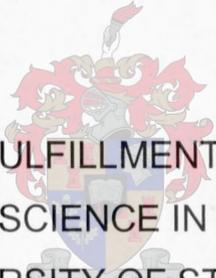


**THE DESIGN OF A COMMUNICATION PROTOCOL
FOR A DISTRIBUTED ADCS FOR SUNSAT 2.**

Abram Tshwaro Magano



THIS IS PRESENTED IN FULFILLMENT OF THE REQUIREMENTS
FOR THE MASTERS OF SCIENCE IN ENGINEERING SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH.

Supervisor: Mr. N.L. Steenkamp

December 2001

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my original work, and has never been submitted, in part or in its entirety, to any University, for any requirements towards the achievement of any degree.


A . T. Magano


Date

Abstract

One of the main subsystems of SUNSAT is the Attitude Determination and Control System (ADCS), responsible for the orientation and positioning of the satellite. Due to the integrated architecture of the system, several shortcomings were identified. A possible solution to the problems is the implementation of a distributed system. The design of a communication protocol for a distributed system is the focus of this thesis.

An investigation on different network topologies and communication protocols as well as error control techniques is carried out to establish a combination that meets the requirements of the ADCS. Based on defined protocol specifications a detailed protocol design is proposed. Then the protocol is implemented using a layered software structure that emanates from the OSI layering model, to provide well defined software structures and interfaces. A series of measurements have shown that the protocol meets the functional requirements of the ADCS and further provides reliable data transfer on the network.

Opsomming

Een van die vernaamste dele van SUNSAT is die “Attitude Determination and Control System” (ADCS) wat verantwoordelik is vir die oriëntasie en posisionering van die satelliet. Verskeie tekortkominge as gevolg van die geïntegreerde argitektuur van die stelsel, is geïdentifiseer. ’n Moontlike oplossing vir die probleme is die implementering van ’n verspreide stelsel. Die ontwerp van ’n kommunikasie protokol vir ’n verspreide stelsel is die fokus van die tesis.

’n Ondersoek na verskeie netwerk topologieë en kommunikasie protokolle, asook foutbeheer tegnieke is uitgevoer om vas te stel watter kombinasie die ADCS se vereistes sal bevredig. ’n Gedetailleerde protokol ontwerp is voorgestel gebaseer op gedefinieerde protokol spesifikasies. Hierdie protokol is toe geïmplementeer deur gebruik te maak van vlak gestruktureerde sagteware wat afkomstig is van die OSI model, met die oog op goed gedefinieerde sagteware strukture en koppelvlakke. ’n Reeks meetings het aangedui dat die protokol die funksionele vereistes van die ADCS bevredig en dat dit verder betroubare data verplasing oor die netwerk verskaf.

To my Mother and in Memory of My Grandparents.

Acknowledgements

I would like to thank the Lord for giving me strength and courage to complete this thesis. I would also like to express my sincere gratitude to the following people:

- Niki Steenkamp, my supervisor for his guidance and insight.
- Xandri Farr, my mentor for his academic example and continued support for this work.
- My family for their understanding and support.

Contents

ACKNOWLEDGEMENTS	I
CONTENTS	II
LIST OF ABBREVIATIONS AND ACRONYMS	VI
LIST OF FIGURES	VII
LIST OF TABLES	IX
CHAPTER 1	1
1 INTRODUCTION	1
1.1 SUNSAT ADCS SYSTEM ARCHITECTURE AND LIMITATIONS	1
1.2 THESIS OVERVIEW.....	4
CHAPTER 2	6
2 BACKGROUND	6
2.1 DISTRIBUTED SYSTEMS.....	6
2.2 EVALUATION OF NETWORK TOPOLOGIES	7
2.2.1 Ring Network.....	7
2.2.2 Star Network	8
2.2.3 Bus Network.....	10
2.3 CHOICE AND MOTIVATION FOR A PREFERRED NETWORK.....	10
CHAPTER 3	13

3	COMMUNICATION PROTOCOLS	13
3.1	MEDIA ACCESS PROTOCOL/CONTROL	13
3.1.1	Master Slave Polling	14
3.1.2	Token Bus	15
3.1.3	Carrier Sense Multiple Access (CSMA)	15
3.2	FLOW CONTROL	16
3.3	ERROR CONTROL	17
3.4	CHOICE OF A PROTOCOL AND ERROR CONTROL	18
CHAPTER 4.....		20
4	PROTOCOL DESIGN	20
4.1	PROTOCOL LAYERS	20
4.1.1	Data Link Layer	21
4.1.2	Physical Layer	22
4.2	PROTOCOL SPECIFICATION.....	23
4.2.1	Service Specification	23
4.2.2	Assumption about the environment.....	24
4.2.3	Protocol Vocabulary	24
4.2.4	Message Format	25
4.2.5	Procedure Rules	28
	4.2.5.1 <i>Recovery From Transmission Errors</i>	28
	4.2.5.2 <i>Token Passing and Generation</i>	31
	4.2.5.3 <i>Master Node State Machine</i>	33
	4.2.5.4 <i>Slave Node State Machine</i>	35
CHAPTER 5.....		37
5	PROTOCOL IMPLEMENTATION	37

5.1	PHYSICAL LAYER FUNCTIONS	38
5.1.1	Interrupt Service Routines.....	39
5.1.2	Physical Layer Transmission	41
5.1.3	Physical Layer Reception.....	42
5.2	DATA LINK LAYER FUNCTIONS	44
5.2.1	Data Link Layer Transmission.....	44
5.2.2	Data Link Layer Reception	45
5.3	TOKEN IMPLEMENTATION.....	49
5.4	IMPLEMENTATION SETUP.....	53
CHAPTER 6	55
6	TESTING AND EVALUATION	55
6.1	HARDWARE.....	55
6.2	SYSTEM INTEGRATION	57
6.3	TEST DESCRIPTION AND FAULT INJECTION.....	58
6.3.1	Physical Fault Injection	60
6.3.2	Software-Implemented Fault Injection.....	60
6.4	MEASUREMENTS AND RESULTS	61
6.4.1	Results	62
6.4.2	Transfer Rates	64
6.4.3	Memory Loading	65
6.5	EVALUATION.....	66
CHAPTER 7	68
7	CONCLUSION AND FUTURE RESEARCH.....	68
APPENDIX A	71
A. PROTOCOL TEST SETUP AND DATA SHEETS	71

Contents

v

A.1 TEST SETUP	71
A.2 DIAGNOSTIC SOURCE CODE	72
A.3 SCHEMATICS	73
A.4 DATA SHEETS	74
APPENDIX B.....	75
B. PROTOCOL SOURCE CODE	75
BIBLIOGRAPHY	76

List of Abbreviations and Acronyms

ACP	Attitude Control Processor
ADCS	Attitude Determination and Control System
CSMA	Carrier Sense Multiple Access
CSMA/CD	Carrier Sense Multiple Access with Collision Detection
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance
EEPROM	Electrically Erasable Programmable Read Only Memory
EOM	End of Message
ICP	Interface Control Processor
ISO	International Standard Organization
ISR	Interrupt Service Routine
OBC	On Board Computer
OSI	Open System Interconnection
PC	Personal Computer
RAM	Random Access Memory
SOM	Start of Message
SPI	Serial Programming Interface
SRAM	Static Random Access Memory
SUNSAT	Stellenbosch University Satellite
UART	Universal Asynchronous Receiver and Transmitter

List of Figures

FIGURE 1.1: BASIC STRUCTURE OF THE ADCS	2
FIGURE 2.1: RING NETWORK.....	8
FIGURE 2.2: STAR NETWORK	9
FIGURE 2.3: PROPOSED ADCS BUS NETWORK.....	12
FIGURE 3.1: MASTER NODE POLLING SLAVE NODES FOR INFORMATION.....	14
FIGURE 4.1: PROTOCOL LAYER	21
FIGURE 4.2: FRAME FORMAT.....	25
FIGURE 4.3: EFFICIENCY VERSUS NUMBER OF DATA BYTES	28
FIGURE 4.4: MESSAGE TRANSMITTED WITHOUT ERRORS.....	29
FIGURE 4.5: MESSAGE NOT RECEIVED BY THE SLAVE NODE	30
FIGURE 4.6: MESSAGE NOT RECEIVED BY THE MASTER NODE.....	31
FIGURE 4.7: TOKEN PASSING	32
FIGURE 4.8: MASTER NODE STATE MACHINE	33
FIGURE 4.9: SLAVE NODE STATE MACHINE	35
FIGURE 5.1: COMMUNICATING FUNCTIONS	38
FIGURE 5.2: SLIP FUNCTION FLOW DIAGRAM.....	42
FIGURE 5.3: DESLIP FUNCTION FLOW DIAGRAM.....	43
FIGURE 5.4: PROCESS MESSAGE STATE MACHINE	46
FIGURE 5.5: TOKEN FRAME.....	49
FIGURE 5.6: TOKEN GENERATION.....	50
FIGURE 5.7: TOKEN PASSING FLOW DIAGRAM	51
FIGURE 5.8: DUPLICATE TOKEN.....	52
FIGURE 5.9: IMPLEMENTATION SETUP.....	53
FIGURE 5.10: PC COMMUNICATION	54
FIGURE 6.1: TEST BOARD.....	57
FIGURE 6.2: DIAGNOSTIC NODE	59
FIGURE 6.3: TRANSFER RATES	64
FIGURE 6.4: MESSAGE TIMING DIAGRAM.....	67

List of Figures

viii

FIGURE A. 1. PROTOCOL TEST SETUP71

List of Tables

TABLE 1.1: SUMMARY OF DATA ON THE ADCS.....	3
TABLE 6.1: COMMUNICATION PERIOD.....	62
TABLE 6.2: FRAME FORMING PERIOD.....	63
TABLE 6.3: RETRANSMISSION PERIOD.....	63

CHAPTER 1

1 INTRODUCTION

The Attitude Determination and Control System (ADCS) stabilizes the satellite and orients it in desired directions during the mission despite the external disturbance torques acting on it [WL99, p.354]. This requires that the satellite determine its attitude, using sensors, and control it, using actuators. This is achieved by reading sensor data and transmitting it to the Attitude Control Processor (ACP). The ACP will then send control data to the actuators. The primary and secondary on board computers (OBC1 and OBC2) can take over most of the functions of the ACP.

1.1 SUNSAT ADCS System Architecture and Limitations

The ADCS of SUNSAT has two main processors, the Attitude Control Processor (ACP), a T800 transputer that implements all the control system software, and the Interface Control Processor (ICP). The latter is a 80C31 based micro-controller which interfaces directly with all actuators, sensors, ACP and the onboard computers (OBC1 and OBC2) as shown in figure 1.1.

The ICP performs mainly four functions, namely.

- ◆ Control the actuators, which include the 3-axis Reaction Wheels (RW's) and the 3-axis Magnetotorquers (MT's).
- ◆ Read the attitude sensors that includes the sun and horizons sensors, the course sun sensors and the magnetometer.
- ◆ Transmit attitude sensor data to the OBC's or the ACP.
- ◆ Receive and decode messages from the OBC's and the ACP.

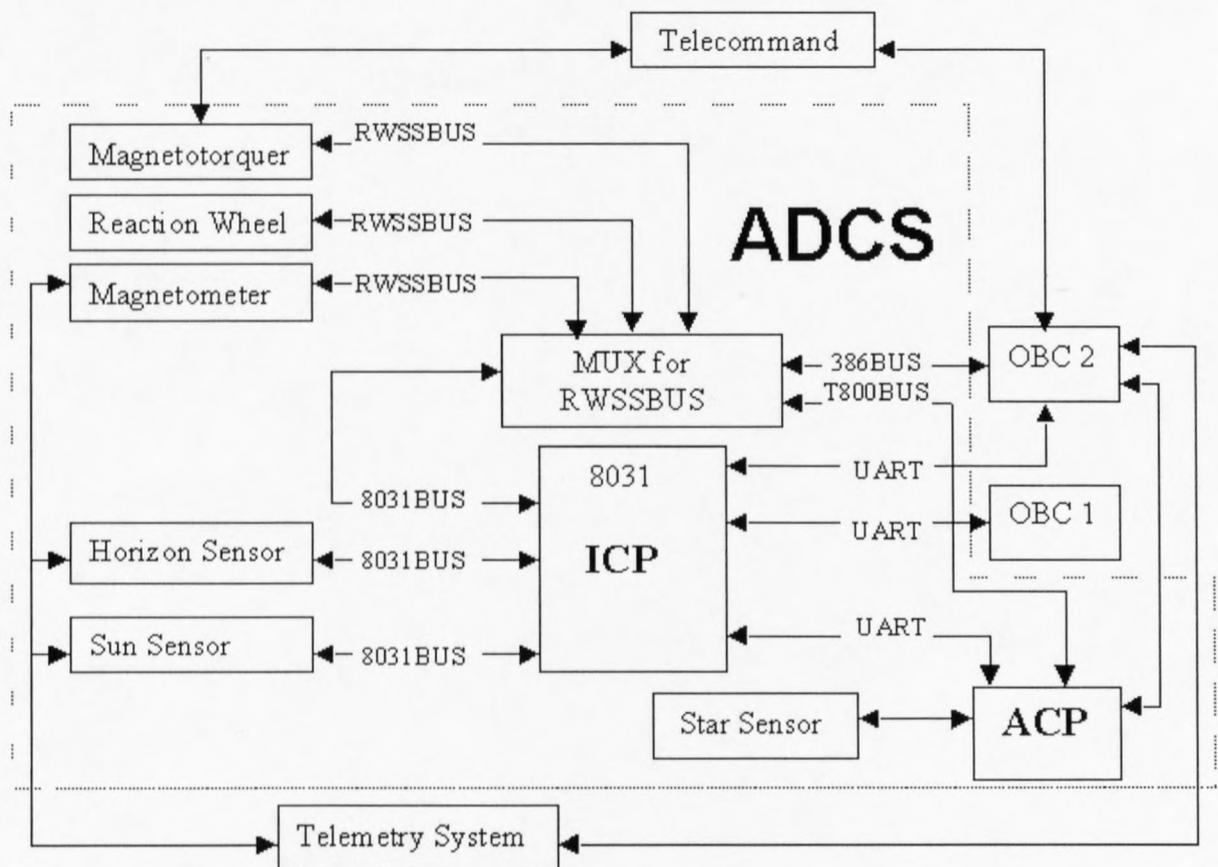


Figure 1.1: Basic Structure of the ADCS

Every second, the ICP receives a data packet containing the control signals for the reaction wheels from the ACP or OBC's and return a packet containing the sensor data to it via a bi-directional UART. Every ten-seconds the ICP receives a data packet containing the control signals for the magnetic torquers from the ACP or OBC's. Table 1 summarized the number of data bytes transmitted between the components of the ADCS.

Table 1.1: Summary of Data on the ADCS.

Source	Destination	Data Bytes	Rate	Description
ACP/OBC's	ICP	02	On Request	Request Sensor Data
ACP/OBC's	ICP	02	Every Tenseconds	MT's Control Data
ACP/OBC's	ICP	08	Every Second	RW's Control Data
ICP	ACP/OBC's	46	Every Second	Sensor Data
ICP	RW's	08	Every Second	Command RW's
RW's	ICP	16	Every Second	RW's Speed Data

Figure 1.1 also shows the four communication routes on the ADCS. The three UART-routes to OBC1, OBC2 and the ACP as well as the general 8031BUS that is connected to a multiplexer and to horizon and sun sensors. The ICP is the central node of communication between sensors, actuators and the ACP as well as the OBC's.

In the event the centralized ICP fails, communication with actuators and magnetometer will be through the multiplexer while the sun and horizon sensor data will be read by the telemetry subsystem. If the multiplexer fails, reaction wheels will be lost and communication with the magnetotorquer will be through the telecommand subsystem. The magnetometer data will be read by the

telemetry subsystem. The use of telecommand subsystem to control the magnetorquer creates overheads for OBC 2.

The ADCS is integrated and the central ICP facilitates communication between actuators, sensors and the ACP or OBC's. Due to the integrated architecture of the system and centralized ICP it makes it difficult to change an interface like the magnetometer or add new interfaces necessary for control systems of the satellite, without changing the whole ADCS system architecture.

These shortcomings can be overcome by decentralizing the ICP and implementing a distributed system. Distributed system provides increased computing power and reduces system complexity [Sla98]. The network for a distributed ADCS is discussed in detail in chapter 2.

1.2 Thesis Overview

The aim of the thesis is to design a communication protocol for a distributed ADCS for SUNSAT 2. The development of the design is structured as follows:

- ◆ Chapter 2 surveys the types of network topologies. The chapter concludes with the choice and motivation of a suitable network topology.
- ◆ Chapter 3 presents background information on communication protocols and media access protocols. It also presents the types of error control and the need for error detection and correction.

- ◆ General specifications of the protocol design are discussed in Chapter 4. The design specifications include the services to be provided by the protocol, messages types and formats. The chapter also defines the protocol layers.
- ◆ Chapter 5 describes software implementation of the protocol designed in Chapter 4. It describes the implementation of error recovery mechanism and media access protocols.
- ◆ Testing and evaluation of the protocol is described in Chapter 6. The chapter presents the test setup and software/hardware integration. Various testing methods, measurements and results are presented.
- ◆ Chapter 7 concludes the thesis by evaluating the results obtained during testing. The chapter investigates the extent to which the initial objectives are met and the future recommendations on the protocol design are discussed.

CHAPTER 2

2 Background

The previous chapter indicated the possible flaws with the current integrated architecture of the ADCS of SUNSAT. This chapter surveys different types of distributed architecture that can be implemented to alleviate the shortcomings mentioned in the previous chapter. The details presented in this chapter are necessary for the complete understanding of the descriptions and reasoning in chapter 4 and form the basis of this thesis.

2.1 Distributed Systems

Distributed system architecture consists of a set of computational nodes and a communication network that interconnects them. In a distributed system, individual processors are assigned fixed, specific task and perform significant processing without the cooperation of the other processors [Lap97, p.283]. A distributed system provides system flexibility and eases integration [UK94]. The

benefits lie in dividing the computations that need to be done among the nodes, and coordinating their activities to achieve higher performance [Sla98]. This coordination, however, requires synchronization mechanisms and protocols to control the system. Coordinating the communication between nodes begins with the network. The next section discusses and evaluates common network topologies

2.2 Evaluation of Network Topologies

Network topology describes how the distributed communities of nodes are physically connected in a network. The three topologies in common used in distributed systems are ring, star and bus networks. This section surveys these network topologies.

2.2.1 Ring Network

A ring network consists of a number of nodes, each connected to two others by unidirectional transmission links to form a closed path as shown in Figure 2.1. Data messages are transmitted in packets, each of which contains the source and destination address field. Messages travel around the ring, with each node reading those messages addressed to it. However, because the ring is a closed loop, a packet will circulate indefinitely unless it is removed. A packet maybe removed by the addressed node. Alternatively, the transmitting node could remove each packet after it has made one trip around the loop. This latter approach is more desirable because it permits automatic acknowledgement and also permits multicast addressing [Sta00, p.444]. Listed below are some of the advantages and the disadvantages of the ring network.

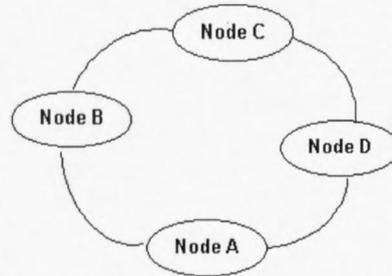


Figure 2.1: Ring Network

Advantages

- ◆ There is no dependence on the central node.
- ◆ Messages travel in one direction so there are no collision of data.

Disadvantages

- ◆ The system depends on the network keeping its ring integrity i.e. if the connection is broken at any point the whole system goes down.
- ◆ The system can be difficult to extent. Problem can arise when adding another node. The whole loop must be broken to add a new node.

2.2.2 Star Network

In a star network, all devices are connected to a central node. Typically, each node attaches to a central node via a two point-to-point links, one for transmission and one for reception. The message transmitted by node has to pass through the central node before it reaches the other node. The central node

acts as a switching device. This requires the central node to manage message traffic on the network. An example of star network is shown in Figure 2.2 below.

