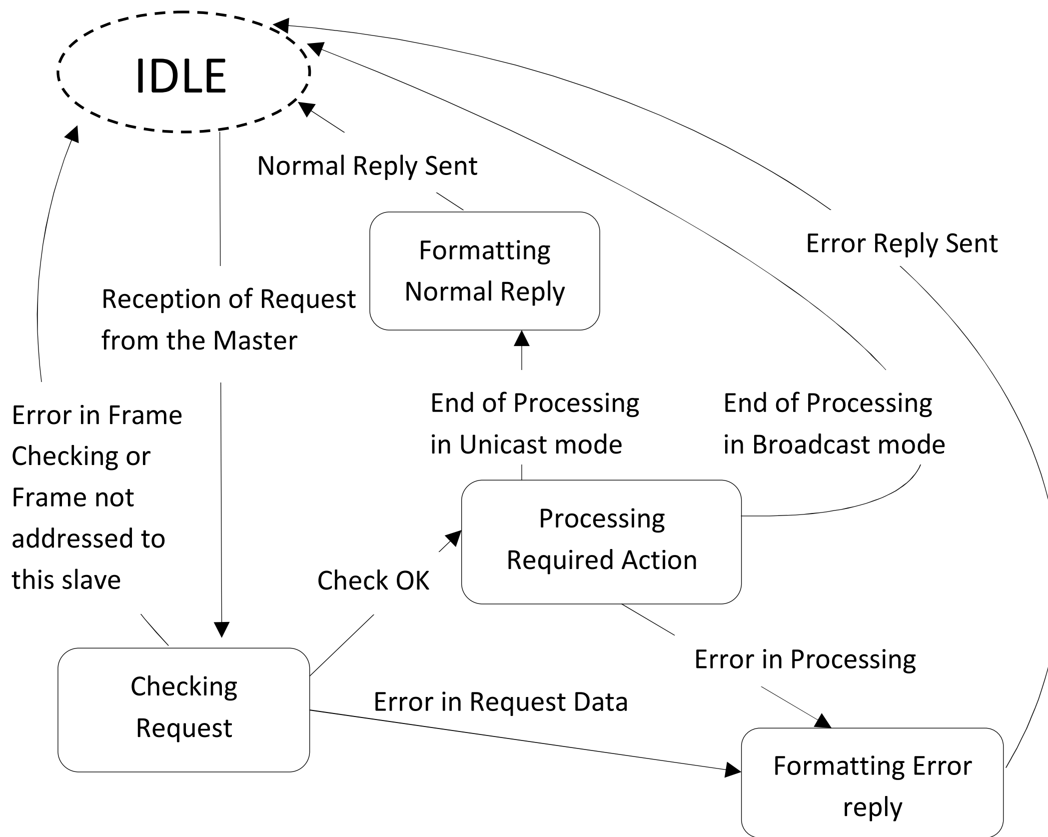


Figure 3.8: Slave State Diagram

there are no requests from the Master pending for processing. When the Slave leaves this state, it would thereafter be able to receive any new requests from the Master.

When the Slave receives a request from the Master, irrespective of its mode of operation, it checks the received packet for various kinds of errors before performing the action requested. The errors may include framing errors, errors in frame checking or the frame being not addressed to that particular slave. In such cases, the Slave does not respond to the Master and goes back into the 'Idle' state and waits for further requests.

When no errors occur in checking the packet, the Slave goes ahead and processes the request. Prior to this it also checks if there is any error in the data requested by the Master. If such an error occurs the Slave responds to the Master with an error reply, indicating the error type using an exception code. When there is no error in the data requested, the Slave performs the requested action determined by the Function Code.

When performing the required action, if the Slave encounters any problem or error in the processing, it formats an error reply and sends it to the Master, with the exception code indicating to the Master the kind of error that occurred while processing. Once the Slave successfully processes the required action, it responds to Master with the required data. In the Unicast mode it is mandatory for the Slave to respond to the Master once it completes performing

the requested action. In the Broadcast mode, the Slave just goes back to the 'Idle' state after performing the mandatory action requested by the Master.

MODBUS Addressing

MODBUS allows a space of 256 bytes for addressing. The Figure 3.9 below shows the addressing model. The MODBUS Master node does not have a specific address, however, the Slave nodes have specific addresses which are distinct and unique on a serial bus. MODBUS provides the address 0 for a Broadcast exchange and this must be recognized by every Slave node. It also provides Slave addresses from 1 to 247 with addresses from 248 to 255 being reserved.

MODBUS Frame

The mapping of the MODBUS protocol on a serial bus introduces few additional fields on the Protocol Data Unit (PDU). The Master when initiating a MODBUS transaction or a Slave when responding to the Master, builds the basic MODBUS PDU and then adds fields in order to build the communication PDU. This communication PDU is called the MODBUS Serial Line PDU as shown in the Figure 3.10. The Address Field of the Serial Line PDU contains only the Slave addresses in the range 0 to 247 decimal with 0 being assigned for the Broadcast mode and 1 to 247 being assigned for the individual Slave addresses.

The Master addresses a particular Slave by placing the Slave's address in the Address Field of a message. The Slave responds back with a reply to the

0	From 1 to 247	From 248 to 255
Broadcast Address	Slave Individual Addresses	Reserved

Figure 3.9: MODBUS Addressing Model

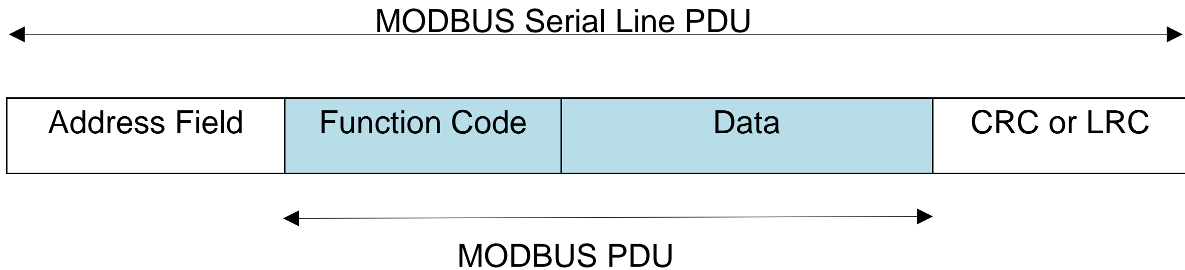


Figure 3.10: MODBUS Serial Line PDU

Master by placing its own address in the Address Field thereby enabling the Master to know which particular Slave is responding.

The Function Code of the Serial Line PDU indicates to the Slave the kind of action it needs to perform. The Slave echoes the same Function Code in its reply to the Master if the requested action is performed without any error. In case of an error it sends the same Function Code with its most significant position (MSB) bit set to logic 1.

The Data field of the Serial Line PDU contains parameters that are a part of the requests or responses. In case of a normal action performed by the Slave, the data requested by the Master is placed in this field. In case, an error occurs in the course of taking the requested action, the Slave sends an exception code in the Data field of the Serial Line PDU.

The Error Checking Field contains the result of the Error Checking algorithm employed on the entire message contents. The Error Checking algorithm may be the Cyclic Redundancy Check algorithm or the Longitudinal Redundancy Check algorithm. However, the type of algorithm employed depends on the type of transmission mode used viz. RTU or ASCII.

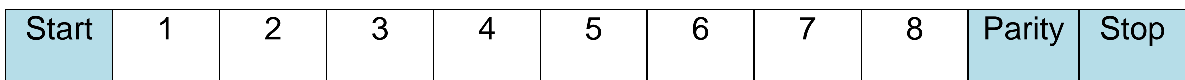
Transmission Modes

MODBUS defines two serial modes of transmission modes, the RTU mode and the ASCII mode. The modes define the bit contents of the individual message fields which are transmitted on the Serial bus, the method in which the information is embedded into the message and the coding and decoding techniques employed on the information. All the devices on the Serial bus i.e. the Master and the Slaves should transmit in the same mode with the serial parameters also being the same one. MODBUS requires the RTU mode to be implemented on all devices on the Serial bus as a mandatory mode even though few applications require communication in ASCII mode. The ASCII mode is an option which can be used by the users for some specific applications, however the RTU mode has to be set as the default. At any baud rate the RTU mode has a greater character density thereby allowing a better data throughput when compared to the ASCII mode.

RTU Transmission Mode

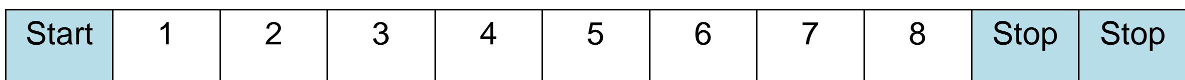
In the Remote Terminal Unit (RTU) mode, each 8-bit byte of a message is coded as two 4-bit hexadecimal characters with each message being transmitted as a continuous stream of characters. An 8-bit byte of a message is coded into an 11-bit byte in this mode. The 11-bit byte includes a start bit, the 8-bit message byte, a parity bit for completion and a stop bit. The Figure 3.11 shows the bit sequence in the RTU mode. Each 11-bit byte or character is transmitted from the left to the right side i.e. the Least Significant Bit (LSB) is sent first and the Most Significant Bit (MSB) is sent last. The RTU mode requires EVEN parity to be employed as a mandatory and default option. However, the ODD and NO parity can also be employed. This ensures more compatibility with the devices.

In case of NO parity being employed, there would be 2 stop bits in the 11-bit byte. The Figure 3.12 shows the bit sequence in the case of NO parity being employed.



Start	1	2	3	4	5	6	7	8	Parity	Stop

Figure 3.11: Bit sequence in RTU mode



Start	1	2	3	4	5	6	7	8	Stop	Stop

Figure 3.12: Bit Sequence in RTU mode for NO parity

The RTU MODBUS frame used is shown below in the Figure 3.13. The maximum size of the frame is 256 bytes. The RTU mode employs Cyclic Redundancy Check (CRC) method as the error checking method. The CRC checks the contents of the message irrespective of the type of parity used. The CRC is performed on the individual characters of the message with the resulting checksum field containing 16 bits which are implemented as two 8 bit bytes viz. the lower-order byte and the higher-order byte. These 2 bytes are appended at the end of a message with the lower-order byte appended first followed by the higher-order byte.

The device transmitting the MODBUS message calculates the CRC bytes and appends it to the end of the message making it the last field to be transmitted. The device which receives the message recalculates the CRC and compares it with the two CRC bytes that were sent in the message. If there is any difference between the values, an error occurs indicating that few bits were transmitted incorrectly.

Every 8-bit character of the message is applied to a 16-bit register which is preloaded with 1's. The entire contents of the message are applied to the

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 to 252 bytes	2 bytes

Figure 3.13: RTU MODBUS Frame

register. However, the process of calculation of the checksum does not take into account the Start, Stop and Parity bits of a character in the message. Only the 8-bits of the individual character bit sequence are used in this process.

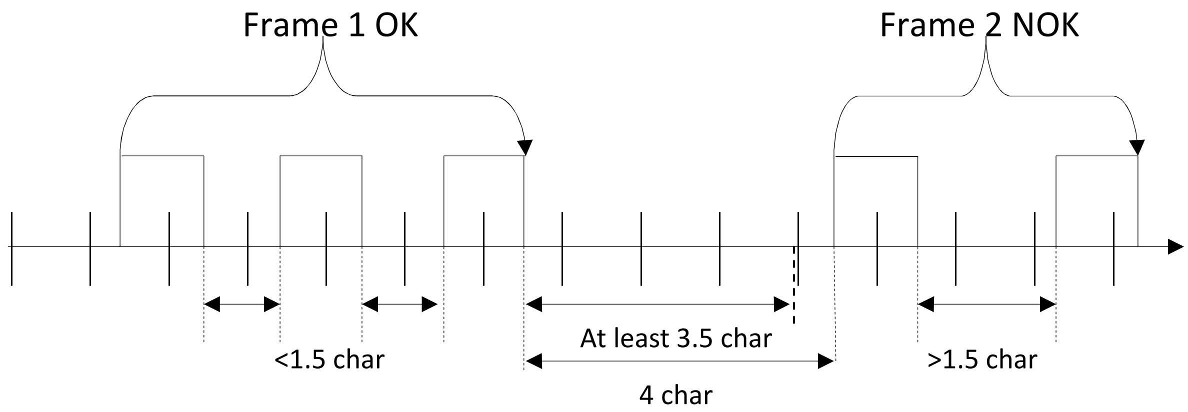
MODBUS Messaging using RTU Frame

MODBUS messaging in an RTU frame enables the devices receiving the message to be informed about the beginning and ending of a message. It facilitates the device in reception to start receiving a message from the beginning and to stop receiving promptly when the tail end of the message is received. Any message frame that is received incompletely is discarded and an error is reported.