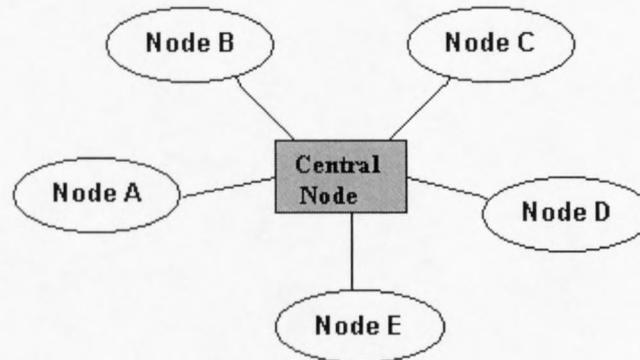


Figure 2.2: Star Network

The star topology has the following advantages and disadvantages:

Advantages

- ◆ Each node is independent from the rest i.e. one malfunctioning node does not affect the rest of the network
- ◆ Centralized control of message switching allows a high degree of security.
- ◆ Transmission speed can vary from one spoke to another.
- ◆ Easy to add and remove nodes

Disadvantages

- ◆ If the central node goes down the whole system goes down.
- ◆ Requires more cabling than other networks

2.2.3 Bus Network

Unlike the star network, the bus network is different in the sense that each node controls its own communications. In a bus network, all the nodes in the network are connected to one cable line, called a bus. The nodes at both ends of the bus are terminated internally or externally. Data messages are broadcast along the bus to all the nodes. Just like the ring network, data packets have the sender and the destination address field. The advantages and disadvantages of a bus network are listed below:

Advantages

- ◆ Nodes can be added or removed without affecting the transmission of data along the network
- ◆ Less cabling is required compared to other topologies.
- ◆ The network can be extended easily.

Disadvantages

- ◆ Nodes transmitting at high speed tend to hog the transmission lines.
- ◆ The nodes share the same medium, thus only one node can transmit at a time.

2.3 Choice and Motivation for a preferred network

The requirements and the nature of the flaws of the current ADCS subsystem of SUNSAT have been taken into account, resulting in the bus topology being chosen. The main motivation in this choice, is the requirement of reliability. There

are a number of potential problems with a ring network. A failure of a node disables the entire system. Since a ring is closed, a means is needed to remove circulating packets with backup techniques to guard against errors. A failure of a central node in star network also disables the whole network. This renders the ring and star networks to be more unreliable than the bus network. In a bus network a faulty node does not affect transmission along the network. This is possible of the node if implemented such that when it fails, it goes to high impedance state.

Bus network allows nodes to be added or removed from the network without affecting the transmission of data. Although this cannot be done when the satellite is in space, it makes it easier in the development of a new satellite to add or remove an interface like magnetometer without changing the whole architecture. The proposed ADCS subsystem on the bus network will consist of processors linked to their respective interface as shown in Figure 2.3 The system will perform the same function as the current ICP. Each processor will perform some of the function of the ICP. These functions depend on the interface the processor is link to as shown in Figure 2.3. For instance, a processor interface to a horizon sensor will read the data from the particular sensor and transmit it to the ACP or the OBC 2 per request.

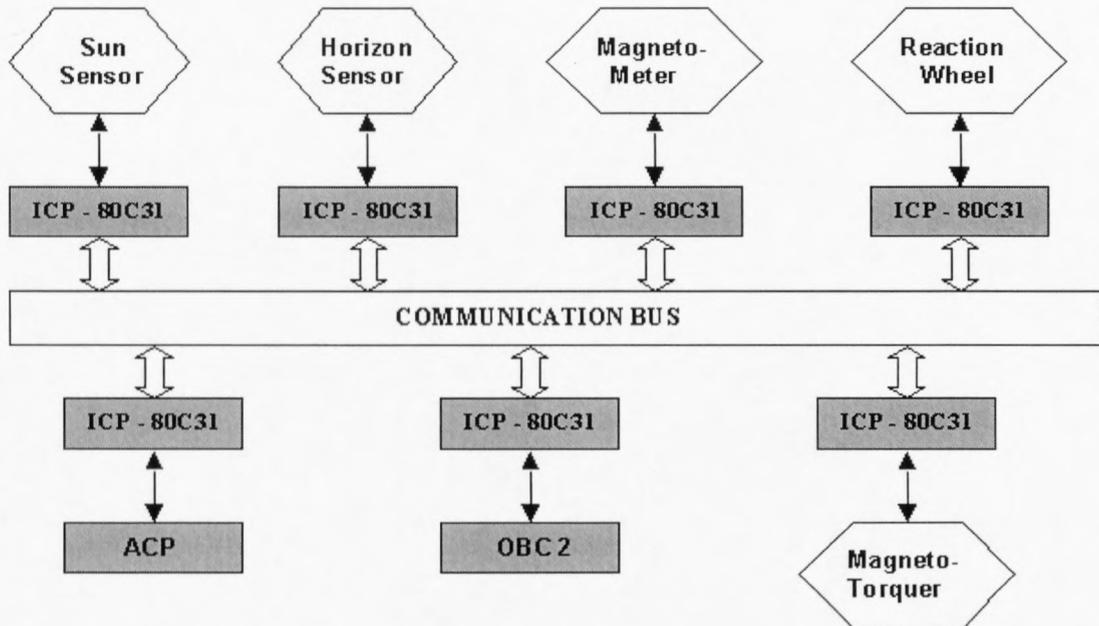


Figure 2.3: Proposed ADCS Bus Network

Following the choice of a network topology, a set of rules that govern the interaction of concurrent processes is needed. A set of rules to exchange information between the nodes on the network and rules which specifies which node is allowed to access the single communication medium at a particular point. The set of rules and the agreement between the sender and the receiver about those rules is called a protocol. This is the main focus of this thesis.

CHAPTER 3

3 Communication Protocols

The chosen network makes use of a shared network. To communicate with one another, the interconnected network nodes must follow some rules concerning which node can transmit at which time. If the messages transmitted over the network are to be intelligible, their syntax and semantics must conform to a set of rules. These rules, along with electrical and physical interconnection specifications, constitute a network protocol.

3.1 Media Access Protocol/Control

Another factor that any network protocol must define is how the communication medium is shared. In embedded systems, three popular medium access control methods are polling, token-bus, and CSMA (carrier-sense/multiple access).

3.1.1 Master Slave Polling

Polling is one of the more popular protocols for embedded systems because of its simplicity and deterministic nature. In this protocol, a centrally assigned master periodically polls the slave nodes, giving them explicit permission to transmit on the network. In some protocols, slaves can send messages to any network node; in others, the slaves can send messages only to the master.

Figure 3.1 shows the polling order (dotted lines) of a simple four node bus network. The majority of the protocol software is stored in the master and the communication work of slave nodes is minimal. This protocol is ideal for a centralized data acquisition system where peer-to-peer communication is not required.

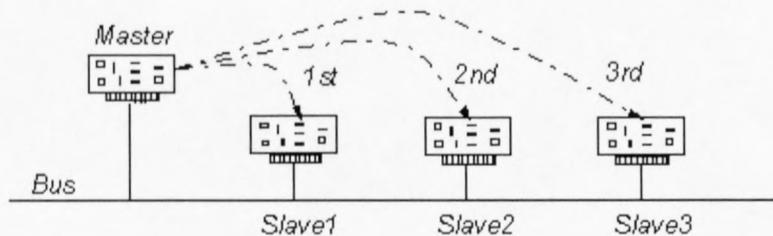


Figure 3.1: Master node polling slave nodes for information

3.1.2 Token Bus

Token-bus is similar to polling in that only one node has permission to transmit at any time. That permission takes the form of a token. The node possessing the token can transmit a packet to any other node. The token passes from node to node in a special packet as if the nodes were connected in a ring or a circular queue. A node that has nothing to transmit passes the token to the next node. There are two time parameters that determine the response of the token bus system [Kop97, p.161], token-hold time and the token-rotation time. The complex part of token-bus protocols is handling transient conditions and errors: how the system starts up, what happens if a token is lost, and procedures for allowing nodes to enter or leave the network.

3.1.3 Carrier Sense Multiple Access (CSMA)

In CSMA, if a node wants to transmit messages, it waits for the network to go idle before transmission. In other words the node listens electronically to the transmission medium whether a frame is currently being transmitted. If a carrier signal is sensed, the node defers its transmission until the passing frame has been transmitted. Only then does the node attempt to transmit the message. However multiple nodes wishing to transmit messages may simultaneously determine that there is no transmission on the network. Consequently they all transmit their messages simultaneously. The messages collide and get corrupted as a result.

The nodes must detect this collision, and resolve it by waiting for a random time before retransmission. One of the nodes informs the other nodes about the collision by continually sending a jam signal for a short period. The involved node then waits for a further short random time before trying to retransmit the affected messages. Examples of CSMA are carrier sense multiple access with collision detection (CSMA/CD) and carrier sense multiple with collision avoidance (CSMA/CA).

3.2 Flow Control

Idle RQ is a protocol that has been defined to enable blocks of information to be reliably transferred to a high probability, without error or replication and in the same sequence as they were submitted over the a serial data link between a sender and receiver [Hal96, p.170]. It is a stop-and-wait protocol that operates in a half-duplex mode since the sender after transmitting information, waits until it receives an indication from the receiving node whether information was correctly received or not. Implicit retransmission and explicit request are two ways in which Idle RQ protocol can be implemented.

In implicit retransmission the receiver acknowledges only correctly received frames and the sender interprets the absence of an acknowledgement as an indication that the previous frame was corrupted. In explicit request, if the receiver detects that the frame has been corrupted, it returns a negative acknowledgement to request another copy of the frame to be transmitted otherwise it returns an acknowledgement for correctly received frames.

In order for the receiving node to discriminate between the next valid frame and a duplicate, each frame transmitted contains a unique identifier known as the sequence number. As a result the receiver must retain a record of the sequence number contained in the last frame it correctly received. In an event a duplicate frame is received, the copy is discarded and the receiver returns a previous acknowledgement frame with the related identifier to resynchronize the sender.

3.3 Error Control

During the transmission of data between two nodes, it is very common particularly when the physical separation is large and transmission lines are in an electrically noisy environment, for transmitted information to be corrupted [Hal96, p.105]. To ensure that the data on the receiving node is the same as that transmitted by the sending node, there must be a way for the receiver to detect errors. Moreover, if errors are detected a mechanism is needed to obtain a copy of the correct information. This is achieved by forward error control and backward error control. Both approaches transmit a frame with additional information so that the receiver can detect errors.

In forward error control, the receiver can detect the presence of errors and also determine the location of the errors in the bit stream. The correct data is then obtained by inverting these bits. However, not all errors patterns can be corrected using the forward error control techniques. As a result, retransmission of messages whose errors were uncorrected is required. In forward error control less bandwidth is used due to fewer retransmission of messages. Reed Solomon and Convolutional codes are some of the forward error control techniques.

Another error control technique is the backward error control. In this technique, the receiver can only detect the presence of errors but not their location. A retransmission control scheme is required to request a correct copy of data to be sent. An example of backward error control is the Longitudinal Redundancy Check (LRC) which only Xor the data bytes.

3.4 Choice of a Protocol and Error Control

An Idle RQ protocol that implements explicit request is more effective than an Idle RQ protocol that implements implicit retransmission. For instance, with explicit request, retransmission of messages occurs immediately upon reception of a negative acknowledgement. In an implicit retransmission, the transmitter waits for time out before retransmission. The wait-time makes the Idle RQ protocols with implicit retransmission to be less effective. An Idle RQ with explicit retransmission is adequate for the time critical ADCS and will be implemented in the design of the protocol.

Data transfer among the communicating components of the current ADCS was taken into consideration when choosing Master/Slave Polling as a media access protocol. For instance, the ICP collect data from the sensors and transmit control messages to the actuators. The ACP or OBC's request data from or send control data to the ICP. The ICP then responds to the ACP or OBC's, depending on which one requested data.

Both the ACP and the OBC's can request sensor data and command the actuators. Therefore both the ACP and the OBC's should have access to communicate with the sensors and actuators. To achieve this a bus network shown in Figure 2.3, should have two master nodes. Since only one node is allowed to transmit at a time, a token passing mechanism is used to give each master node explicit permission to transmit messages. Two master nodes on the network also provide the system with some reliability in case one master node fails. The protocol design will implement Master/Slave polling and token-bus to access the medium.

Forward error control techniques require more overhead bytes for error detection and correction. As a result a detection and correction code will increase the source code thus making message processing to take a long time. Normally these source codes are complex and difficult to implement in the types of processors used in the ADCS, like the 80C51 processor. Backward error control techniques only require error detection code with less overhead bytes. This type of error control techniques is easy to implement on the 80C51 type processor and will be used for data protection in the design.

In conclusion, an Idle RQ protocols with explicit request, the media access protocol and the error control chosen in this section can adequately satisfy the reliability requirement of ADCS. The following chapter describes how they are implemented in the communication protocol design.

CHAPTER 4

4 Protocol Design

All the rules, formats, and procedures that the two communicating nodes agree upon are collectively called a protocol. In a way, the protocol formalizes the interaction by standardizing the use of a communication channel [Holz91, p.20]. This chapter describes the protocol design for the bus communication network. The requirements and the functions of the ADCS of SUNSAT were taken into consideration during the design of the protocol. Section 4.1 and section 4.2 describe the protocol layers and the protocol specification respectively.

4.1 Protocol Layers

In 1979 (ISO 1979) the International Standards Organization (ISO) published a seven-layer Open Systems Interconnect (OSI) model for organizing emerging network communication protocols [Zim80, p.425]. This model separates various functions of communication network protocols into layers so that there is an agreed concept of what functions are to be performed and what the interfaces are between these functions.

This protocol design is mainly concerned with the two lower non-dependent network layers of the OSI reference model namely the data link and physical layers. The exchange of information and how data is appended at the data link and physical layers is shown in Figure 4.1 and the service provided by the two layers are described in section 4.1.1 and 4.1.2 below.

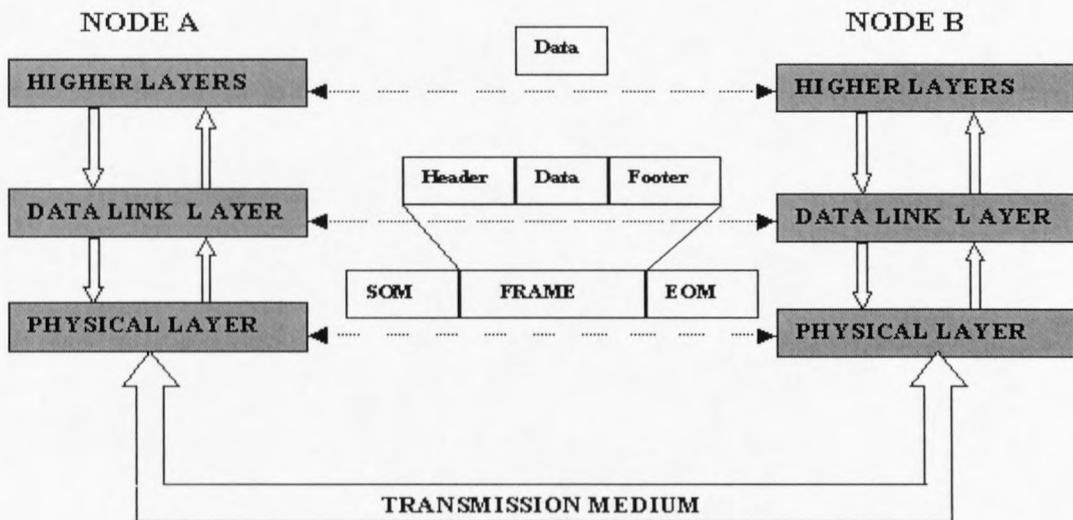


Figure 4.1: Protocol Layer

4.1.1 Data Link Layer

The data link layer attempts to make the physical link reliable and provides the means to activate, maintain, and deactivate the link. It is responsible for functions such as error detection and controls. It provides the higher layers with reliable information transfer facility.

The sending node calculates the checksum and appends it to data as the footer. It also appends a header that consists of the address field and the sequence number together with length. The frame format is described in section 4.2.4. Upon receiving the frame, the data link of the receiving node strips the headers and footers and passes data to the higher layers.

4.1.2 Physical Layer

The physical layer covers the physical interface between devices and the rules by which bits are passed from one to another. This includes the physical properties of the interface to a transmission medium, bit representation and data transmission rate. It also specifies the functions performed by the circuits of the physical interface and the sequence of events by which bit streams are exchanged across the physical medium.

The physical layer receives the data frame from the data link layer and appends the Start of Message (SOM) and End of Message (EOM) bytes. The end and start of messages are unique bytes to indicate where message starts and stops. These two bytes can accidentally occur in a data frame, which causes misinterpretation of the message at the receiving end. To handle this accidental occurrence a mechanism similar to the SLIP protocol is implemented. This mechanism replaces any SOM or EOM bytes found in the data frame with sequences of bytes, according to the following rules:

- <SOM> is replaced with sequence <A>
- <EOM> is replaced with sequence <A><C>
- <A> is replaced with sequence <A><D>

The physical layer of the receiver always checks for these sequences and immediately replaces them with the correct byte and passes the data frame to the data link. The data bits are transmitted serially on the transmission medium via an 8-bit uart at 9600 bits per second.

4.2 Protocol Specification

Protocol specification consists of five distinct parts [Holz91, p.21] namely:

- ◆ Service specification
- ◆ Assumptions about the protocol environment
- ◆ Protocol vocabulary
- ◆ Message formats
- ◆ Procedure rules

These are the essential elements in a protocol definition.

4.2.1 Service Specification

The purpose of the protocol is to reliably transfer data among the communicating nodes on a multi-drop network. The service includes:

- ◆ Recovery from transmission errors.
- ◆ Recovery from duplicate messages and tokens.
- ◆ Acknowledgements of received messages.
- ◆ Limited retransmission of messages.
- ◆ Detection of faulty nodes.
- ◆ Avoidance of deadlocks and time out procedures.

The protocol is to be designed for half duplex transfer of messages, which means only one node is allowed to transmit at a time.

4.2.2 Assumption about the environment

The environment in which the protocol is to be executed consists of two master nodes, the distributed community of slave nodes and a transmission medium. The master nodes request data from the slave nodes with the slave nodes responding to the requesting master within a specified time.

4.2.3 Protocol Vocabulary

The protocol is to be designed for the following types of messages that are of variable lengths. The master nodes will transmit three types of messages:

- ◆ Request messages for slave nodes.
- ◆ Data messages for slave nodes.
- ◆ Token Messages between the master nodes

Slave nodes transmit two types of messages:

- ◆ Data messages to the requesting master
- ◆ Acknowledgement or Negative Acknowledgement messages in case of error messages.

4.2.4 Message Format

Each message transmitted on the network has the format shown in Figure 4.2.

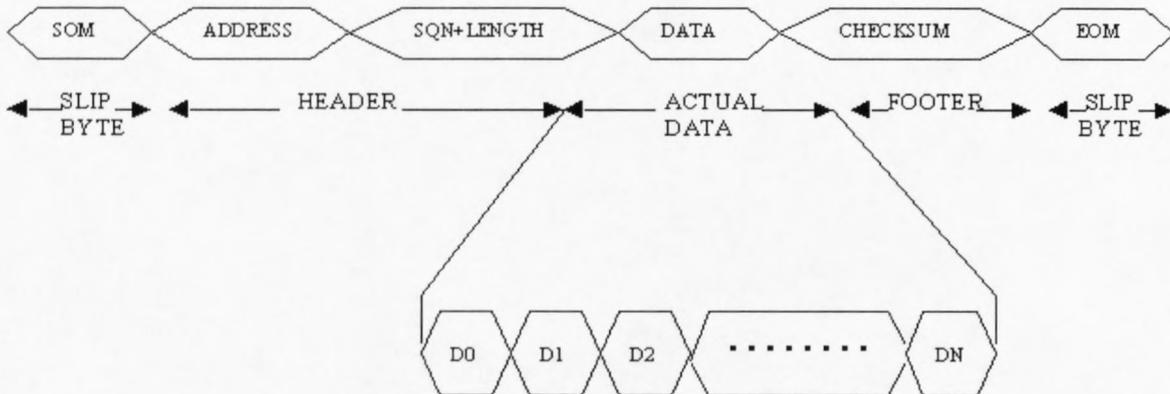


Figure 4.2: Frame Format

Start and End of Message.

The start and end of message bytes are frame delimiters, to enable the receiver to recognize where the frame starts and stops.

Address Field.

The address field is used to identify both source of the frame and its destination. Both addresses uses 4 bits with the source address using the high nibble and the destination the lower nibble of the byte. Messages on the bus network are broadcast, therefore each node on the network has a unique address to only process messages addressed to it. The maximum number of addresses, which implies the maximum number of nodes on the network, is $2^4 = 16$, from 0 to 15.

Sequence Number and Length Field.

The third byte consists of sequence number (2 most significant bits) and the length of the message (6 least significant bits). The message sequence number can only have four numbers 0,1,2,3. To avoid processing the duplicate message, the receiving node checks the sequence number and if it is the same as the one in the previous message, it sends a previous reply with the same sequence number. The sender also sends the length of data bytes to be transmitted which indicates to the receiving node the number of data bytes to expect. If there is no correlation, then the message is discarded.

Data Field.

The data field has the actual data information that is used to convey user data from one end of the link to the other. The length of the data field is expressed in the sequence number and length field and is variable per data message. The maximum data message is 63 bytes. This is due to the message length represented by 6 bits, i.e. $2^6 - 1 = 63$.

Checksum Field.

The footer of the frame is the checksum field. An LRC error detection technique is used to XOR bytes to be transmitted. XORing all the data bytes calculates the checksum. In other words:

$$\text{Checksum} = D0 \text{ xor } D1 \text{ xor } D2 \text{ ----- xor } DN.$$

The transmitting node calculates the checksum and sends them as the footer of the frame. The receiving node calculates the checksum in the same manner as the transmitter and compares it with the received checksum.

Efficiency

Efficiency of transmitting a frame is defined as the number of data bytes to be transmitted divided by the sum of the data bytes to be transmitted and the overhead bytes. In other words:

$$\text{Efficiency} = \frac{\text{Data}}{\text{Data} + \text{Overheads}} \quad (1)$$

Reducing the overhead bytes of the frame increases efficiency of transmitting a frame. The message format shown in Figure 4.2 is reduced by two bytes. The address field consists of the source and destination address and the sequence number and length of the message are combined in one byte giving the frame a total of 5 overhead bytes instead of 7 overhead bytes. The graph in Figure 4.3 shows that the efficiency of transmitting a frame with 5 overheads is about 5% more than the efficiency of transmitting a frame with 7 overheads bytes.

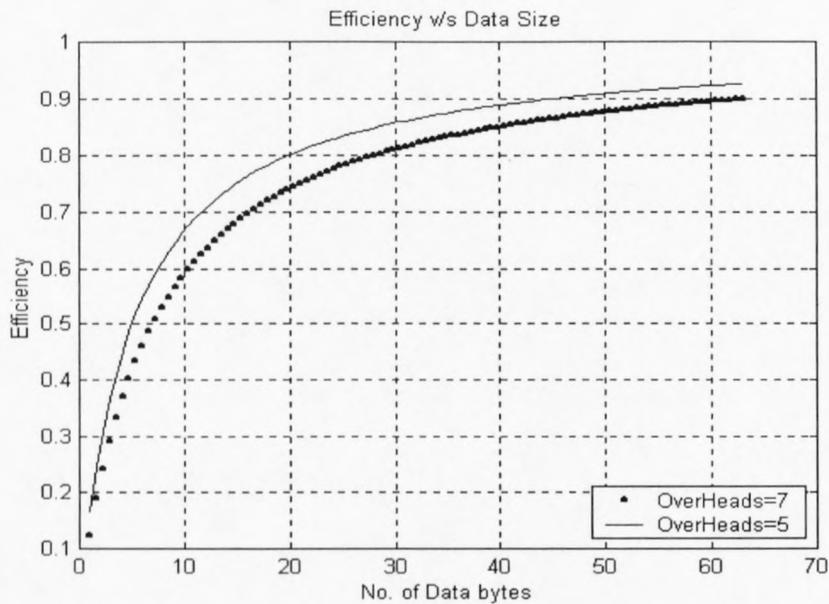


Figure 4.3: Efficiency versus Number of Data Bytes

4.2.5 Procedure Rules

4.2.5.1 Recovery From Transmission Errors

Recovery is a concept distinct from that of error correction [Sta00, p.7]. Recovery techniques are needed in situations in which an information exchange is interrupted due to a fault somewhere in the system. To detect a faulty node on the network, the master node after sending consecutive specified number of messages without a reply from a certain slave node, terminates communication between itself and that particular node. The master node should then relay a message to systems capable of resetting faulty nodes about the fault. These systems will in turn alert the master nodes when the fault has been corrected. Communication is then re-established with the particular node.

Retransmission of messages either due to negative acknowledgement and no response is limited. This technique prevents a node from transmitting messages infinitely to a faulty node or from continuously transmitting incorrect messages. In the event there is a duplicate token whereby both masters transmit at the same time, a higher priority is given to one master.

The sequence diagram shown in Figure 4.4 demonstrates how messages are exchanged between the master node and the slave node. Each message transmitted has a unique sequence number to identify a message. The receiving node must therefore retain the copy of the sequence number contained within the last frame it correctly received.

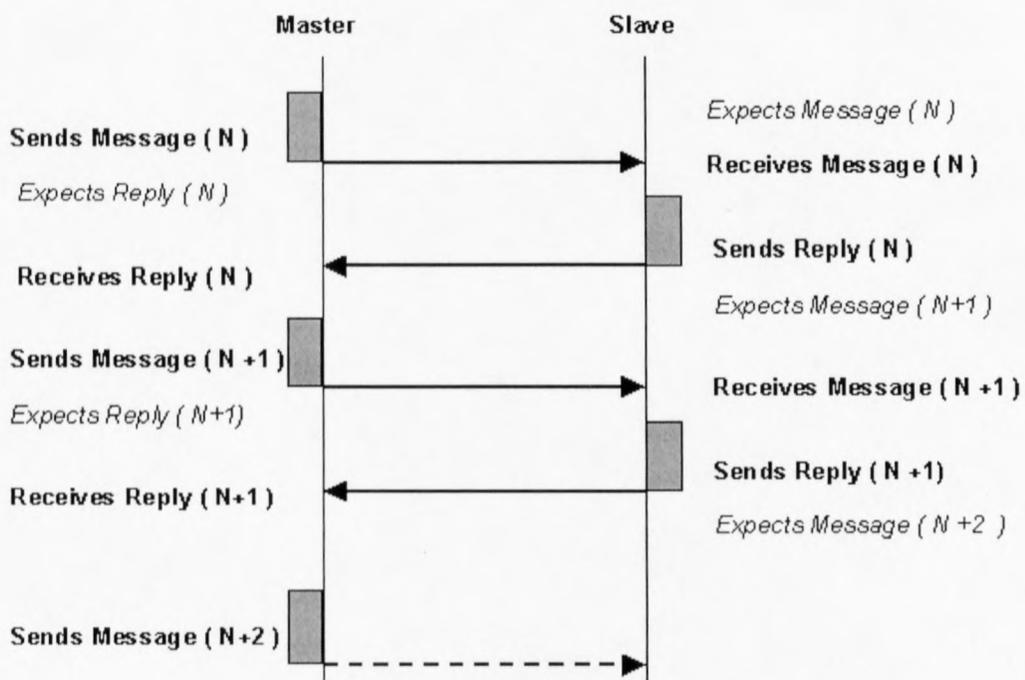


Figure 4.4: Message Transmitted without Errors

However, two sorts of errors could occur. Firstly, the messages sent by the master node may not arrive at the slave node as illustrated in Figure 4.5. The failure of the message to arrive at the other side for example, be due to noise burst. Noise burst may damage messages to an extent that the receiver is not aware that a frame has been transmitted. To account for the possibility of losing messages, the master node is equipped with a timer. After a message is sent, the master node waits for a reply. If no reply is received by the time the timer expires, the message is resent.

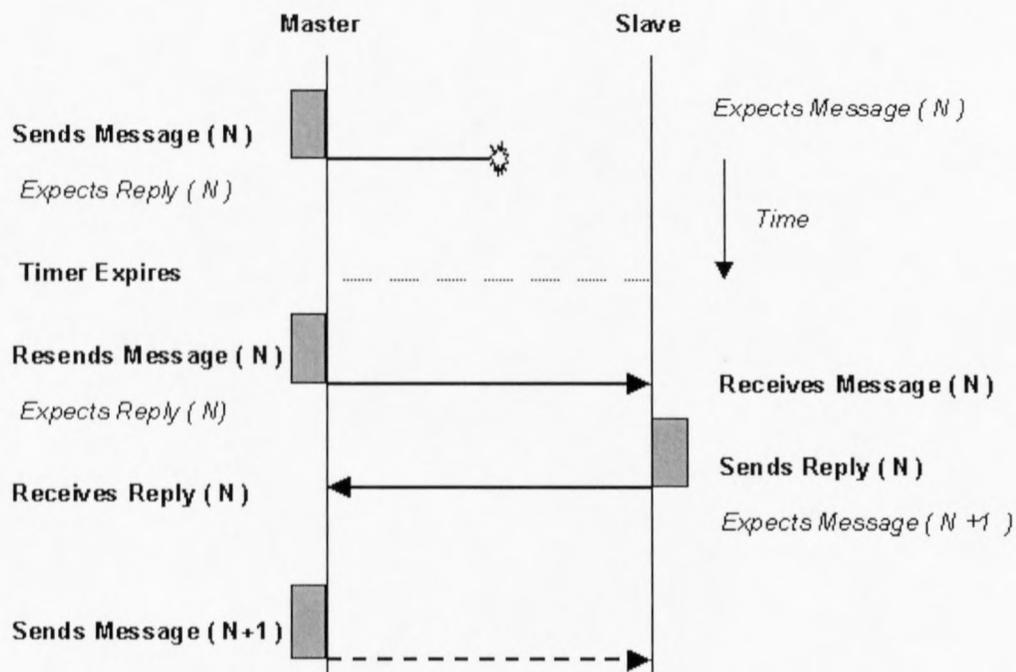


Figure 4.5: Message not received by the Slave Node

The second sort of error is when the reply fails to arrive at the master node. The message sent by master node is successfully received by the slave node. The

slave node then sends a reply, but the reply message gets lost as shown in Figure 4.6. Consequently, upon timer expiry, the master node resends the same message. The duplicate message arrives and is discarded by the slave node because it was expecting a message with a different sequence number. The slave node then resends the previous reply.

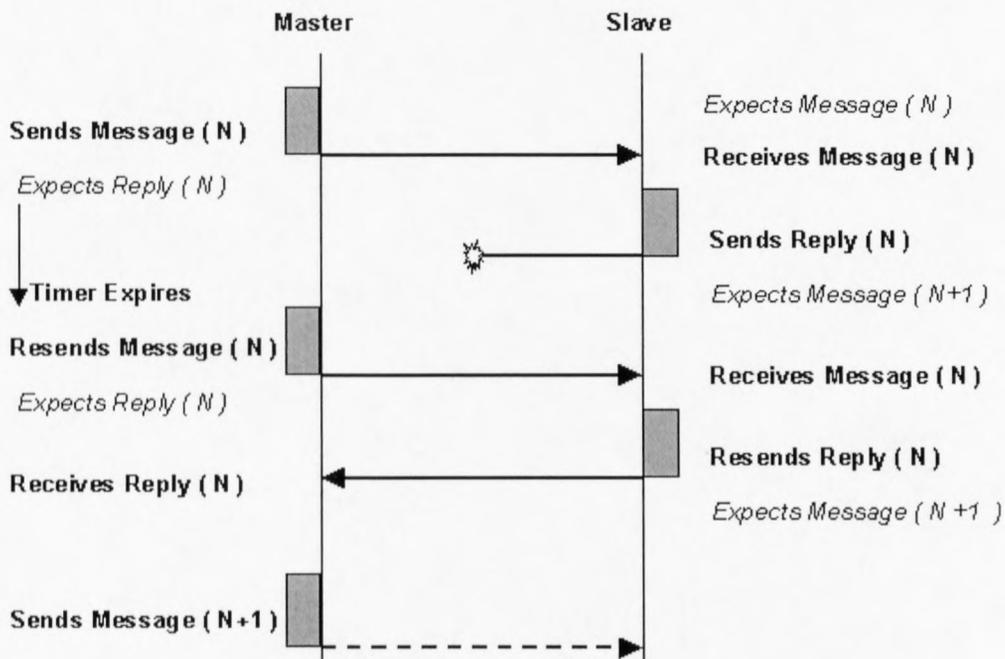


Figure 4.6: Message not received by the Master Node.

4.2.5.2 Token Passing and Generation

To access the media the protocol implements two mechanisms. Master slave polling and token bus. In polling mechanism, a master in possession of a token

request or send data to slave nodes giving them explicit permission to transmit messages to the requesting master. Communication among the slave nodes is not possible. The two master nodes pass the token among themselves to attain mastership.

After sending messages to slave nodes on the network and there are no more messages to transmit, the master in possession of the token, sends a token ready message to the other master, to indicate that the token is available as illustrated in the sequence diagram shown in Figure 4.7. To avoid losing the token, after passing the token the master node (previous token holder) starts the timer and listens to the bus. If the timer expires without any activity on the bus, that master generates a new token and starts with transmissions.

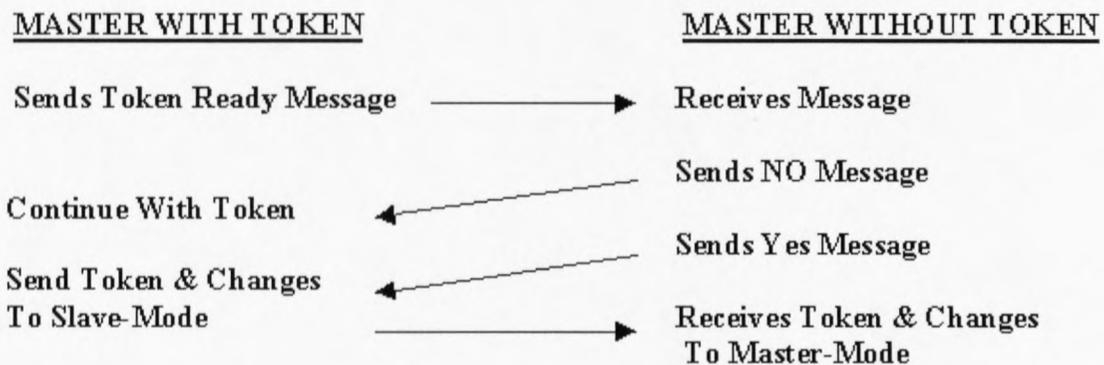


Figure 4.7: Token Passing

When the system is initialized, all the nodes on the network cannot transmit messages. The two master nodes start their timers that have different timeouts. The master with a short timeout generates the token and the one with a longer timeout becomes a slave as soon as it realizes there is an activity on the bus. In

an event there are no messages transmitted on the network, until the master with longer timeout expires, then that master generates the token. The following sections describe the state machines of the protocol.

4.2.5.3 Master Node State Machine

The state flow diagram of the master node is shown in Figure 4.8. It consists of five states: *Idle*, *MakeFrame*, *waitForAck*, *RtxMsg* and *ProcessMsg*. The state machine and the functions of the states are explained as follows:

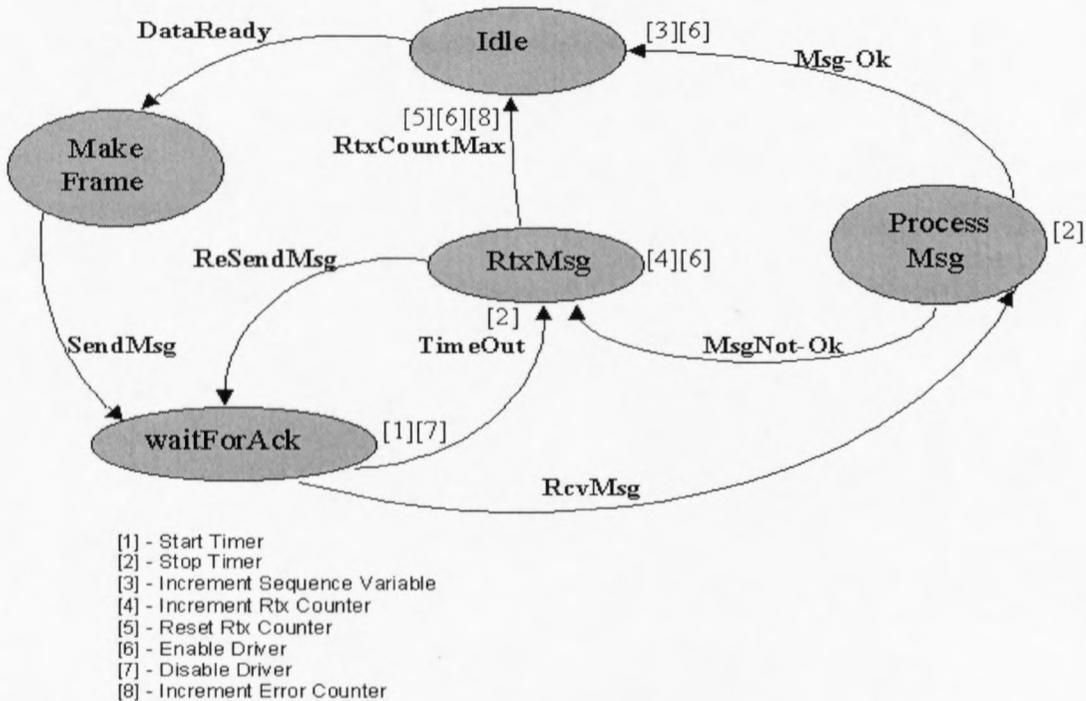


Figure 4.8: Master Node State Machine

The process starts in an *Idle* state. The process is waiting for data to be transmitted. When data is ready, the process moves to *MakeFrame* state.

The function of the **MakeFrame** state is to append headers and footers to information data. On completion, the frame is transmitted and the process changes to **waitForAck** state.

In the **waitForAck** state, the driver is disabled in order to receive acknowledgement. The timer is also started. The process stays in this state waiting for a response. In an event there is no response in a specified time, the process will timeout and move to **RtxMsg** state. During the move from **waitForAck** state to **RtxMsg** state the timer is stopped. If a message is received within the specified time, the process will change to **ProcessMsg** state.

During the **RtxMsg** state, the retransmission counter is incremented and the driver is enabled. The previous message is resent and the process changes to **waitForAck** state. If the retransmission counter reaches maximum, the state will move to **Idle** state and reset the retransmission counter, increment the error counter and enable the driver for the next transmission. If an error counter reaches maximum an error message is sent to the higher layers.

The function of the **ProcessMsg** is to determine the validity of the message. During this state, a timer is stopped. If an invalid message is received the process will move to a **RtxMsg** state. For a valid message the process will change to an **Idle** state. During the move the sequence number variable is increased and the driver is enable to allow for the next transmission.