MODBUS stipulates a time interval of at least 3.5 character times (known as t3.5) between two frames transmitted in the RTU mode. However, it also stipulates that the entire message frame should be transmitted as a continuous stream of characters. If two successive characters in a frame are separated by an interval of more than 1.5 character times (known as t1.5), the message is known to be received incompletely and the receiver discards it. The Figure 3.14 shows the message framing in RTU mode and the timing constraints.

State Diagram of RTU Transmission Mode

The state diagram shown in the Figure 3.15 gives an overview with respect to both the Master and the Slave simultaneously. For any device to move from the ‘Initial’ state to the ‘Idle’ state, the t3.5 timeout must expire. This ensures



Timing between Successive Characters and Frames

MODBUS Message

Start	Slave Address	Function Code	Data	CRC	End
≥ 3.5char	8 bits	8 bits	N × 8 bits	16 bits	≥ 3.5 char

Figure 3.14: Message Framing in RTU mode and Timing constraints

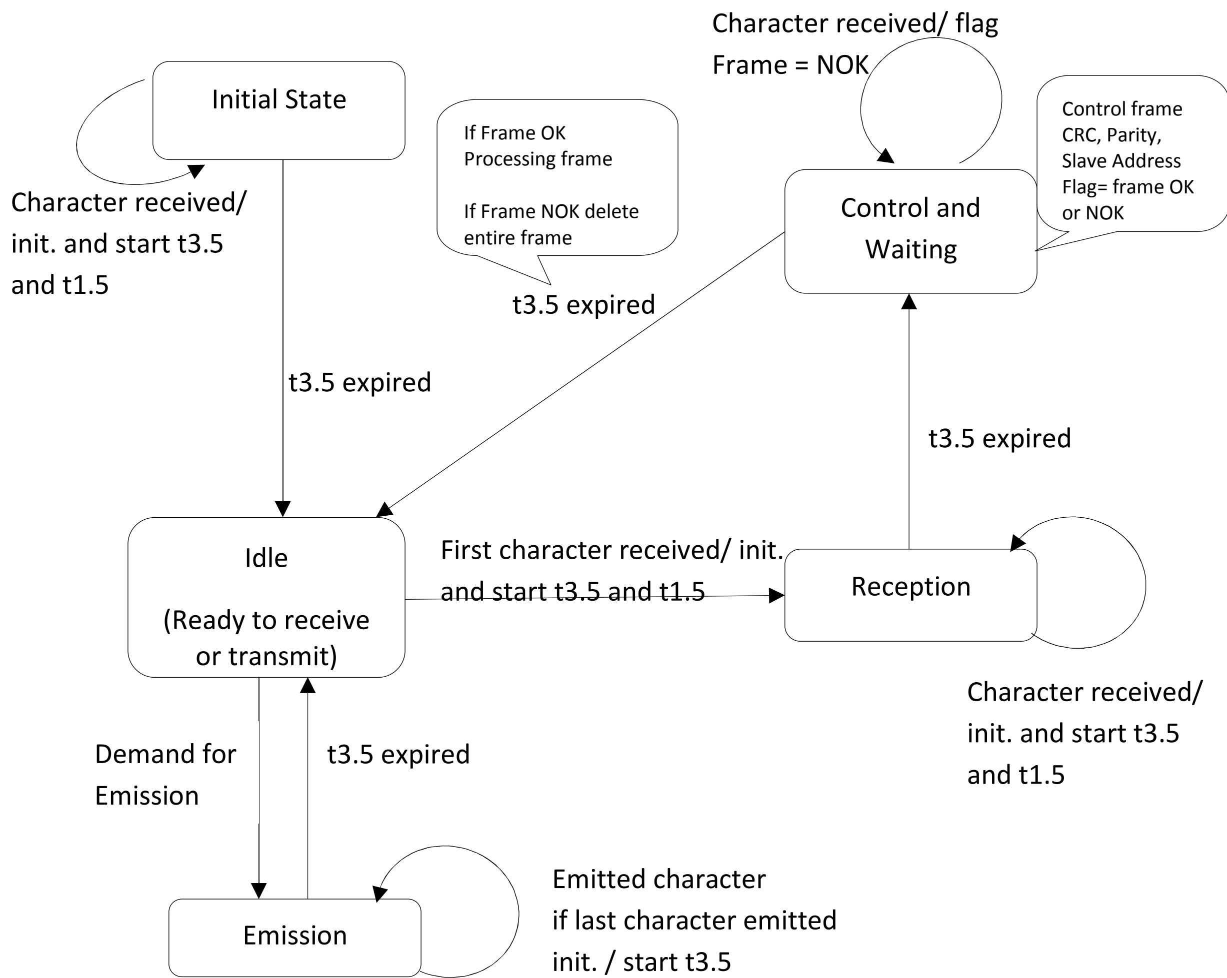


Figure 3.15: State Diagram of RTU Transmission Mode

that no two consecutive frames are overlapped. When the device is in the 'Idle' state, it is ready to transmit or receive. The communication link is itself declared as 'Idle' when there is not activity for more than 3.5 character times. When the communication link is in the 'Idle' state, any character that is transmitted on the link is identified as the starting character of a message frame. The link becomes active from now onwards. The device on the link which transmits the character now moves into the 'Emission' state. It transmits the characters of the entire message frame and when it transmits the last character, it moves into the 'Initial' state and the t3.5 timeout is started. It goes back to the 'Idle' state when the t3.5 timer expires.

The device which receives the first character goes into the 'Reception' state form the 'Idle' state. As soon as the first character is received, the t3.5 and the t1.5 timeouts are started. When no character is received even after the t1.5 timer expires, the receiving device goes into the 'Control and Waiting' state. In this state the receiving device checks the frame characters which would had been received until the t1.5 timer expired. The device checks for the Slave address, Parity and CRC. The frame is processed if there are no errors in any of those parameters, else the entire frame is deleted. All through this process starting from the reception of the first character of the frame to the checking of the parameters, the t3.5 timer is kept running. The end of the frame is identified when the t3.5 timer expires. As and when the t3.5 timer expires the receiving device goes back into the 'Idle' state enabling it to further receive or transmit a

frame. In order to reduce the reception processing time, the device first checks the address field of the frame to determine if the frame was intended to it. Only after confirming that the frame was indeed addressed to it, the device calculates the CRC and checks the parity.

ASCII Transmission Mode

In the American Standard Code for Information Interchange (ASCII) mode, each 8-bit byte is coded as two 4-bit ASCII characters. This mode finds it use in applications where the communication link would not able to handle the timer constraints of the RTU mode. However, the ASCII mode is less efficient when compared to the RTU mode, because each 8-bit byte needs two characters. The coding system includes the hexadecimal characters and the ASCII characters 0-9, A-F with each hexadecimal character containing 4 bits of data within each ASCII character that is part of the message.

Each 7-bit data of the message is encoded as a 10-bit byte in this mode. The 10-bit byte comprises of the Start bit, the 7-bit data, the Parity bit and the Stop bit. Figure 3.16 shows the 10-bit sequence of the ASCII mode.

Each 10-bit byte or character is transmitted from the left to the right side i.e. the Least Significant Bit (LSB) is sent first and the Most Significant Bit (MSB) is sent last. The ASCII mode requires EVEN parity to be employed as a mandatory and default option. However, ODD and NO parity can also be employed. This ensures more compatibility with the devices. In case of NO parity

being employed, there would be 2 stop bits in the 10-bit byte. The Figure 3.17 shows the bit sequence in the case of NO parity being employed.

The ASCII MODBUS frame is shown in the Figure 3.18. Each data byte in the ASCII mode requires two characters and hence the data field could be made up of a maximum of 2x252 characters. This ensures compatibility between the ASCII and the RTU mode at the MODBUS application level and hence maximum size of the ASCII frame is 513 characters.

The ASCII mode employs Longitudinal Redundancy Check (LRC) method as the error checking method. The LRC checks the contents of the message except the Start character ‘colon’ and the end characters ‘CRLF’. LRC checks the contents irrespective of the type of parity used. The LRC is performed on the individual characters of the message with its field containing 8 bits. At the end of the calculation process of LRC, the resulting two ASCII bytes are appended to the message frame after the data characters, prior to appending the ending characters CRLF.

Start	1	2	3	4	5	6	7	Parity	Stop

Figure 3.16: Bit Sequence in ASCII mode

Start	1	2	3	4	5	6	7	Stop	Stop

Figure 3.17: Bit Sequence in ASCII mode for NO parity

Start	Address	Function	Data	LRC	End
1 char	2 chars	2 chars	0 to 2 × 252 chars	2 chars	2 chars

Figure 3.18: ASCII MODBUS Frame

The device transmitting the MODBUS message calculates the LRC bytes and appends it to the message. The device which receives the message recalculates the LRC and compares it with the two LRC characters that were sent in the message. If there is any difference between the values, an error occurs indicating that few bits were transmitted incorrectly.

LRC calculation involves adding successive 8-bit bytes of the message and then two's complementing the result. The carries resulting from the addition process are discarded prior to two's complementing the result. However, the process of calculation of the LRC does not take into account the Start character 'colon' and the ending 'CRLF' characters in the message frame.

MODBUS Messaging using ASCII Frame

MODBUS messaging in an ASCII frame facilitates the devices in reception to be informed about the beginning and ending of a message. It enables the device in reception to start receiving a message from the beginning and to know when the end of the message occurs. Any message frame that is received partially is discarded and an error is reported. Refer Figure 3.18 for the MODBUS ASCII frame.

A MODBUS ASCII frame is delimited by specific characters known as the Start of Frame and End of Frame characters. Every message frame starts with a 'colon (:)' i.e. ASCII 3A hexadecimal as the Start of Frame character and ends with the two characters 'CRLF' i.e. ASCII 0D & 0A hexadecimal as the End of Frame characters. 'CRLF' starts for 'Carriage Return Line Feed'. Every device present on the Serial bus looks for the 'colon' character and as soon it is detected, the devices start receiving characters until the 'CRLF' characters are detected indicating the end of the frame.

MODBUS allocates a space of two characters for the address field. It stipulates a time interval of one second between two successive characters in a message. However, few applications might need a higher time interval, in which case the user can set a timeout which suits the application.

State Diagram of ASCII Transmission Mode

The state diagram shown in the Figure 3.19 gives an overview with respect to both the Master and the Slave simultaneously. Initially the devices are in the 'Idle' state wherein there is no transmission or reception taking place. As and when the need to transmit a message arises, the device moves into 'Transmit Start' state. It starts the transmission by first sending the Start of Frame character 'colon (:)' after which it goes into the 'Transmission' state. In this state the device transmits the remaining message until it encounters the LRC characters. The device after appending the LRC characters to the message and sending them, starts transmitting the 'Carriage Return (CR)' character. During