4.2.5.4 Slave Node State Machine

The state flow diagram of the slave node is shown in Figure 4.9 and consists of four states: *Idle*, *ProcessMsg*, *MakeFrame* and *RtxMsg*. The state machine and the functions of the states are explained as follows:

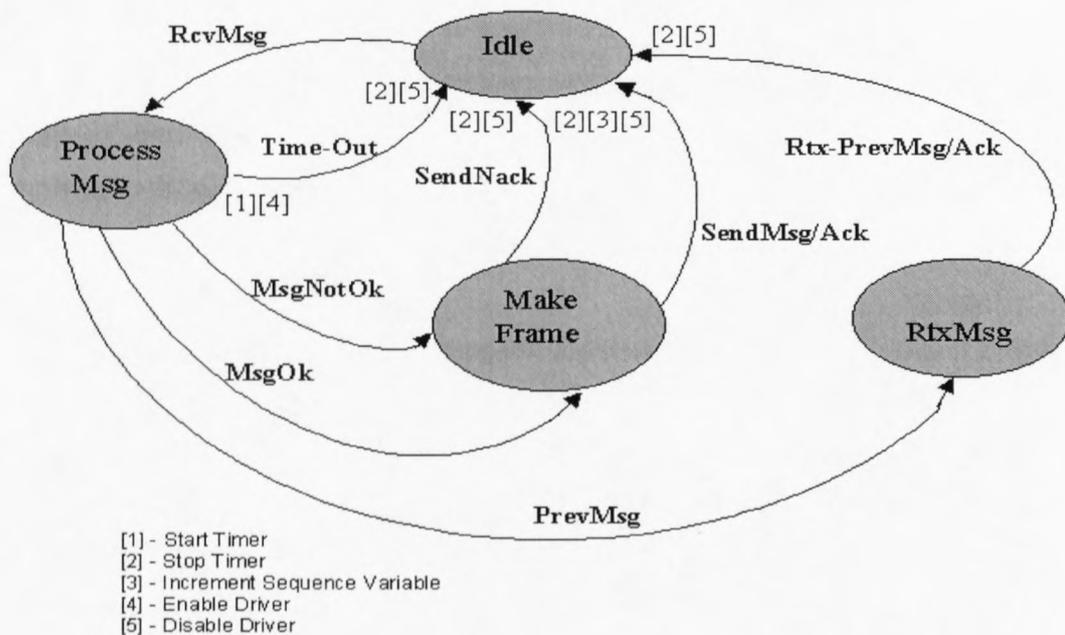


Figure 4.9: Slave Node State Machine

The process starts in an *Idle* state. This is the state the process is in while waiting to receive messages. Upon receiving the message, the process will move to the *ProcessMsg* state. During the move the timer is started and the driver is enabled for transmission.

The function of the *ProcessMsg* is to determine the validity. If the message is valid, i.e. *MsgOk* as shown in the state machine, the state advances to the

MakeFrame state. The state will also move to a **MakeFrame** state if an erroneous message is received. If the message is not processed within a specified time, the process time out and move to an **Idle** state. During the move the timer is stopped and the driver is disabled to allow reception of new messages. If the message received is the previous message, it is discarded and the state moves to **RtxMsg**.

The function of the **MakeFrame** state is the same as that of the Master node state machine. It appends the header and the footer to the message data. If the correct message was received, a header and footer will be appended on the acknowledgement message or data depending on the received message. For an error message, appending is done on a negative acknowledgement. The frame is then sent and the state reset to an **Idle** state. During the move, timer is stopped and the sequence variable is increased. The sequence number variable is only increased if the correct message was received.

The **RtxMsg** state sent the previous message or the previous acknowledgement to the sender and the process moves to **Idle** state. During the move the timer is stopped and the driver is disabled.

In conclusion, this chapter described the protocol layers on which the design is based and their functions. The protocol specification can be considered to be adequately satisfied. The protocol provides the services described in section 4.2.1 and can reliably transfer data among the communicating nodes on the network. The use of error detection and recovery of transmission error mechanisms ensures reliability of data transfer.

CHAPTER 5

5 Protocol Implementation

This section explains the software implementation of the protocol described in chapter 4. Software programs were written in C and Appendix B contains the source code. C provides the ability to write separate software modules with well defined interfaces. This allows for encapsulation and information hiding which promotes modular development. As a result different functions can operate at different layers that are independent of each other. C is also portable since most of the hardware platforms have C compilers.

Figure 5.1 illustrates the flow of data between communicating nodes and also shows where each *function* (procedure) operates on the layers and its functions. The sender could be the master node transmitting data or requesting data from the slave node or the sender could be the slave node responding to the requesting master with data. Communication could be taking place between the two masters passing the token. The one master sending the token and the other being the receiver of the token.

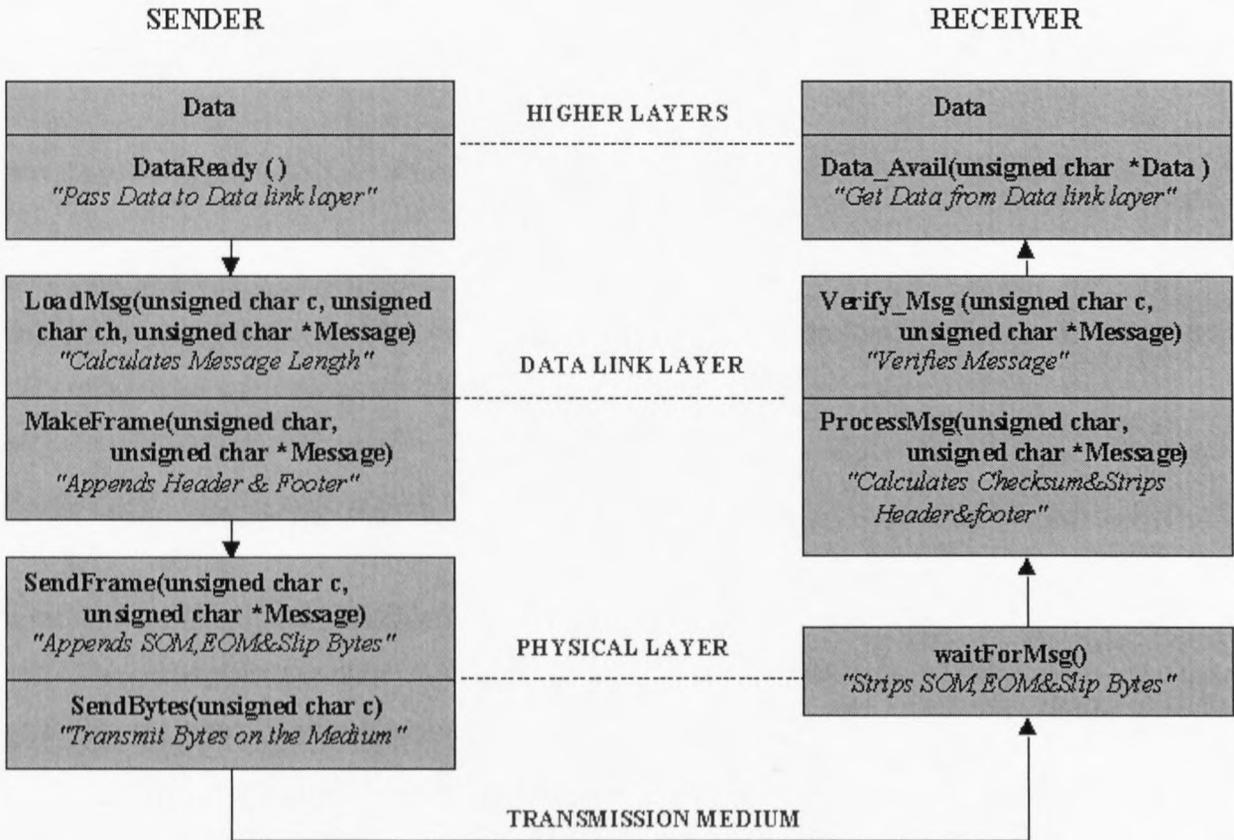


Figure 5.1: Communicating Functions

5.1 Physical Layer Functions

This section explains the functions and processes taking place at the physical layer. The software program implements the serial interrupt service routines and the timer interrupt service routines on the 80C51 type microprocessor.

5.1.1 Interrupt Service Routines

An interrupt service routine (ISR), is a software routine that is invoked by the processor hardware when an interrupt request has been granted. Interrupt requests are automatically vectored to the correct address by the processor, according to the interrupt vector table. A typical *serial interrupt service routine* is implemented as follows:

```

/*----- SERIAL INTERRUPT -----*/
void Serial_Int (void) interrupt SIO_VECTOR {
  if ( RI ) {
    RI = FALSE;           // Clear RI
    if ( ( rbin+1^rbout ) != 0 ){
      RXBUF[rbin] = SBUF; // Store Received Bytes
      rbin = ++rbin & 0X1f; // Rest RXBUF
    }
  }
  if ( TI ) {
    TI = FALSE;           // Clear TI
    CLEAR = TRUE;         // Set Clear Flag
  }
}
/*-----*/

```

Upon receipt of the serial byte, the interrupt flag (RI), which is set by hardware during serial interrupt, is cleared and the byte is stored in the cyclic buffer, *RXBUF*. A cyclic buffer is implemented to allow reception of variable lengths of messages. This byte is then transferred to another buffer to be processed. Section 5.2.3 explains how this byte is processed. When transmitting data, an interrupt flag (TI) is set by the hardware after successful transmission of a byte to allow other bytes to be transmitted. Everytime this occurs TI is cleared.

The processor is also set up to create a timer interrupt every 10ms. The function of the *timer interrupt service routine* is to provide time for time critical functions such as, message transmissions and retransmissions. For instance, some messages are transmitted after every second and some every ten seconds. Typical *timer interrupt service routine* for these functions would be:

```

/*----- TIMER INTERRUPT -----*/
void Timer_Int (void) interrupt TF0_VECTOR {
/*----- 1s & 10s MESSAGE TIMER -----*/
  if ( Msg_Timer ) {
    Second++;           // Increment Second Counter
    If ( Second==100 ) { // Second =100 x 10 ms
      Second=0;        // Reset Second Counter
      SECONDFLAG=TRUE; // Set flag
      TenSecond++;    // Increment TenSecond Counter
      If ( TenSecond==10 ) { // 10 Seconds = 10 x 1s
        TenSecond=0;    // Reset TenSecond Counter
        TENSECONDFLAG=TRUE; // Set flag
      }
    }
  }
}
/*-----*/

```

Since a timer interrupt occurs every 10ms, a second expires after 100 timer ticks. Other time critical functions include the token generation described in section 5.3 and timeouts. To protect against the loss of essential information the sender has to keep track of elapsed time. The sender tries to predict the worst turn around time for each acknowledgment. A timeout occurs when a sending node fails to receive a response within a specified time interval. The two interrupt routines play an integral part in data transfer among different processes.

5.1.2 Physical Layer Transmission

The *functions* operating on physical layer of the transmitting node is the *SendFrame(unsigned char c, unsigned char *Message)* function and the *SendBytes (unsigned char c)*. The *SendFrame* function receives data frame from the data link layer. It transmits the SOM byte, the frame bytes and the EOM byte on the transmission medium by calling the *SendBytes function*. While transmitting the frame bytes, the *SendFrame function* call the *Slip (unsigned char c) function* that checks whether the SOM (254) or EOM (255) bytes didn't occur accidentally in the frame and add the SLIP bytes if SOM and EOM bytes occurred. The SLIP bytes are bytes sent in sequence if SOM or EOM accidentally occur in the actual data. The sequence bytes replacing the SOM and EOM bytes are shown in Figure 5.2. The Figure also shows a <65><68> sequence being sent if byte 65 is in the actual data. This sequence is sent to avoid a situation whereby byte 65 and 66 or 67 are sent sequentially as actual data and thus being misinterpreted by the receiver to be SOM byte and EOM byte respectively.

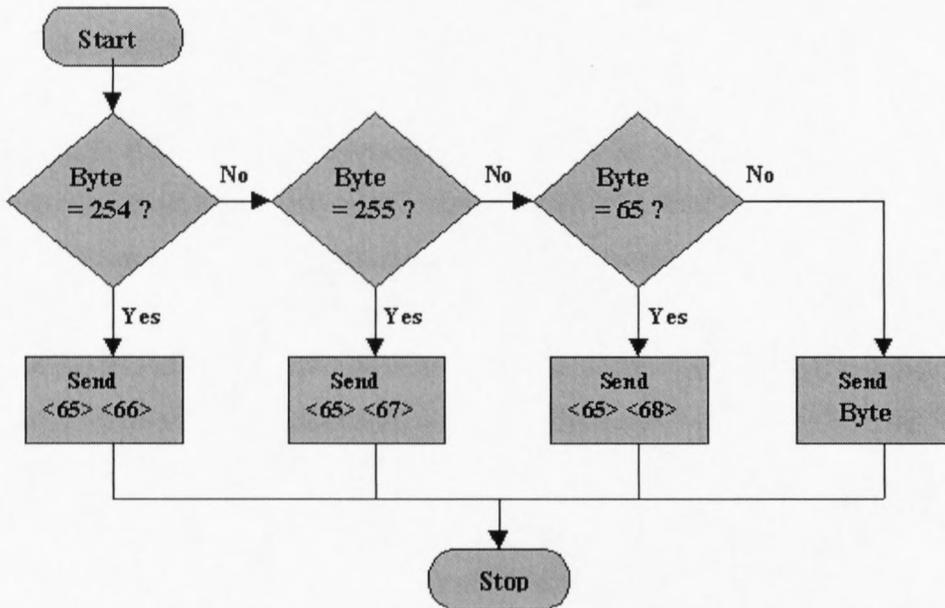


Figure 5.2: Slip Function Flow Diagram

5.1.3 Physical Layer Reception

The *waitForMsg()* function operates on the physical layer of the receiving node. This function strips the SOM,EOM and the SLIP bytes. The *waitForMsg ()* function strips the SLIP bytes by calling *deSlip()* function. *deSlip()* function identifies slip sequence and replaces them with the correct byte as illustrated in Figure 5.3.

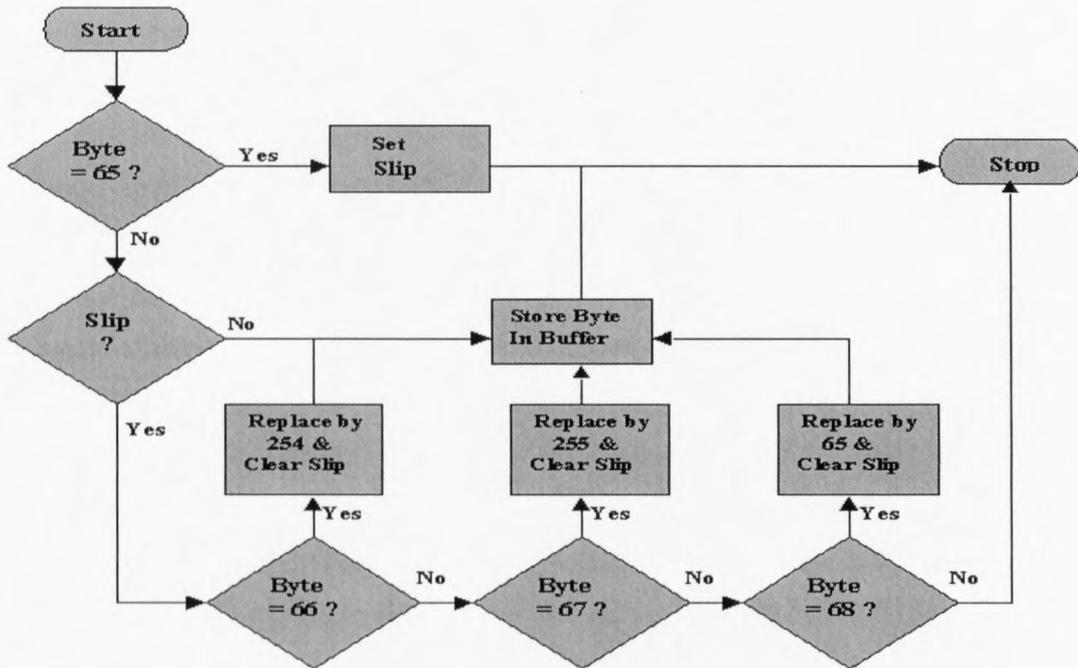


Figure 5.3: deSlip Function Flow Diagram

If byte 65 is received, the next expected byte is either 66 or 67 or 68. Anything other than the three bytes would mean the possibility of data corruption. The erroneous byte is not stored in the buffer thus leaving the data link layer to receive incorrect length of data, hence an incorrect message. Sequences <65><66>, <65><66> and <65><68> are replaced by 254, 255 and 65 respectively as the original bytes send in the data frame. The data frame is passed to the data link layer by a procedure call *ProcessMsg* (*RxLength*, &*PROCBUF*). The *RxLength* is passed to inform the data link layer about how many bytes are to be processed.

5.2 Data Link Layer Functions

The data link layer consists of two functions for both transmitting and receiving node. These *functions* are mainly concerned with error detection and control.

5.2.1 Data Link Layer Transmission

Data to be transmitted is passed from the higher layers to the data link by a procedure call *Data_Ready()* function. The *Data_Ready()* function is the interface function that allows communication between the higher layer and data link layer. It notifies the data link layer when data is ready to be transmitted. It also indicates where the message should be transmitted and when there is no data to be transmitted. This information and the data contained in the *DataReady* buffer is passed to the data link layer.

DataReady buffer is passed by a procedure call *LoadMsg(Addnum, No_data, &DataReady)*. The *Addnum* indicates where the message should be sent and *No_data* indicates when there is no data to be transmitted. The *LoadMsg(unsigned char c, unsigned char ch, *Data)* function calculates the number of data bytes and put them in a *DataBuf*.

The calculated number is then passed as the length of the message together with the buffer (*DataBuf*) containing data by procedure call *MakeFrame(Length, &DataBuf)*. The *LoadMsg()* function also set a flag to indicate where the message is to be transmitted. If there is no data to be transmitted, the *LoadMsg()* function set the *SendTokenflag*. The *SendTokenflag* is used for token passing. The token passing implementation is described in section 5.3.

The *MakeFrame* function calculates the checksum by XORing all the data bytes and then appends the header and the checksum as the footer. The *MakeFrame*

function then passes the whole frame to the physical layer by a procedure call *SendFrame (Length + Overheads, &TxBuf)*. The number of data bytes and the number of bytes appended while forming the frame constitute the length of the frame. This informs the next functions on the physical layer about the number of bytes to be transmitted.

5.2.2 Data Link Layer Reception

Incoming messages are recognized by the SOM byte. The next coming bytes are then store in the *PROCBUF* buffer. The *ProcessMsg(unsigned char c, unchar *Message)* function is called if the EOM byte is detected while storing the bytes. The SOM and EOM bytes are not stored in the buffer. This function reads the buffer (*PROCBUF*) passed by the physical layer (see section 5.1.3) and validates the message using the state machine shown in Figure 5.4.

The state machine consists of four states and starts with the *waitForAddress* state. In this state, the byte to be processed is assumed to be the source and destination address byte. To verify an address byte of a message from SLAVE 1 to MASTER 2, the procedure is as follows:

```
switch(Byte){
/*----- VERIFY ADDRESSES -----*/
  case waitForAddByte:
/*----- SLAVE 1 ADDRESS -----*/
    if ( (ch>>4) == SLAVE1 ) {           // Source Address
      SLAVE1FLAG=TRUE;                 // Set flag
      If ( ( ch&0x0f ) == MASTER2 ) {  // Destination Address
        MASTER2FLAG=TRUE;             // Set flag
        Byte=waitForSequenceNumAndLength; // Wait for next byte
      }
    }
    else{                               // Discard the message
      Byte=endLoop;
    }
  }
}
```

The source address (**SLAVE1**) is verified first by shifting the address byte by four bits to the right. By masking the address byte with 0FH gives the lowest nibble of the byte which represents the destination address (**MASTER2**). If the address byte is valid, the current state is set to *waitForSequenceNumAndLength* state for the next byte (Byte=*waitForSequenceNumAndLength*) otherwise no further message processing takes place. The message is discarded and the current state is moved to *waitForAddress* state.

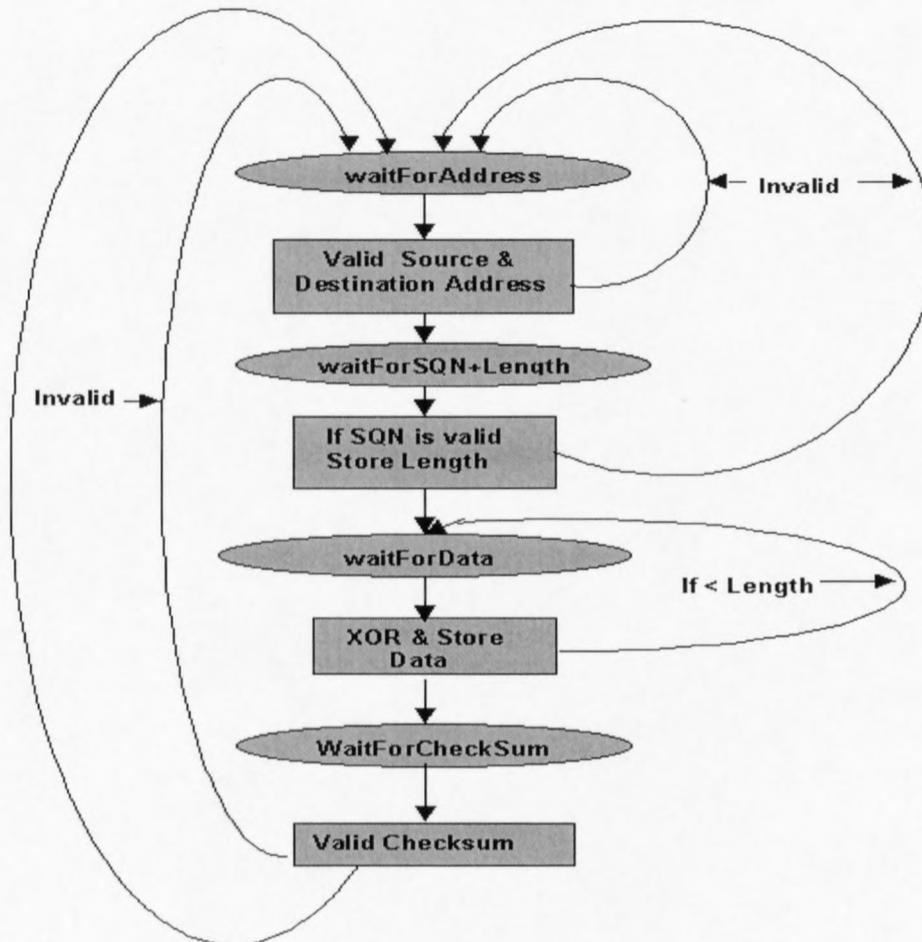


Figure 5.4: Process Message State Machine

In *waitForSequenceNumAndLength* state the byte to be processed is assumed to be the sequence number and the length of the message. This byte is processed almost the same way as the address byte except that, to get the sequence number the byte is shifted by six bits to the right and masked with 3FH to get the length of the message. The message is discarded if the previous sequence number is received and the previous message is retransmitted to resynchronize with the sender. The current state is reset to *waitForAddress* state.

The value of the sequence number can either be 1 or 2. Initially the sequence number is 1 and the sender increments it by one for a new message. If the number reaches 2, it is reset to 1. The receiver acknowledges the receipt of messages by echoing the sequence number. The number must be different from the one previously stored. For the very first message, the receiver accepts any number, 1 or 2 and stores it if the message is valid.

The message will also be discarded if an incorrect length is received and the current state returns to *waitForAddress* state. The difference of *RxLength* minus Overhead bytes gives the length of information data bytes. The length of the message received is expected to be equal to this difference. Note that *RxLength* is the total length of the received frame and overhead bytes. The overhead bytes are the header and the footer of the frame. The header consists of two bytes and one byte footer making a total of three overhead bytes of the frame. If the sequence number and the message length are valid, the message length is stored. The current state is advanced to *waitForData* state.

The *waitForData* state assumes the next coming bytes to be data information. The state machine remains in this state receiving bytes that are equal to the

length received in the previous state while calculating the checksum. The checksum is calculated by XORing all data bytes. The data bytes are stored in the buffer (*BUF*) and the current state move to *waitForChecksum* state.

In the *waitForChecksum* state the calculated checksum from the *waitForData* state is compared with the received checksum to detect errors in the received message. In the absence of errors the current state is reset to *waitForAddress* state, ready for the next frame. If the difference between two checksums is not zero, the message is discarded and the current state is advanced to *waitForAddress* state. The zero difference between the two checksums implies an error free or undetected error in the message.

The message is complete with the reception of EOM byte. The data bytes stored in the *waitForData* state is passed to the *verify_Msg* (*unsigned char c, unsigned char *Message*) function by procedure call *verify_Msg(RxLength-Overheads, &BUF)*. The *verify_Msg(unsigned char c, unsigned char *Message)* function verifies the message by checking whether all the control bytes of the frame are valid. The control bytes include the checksum, destination address, sequence number and length. If all the four control bytes are valid, the received sequence number is stored. The stored sequence number is used when processing the next message to verify whether it is the previous message or the new message.

The *verify_Msg* function then passes the buffer (*DataAvail*) with data bytes by procedure call *Data_Avail(&DataAvail)* to the higher layers. The *Data_Avail* function is an interface function between higher layers and the data link layer for the receiving node. The software programs are contained in Appendix B.

5.3 Token Implementation

Two master nodes are responsible to generate a one byte long token with a value of 16. A token frame has the same format as any message transmitted on the network as shown in Figure 5.5 below. The message format is the same to make the generation of frames and processing of messages standard for every message on the network.

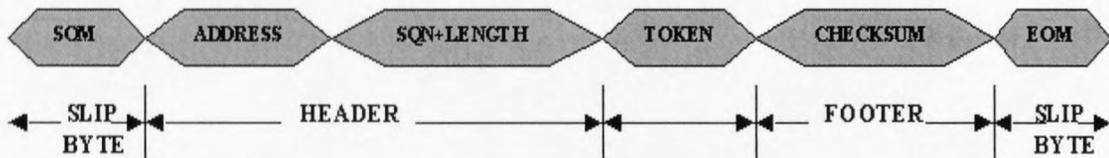


Figure 5.5: Token Frame

To generate the token, the two master nodes go through the process shown in Figure 5.6 during reset. They both start their token timer and listen to whether there are messages transmitted on the network. The time-outs for Master 1 and Master 2 are 1.6 seconds and 3.2 seconds respectively. Under normal conditions Master 1 timer expires first and then generates a token. After generating a token, Master 1 waits for the 1-second message timer to expire.

As a result, the first message transmitted on the network by Master 1 will be after 2.6 seconds. Note that a 1-second message timer is an ADCS requirement and not the protocol requirement. At this stage the bus is active and Master 2 stops the token timer and becomes a slave node and monitors silence on the bus as illustrated in Figure 5.6. In an event of silence for 3.2 seconds Master 2 will generate a token.

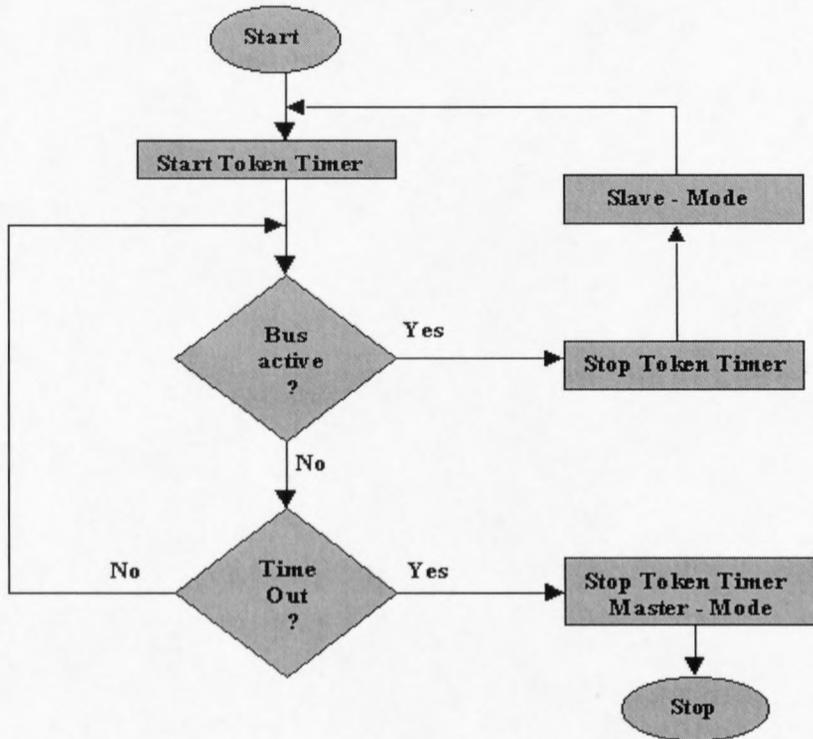


Figure 5.6: Token Generation

Upon completion of transmitting messages to the slave nodes, Master 1 notifies Master 2 about the availability of a token by sending a *Token Ready message*. The *Token Ready message* is a one byte long message and has the same message format as the token. If *Request tokenflag* is set, Master 2 sends back a *Request Token message*, otherwise it sends a *Decline Token message* as shown in Figure 5.7. *Request tokenflag* is set if Master 2 has messages to send.

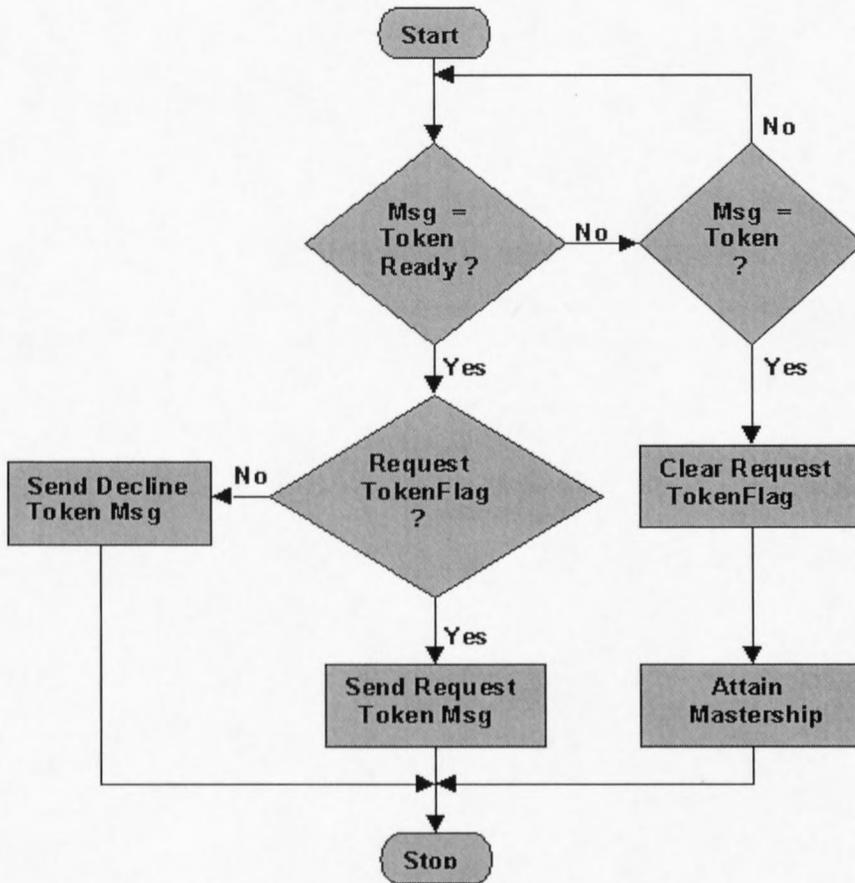


Figure 5.7: Token Passing Flow Diagram

On receipt of a *Request Token* message, Master 1 then sends a token to Master 2 and loses mastership. Upon reception of a token, Master 2 then gains access to transmit messages. Master 2 restarts the token passing process when it has transmitted all its messages. Master 1 retains mastership if it receives a *Decline Token* message.

Token generation and passing implementation described above assumes that the system operates under normal conditions. Token generation mechanism will succeed if both master nodes are switched on at the same time. In the event Master 1 is switched on 1.6 seconds later than Master 2 has been switched, both masters will generate a token and transmit at the same time. Master 2 has less token possession priority. While they are both transmitting, Master 2 keeps on checking whether the duplicate flag is set. The duplicate flag is set if it receives messages from the other master while it is transmitting. This event is demonstrated in Figure 5.8. If the duplicate flag is set, the token is destroyed. The process shown in Figure 5.8 will take place every-time when there is a duplicate token on the network .

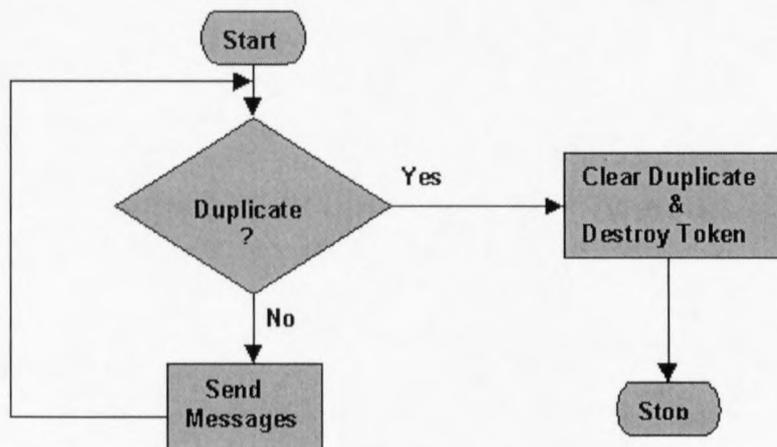


Figure 5.8: Duplicate Token

A token can get lost during token passing process. After sending a token, the sender loses mastership without an acknowledgment of the successful arrival of the token from the other master. As a result, a token will get lost if token transmission is unsuccessful due to transmission errors or node failure. Lost

token leads to network communication failure. To avoid the failure, the master sending the token go through the token generation process illustrated in Figure 5.7. If the token is lost, it generates a new token.

5.4 Implementation Setup

The setup used for the development of software is shown in Figure 5.9 below. The setup consists of three personal computers (PC) connected in an RS485 bus network. The PC setup is used because it allows easy debugging of the source code. Software testing is carried out by the insertion of *probes* into the program. A *probe* is a simple output statement to track the values of variables during program execution.

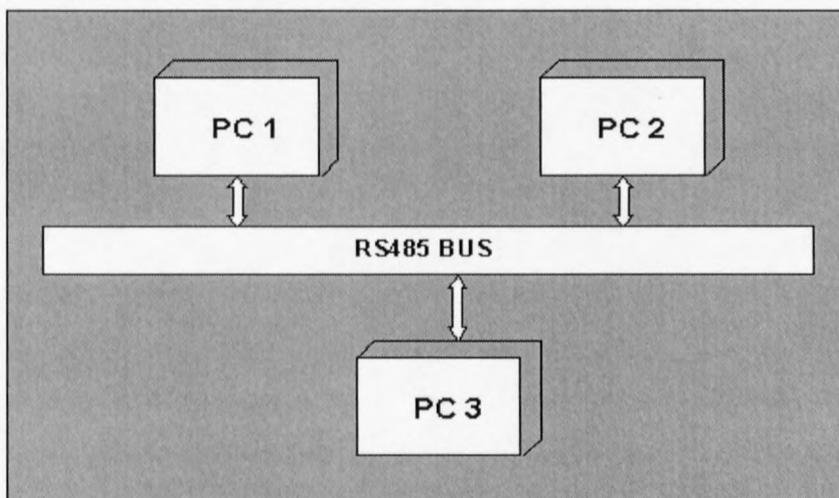


Figure 5.9: Implementation Setup

The MAX232 was used to facilitate communication between RS232 driver of the PC and the RS485 driver. The MAX232 converts the TTL/CMOS input levels into RS232 levels. Data sheets for the MAX232 and RS485 drivers are in appendix A.

A Topspeed C compiler was used to compile the source codes and generate the executable files. PC 1 was used to run the executable file of the Master 1 program while PC 2 and PC 3 ran Slave 1 and Slave 2 executable files respectively. PC 1 transmits messages over the network through the process shown in Figure 5.10. To test communication between two masters, a Master 2 executable file is run on PC 2 with PC 3 running the Slave 1 program. The main focus of testing communication between two masters was token passing and generation.

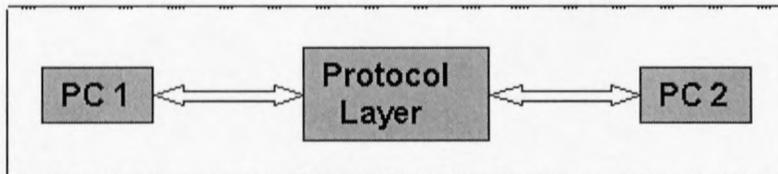


Figure 5.10: PC Communication

After successful development of software on the PC, software was downloaded onto the test board to do testing and to take measurements. The next chapter describes how this was done.

CHAPTER 6

6 Testing and Evaluation

Although testing will flush out errors, this is not the goal of testing [Lap97, p.263]. Testing can only detect the presence of errors, not the absence of them. The goal of testing is to ensure that the software meets its requirements. To determine whether the system provides all specified function. This chapter focuses on the testing of the protocol described in chapter 4. The design details of the test setup are presented.

6.1 Hardware

The network used for testing the protocol software consists of four nodes. The microprocessor, data memory and the RS485 driver are the main components of each node. The test board node is shown in Figure 6.1 and the whole test setup and schematics of test board are contained in Appendix A. The following is the description of the main components of the test board.

- ◆ **Microprocessor** . The microprocessor on the node is an Atmel AT89s8252 clocked at 11.0592 MHz. It is compatible with the industry standard 80C51 instruction set and pin-outs. It is high performance CMOS 8-bit microcomputer with 8k bytes of downloadable flash programmable and erasable read only memory, and 2k bytes of EEPROM. The ATMEL AT89s8252 is a powerful microcomputer that provides a highly flexible and cost effective solutions to many embedded control applications. The on chip downloadable program flash allows program memory to be reprogrammed through the SPI serial Interface or by conventional nonvolatile memory programmer.

- ◆ **Data memory**. A volatile Hitachi HM62265B SRAM is used to store data. It is a CMOS static 32KB 8-bit RAM with a low operating current of 33mA maximum and standby current of 0.2 μ A typical.

- ◆ **RS485 Driver**. The data interface is an RS485 transceiver that facilitates communication between communicating nodes. The DS75176B used is a high speed differential high impedance bus/line transceiver and has the following main features:
 - i. Meets EIA standards RS485 multi-point bus transmission and compatible with RS422.

 - ii. Common mode range permits +/-7V ground difference between devices on the bus.

 - iii. Combined impedance of driver and receiver output is less than one RS485 unit load, allowing up to 32 transceiver on the bus.

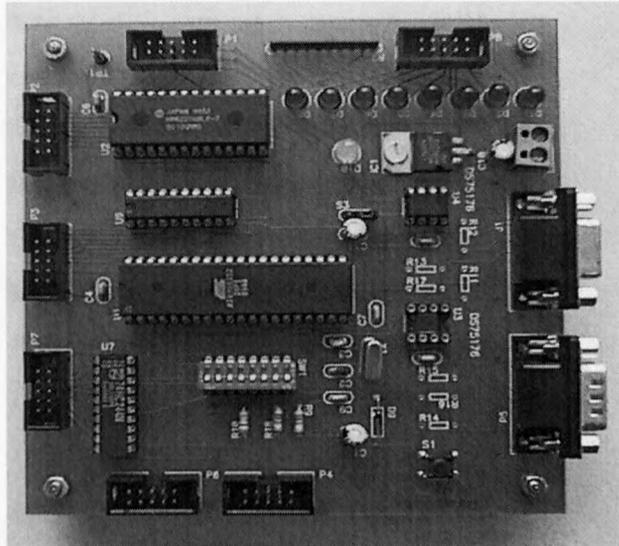


Figure 6.1: Test Board

6.2 System Integration

The goal of integration in real time systems is to unite the parts of the system to create the larger whole in a way that conforms to response-time constraints [Lap97, p302]. Although the hardware and software have been tested and verified separately, the overall timing behavior of the system cannot be tested until the parts are integrated. This section describes the software and hardware integration.