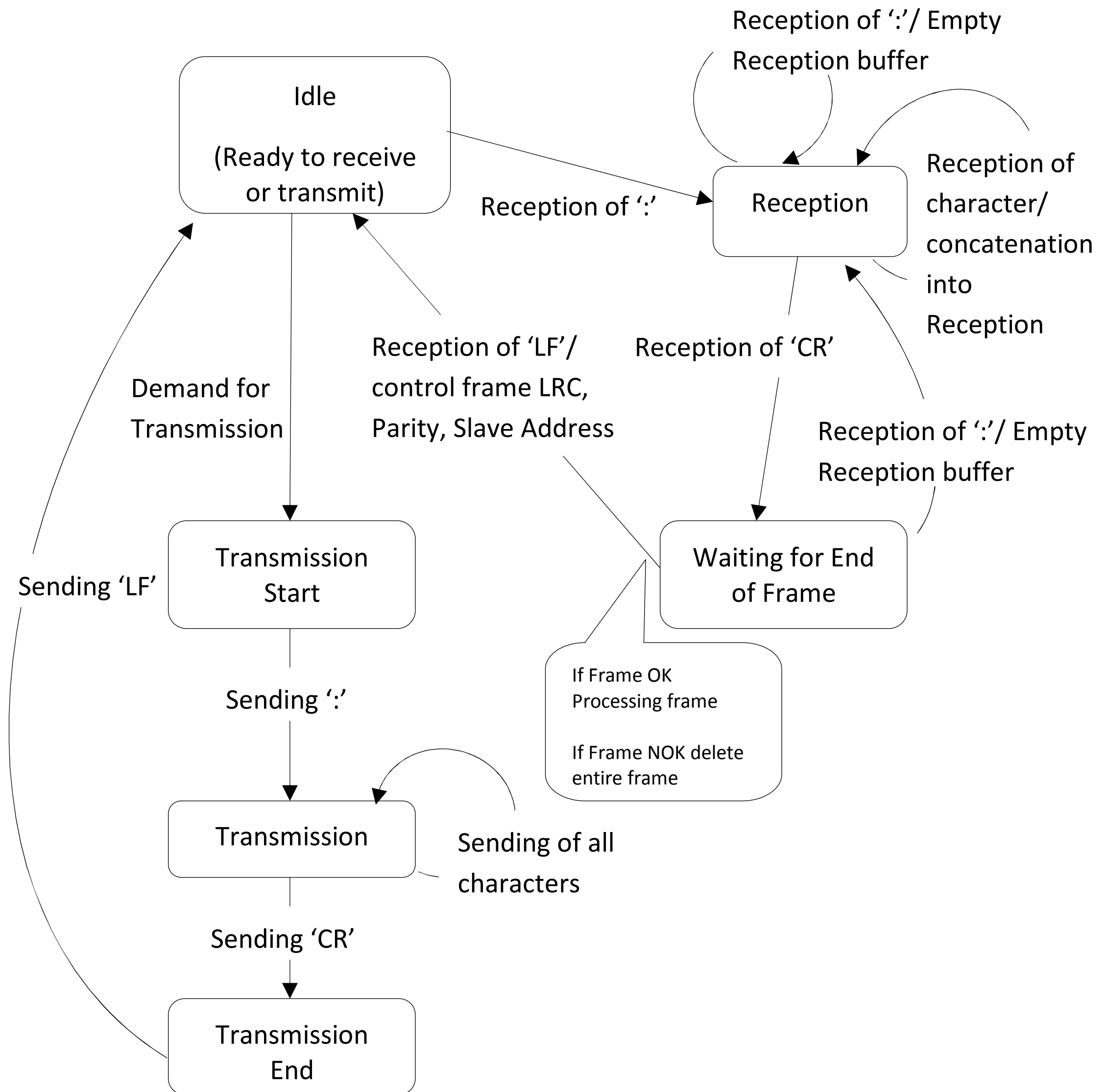


Figure 3.19: State Diagram of ASCII Transmission Mode

this process the device moves into the ‘Transmission End’ state and starts transmitting the last character ‘Line Feed (LF)’. Once the device sends the ‘LF’ character, it goes back to the ‘Idle’ state enabling it to further transmit or receive a new message.

The device which is intended to receive looks for the Start of Frame character ‘colon (:)’ on the communication link. As and when it detects the ‘colon (:)’ on the link, it starts its reception and in this process moves from the ‘Idle’ state to the ‘Reception’ state. In this state, the device empties the reception buffer on receiving the ‘colon (:)’ character. The device thereafter starts receiving the remaining characters of the message frame and in doing so concatenates the received characters in the reception buffer. The device remains in the ‘Reception’ state until it encounters the End of Frame character ‘CR’. During the process of receiving the ‘CR’ character the device moves from the ‘Reception’ state to the ‘Waiting for End of Frame’ state. In this state the device checks the received frame for the Slave address, Parity and LRC characters after it receives the other End of Frame character ‘LF’. Once the ‘LF’ character is received the device checks the above mentioned parameters and in case of no errors, the device goes ahead and processes the frame. If any error occurs in the parameters, the frame is discarded by the device. In order to reduce the reception processing time, the device first checks the address field of the frame to determine if the frame was addressed to it or not. Only after confirming that the frame was indeed addressed to it, the device calculates the LRC.

The device after processing the frame or deleting the frame goes back into the ‘Idle’ state from the ‘Waiting for End of Frame’ state which thereby enables it to further receive or transmit a new message. However, if the device receives the Start of Frame character ‘colon (:)’ when it is still in the ‘Waiting for End of Frame’ state, it goes back to the ‘Reception’ state. The device meanwhile empties the reception buffer during this process in order to receive the new characters of an incoming message.

Chapter 4: EXPERIMENTS & RESULTS

Previous Tasks Accomplished

The firmware established previously had its limitations and errors in implementation. Figure 4.1 shows the action flow based on which the Console was designed previously. The Console was configured as a Client and it had to transmit a MODBUS read packet. It had to keep transmitting the read packet until it received any response. On receiving a response it had to process the data and store the data until it received the entire packet. After that it had to check the CRC and if it was correct, the Console had to indicate that a correct packet was received. A receive time-out also had to be configured which would set a time limit for the packet to be received. When the receive time-out expired without the entire packet being received, or when the CRC was incorrect, the read packet had to be sent a second time. When a time-out occurred while the device was still being in reception of the response for the second time, the packet had to be dropped.

The Console had errors in its design and implementation in the previous work. The Console kept transmitting the MODBUS packet until it received a response for the packet previously sent. It would get stuck in a while loop in the software. Also, the Console never knew when the end of packet arrived. Errors in the implementation included the inability of the Console to transmit a MODBUS write packet. Also, the Console was able to receive only four bytes of data, not

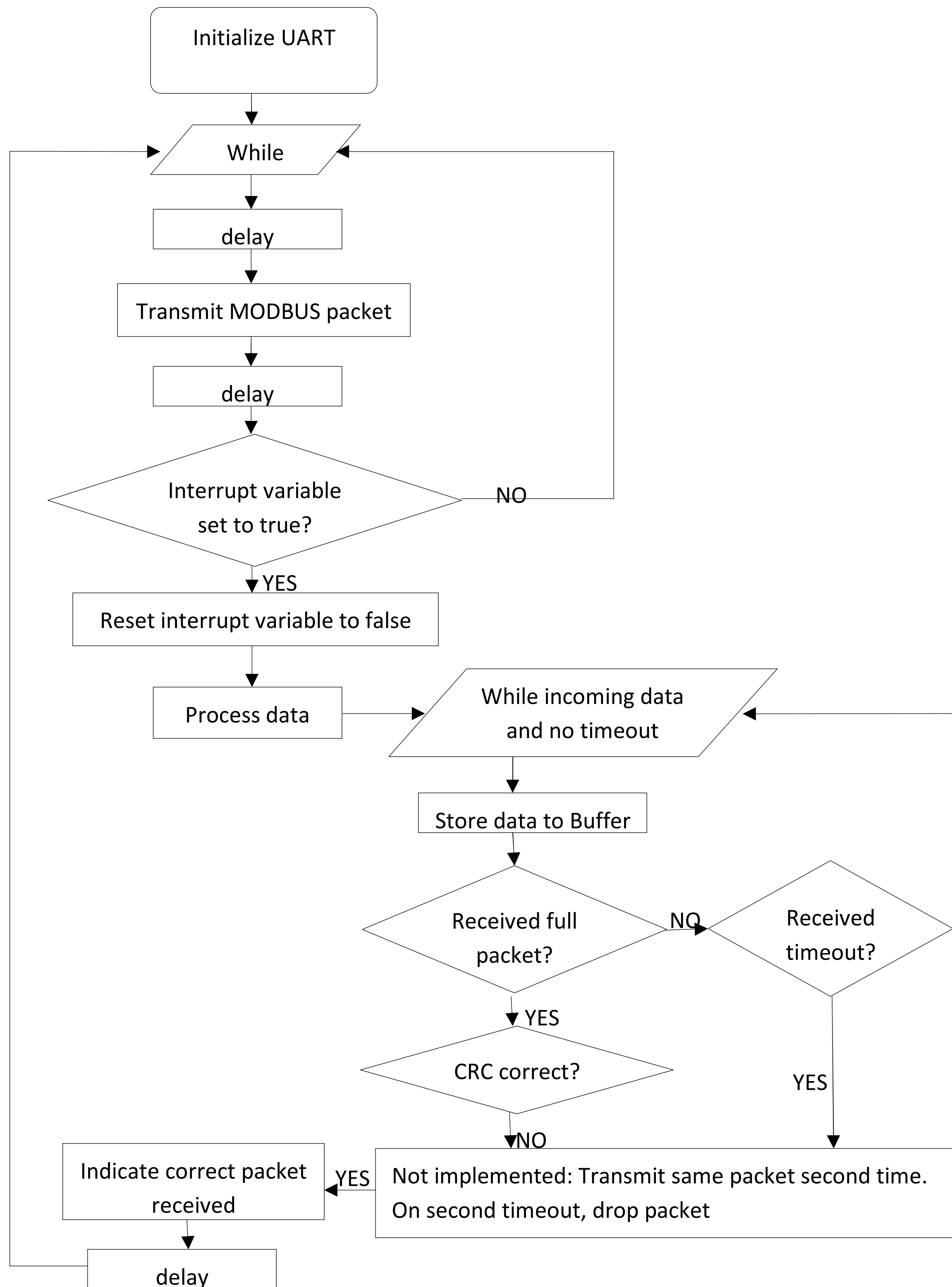


Figure 4.1: Previous Console Action Flow

an entire packet at a time. The Console had problems in implementing the CRC on any of the data. The Console would also reset after some random amount of time which was probably because of no timer being set for receiving the data from the Relay Unit. The receive timer not being implemented lead the Console to stay in the receiving state until any data was detected. This prevented the Console to send further packets of data to any other device until it received a response from the Relay Unit. The implementation did not include the error packets in its design of packets and the memory mapping of the Tank Console and Tank structure was also not performed.

Figure 4.2 shows the action flow based on which the Relay Unit was designed previously. The Relay Unit was configured as a Master and had to receive the MODBUS read packet from the Console. When a receive interrupt variable was set indicating the detection of data, the receiving process had to be started. The incoming data was stored to a buffer until an entire packet was received. On completely receiving the packet, the CRC had to be checked. If the CRC was correct, a MODBUS response packet had to be transmitted to the Console and the Relay Unit had to go back to the state in which it waited for reception of further messages. If the CRC was incorrect, an error packet had to be sent. A receive time-out had to be configured which would set a time limit for the packet to be received. When the receive time-out was set without the entire packet being received, the packet had to be discarded and an error packet had to be sent.

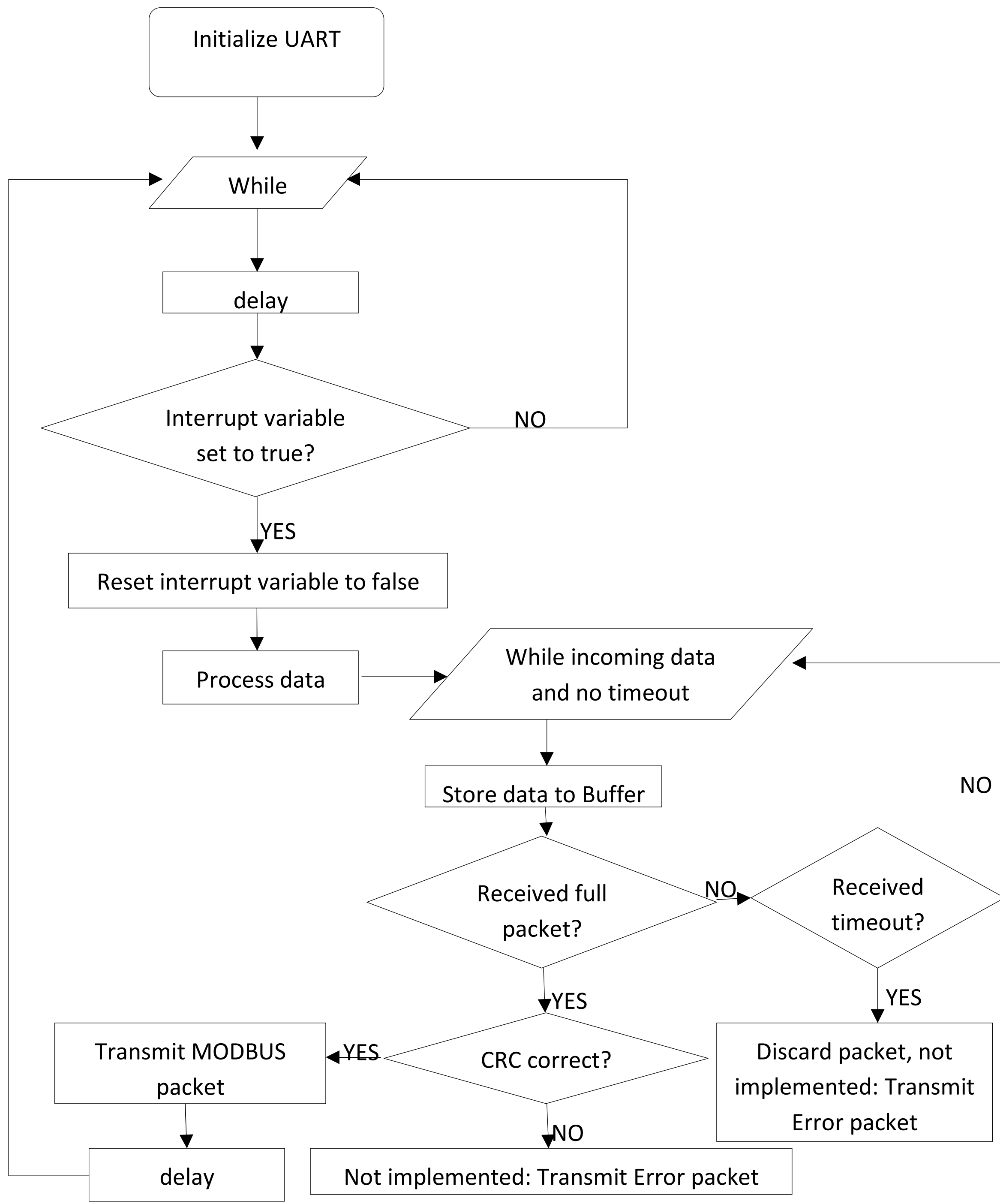


Figure 4.2: Previous Relay Unit Action Flow

The Relay Unit had errors in its design and implementation in the previous work. It would receive the MODBUS read packet, but failed to send any response packet. It was not tested for receiving a MODBUS write packet from the Console. Also, it had issues with the timer for receiving not functioning properly. The error packets were not implemented and the memory mapping also was not implemented.

Procedure & Experimental Results:

The firmware established previously was studied for getting an overview of the design and implementation. There seemed to be too many errors in the implementation and also a few design issues had to be addressed. The entire design and logic was redone by learning from the fallacies of the previous work. The software programming of the boards was done in C/C++ [6], [7]. The IAR Embedded workbench tool was used for compiling and debugging operations of the code [8].

The Console and the Relay Unit hardware included the AVR Atmega-163 processors along with 512 bytes of memory [9]. A new programmer AVR ISP mkII was used for programming the boards. This new programmer was not compatible with the boards and had its hardware fixed and programmed for proper functioning [10]. The AVR Studio [11] was used for programming the boards using the programmer. In order to accomplish the final objective of having the MODBUS read and write packets being transmitted and their responses

being exchanged between the Console, Relay Unit and the Personal Computer, a step by step approach was taken up. These steps included

1. Setting up interrupts at a one second and one minute interval. This would enable the Console to switch between communications involving the Serial port (used for communicating with the Relay Unit or the Personal Computer) and the Modem (used for communicating with the Switch Monitors).
2. Transmitting a character from the boards to a Hyper-terminal [12] on a desktop computer.
3. Having a character transmitted and received between the two boards.
4. Transmitting a stream of characters and thereby a packet from the boards to the Hyper-terminal.
5. Transmitting and receiving a packet between the two boards.
6. Transmitting a MODBUS read/write packet from the boards to the Hyper-terminal.
7. Finally having the MODBUS read/write packet transmitted and received between the two boards.

The boards have three different LEDs each which were used for debugging and for checking the results. The three LEDs were classified as the "Control LED', the 'Modem LED' and the 'RF' LED. For generating the interrupts at a one second and one minute interval, two flags were used which were set based on the timer counters. As and when the flags were set, an LED on the

board was set to glow. The LED would glow ‘Green’ for every one second interrupt generated and ‘Red’ for every one minute interrupt generated.

After generating the interrupts in software, the Console board was programmed to transmit a character at every second at a rate of 19200 kbps. A function named ‘TransmitByte’ was written to transmit the character. After it was programmed, the board was connected to a desktop computer through a com-port and the Hyper-terminal connection. Figure 4.3 shows the snapshot of the character being received by the Hyper-terminal window.

Thereafter, the Console board was programmed to transmit a character for every minute. Figure 4.4, 4.5, 4.6, 4.7 show the character being received at the Hyper-terminal window after one minute, 2 minutes, 5 minutes and 10 minutes respectively. After successfully transmitting a character at the required intervals, the Relay Unit was also programmed to transmit characters. Once this was accomplished, the Relay Unit was programmed to receive a character from the Console. A function named ‘ReceiveByte’ was written to handle the reception of each character. Thereafter, the Console board and the Relay Unit were programmed individually for transmission and reception of a character at every one minute interval respectively. Both the boards were then connected using an RS-232 cable. On successful transmission of a character, an LED on the Console board was set to glow and on successful reception of the character, an LED on the Relay Unit was set to glow. Once this task was accomplished, the next step to be taken was to have the Relay Unit send a response character to