A Topspeed C compiler that was used for the development software in the previous chapter assumes the target platform to be the PC. To test software on the test boards shown in Figure 6.1, Keil C compiler was used to generate code

for such hardware platform. Keil C compiler is a Windows-based environment that takes advantage of all the tools available to minimize the time and effort required during software development [Sch98, p6].

However several alterations were made to the software developed with the Topspeed C compiler. The initialization of communication ports were changed because the serial port registers of the PC are different from the ATMEL A89s5282 registers. A serial interrupt service routine and the timer interrupt service routines were also changed to comply with the ATMEL processor. P1.3 of the ATMEL processor was used to enable and disable the driver and the rest of the software was left unchanged. New software versions were compiled with the Keil C compiler taking advantage of efficient software development where source code can easily be reused as new assignments come along.

Software programs were downloaded from the development environment to the target boards. A PC based programmer (Pony Programmer) was used to program the processor through the SPI serial interface. The on chip downloadable program flash of Atmel AT89s8252 allows program memory to be reprogrammed through that interface.

6.3 Test Description and Fault Injection

Messages transmitted on the network are broadcast. The broadcast communication bus has the advantage that all messages on the bus can be observed by a non-intrusive test monitor. Figure 6.2 shows the diagnostic PC connected to the bus to monitor the activities on the network. Appendix A contains the software program running on the PC.

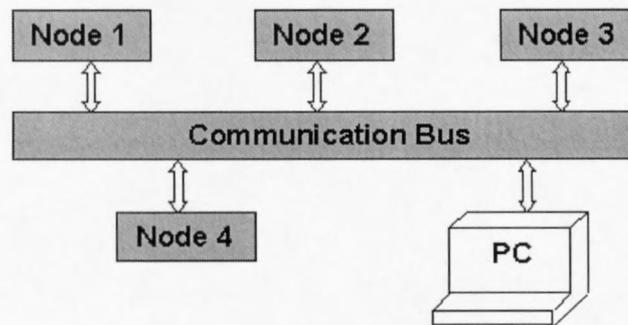


Figure 6.2: Diagnostic Node

Fault injection is the intentional activation of faults by hardware or software means to be able to observe the system operation under fault conditions [Kop97, p253]. Fault injection serves two purposes during system evaluation:

- ◆ Testing and debugging. Under normal operation, faults are rare events that occur only infrequently.
- ◆ Dependability forecasting.

To test whether the protocol can recover from transmission errors, physical level faults and software-implemented faults were injected.

6.3.1 Physical Fault Injection

During physical fault injection the target hardware is subjected to adverse physical phenomena that interfere with the correct operation of the node. A node running the Master 1 program was switched off while in possession of the token. The token is lost as a result of this deliberate action. Master 2 is then expected to generate a new token and continue with normal operations. The Master 1 node is switched on again and automatically becomes a slave node since the Master 2 node will be in possession of token.

The process was repeated but this time by completely disconnecting Master 1 from the network instead of switching the Master 1 node off. Master 2 becomes a master while Master 1 keeps on transmitting message without receiving responses since it is not on the network. Effectively Master 1 is also in possession of the token. By reconnecting Master 1 on the network, both masters are in possession of the token. Duplicate token leads to message collision and creates deadlocks. Master 2 with less priority of token mastership becomes a slave node leaving one token on the network.

6.3.2 Software-Implemented Fault Injection

In software-implemented fault injection, errors are seeded into the software program. These seeded errors mimic the effects of design faults in the software. For instance, instead of transmitting a calculated checksum byte, a wrong checksum byte can be transmitted thus creating an incorrect message. The receiving node discards incorrect messages and sends a negative acknowledgment. Upon receipt of negative acknowledgement, the transmitting master is expected to retransmit the message.

Switching or disconnecting a node from the network or simply assigning it an unknown address by software means, creates a faulty node. The node becomes faulty in a sense that, the transmitting master will not receive response messages from this particular node for a specified number of times.

6.4 Measurements and Results

Measurements were done with the basic two-channel digital real time oscilloscope (Tektronix, TDS 210). A *probe* statement was inserted into a software program to toggle a bit. An execution period of a function can be measured using a typical probe statement as illustrated in code extract below.

```
/*----- LOAD MESSAGES -----*/  
void loadMsg(unsigned char *Data){  
    Test=TRUE; //Test = Pin 1.3  
    Length=0;  
    while(*Data!=0){  
        DATABUF[Length++]=*Data;  
        Data++;  
        Test=FALSE; //Test = Pin 1.3  
    }  
}  
/*-----*/
```

The time difference is measured at pin 1.3 of a microprocessor with an oscilloscope probe. The time difference is the approximate time the *loadMsg* (*unsigned char *Data*) function takes to execute. The measurement results presented in this section were obtained using the same technique. Most of the

measurements taken were influenced by the current operations of SUNSAT ADCS.

6.4.1 Results

The results listed in Table 6.1 were obtained to determine the communication period between two nodes at 9600-baud rate. The data bytes used during these measurements were taken from Table 1.1. These are the data bytes used by the current ADCS. An assumption was also made that, for a distributed system some data bytes will be reduced in the sense that each interface data will be collected independently. For example 46 bytes of sensor data transmitted by ICP to the ACP/OBC's every second, includes the following:

Magnetometer Data = 8 bytes.

Horizon Sensor Data = 4 bytes.

Sun Sensor = 2 bytes.

16 channel A/D converter Data = 16 bytes.

Reaction Wheel Speed and Reference Counter Data = 16 bytes.

Table 6.1: Communication Period.

Description	No. of Bytes Transmitted	No. of Bytes Received	Period	Units
Master - Slave 1	01	08	22.3	ms
Master - Slave 2	01	04	18.3	ms
Master - Slave 3	01	02	14.6	ms
Master - Slave 4	08	16	41.3	ms
Master - Slave 5	01	16	40.1	ms
Master - Slave 6	02	01	18.6	ms

General measurements were made to assess the network performance. The results of these measurements are listed in Table 6.2 and 6.3. The results in Table 6.2 were obtained for transmission of message onto the network only. The purpose of this measurement was to determine the period of forming the message frame and sending it.

Table 6.2: Frame Forming Period.

Description	No. of Bytes	Period	Units
Short Message	01	6.6	ms
Average Message	32	42.9	ms
Long Message	63	78.8	ms

The results listed in Table 6.3 were obtained to determine the period of retransmission of messages.

Table 6.3: Retransmission Period.

Description	Rtx = 1	Rtx = 2	Rtx = 3	Units
Minimum Message	120	240	360	ms
Average Message	155	310	465	ms
Maximum Message	190	380	570	ms

Using the Long message (63 bytes) on both the requesting node and the replying node the following results were obtained:

1. The time the requesting node has to wait before retransmission = 92 ms.
2. The total communication period between the nodes = 176 ms.

Successful transmission and passing of a token = 920 ms.

6.4.2 Transfer Rates

The maximum transfer rate is a function of transmission speed and the number of nodes on the network. Consider results in Table 6.1, if the ACP as the master node request sensor data at 9600 bits/sec from magnetometer, horizon and sun sensors as slave nodes (see Figure 6.3 below), then the maximum transfer rate of each slave node is as follows:

The total transmission time = 55.2 ms

Slave 1 = 144 bytes/sec

Slave 2 = 72 bytes/sec

Slave 3 = 36 bytes/sec

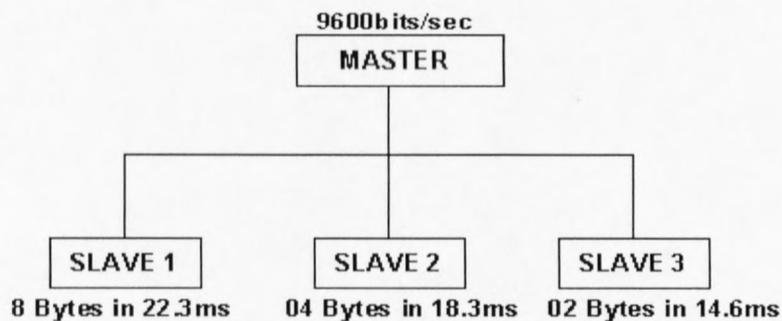


Figure 6.3: Transfer Rates

Increasing the number of nodes on the network decrease the maximum transfer data rates. If slave 4 is added to the network and transmitting 16 bytes in 41.3 milliseconds (see Table 6.1), then the maximum transfer rate for each node is as follows:

The total transmission time = 96.5

Slave 1 = 80 bytes/sec

Slave 2 = 40 bytes/sec

Slave 3 = 20 bytes/sec

Slave 4 = 160 bytes/sec

Thus the number of the nodes on the network affect the data rates.

6.4.3 Memory Loading

With memory becoming denser and cheaper, memory-loading analysis has become less of a concern [Lap97, p.234]. Still, its efficient use is important in satellite systems where power and cost are important limiting factors. However, this section only determines how much memory was used and not how efficiently it was used. The total memory-loading is typically the sum of the individual memory-loading for the program, stack, and RAM areas [Lap97, p.235]. That is,

$$M_T = M_P \cdot P_P + M_R \cdot P_R + M_S \cdot P_S + M_{EXT} \cdot P_{EXT} \quad (2)$$

where M_T is the total memory loading, M_P , M_R , M_S and M_{EXT} are the memory loading for the program, RAM, stack areas, and external memory respectively, and P_P , P_R , P_S and P_{EXT} are the percentages of total memory allocated for the program, RAM, stack areas and external memory respectively. The total memory loading was calculated to be 0.1% from the total memory of 41 kilobytes. This implies that there was more memory available than required.

6.5 Evaluation

In the current ADCS, reaction wheels are commanded and ADCS sensors are read by the ICP every second, and sensor data is transmitted to the ACP/OBC's by the ICP. After ten seconds, the magnetic torquers are commanded. Consider results in Table 6.1 and assume the following:

Slave 1 represents magnetometer.

Slave 2 represents Horizon Sensor.

Slave 3 represents Sun Sensor.

Slave 4 represents Reaction Wheel.

Slave 5 represents 16-channel A/D converter

Slave 6 represents magnetic torquers.

The process of communication between the master node and slave nodes (1-5) which is performed every second takes about 136.6 ms. The timing diagram shown in Figure 6.4 illustrates this process.

After every ten seconds, two bytes of control data are transmitted to the magnetic torquers. Magnetic torquers respond by sending a one-byte long acknowledgement message. The results in Table 6.1 (Master-Slave6) shows that, it takes about 18.6 ms to complete this process without transmission errors. The total communication time between the master node and six slave nodes is 115.2 ms. This will allow the ACP to collect sensor data, 16 channel A/D converter data, and reaction wheel speed and reference data within a second. It will also send command data for reaction wheel and magnetic torquers within the same time.

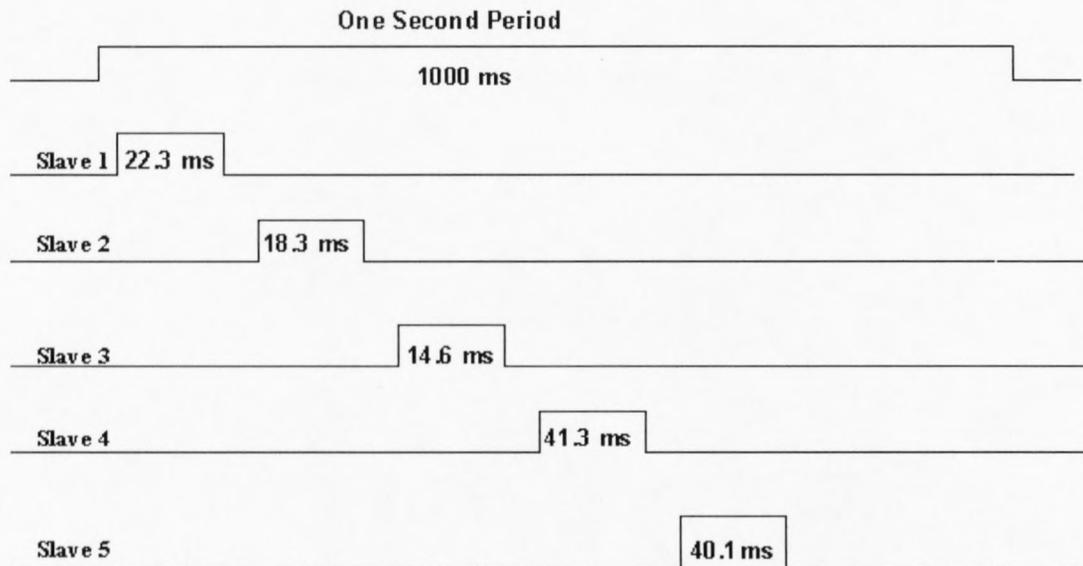


Figure 6.4: Message Timing Diagram

The message frame for all messages transmitted on the network has five overhead bytes. The message transmission efficiency on the network is 94%, 86% and 20% for the long, average and short messages respectively.

In conclusion, the results obtained in this section have shown that the functional requirements of the current ADCS are met. For instance, it takes a master node about 136.6 ms to communicate with five slave nodes, which is within the one second period specified for the system. This leaves enough time for the master to send command data to magnetic torquers and to receive back an acknowledgement every ten seconds. Furthermore the general network performance results are provided, which include frame forming periods and retransmission periods for various message lengths as well as data rates.

CHAPTER 7

7 Conclusion and Future Research

A communication protocol for a distributed system was designed. The distributed nature of the system coupled with the need for half-duplex communications resulted in a set of procedure rules to facilitate the exchange of messages between different nodes. The layering model (OSI) was used to separate various functions of the protocol into layers to define what functions are to be performed and what the interfaces are between these functions.

The two lower layers of the OSI reference model were implemented, i.e. the data link and physical layers. Error detection in the form of LRC and error control such as retransmissions were implemented on the data link layer to ensure reliable transfer of data. The physical layer is mainly concerned with the addition of delimiters to frames to allow message recognition at the receiving end.

The design done here can be considered functionally adequate for the application. The protocol specifications described in chapter 4 are met. The protocol can reliably transfer messages on the RS485 bus network as

demonstrated during testing. It meets the reliability requirement of the ADCS system by recovering from transmission errors through retransmission of messages, detecting duplicate messages and token, and detecting faulty nodes on the network. The protocol also implements two master nodes to enhance reliability. If one master fails, the other master takes over control of the network.

The protocol allows the addition of new nodes on the network without changing the system architecture. This feature is advantageous for the design of a new satellite with a new interface to be added on the system. For instance a star sensor node can be added to the network without changing the architecture of the network. In the current integrated architecture of the ADCS it is difficult to add a new node without changing the whole system architecture.

A design can never be seen as optimal in all senses. The protocol design presented in this thesis form the bases for future designers as to how they think it can be better optimized or even redesigned radically. For instance, the transmission of a one byte long message has an efficiency of 20%. This is due to the number of overhead bytes appended on the message frame. The most affected messages are token ready message, token request and decline token message, and the token. An investigation on how to improve the data efficiency should be carried out.

One of the limitations of the protocol is the implementation of detecting a duplicate token efficiently. The process of detecting a duplicate token is difficult to implement. When both masters are transmitting simultaneously, collisions occur and as a result the messages are unrecognizable. Since the master with less priority detects a duplicate token when it receives messages from the other master while it is transmitting, then with collisions it will not recognize these

messages. However this process becomes possible while the master with less priority is in a wait for acknowledgement state and there is no collision between the other master and the responding slave node. Focus should be on this area during the future development and implementation of this protocol.

Further more, although not likely, if the nodes have to be increased to more than 16, two address bytes will be needed. One byte for the source address and the other for the destination address, thus increasing the number of overhead bytes. Another limitation of the protocol is that, the nodes can transmit a maximum 63 bytes of data. If more than 63 bytes is to be transmitted, the sequence number and length field byte should be separated into two bytes. Then, the maximum length of the message will be $2^8 = 256$. Alternatively a windowing mechanism can be implemented in the protocol. Finally, the protocol can be redesign to support a dual bus network thus making it more reliable.

Appendix A

A. Protocol Test Setup and Data Sheets

A.1 Test Setup

Four identical test boards were connected in an RS485 bus network as shown in Figure A.1 below. The main components of the boards are the ATMEL AT89s5282 microprocessor, Hitachi HM62265B SRAM and DS75176B RS485 transceiver. A PC was connected in the network as non-intrusive device to monitor the network activities. It was mainly for diagnostic purposes and the software used for this purpose is presented in section A.2

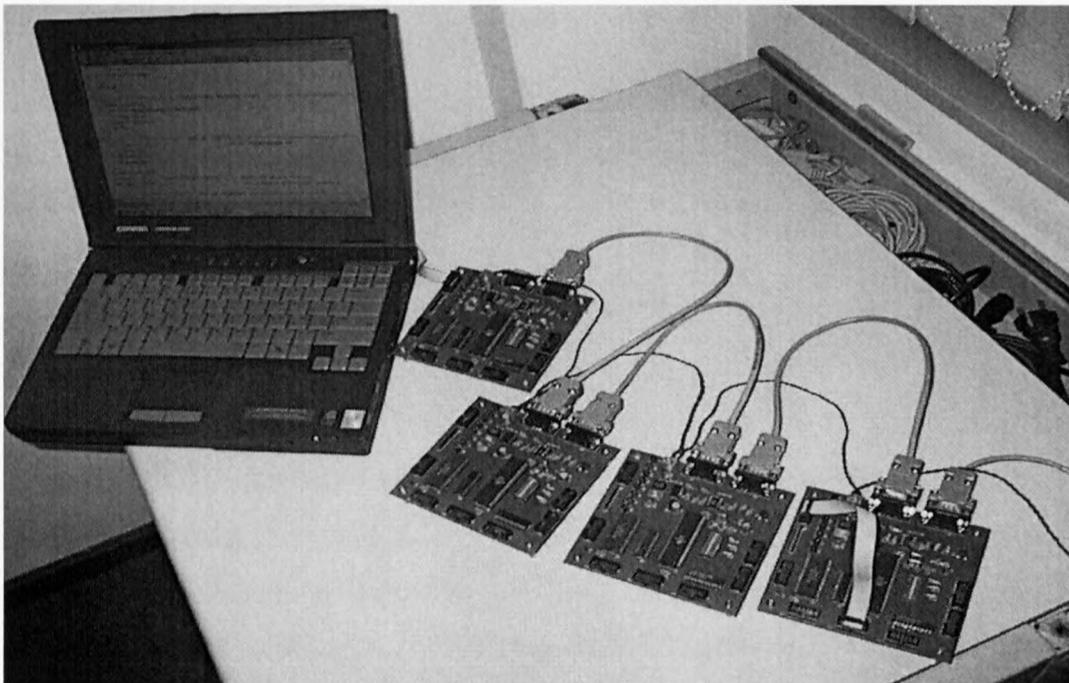


Figure A. 1. Protocol Test Setup

A.2 Diagnostic Source Code

This section consists of the Diagnostic Source code. It was generated with a Topspeed C compiler. The source code was used to monitor the activities of the proposed ADCS bus network.

```

/* Name: (NWM.C), Diagnostic Program
Descr: This program monitors the activities of the RS485 Bus Network.
/*----- Include Files -----*/

#include <dos.h>
#include <stdio.h>
#include <conio.h>

/*----- Define Constants -----*/

#define PORT1 0X3F8                /* COM1 Serial Port Base Address */
#define INTVECT 0X0C              /* Serial Interrupt Vector Address */
#define TIMVECT 0X1C              /* Timer Interrupt Vector Address */
#define STX 254                    /* Start of Frame */
#define ETX 255                    /* End of Frame */
#define MASTER1 1                  /* Master1 Address */
#define MASTER2 2                  /* Master2 Address */
#define SLAVE1 3                    /* SLAVE1 Address */
#define SLAVE2 4                    /* SLAVE2 Address */
#define SLAVE3 5                    /* SLAVE3 Address */
#define SLAVE4 6                    /* SLAVE4 Address */
#define SLAVE5 7                    /* SLAVE5 Address */
#define Slip1 65                   /* Caps A */
#define Slip2 66                   /* Caps B */
#define Slip3 67                   /* Caps C */
#define Slip4 68                   /* Caps D */
#define ACK 06                     /* Acknowledgement */
#define NACK 21                    /* Negative Acknowledgement */
#define TOKEN 15                   /* Token Message */
#define TOKENREADY 15              /* Token Available Message */
#define TACK 05                    /* Token Acknowledgement */
#define TNACK 22                   /* Negative Token Acknowledgement */
#define SIZE 65                    /* Define Buffer Length */
#define overheads 3                /* Define Overheads Bytes */

#define TRUE 1
#define FALSE 0
#define boolean int
#define uchar unsigned char

/*----- Declarations -----*/

uchar BUF[SIZE];
uchar RXBUF[SIZE];
uchar PROCBUF[SIZE];
uchar DATABUF[SIZE];

boolean MASTER1FLAG,MASTER2FLAG,SLAVE1FLAG;