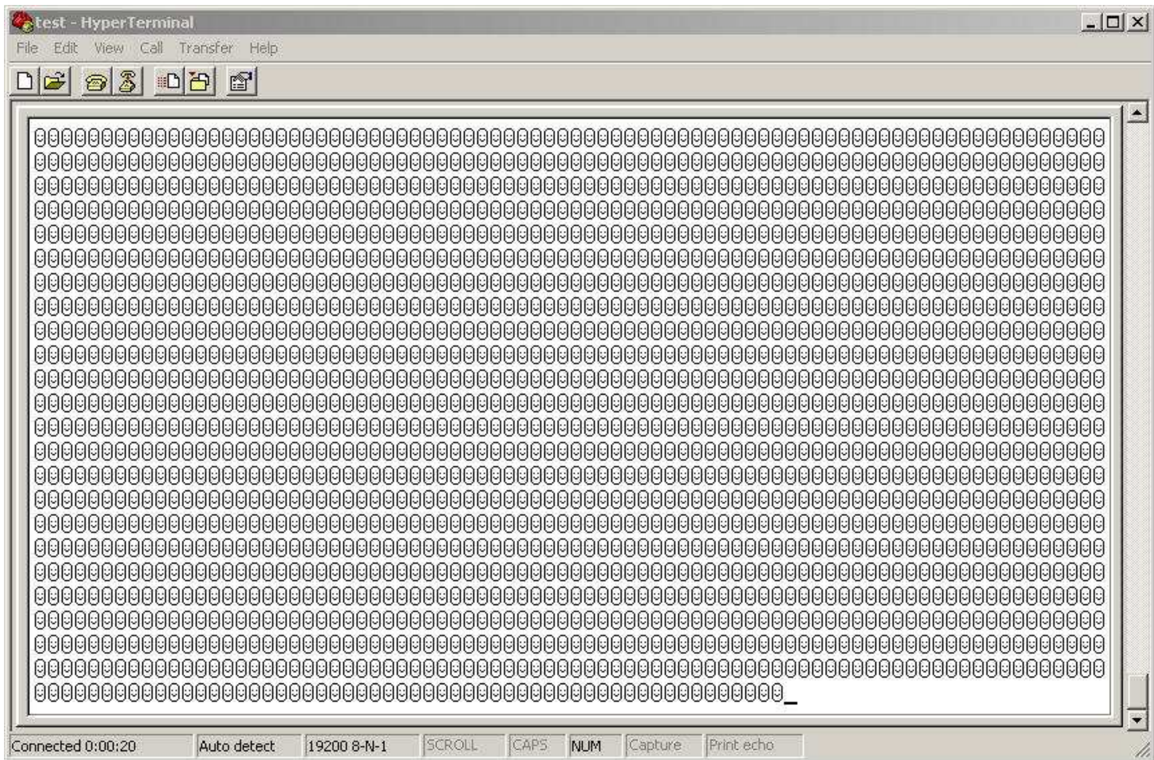


Figure 4.3: Hyper-Terminal window receiving characters every second

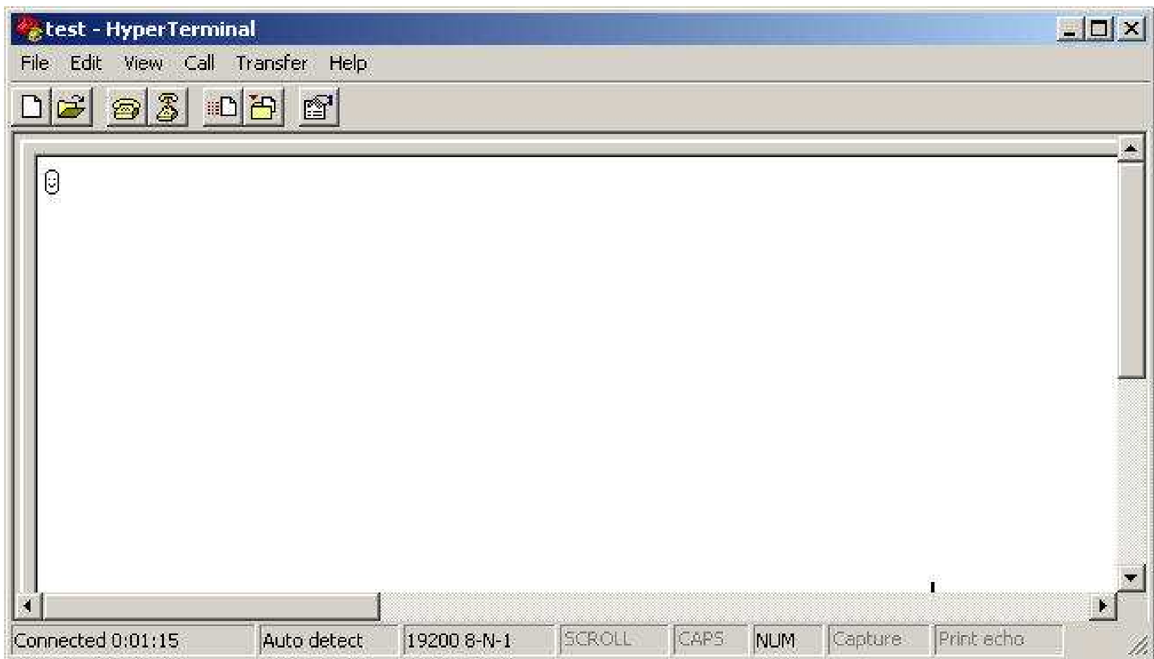


Figure 4.4: Hyper-terminal window after one minute

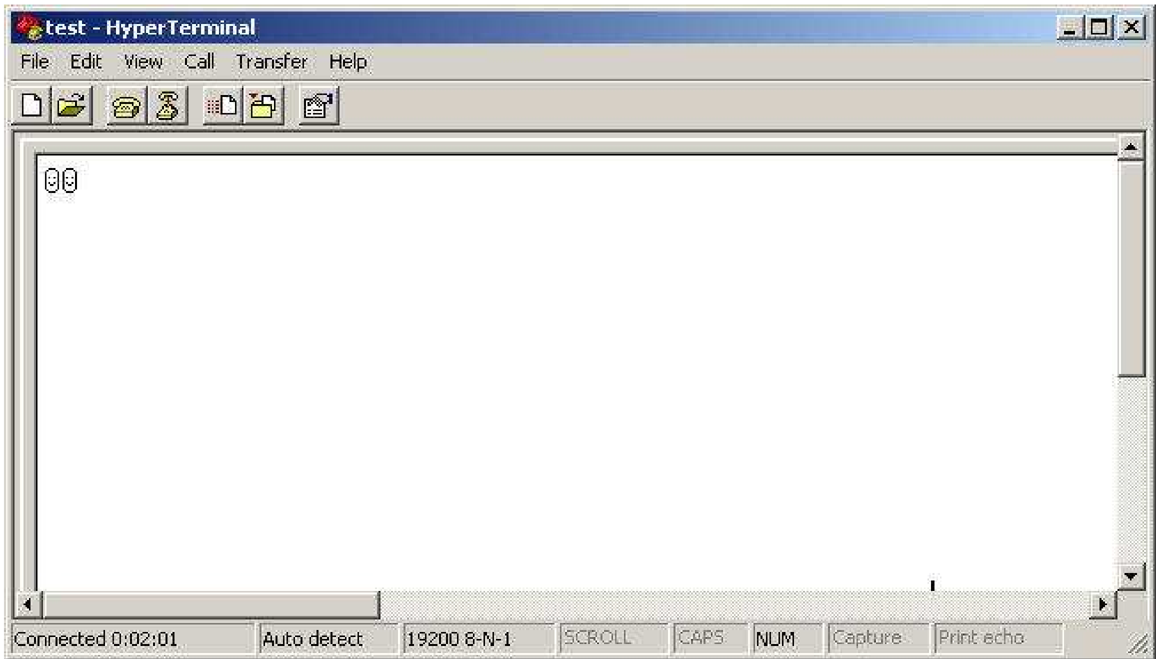


Figure 4.5: Hyper-terminal window after 2 minutes

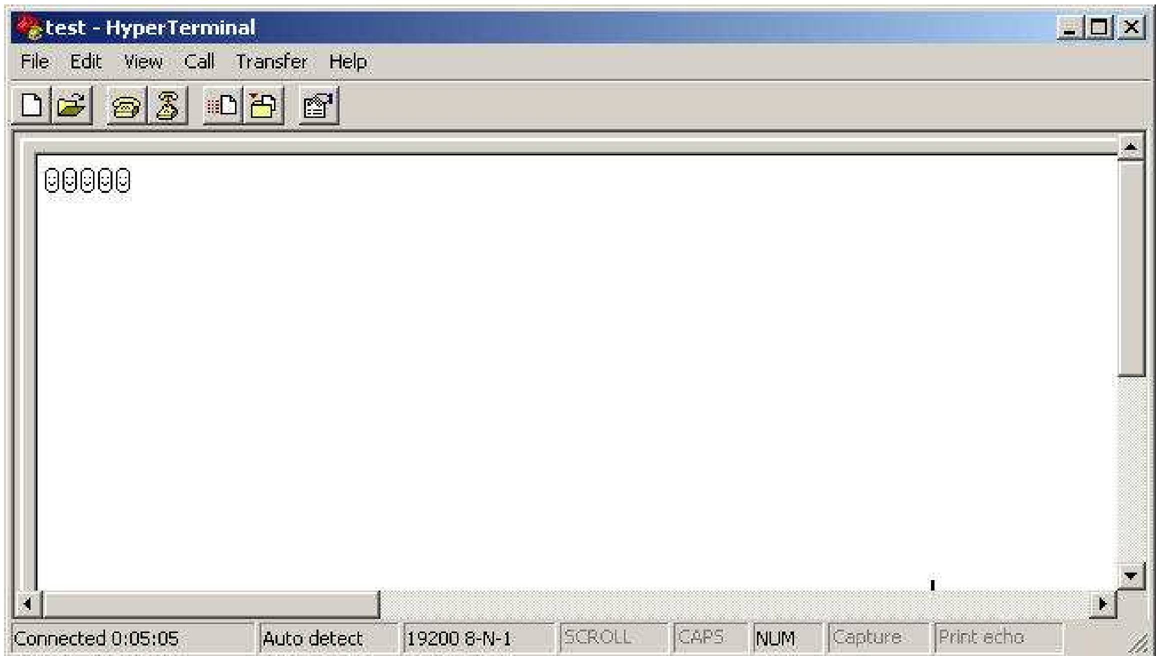


Figure 4.6: Hyper-terminal window after 5 minutes

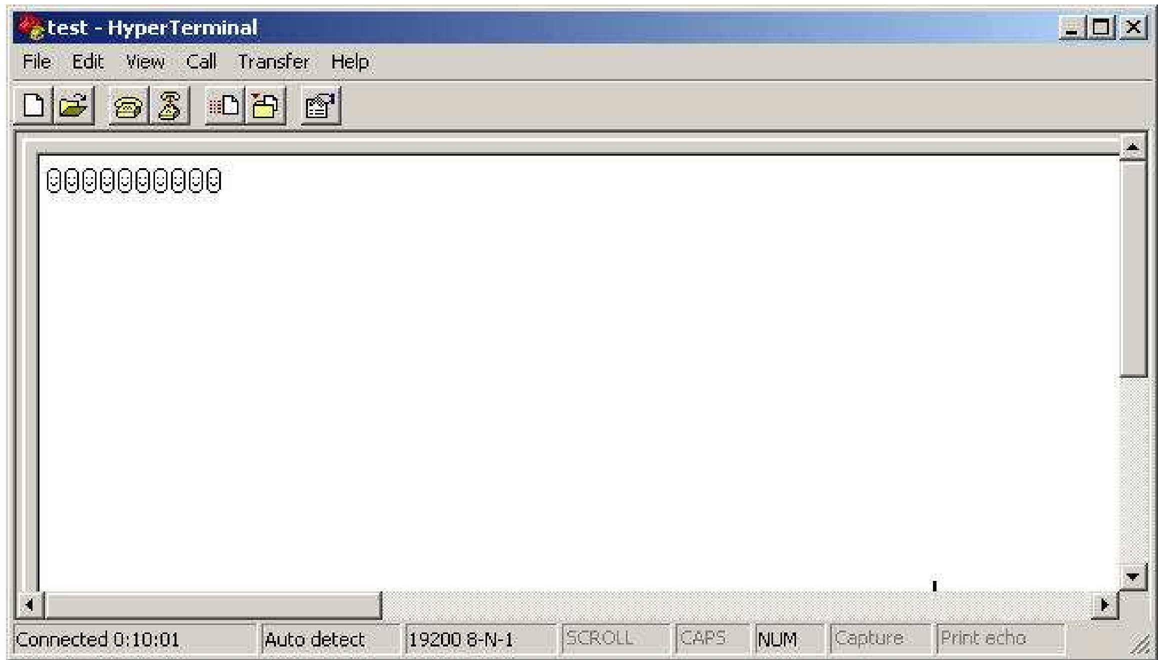


Figure 4.7: Hyper-terminal window after 10 minutes

the Console board. The Relay Unit's logic was again modified in software to transmit a response character, after it successfully received and processed the character sent by the Console board. The 'TransmitByte' function was used in the Relay Unit this time, to send a response character. This function was used after the 'ReceiveByte' function, i.e. only after the incoming character was validated. On the Console board, the 'ReceiveByte' function was used to handle the reception of the response character sent by the Relay Unit. Different LEDs were set to glow for each of the actions taking place. The Console board would transmit a character at every one minute interval which was validated by the glow of an LED and the Relay Unit would receive the character. Once it was validated, an LED on the Relay Unit would glow, and thereafter the Relay Unit would transmit a response character. The Console which would be in the 'Waiting for Reception' state would detect the response character and would glow an LED as and when the received character was validated. The Console was designed to transmit a character at every one minute interval irrespective of it receiving or not receiving the response character. A certain timer was put in place which would let the Console come out of the 'Waiting for Reception' state and get back into the transmission mode. A flag was used by the Console which would be set once any character was detected for reception. However, when the Console was transmitting a character, any incoming character was rejected.

Once the task of having the two units communicate with characters was accomplished, the next step of having packets or stream of characters being

exchanged had to be undertaken. Initially a stream of two characters was transmitted at a time as a packet from the Console board. This was received by the Hyper-terminal window for every one minute. A function named 'TxPacket' was used for transmission of the stream of characters as a packet. The function made use of pointers and data buffers for transmission. Figure 4.8, 4.9, 4.10 show the Hyper-terminal window receiving the simple packet consisting of two characters 1 and 2 at one minute, 2 minutes and 5 minutes.

Thereafter, the Relay Unit was programmed to receive the continuous stream of characters. A function named 'RxPacket' function was used to handle the reception of the packet. However, even though the Relay Unit received the packet consisting of 2 characters, the design lacked the logic to determine the end of a packet. In order to accomplish this, a timer was set for receiving every

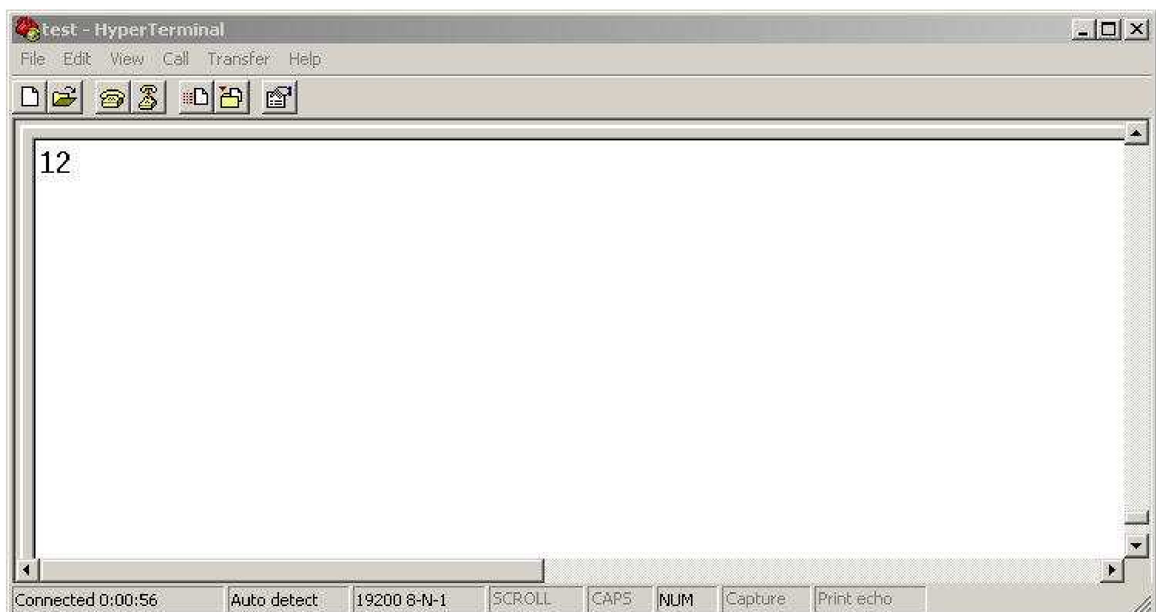


Figure 4.8: Hyper-terminal window receiving the packet after 1 minute

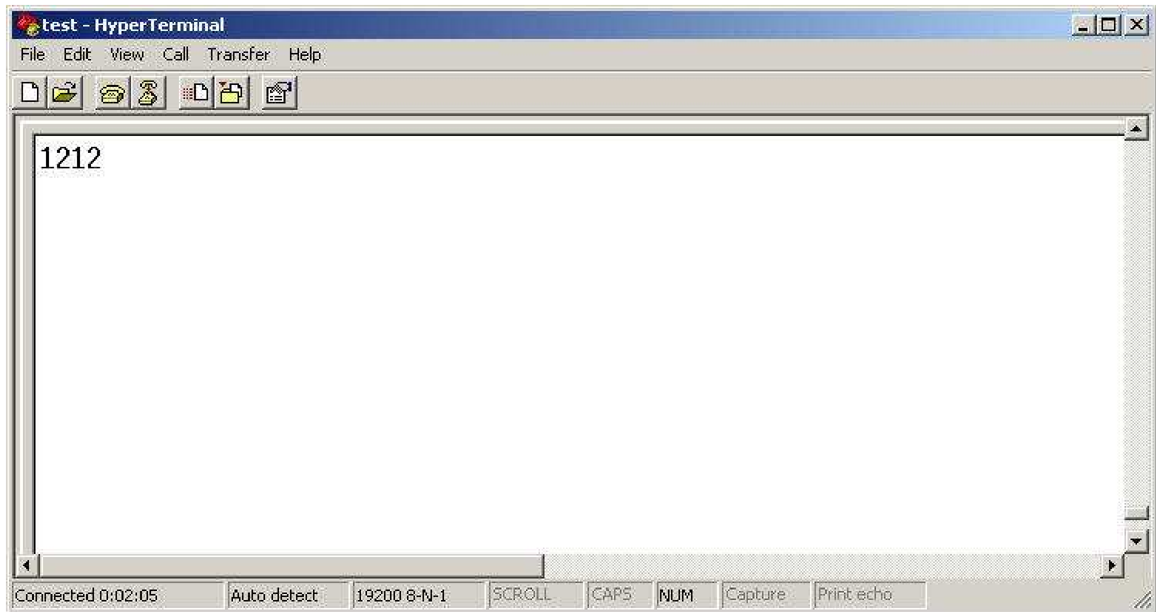


Figure 4.9: Hyper-terminal window receiving the packet after 2 minutes

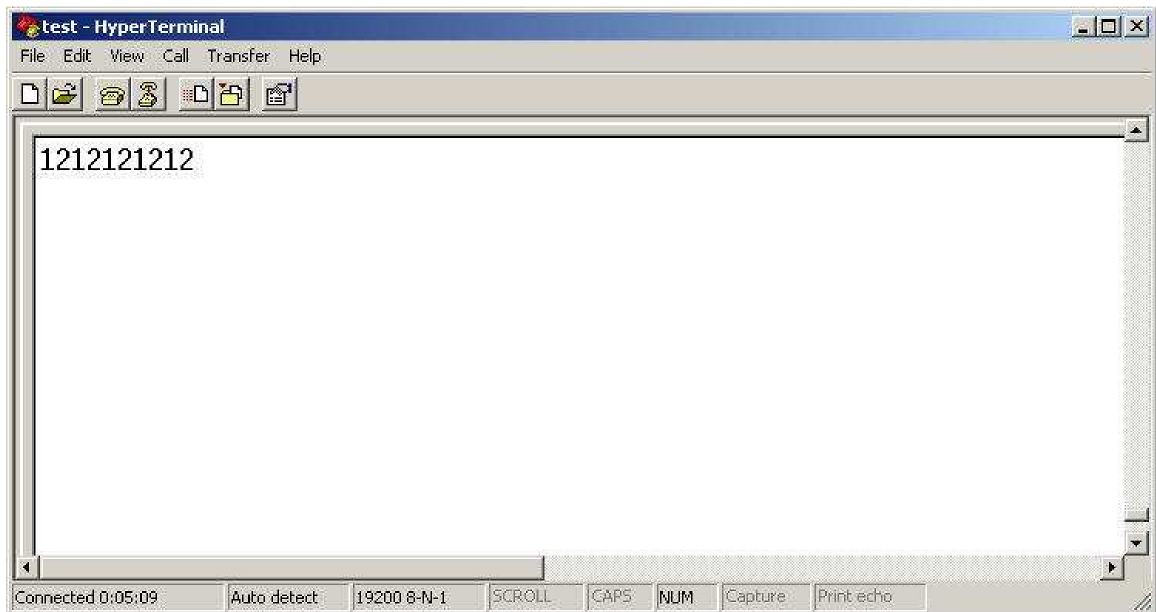


Figure 4.10: Hyper-terminal window receiving the packet after 5 minutes

consecutive byte in a packet. This time-out would be set after 2 milliseconds after which the Relay Unit would not classify any incoming character as a part of the previous packet. Once this was accomplished, the Relay Unit was programmed for transmitting its response packet to the Console.

The ‘TxPacket’ function was now used in the Relay Unit for transmitting a packet and the ‘RxPacket’ function along with the time-outs was used in the Console for handling the response packet from the Relay Unit. The Relay Unit transmitted a response packet only after the packet received from the Console was validated. The Console also employed timers for reception and would go back into the transmission mode if any time-out in reception occurred before receiving the entire response packet. The incompletely received response packet was discarded by the Console and it kept transmitting the packet at every one minute interval. The reception flag would be set if any further incoming data was detected. However, if the Console would be transmitting at that point of time, the incoming data is rejected. This logic was implemented in the Relay Unit as well. Any incoming data was rejected while it would be transmitting a packet.

As in the earlier case of transmission and exchange of a single character, the LEDs were set to glow for each of the actions performed. The initial transmission of the packet from the Console, the reception by the Relay, its validation of the received packet and subsequent transmission of the response packet and finally the reception and validation of the response packet by the Console were indicated by the glow of different LEDs on each of the boards.

Once the data exchange in packet form was accomplished, the next step was the crucial segment of the work. The MODBUS serial line protocol had to be implemented on the data packets and this involved the design of frames for different functions to be performed.

The MODBUS read packet was first designed which is shown in the Figure 4.11.The MODBUS read packet started with a Slave Address which comprised of one byte, ‘0x01’ pertaining to the Relay Unit in this case. It was followed by the Function Code which again was a byte. The Function Code used was one among ‘0x03’, ‘0x04’ and ‘0x07’ for the read function. The Function Code was followed by the Start Address High byte and Start Address Low byte. These two bytes determined the address from which the Relay Unit or the Personal Computer had to read the required data. The Start Address Low byte was followed by the Number of Points High byte and Number of Points Low byte. These two bytes provided the number of bits that had to be read by the Relay Unit or by the Personal Computer. The Number of Points Low byte was followed by the CRC Low byte and CRC High byte. The ‘create_req_packet’ function was designed to frame the read packet and the ‘TransmitByte’ function was used to

Slave Address	Function Code	Start Address Hi	Start Address LO	No of Points Hi	No of Points Lo	CRC Lo	CRC Hi

Figure 4.11: MODBUS read packet

transmit the read frame to the Hyper-Terminal. The Console was programmed to transmit the MODBUS read packet for every minute and connected to the desktop computer. Figure 4.12 and 4.13 show the Hyper-Terminal receiving the read packet from the Console after one minute and 2 minutes. The Hyper-terminal converts the Hexadecimal contents of the read packet into ASCII characters and displays them. Thus, the characters "130:PL'" in the window represent the Hexadecimal values "31,33,30,3A,30,50,4C,60" respectively. While transmitting the packet, the hexadecimal values are converted to ASCII by adding the value '0x30' to every byte. Hence, the values obtained on the Hyper-terminal window represent the hexadecimal sequence '0x01, 0x03, 0x00, 0x0A, 0x00, 0x20, 0x1C, 0x12'.

The first byte '0x01' indicates the Slave address followed by the read Function Code '0x03'. In this case, the Console is trying to read the parameter at the 16th address byte, which is indicated by the Start Address '0x00 0x0A'. The numbers of points or bits to be read are 32 which is specified by the Number of Points bytes '0x00 0x20'. The last 2 bytes '0x1C' and '0x12' are the CRC bytes which are computed before transmitting the read frame. The 'calc_crc' function used for computing the CRC is called by the 'create_req_packet' function before the transmission is initiated. The CRC is performed on every message content of the frame and the final result is appended at the end of the frame.

After successfully transmitting the read packet from the Console to the Hyper-terminal, the next obvious step was to handle the reception of the read

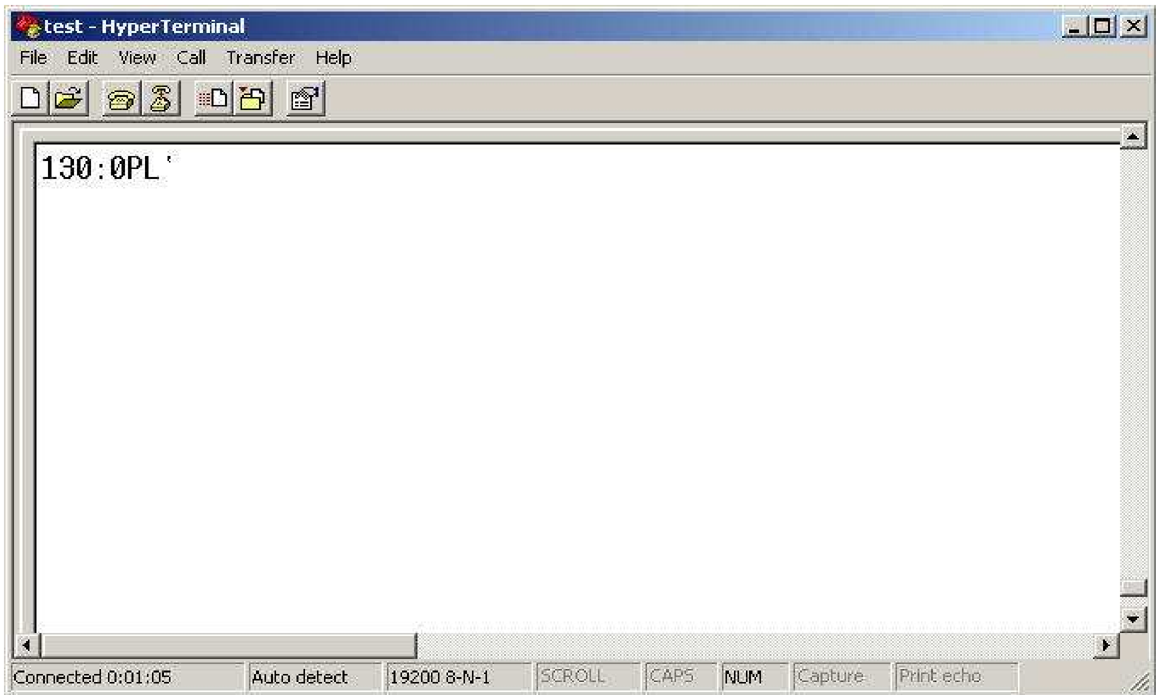


Figure 4.12: Hyper-terminal receiving the MODBUS read packet after 1 minute

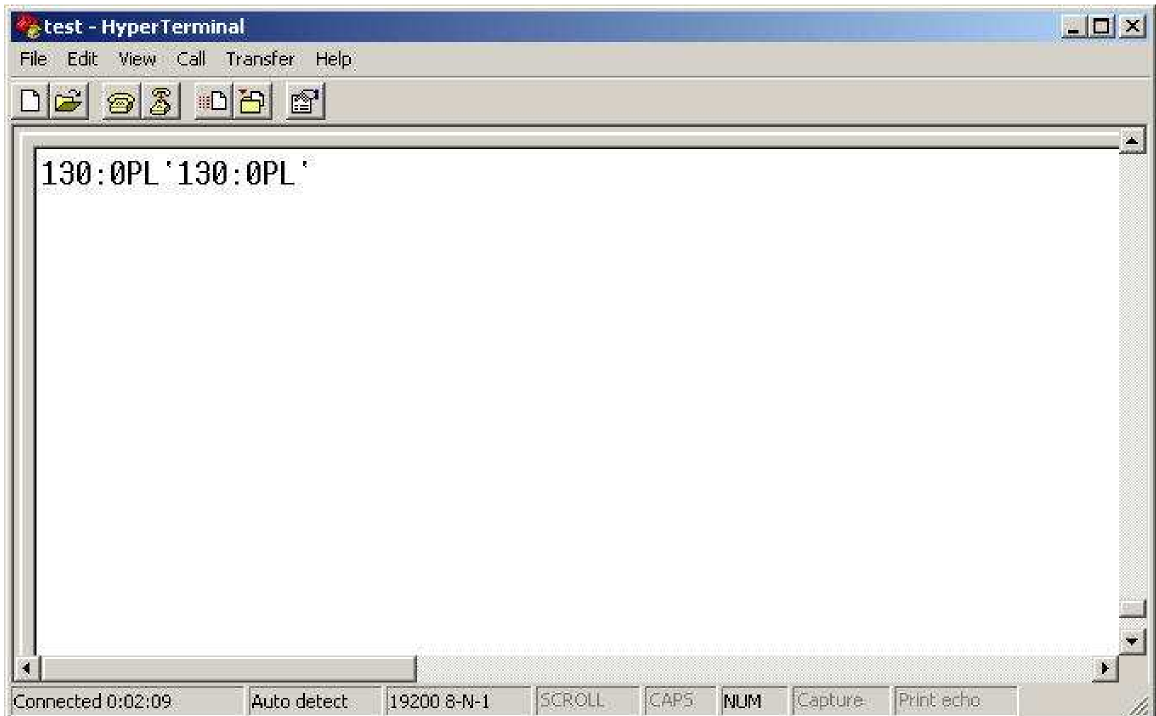


Figure 4.13: Hyper-terminal receiving the MODBUS read packet after 2 minutes

packet on the Relay Unit. In order to handle this task, the receiving buffer size was increased to accommodate the packet which was of bigger size now. The Relay Unit would go into the receiving mode once an initial character was detected. The first character i.e. Slave Address received is checked by the Relay Unit to determine if the packet was intended for it. After validating the Slave address, the Relay Unit starts accepting the remainder of the packet starting from the Function Code. Since the packet received is a read packet, the Relay Unit was configured with a byte counter or pointer which would enable it to receive only the remaining 6 bytes i.e. the Start Address High and Low bytes, the Number of points High and Low bytes and the CRC High and Low bytes. Any further bytes received after that point of time were discarded. The Relay Unit would then validate the entire frame by computing the CRC on each and every content of the frame except on the last two bytes. The resulting CRC computed is checked with the two bytes of CRC sent in the original message frame and is validated. As before, the transmission, reception and validation of the packet are indicated by the glow of LEDs on both the boards. The Relay Unit is employed with timers which employ a time constraint on successive reception of bytes and also on the total time in which an entire frame must be received. If any time-out occurs during the reception of successive bytes, the reception is stopped and the frame is discarded if it had not been received in its entirety. The frame is also discarded when the timer controlling the reception of the entire frame expires provided the frame not being received completely by that time.