```

```

boolean NO_ERRFLAG,VALID_LENGTH;
boolean Msg_ok,txflag,spflag1,GotSTX,Run_Timer,Time_Out;

uchar RxLength,RxChecksum;
uchar rbin,rbout,fin;

/*----- Define States and Events -----*/

enum States {Idle,ProcMsg};
enum Events {noEvent,RcvMsg};
enum Bytes {waitForAddByte,waitForSequenceNumAndLength,waitForData,
            waitForChecksum,endLoop};

enum Bytes Byte;
enum States State;
enum Events Event;

FILE *fp;                                // Declare File Pointer

/*----- Define Interrupt Vector Pointer -----*/
void (interrupt far *old)();

/*----- Serial Interrupt Service Routine -----*/

void interrupt far PORT1INT(){
if((rbin+1^rbout)!=0){
  RXBUF[rbin]=inportb(PORT1);
  rbin=++rbin&0X1f;
}
outportb(0X20,0X64);                    /* Reset 8259 */
}

/*----- DeSlip RX Bytes -----*/

void Slip(uchar ch){
if(ch==Slip1){
  spflag1=TRUE;
  return;
}
if((ch==Slip2)&&(spflag1)) {
  spflag1=FALSE;
  ch=STX;
}
if((ch==Slip3)&&(spflag1)){
  spflag1=FALSE;
  ch=ETX;
}
if((ch==Slip4)&&(spflag1)){

```

```

spflag1=FALSE;
ch=Slip1;
}
PROCBUF[RxLength]=ch;
putc(ch,fp1);
RxLength++;
}

/*----- Wait For Message -----*/

void waitForMsg (void ){
while((rbout^rbin)!=0){
  unchar ch=RXBUF[rbout];
  putc(ch,fp); // Write Received Bytes to the File
  if(ch==STX){
    GotSTX=TRUE;
    RxLength=0;
  }
  else if (ch==ETX){
    GotSTX=FALSE;
    Event=RcvMsg;
  }
  else if (GotSTX)Slip(ch);
  rbout=++rbout&0X1f;
}
}

/*----- Process Message -----*/
void ProcessMsg(unchar c,unchar *Message){
  unchar i=0;
  unchar j=0;
  RxCheckSum=0;
  while(i<c){
    unchar ch=*Message;
    Message++;
    i++;
    switch(Byte){
/*----- Verify Source Address -----*/
      case waitForAddByte:
/*-----
      Master 1 Messages to other Nodes on the network.
      -----*/
      if(ch>>4==MASTER1){
        printf("SOURCE ADD = %d MASTER1\n",ch>>4);
        if((ch&0x0f)==MASTER2){
          printf("DEST ADD = %d MASTER2\n",ch&0x0f);
          Byte=waitForSequenceNumAndLength;
        }
      }
    }
  }
}

```

```

else if((ch&0x0f)==SLAVE1){
printf("DEST ADD = %d SLAVE1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE2){
printf("DEST ADD = %d SLAVE2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE3){
printf("DEST ADD = %d SLAVE3\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE4){
printf("DEST ADD = %d SLAVE4\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE5){
printf("DEST ADD = %d SLAVE5\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else{
printf("DEST ADD = %d UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
}
/*-----
Master 2 Messages to other Nodes on the network.
-----*/
else if(ch>>4==MASTER2){
printf("SOURCE ADD = %d MASTER2\n",ch>>4);
if((ch&0x0f)==MASTER1){
printf("DEST ADD = %d MASTER1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE1){
printf("DEST ADD = %d SLAVE1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE2){
printf("DEST ADD = %d SLAVE2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE3){
printf("DEST ADD = %d SLAVE3\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
}

```

```

else if((ch&0x0f)==SLAVE4){
printf("DEST ADD = %d SLAVE4\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==SLAVE5){
printf("DEST ADD = %d SLAVE5\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else{
printf("DEST ADD = %d UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
}
/*-----
Slave 1 Messages to other Nodes on the network.
-----*/
else if(ch>>4==SLAVE1){
printf("SOURCE ADD = %d SLAVE1\n",ch>>4);
if((ch&0x0f)==MASTER1){
printf("DEST ADD = %d MASTER1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==MASTER2){
printf("DEST ADD = %d MASTER2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else{
printf("DEST ADD = %d UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
}
/*-----
Slave 2 Messages to other Nodes on the network.
-----*/
else if(ch>>4==SLAVE2){
printf("SOURCE ADD = %d SLAVE2\n",ch>>4);
if((ch&0x0f)==MASTER1){
printf("DEST ADD = %d MASTER1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==MASTER2){
printf("DEST ADD = %d MASTER2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
}
}

```

```

else{
printf("DEST ADD = %d UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
/*-----
Slave 3 Messages to other Nodes on the network.
-----*/
else if(ch>>4==SLAVE3){
printf("SOURCE ADD = %d SLAVE3\n",ch>>4);
if((ch&0x0f)==MASTER1){
printf("DEST ADD = %d MASTER1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==MASTER2){
printf("DEST ADD = %d MASTER2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else{
printf("DEST ADD = %d UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
/*-----
Slave 4 Messages to other Nodes on the network.
-----*/
else if(ch>>4==SLAVE4){
printf("SOURCE ADD = %d SLAVE4\n",ch>>4);
if((ch&0x0f)==MASTER1){
printf("DEST ADD = %d MASTER1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==MASTER2){
printf("DEST ADD = %d MASTER2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else{
printf("DEST ADD = %d UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
/*-----

```

Slave 5 Messages to other Nodes on the network.

```

-----*/
else if(ch>>4==SLAVE5){
printf("SOURCE ADD = %d  SLAVE5\n",ch>>4);
if((ch&0x0f)==MASTER1){
printf("DEST ADD = %d  MASTER1\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else if((ch&0x0f)==MASTER2){
printf("DEST ADD = %d  MASTER2\n",ch&0x0f);
Byte=waitForSequenceNumAndLength;
}
else{
printf("DEST ADD = %d  UNKNOWN\n",ch&0x0f);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
else{
printf("SOURCE ADD = %d  UNKNOWN\n",ch<<4);
Byte=endLoop;
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
break;
/*----- Verify Source Address -----*/
case waitForSequenceNumAndLength:
/*----- Verify Sequence Number -----*/
if((ch>>6==1)||(ch>>6==2)){
printf("SEQUENCE NUMBER = %d\n",ch>>6);
if((ch&0x3f)==(c-overheads)){
Byte=waitForData;
printf("MESSAGE LENGTH = %d\n",ch&0x3f);
printf("----- DATA -----\n");
}
}
/*----- Invalid Length -----*/
else{
Byte=endLoop;
printf("MESSAGE LENGTH = %d INVALID\n",ch&0x3f);
printf("----- INVALID DATA -----\n");
printf("*****\n");
}
}
/*----- Unexpected Sequence Number -----*/
else{
Byte=endLoop;
printf("SEQUENCE NUMBER = %d UNEXPECTED SEQUENCE NUMBER\n",ch>>6);
}

```

```

printf("----- INVALID DATA -----\n");
printf("*****\n");
}
break;
/*----- Store Message -----*/
case waitForData:
if(j<c-overheads){
RxChecksum=RxChecksum^ch;
j++;
if(ch==TOKENREADY)printf("Token Ready Message %c",ch);
else if(ch==TACK)printf("***Token Requested*** %c",ch);
else if(ch==TOKEN)printf("Token SEND!! %c",ch);
else if(ch==ACK)printf("Acknowledged %c",ch);
else if(ch==NACK)printf("--Acknowledged-- %c",ch);
else if(ch==TNACK)printf("Token Declined %c",ch);
else printf("%c",ch);
if(j==c-overheads){
printf(" \n");
printf("-----\n");
Byte=waitForChecksum;
}
}
else{
printf("Surprise!! No DATA\n");
printf("-----\n");
printf("*****\n");
Byte=endLoop;
}
break;
/*----- Verify Checksum -----*/
case waitForChecksum:
if(ch==RxChecksum){
printf("CHECKSUM = %d VALID\n",ch);
printf("*****\n");
Byte=endLoop;
}
else{
printf("CHECKSUM = %d INVALID\n",ch);
printf("*****\n");
Byte=endLoop;
}
break;
/*----- END LOOP -----*/
case endLoop:
Byte=endLoop;
printf("Data = %d",ch);
break;
}

```

```

}
printf("\n");
}
/*----- Initialization -----*/

void init (){                               /* Initialize Port */

    outportb ( PORT1+3, 0X80 );              /* Select Baud Rate Registers (9600)*/
    outportb ( PORT1+0, 0X0C );
    outportb ( PORT1+1, 0X00 );
    outportb ( PORT1+3, 0X07 );              /* 8 Bits/Char, 2 Stop Bits, No Parity */
    outportb ( PORT1+4, 0X0A );              /* OUT2 for interrupts to 8259 */
    outportb ( PORT1+1, 0X01 );              /* Enable interrupts */
    old = getvect (INTVECT );                /* Save Old Interrupt Vector */
    setvect(INTVECT,PORT1INT);              /* Set Serial Port Vector to Service Routine */
    outportb(0X21,(inportb(0X21)&0XEF));    /* Enable PIC */

}

/*----- MAIN FUNCTION -----*/
void main(void){

    unchar c;
    init ();
    fp=fopen("Results.txt","w");             /* Open File */
    State=Idle;                              /* Initialize State */
    Event=noEvent;
    printf(" State is Idle<Rx MODE\n");
    printf(" Waiting For Data ..... \n\n");

    /*----- Operations -----*/
    do{
        switch(State){
            /*----- IDLE STATE -----*/
            case Idle:
                Byte=waitForAddByte;
                waitForMsg();
                if(Event==RcvMsg)State=ProcMsg;
                break;

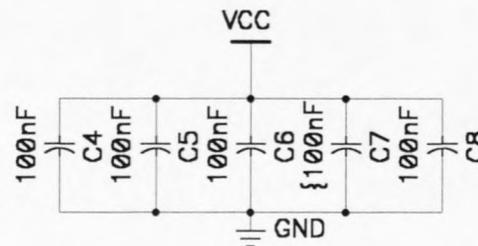
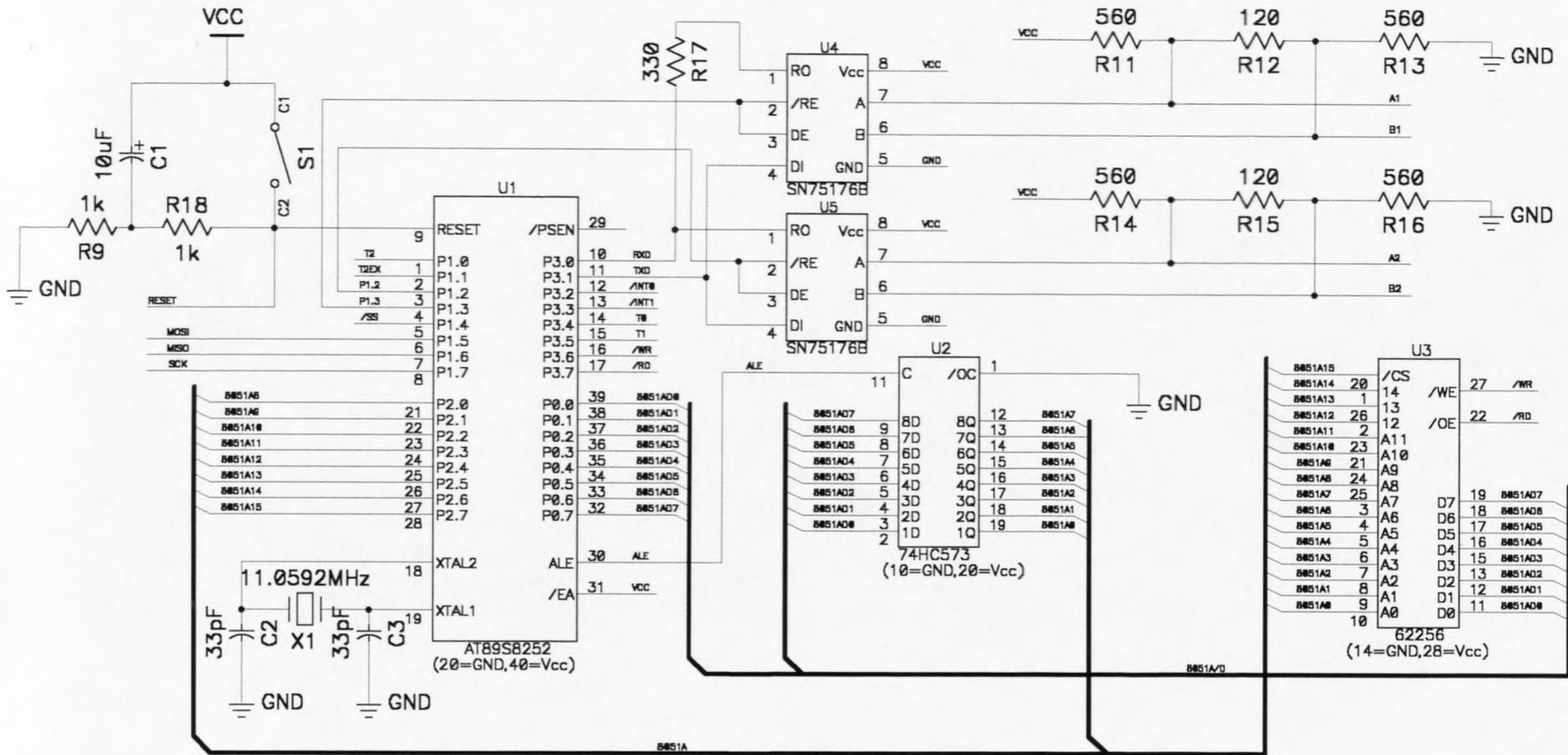
            /*----- PROCESSING MESSAGE STATE -----*/
            case ProcMsg:
                ProcessMsg(RxLength,PROCBUF);
                State=Idle;
                Event=noEvent;
                break;
        }
    }
    /*----- END OF STATES -----*/
}

```

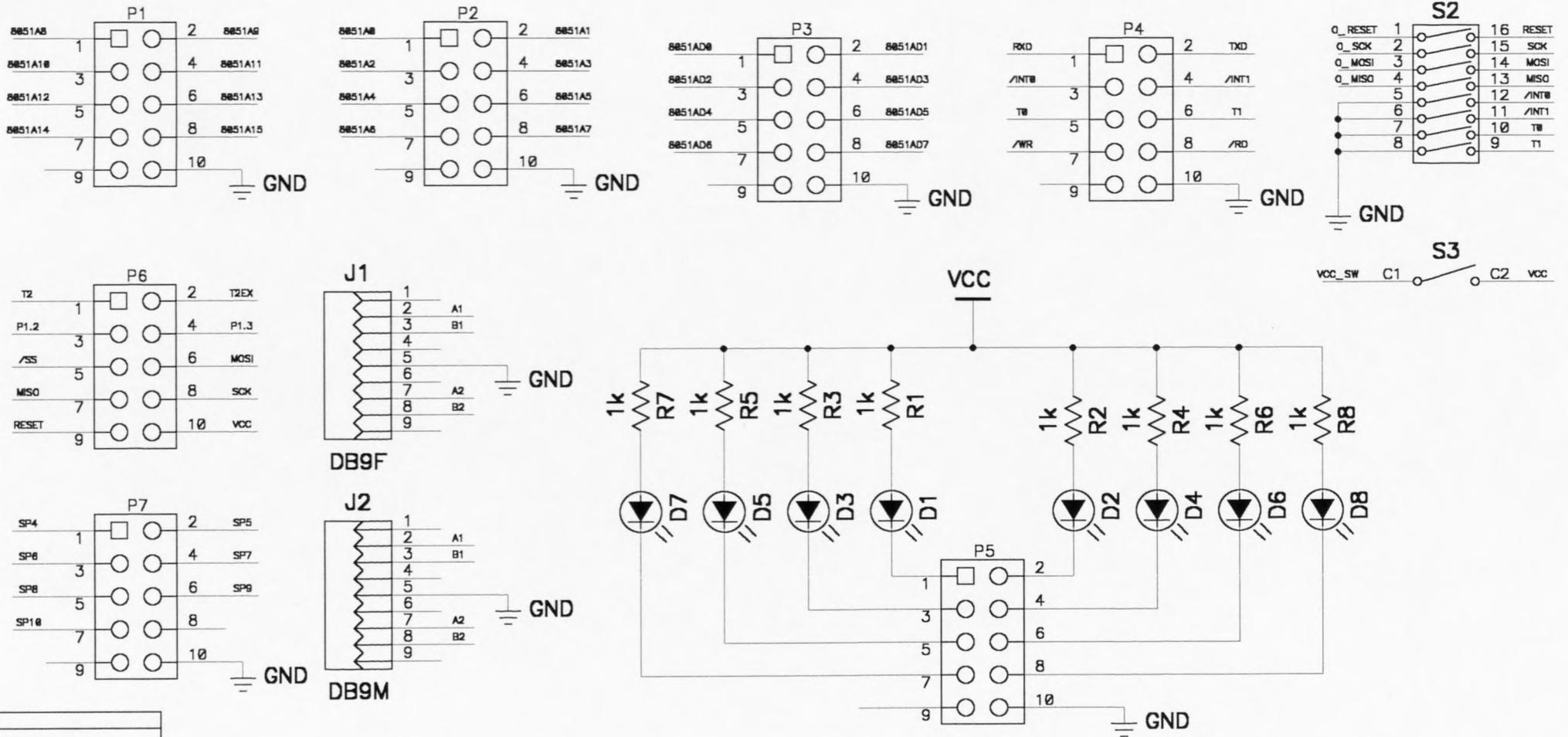
```
if(kbhit()){                                /* If keypressed */
c=getch();                                  /* Read the key */
fclose(fp);                                 /* Close the file*/
}
}while(c!=27);
outportb(0X21,(inportb(0X21)&0X10));        /* Mask IRQ using PIC */
setvect(INTVECT,old);                       /* Restore Old Interrupt Vector */
}
/*===== END OF PROGRAM =====*/
```

A.3 Schematics

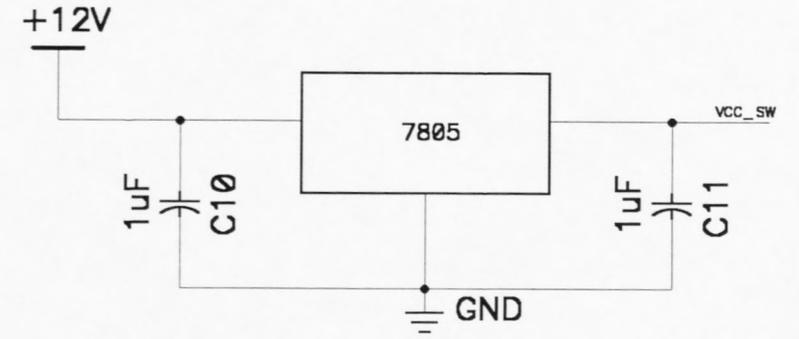
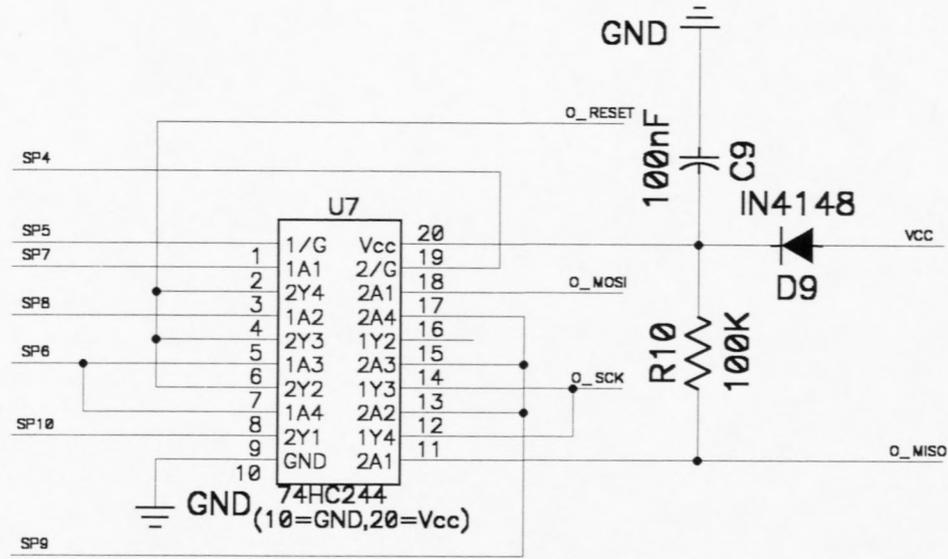
This section consists of the Test Boards schematics generated with *P-CAD 2000*.