After successfully transmitting the read packet from the Console to the Relay, the next step of having the Relay send a response packet with the requested data had to be undertaken. The memory map of the tank structure was implemented in the Console and the Relay Unit. In order to send a response packet, the 'create_resp_pack' function was designed which framed the appropriate reply to the requested action. The response packet would be framed after the requested action was performed by the Relay. Figure 4.14 shows the design of the response packet at the Relay Unit. The response packet contains the Slave Address '0x01' i.e. address of the Relay Unit, followed by the Function Code '0x03' indicating that a read action was requested initially. The Function Code is followed by the Byte Count byte which contains the number of bytes that are read starting from the address specified by the Start Address bytes in the request packet. The Byte Count is followed by the Data High and Low bytes which are the data requested by the Console. The data bytes are followed by the CRC High and Low bytes which are computed by the Relay. The CRC is performed on the entire portion content of the frame and the resulting two bytes of the computation are appended at the end of the message frame.

Once the Relay was programmed for sending the response packet to the Console, the two boards were connected and checked for proper operation. The

Slave Address	Function Code	Byte Count	Data Hi	Data Lo	CRC Lo	CRC Hi

Figure 4.14: MODBUS read response packet

Console sent a MODBUS read packet every minute, requesting to read the 16th byte from the Relay. The Relay received the packet and performed the action required and replied with the data required to the Console.

Thereafter, the Console validated the response packet it received. The validation included the data read and also the CRC being checked. The CRC computation and checking done by the Console is similar to the computation done by the Relay Unit when it receives a packet. All these actions were validated and were checked by having the LEDs glow at each step.

Once the 2-way read exchanges were performed the final step of having the MODBUS write packet transmitted from the Console to the Relay and the pertaining response sent from the Relay to the Console had to be undertaken. Figure 4.15 shows the design of the MODBUS write packet. The MODBUS write packet started with the Slave Address ‘0x01’ pertaining to the Relay Unit. This was followed by the Function Code byte which denoted the write function. The Function Code could be one among ‘0x05’, ‘0x06’, 0x0F’ and ‘0x10’. The Function Code was followed by the Start Address High and Low bytes which indicated to the Relay Unit the address from which the data to be written had to

Slave Address	Function Code	Start Address Hi	Start Address LO	No of Points Hi	No of Points Lo	Byte Count	Data Hi	Data Lo	CRC Lo	CRC Hi

Figure 4.15: MODBUS write packet

start. The Start Address Low byte was followed by the Number of Points High and Low bytes which indicated to the Relay the number of bits that had to be written. The Number of Points Low byte was followed by the Byte Count byte.

This Byte Count though may seem unnecessary, plays a vital role in the reception part of the process. The Byte Count field indicates to the Relay Unit the number of data bytes it has to expect for further reception. The Byte Count is the sum of the number of Data High and Data Low bytes which follow the Byte Count field in the message frame. The CRC bytes which are computed on each and every content of the message are appended at the end of the message.

As in the earlier case of transmitting a read packet, the 'create_req_pack' function was used to frame the write packet and the 'TransmitByte' was used for transmission of the write packet from the Console to the Hyper-terminal on the desktop computer. Figure 4.16 and 4.17 show the write packet being received on the Hyper-terminal window after one minute and two minutes respectively. The ASCII characters "150:0P412342=" in the window represent the Hexadecimal characters '31, 35, 30, 3A, 30, 50, 34, 31, 32, 33, 34, 32, 3D' respectively. While transmitting the packet, the hexadecimal values are converted to ASCII by adding the value '0x30' to every byte. Hence, the values obtained on the Hyper-terminal window represent the hexadecimal sequence '0x01, 0x05, 0x00, 0x0A, 0x00, 0x20, 0x04, 0x01, 0x02, 0x03, 0x04, 0x02, 0x13'.

The first byte '0x01' indicates the Slave address followed by the read

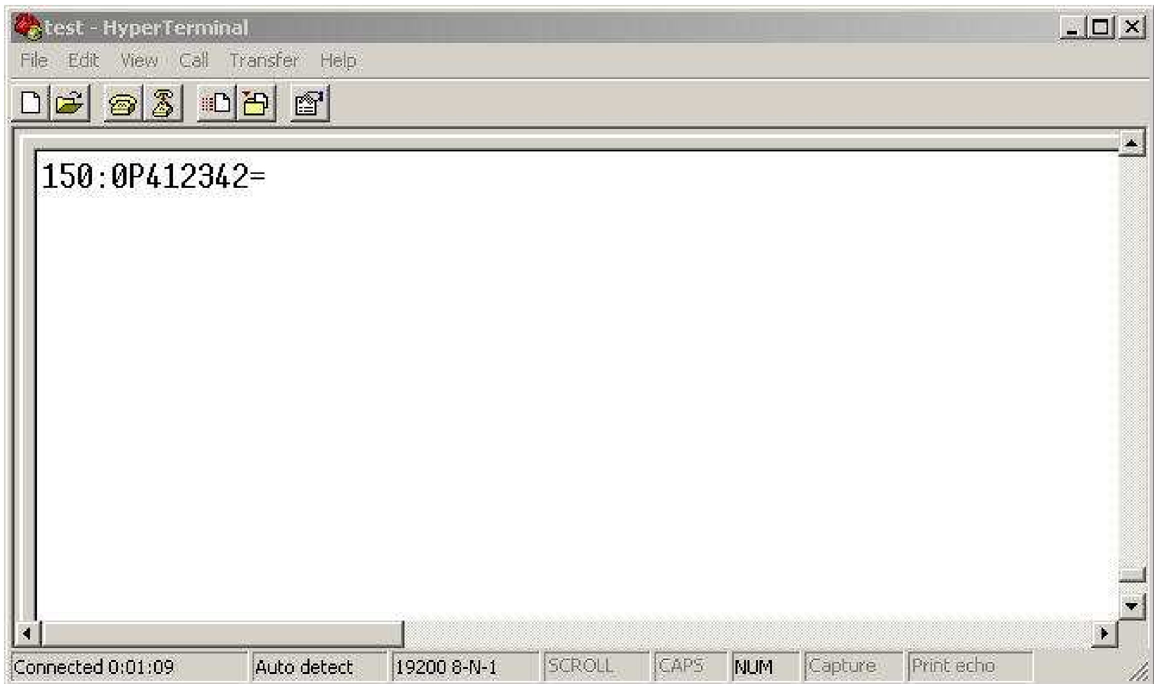


Figure 4.16: Hyper-terminal window receiving the MODBUS write packet after 1 minute

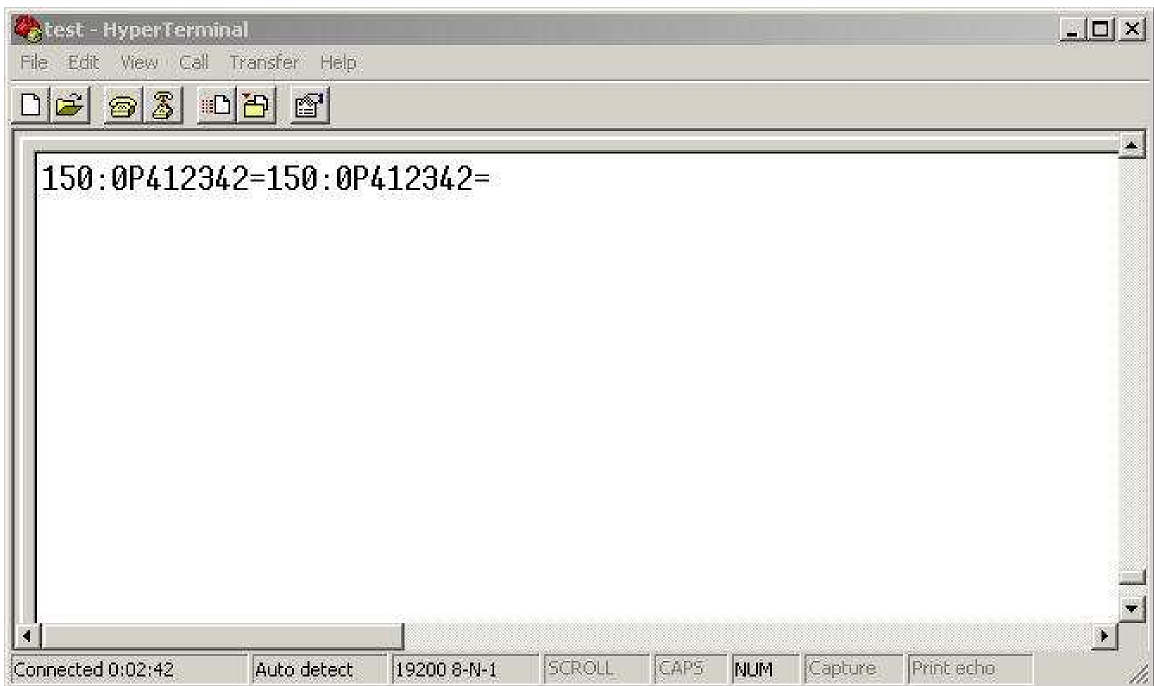


Figure 4.17: Hyper-terminal window receiving the MODBUS write packet after 2 minutes

Function Code ‘0x05’. In this case, the Console requests to write a parameter value at the 16th address byte, which is indicated by the Start Address ‘0x00 0x0A’. The numbers of points or bits to be written are 32 which is specified by the Number of Points bytes ‘0x00 0x20’. The total number of bytes present in the Data High and Low fields is ‘0x04’ which is indicated by the Byte Count field. The Byte Count field is followed by the data bytes which have to be written. The data bytes in this case are ‘0x01 0x02 0x03 0x04’. These data bytes are followed by the CRC Low and High bytes, ‘0x02’ and ‘0x13’ in this case. As earlier, the CRC is computed using the ‘calc_crc’ function which computes the CRC on the entire message contents. The resultant two bytes are appended at the end of the frame.

After the successful transmission of the write packet from the Console to the Hyper-terminal, the remaining task of having the Relay Unit receive the write packet and perform the required writing operations was undertaken. The Relay Unit would go into the receiving mode once an initial character was detected. The first character i.e. Slave Address received is checked by the Relay Unit to determine if the packet was intended for it. After validating the Slave address, the Relay Unit starts accepting the remainder of the packet starting from the Function Code. After determining the packet to be a write packet, the Relay Unit would accept the remainder of the packet starting from the Start Address High byte. It would accept the bytes until the Byte Count field occurred. A counter was used to track the reception of the incoming bytes and as soon as the counter indicated

the reception of the Byte Count field, another data counter or pointer is assigned the value obtained from the Byte Count field. The Relay would thereafter store the incoming bytes in a data buffer until the data counter goes to null. Once the data counter goes to null indicating the end of data bytes, the Relay Unit receives only the next two bytes i.e. CRC High and Low bytes which are used later for comparison and verification. The Relay Unit computes the CRC on the entire message frame except the last two bytes and the resultant CRC is compared with that of the original CRC appended to the incoming message frame. The frame would be discarded if the two set of values do not match with each other. Else the required write action would be performed. As before, the transmission, reception and validation of the packet were indicated by the glow of LEDs on both the boards.

Once this was accomplished, the Relay Unit was designed to send a response packet for the write packet it received from the Console. For this a function named 'create_resp_pack' function was designed to frame the response packet. Figure 4.18 shows the design of the write response packet from the Relay Unit. The write response packet starts with the Slave Address, '0x01' being the case and is followed by the Function Code which is '0x05' for the write function. The Function Code is followed by the Start Address High and Low bytes which denote the starting address in the memory from which the data had been written. The Start Address bytes are followed by the Number of points High and Low bytes which provide information about the number of bits that were written

Slave Address	Function Code	Start Address Hi	Start Address LO	No of Points Hi	No of Points Lo	CRC Lo	CRC Hi

Figure 4.18: MODBUS write response packet

into the memory of the Relay Unit. As in the previous cases, the Number of Points Low byte is followed by the CRC Low and High bytes which are computed by the Relay on the entire frame contents and appended at the end.

Once the Relay was programmed for sending the response packet, the 2 boards were connected and the exchange of write packet was observed. The step by step actions performed starting from the Console transmitting the write packet, the Relay receiving it and validating the received data, the Relay responding to the Console and finally the Console validating the response packet were indicated by the glow of LEDs.

The Console on reception of the response packet irrespective of it being a response for a read or write request sent previously, would check for the Slave Address to determine if the response came from the correct unit. Thereafter, it would start accepting the remainder of the packet starting from the Function Code. The Console was configured with data counters or pointers which would keep check on the remaining incoming data. The data pointers used were different for the read and write cases. This was because the sizes of the response packet for read and write differed. Once the data pointers went to null,

it would not accept any further data. The Console's receiving operation was similar to that of the Relay Unit. Once it went back to transmitting a new packet, any incoming data was rejected. The Console also had timers which kept a check on the reception of successive bytes and also on the reception of the entire message frame. If any of these timers expired before the entire message was received, the packet would be discarded.

After successfully handling the write packet exchanges between the two boards, an additional task was implemented. The Console would initiate a write request to the Relay Unit, which would process the request and reply with a response packet. The Console on validating the response would then initiate a read request which tries to read the same data from the memory location which was written by the earlier write request. The Relay Unit would process the read request and reply back with pertaining data. The Console on receiving the reply would validate the data. This process ensured that the write request initiated in the beginning was accurately performed on the relevant parameter and at the relevant memory location. Figure 4.19 and Figure 4.20 show the final action flow of the Console and the Relay Unit.

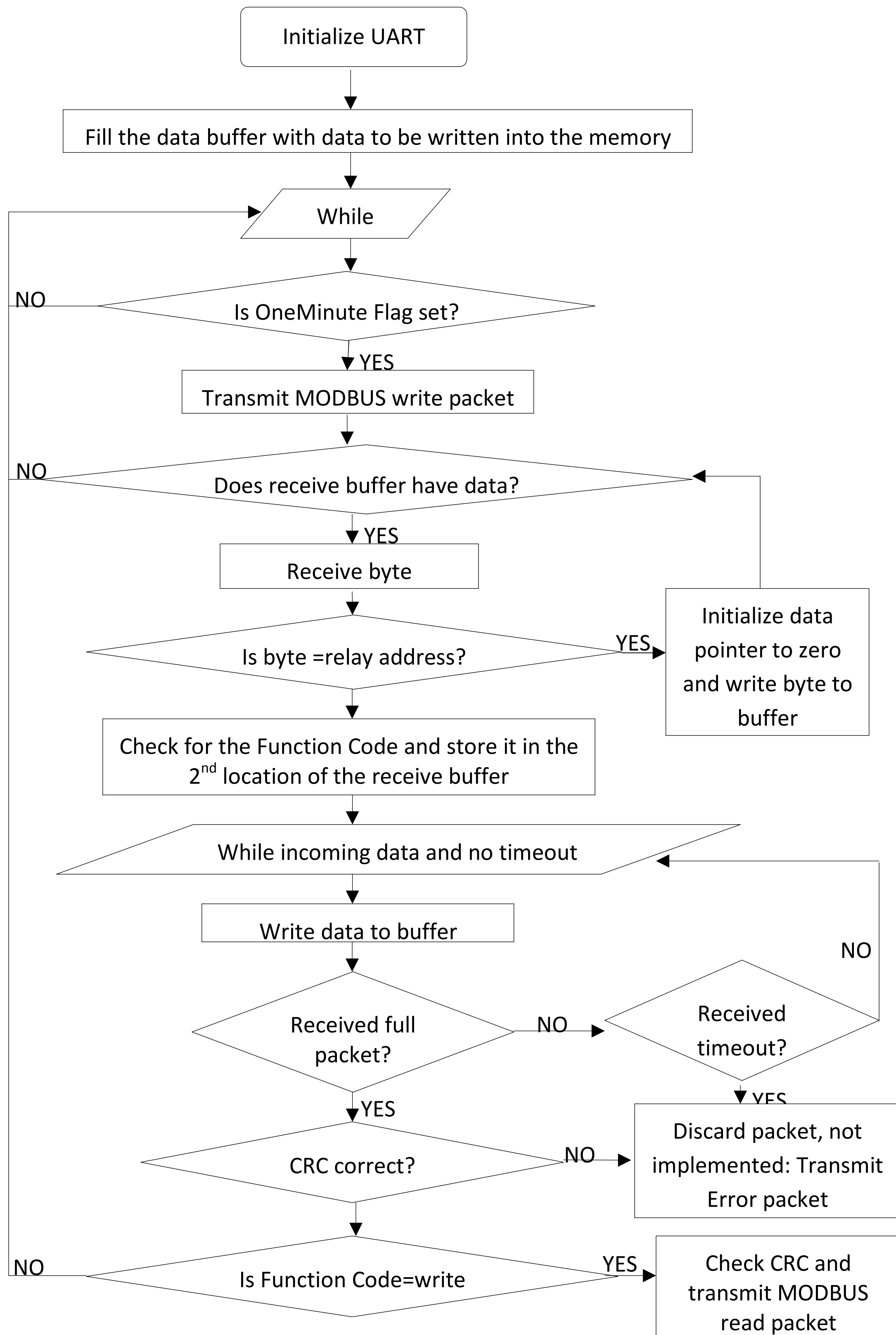


Figure 4.19: Console Final Action Flow

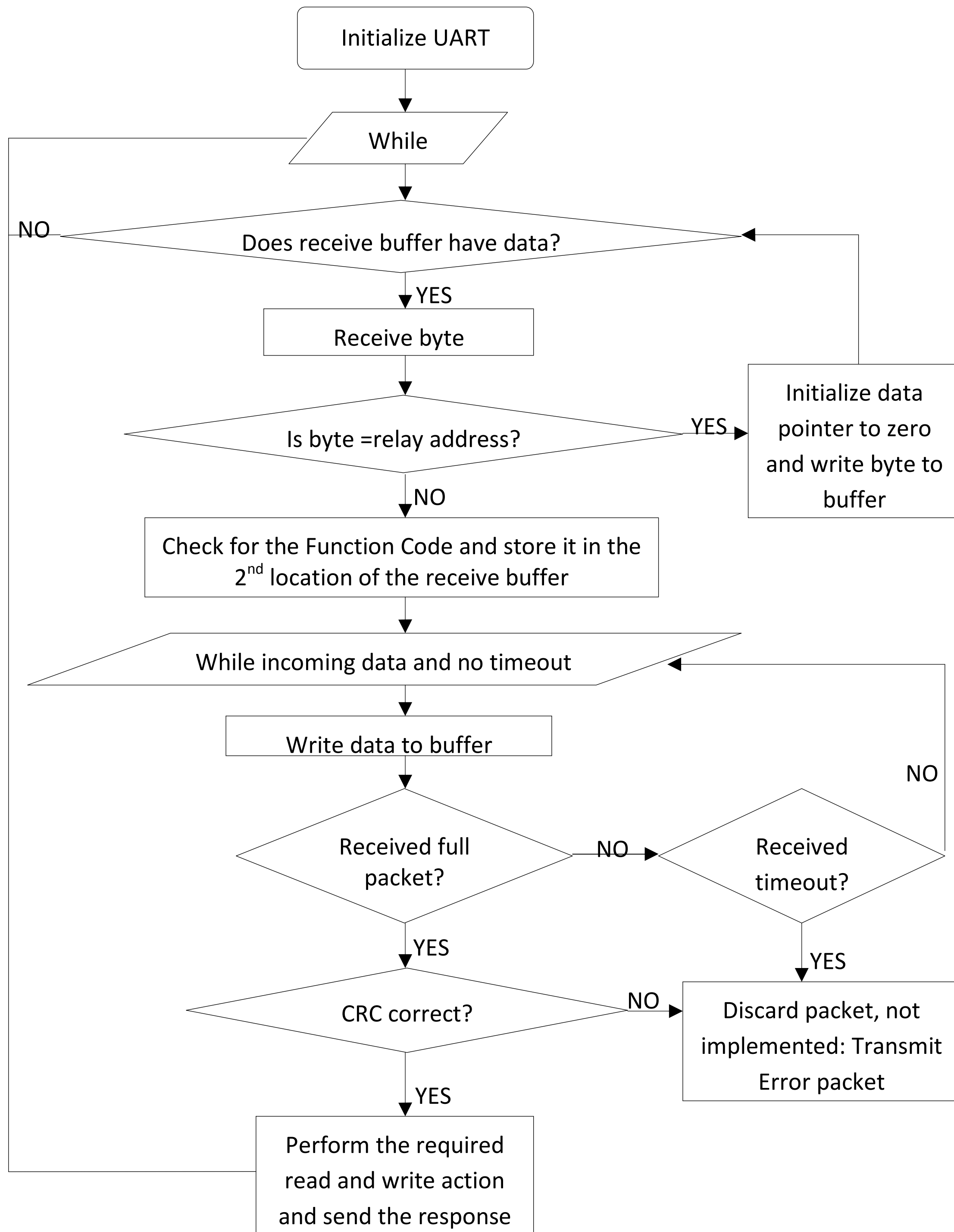


Figure 4.20: Relay Unit Final Action Flow

Chapter 5: CONCLUSIONS

The essential task of having the Console communicate with the Relay Unit and the Personal Computer was accomplished. The communication was based on the Client-Server architecture and was implemented using an RS 232 serial link between the individual entities. The MODBUS protocol was successfully implemented on the model.

The entire model was redesigned and the appropriate software was implemented on the units. The logic in the software was remodeled, written and verified through successful programming of the boards. The previously existing software was modified and functions for creation, transmission and reception of a MODBUS packet were written. Since the data was written into the EEPROM of the units, in case of any power failure the system would return to its most recent configuration and relay settings. However, there are limitations to this work with regard to performance.

The MODBUS as discussed in the previous chapters allows for a total of 256 bytes in a single packet to be transmitted or received in a single transaction. Quite obviously the MODBUS write packet would be the largest size of a packet in a single transaction between the devices. The write packet would include the Slave address, Function Code, Start Address, the Number of Points, the Byte Count which take 7 bytes. Hence the amount of data that could be written by transmitting a single write packet would be limited to 245 bytes. Similarly, the

data that can be transmitted in a read MODBUS packet is limited to 251 bytes due to the framing of the packet. If the number of tanks and their respective number of parameters to be handled increases, then more than a single MODBUS transaction at a time would be required. Though this would not affect the efficiency or throughput of the system, it would need require more computation and more transactions. Also there would also be problems regarding noise when the two units are far apart and connected using the RS-232 cable.

Future scope for this work would involve implementation of the exception codes. All the exception codes specified by the MODBUS need to be implemented in the model.

References

[1]. Martin D. Seyer, *Complete Guide to RS232 and parallel connections,* Englewood Cliffs, N.J.: Prentice Hall, 1988.

[2]. Byron W.Putman, *RS232 Simplified,* Englewood Cliffs, N.J.: Prentice Hall, 1987.

[3]. Alex Berson, *Client- Server Architecture,* New York: McGraw-Hill, 1992.

[4]. Modbus Application Protocol, http://www.Modbus-IDA.org.

[5]. Modbus over Serial Line-Specification & Implementation, http://www.modbus.org.

[6]. Herbert Schildt, *C: The Complete Reference, 4th Edition,* New York: McGraw Hill, 2000.

[7]. E. Balaguruswamy, *Programming in ANSI C, 2nd Edition,* New York: McGraw Hill, 1994.

[8]. IAR Embedded Workbench, http://www.iar.com.

[9]. AVR Atmega-163 processor, 8-bit Microcontroller with 16K Bytes In-System Programmable Flash, http://www.atmel.com.

[10]. AVRISP mkII Programmer Guide, http://www.atmel.com.

[11]. AVR Studio, http://www.atmel.com.

[12]. Hyper-terminal, http://www.hilgraeve.com/htpe/

Vita

Naresh Karnam was born on 24th of June, 1984 in the town of Kadiri, A.P. India. He went to Bharatiya Vidya Bhavan's Public School, B.H.E.L., Hyderabad, India. He received his Bachelor's degree from D.V.R.C.E.T, Hyderabad, India in Electronics & Communications Engineering. He joined the University of Tennessee in Fall 2005 and concentrated mainly on Digital Signal Processing and Digital Communications subjects. He graduated with a Master of Science Degree in Electrical Engineering in December 2007.