Title
AT898252 AND 32K SRAM
ADCS



Title
PIN CONNECTORS
ADCS



A.4 Data Sheets

This section contains the data sheet of the main components of the test boards. ATMEL A89s5282 microprocessor, the Hitachi HM62265B SRAM and a DS75176 RS485 transceiver. It also consists of the MAX 232.

Features

- Compatible with MCS-51™ Products
- 8K bytes of In-System Reprogrammable Downloadable Flash Memory
 - SPI Serial Interface for Program Downloading
 - Endurance: 1,000 Write/Erase Cycles
- 2K bytes EEPROM
 - Endurance: 100,000 Write/Erase Cycles
- 4.0V to 6V Operating Range
- Fully Static Operation: 0 Hz to 24 MHz
- Three-Level Program Memory Lock
- 256 x 8 bit Internal RAM
- 32 Programmable I/O Lines
- Three 16 bit Timer/Counters
- Nine Interrupt Sources
- Programmable UART Serial Channel
- SPI Serial Interface
- Low Power Idle and Power Down Modes
- Interrupt Recovery From Power Down
- Programmable Watchdog Timer
- Dual Data Pointer
- Power Off Flag

Description

The AT89S8252 is a low-power, high-performance CMOS 8-bit microcomputer with 8K bytes of Downloadable Flash programmable and erasable read only memory and 2K bytes of EEPROM. The device is manufactured using Atmel's high density nonvolatile memory technology and is compatible with the industry standard 80C51 instruction set and pinout. The on-chip Downloadable Flash allows the program memory to be reprogrammed in-system through an SPI serial interface or by a conventional non-volatile memory programmer. By combining a versatile 8-bit CPU with Downloadable Flash on a monolithic chip, the Atmel AT89S8252 is a powerful microcomputer which provides a highly flexible and cost effective solution to many embedded control applications.

The AT89S8252 provides the following standard features: 8K bytes of Downloadable Flash, 2K bytes of EEPROM, 256 bytes of RAM, 32 I/O lines, programmable watchdog timer, two Data Pointers, three 16-bit timer/counters, a six-vector two-level interrupt architecture, a full duplex serial port, on-chip oscillator, and clock circuitry. In addition, the AT89S8252 is designed with static logic for operation down to zero frequency and supports two software selectable power saving modes. The Idle Mode stops the CPU while allowing the RAM, timer/counters, serial port, and interrupt system to continue functioning. The Power Down Mode saves the RAM contents but freezes the oscillator, disabling all other chip functions until the next interrupt or hardware reset.

The Downloadable Flash can be changed a single byte at a time and is accessible through the SPI serial interface. Holding RESET active forces the SPI bus into a serial programming interface and allows the program memory to be written to or read from unless Lock Bit 2 has been activated.



**8-Bit
Microcontroller
with 8K Bytes
Flash**

AT89S8252



HM62256B Series

256k SRAM (32-kword \times 8-bit)

HITACHI

ADE-203-135F (Z)

Rev. 6.0

Nov. 13, 1997

Description

The Hitachi HM62256B Series is a CMOS static RAM organized 32,768-word \times 8-bit. It realizes higher performance and low power consumption by employing 0.8 μ m Hi-CMOS process technology. The device, packaged in 8 \times 14 mm TSOP, 8 \times 13.4 mm TSOP with thickness of 1.2 mm, 450 mil SOP (foot print pitch width), 600 mil plastic DIP, or 300 mil plastic DIP, is available for high density mounting. It offers low power standby power dissipation; therefore, it is suitable for battery backup systems.

Features

- Single 5.0 V supply: 5.0 V \pm 10%
- Access time: 55 ns/70 ns/85 ns (max)
- Power dissipation:
 - Active: 25 mW (typ) ($f = 1$ MHz)
 - Standby: 1.0 μ W (typ)
- Completely static memory
 - No clock or timing strobe required
- Equal access and cycle times
- Common data input and output
 - Three state output
- Directly TTL compatible all inputs and outputs
- Battery backup operation



LTC485

Low Power RS485 Interface Transceiver

FEATURES

- Low Power: $I_{CC} = 300\mu\text{A Typ}$
- Designed for RS485 Interface Applications
- Single 5V supply
- $-7\text{V to }12\text{V}$ Bus Common-Mode Range Permits $\pm 7\text{V}$ Ground Difference Between Devices on the Bus
- Thermal Shutdown Protection
- Power-Up/Down Glitch-Free Driver Outputs Permit Live Insertion or Removal of Transceiver
- Driver Maintains High Impedance in Three-State or with the Power Off
- Combined Impedance of a Driver Output and Receiver Allows Up to 32 Transceivers on the Bus
- 70mV Typical Input Hysteresis
- 30ns Typical Driver Propagation Delays with 5ns Skew
- Pin Compatible with the SN75176A, DS75176A and $\mu\text{A}96176$

APPLICATIONS

- Low Power RS485/RS422 Transceiver
- Level Translator

DESCRIPTION

The LTC485 is a low power differential bus/line transceiver designed for multipoint data transmission standard RS485 applications with extended common-mode range (12V to -7V). It also meets the requirements of RS422.

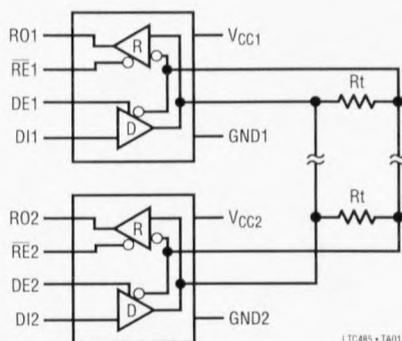
The CMOS design offers significant power savings over its bipolar counterpart without sacrificing ruggedness against overload of ESD damage.

The driver and receiver feature three-state outputs, with the driver outputs maintaining high impedance over the entire common-mode range. Excessive power dissipation caused by bus contention or faults is prevented by a thermal shutdown circuit which forces the driver outputs into a high impedance state.

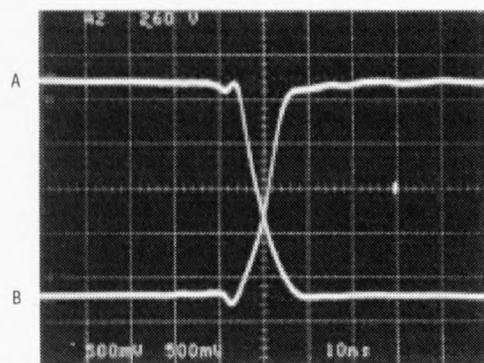
The receiver has a fail-safe feature which guarantees a high output state when the inputs are left open.

The LTC485 is fully specified over the commercial and extended industrial temperature range.

TYPICAL APPLICATION



Driver Outputs





+5V-Powered, Multichannel RS-232 Drivers/Receivers

General Description

The MAX220-MAX249 family of line drivers/receivers is intended for all EIA/TIA-232E and V.28/V.24 communications interfaces, particularly applications where $\pm 12V$ is not available.

These parts are especially useful in battery-powered systems, since their low-power shutdown mode reduces power dissipation to less than $5\mu W$. The MAX225, MAX233, MAX235, and MAX245/MAX246/MAX247 use no external components and are recommended for applications where printed circuit board space is critical.

Applications

Portable Computers
Low-Power Modems
Interface Translation
Battery-Powered RS-232 Systems
Multi-Drop RS-232 Networks

Features

Superior to Bipolar

- ◆ Operate from Single +5V Power Supply (+5V and +12V—MAX231/MAX239)
- ◆ Low-Power Receive Mode in Shutdown (MAX223/MAX242)
- ◆ Meet All EIA/TIA-232E and V.28 Specifications
- ◆ Multiple Drivers and Receivers
- ◆ 3-State Driver and Receiver Outputs
- ◆ Open-Line Detection (MAX243)

Ordering Information

PART	TEMP. RANGE	PIN-PACKAGE
MAX220CPE	0°C to +70°C	16 Plastic DIP
MAX220CSE	0°C to +70°C	16 Narrow SO
MAX220CWE	0°C to +70°C	16 Wide SO
MAX220C/D	0°C to +70°C	Dice*
MAX220EPE	-40°C to +85°C	16 Plastic DIP
MAX220ESE	-40°C to +85°C	16 Narrow SO
MAX220EWE	-40°C to +85°C	16 Wide SO
MAX220EJE	-40°C to +85°C	16 CERDIP
MAX220MJE	-55°C to +125°C	16 CERDIP

Ordering Information continued at end of data sheet.

*Contact factory for dice specifications.

Selection Table

Part Number	Power Supply (V)	No. of RS-232 Drivers/Rx	No. of Ext. Caps	Nominal Cap. Value (μF)	SHDN & Three-State	Rx Active in SHDN	Data Rate (kbps)	Features
MAX220	+5	2/2	4	4.7/10	No	—	120	Ultra-low-power, industry-standard pinout
MAX222	+5	2/2	4	0.1	Yes	—	200	Low-power shutdown
MAX223 (MAX213)	+5	4/5	4	1.0 (0.1)	Yes	✓	120	MAX241 and receivers active in shutdown
MAX225	+5	5/5	0	—	Yes	✓	120	Available in SO
MAX230 (MAX200)	+5	5/0	4	1.0 (0.1)	Yes	—	120	5 drivers with shutdown
MAX231 (MAX201)	+5 and +7.5 to +13.2	2/2	2	1.0 (0.1)	No	—	120	Standard +5/+12V or battery supplies; same functions as MAX232
MAX232 (MAX202)	+5	2/2	4	1.0 (0.1)	No	—	120 (64)	Industry standard
MAX232A	+5	2/2	4	0.1	No	—	200	Higher slew rate, small caps
MAX233 (MAX203)	+5	2/2	0	—	No	—	120	No external caps
MAX233A	+5	2/2	0	—	No	—	200	No external caps, high slew rate
MAX234 (MAX204)	+5	4/0	4	1.0 (0.1)	No	—	120	Replaces 1488
MAX235 (MAX205)	+5	5/5	0	—	Yes	—	120	No external caps
MAX236 (MAX206)	+5	4/3	4	1.0 (0.1)	Yes	—	120	Shutdown, three state
MAX237 (MAX207)	+5	5/3	4	1.0 (0.1)	No	—	120	Complements IBM PC serial port
MAX238 (MAX208)	+5	4/4	4	1.0 (0.1)	No	—	120	Replaces 1488 and 1489
MAX239 (MAX209)	+5 and +7.5 to +13.2	3/5	2	1.0 (0.1)	No	—	120	Standard +5/+12V or battery supplies; single-package solution for IBM PC serial port
MAX240	+5	5/5	4	1.0	Yes	—	120	DIP or flatpack package
MAX241 (MAX211)	+5	4/5	4	1.0 (0.1)	Yes	—	120	Complete IBM PC serial port
MAX242	+5	2/2	4	0.1	Yes	✓	200	Separate shutdown and enable
MAX243	+5	2/2	4	0.1	No	—	200	Open-line detection simplifies cabling
MAX244	+5	8/10	4	1.0	No	—	120	High slew rate
MAX245	+5	8/10	0	—	Yes	✓	120	High slew rate, int. caps, two shutdown modes
MAX246	+5	8/10	0	—	Yes	✓	120	High slew rate, int. caps, three shutdown modes
MAX247	+5	8/9	0	—	Yes	✓	120	High slew rate, int. caps, nine operating modes
MAX248	+5	8/8	4	1.0	Yes	✓	120	High slew rate, selective half-chip enables
MAX249	+5	6/10	4	1.0	Yes	✓	120	Available in quad flatpack package



For free samples & the latest literature: <http://www.maxim-ic.com>, or phone 1-800-998-8800.
For small orders, phone 408-737-7600 ext. 3468.

Appendix B

B. Protocol Source Code

The protocol source code was generated with the Keil C compiler. It consists of Master node and Slave node codes.

```
/* Name: (MASTER1.C), Serial Communication on RS485 Bus Network
Descr: The protocol software code for the Master Nodes.
-----*/
```

```
# include <At89x52.h>          /* SFR declarations for AT89x52*/
```

```
/*----- Define Constants -----*/
```

```
#define SOM  254          /* START of Frame */
#define EOM  255          /* END of Frame   */
#define MASTER1 1        /* Master1 Address */
#define MASTER2 2        /* Master2 Address */
#define SLAVE1  3        /* SLAVE1 Address */
#define SLAVE2  4        /* SLAVE2 Address */
#define SLAVE3  5        /* SLAVE3 Address */
#define SLAVE4  6        /* SLAVE4 Address */
#define SLAVE5  7        /* SLAVE5 Address */
#define Slip1  65        /* Caps A */
#define Slip2  66        /* Caps B */
#define Slip3  67        /* Caps C */
#define Slip4  68        /* Caps D */
#define ACK   06        /* Acknowledgement */
#define TOKEN 16        /* Acknowledgement */
#define TOKENREADY 15   /* Acknowledgement */
#define TACK  05        /* Token Acknowledgement */
#define RSD   48        /* Request Sensor Data */
#define TNACK 22        /* Negative Token Acknowledgement */
#define NACK  21        /* Negative Acknowledgement */
#define SIZE  32        /* Define Buffer Length */
#define overheads 3     /* Define Overheads Length */
#define TokenSize 4     /* Define Token Frame Size */
#define TokenSequenceNum 1 /* Define Token Frame Size */
```

```
#define TRUE 1
#define FALSE 0
#define TxEnable P1_2
#define Test P1_3
#define uchar unsigned char
#define FOREVER for (;;) // Endless loop
```

```
/*----- Declarations -----*/
```

```
uchar xdata BUF[SIZE];
uchar xdata RXBUF[SIZE];
uchar xdata PROCBUF[SIZE];
uchar xdata DATABUF[SIZE];
uchar xdata TXBUF[SIZE];
uchar xdata DataReady[SIZE];
```

```
uchar xdata DataAvail[SIZE];
uchar xdata APPLICATION[SIZE];
uchar TOKENBUF[2];
```

```
/*----- Simulate Data -----*/
```

```
uchar R_SensorData[]={ "SENSOR "};
uchar C_ReactionWheel[]={ "RW Data "};
uchar C_MagnetoTorquer[]={ "CD" };
uchar *SEN;
uchar *CRW;
uchar *CMT;
```

```
/*----- Flags -----*/
```

```
bit SLAVE1FLAG,SLAVE2FLAG,SLAVE3FLAG,SLAVE4FLAG,SLAVE5FLAG,MASTER1FLAG;
bit MASTER2FLAG,CLEAR,DUPLICATE,SECONDFLAG,TENSECONDFLAG;
bit TXFLAG,NO_ERRFLAG,TACKFLAG,TOKENFLAG,TOKENREADYFLAG,SENDTOKENFLAG;
bit MTFLAG,BUS_ACTIVE,VALID_LENGTH,VALID_SEQUENCENUM,TXMODE;
bit ReTxFLAG,Msg_ok,slipflag,GotSTX,Time_Out,TokenOut;
bit Run_Timer,Msg_Timer,Token_Timer,Data_ReadyFlag,Node_Faulty;
bit S1_Faulty,S2_Faulty,S3_Faulty,S4_Faulty,S5_Faulty,M2_Faulty;
bit Send_S1,Send_S2,Send_S3,Send_S4,Send_S5,Send_M2;
```

```
/*----- Counters -----*/
```

```
uchar Length,RxLength,TimerCount,RtxCount,AddCount;
uchar AddNum,Sendto,LastMsg,rbin,rbout,fin;
uchar S1_Fail,S2_Fail,S3_Fail,S4_Fail,S5_Fail,M1_Fail;
uchar SN1,SN2,SN3,SN4,SN5;
uchar TxChecksum,RxChecksum,TxSequenceNum,Second,TenSecond;
int TokenCount;
```

```
/*----- Define States & Events -----*/
```

```
enum States { Idle,RtxMsg,ProcMsg,WaitForAck };
enum Events { sendMsg,reSendMsg,timeOut,rcvMsg,rcvMsg_SlaveMode,
              sendToken,msgOk,msgNotOk,rtxCountMax };
enum Bytes { waitForAddByte,waitForSequenceNumAndLength,waitForData,
             waitForChecksum,endLoop };
enum Replies { TokenReply,TokenAckReply,AckReply,MsgReply };
```

```
enum Bytes Byte;
enum Replies Reply;
enum States State;
enum Events Event;
```

```
/*===== 10ms TIMER INTERRUPT =====*/
```

```
void Timer_Int (void) interrupt TF0_VECTOR {
```

```
    TR0 = FALSE;           // Stop Timer0.  
    TH0 = 0xDC;           // 1's complement of 10ms/[(1/Fosc)*6]  
    TL0 = 0x00;           // =[TH0,TL0].Fosc=11.0592MHz=>[0xDC,0x00]  
    TR0 = TRUE;           // Restart Timer0.
```

```
/*----- 120ms ACK TIMER -----*/
```

```
if(Run_Timer){           // Start ACK Timer  
    TimerCount++;        // Increment Timer Counter  
    if(TimerCount==12){  
        TimerCount=0;    // Reset Timer Counter  
        Event=timeOut;   // Set Time-Out Flag  
        Time_Out=TRUE;   // Set Time-Out Flag  
        Run_Timer=FALSE; // Stop ACK Timer  
    }  
}
```

```
/*----- 1s & 10s MESSAGE TIMER -----*/
```

```
if(Msg_Timer){           // Start Message Timer  
    Second++;           // Increment Second Counter  
    if(Second==100){  
        Second=0;       // Reset Second Counter  
        SECONDFLAG=TRUE; // Set Second Flag  
        TenSecond++;    // Increment TenSecond Counter  
        if(TenSecond==10){  
            TenSecond=0; // Reset TenSecond Counter  
            TENSECONDFLAG=TRUE; // Set Second Flag  
        }  
    }  
}
```

```
/*----- 3.2s TOKEN TIMER -----*/
```

```
if(Token_Timer){        // Start Token Timer  
    TokenCount++;       // Increment Second Counter  
    if(TokenCount==320){  
        TokenCount=0;   // Reset TokenCounter Counter  
        TokenOut=TRUE;  // Set Token Time out flag  
    }  
}
```

```
/*===== SERIAL INTERRUPT =====*/
```

```
void Serial_Int (void) interrupt SIO_VECTOR {  
if(RI){  
RI=FALSE; // Clear RI Flag  
if ((rbin+1^rbout)!=0){ // Buffer Empty  
RXBUF[rbin]=SBUF; // Store Received Byte  
rbin = ++rbin & 0X1f; // Reset Buffer  
}  
}  
if(TI){  
TI=FALSE; // Clear TI  
CLEAR=TRUE; // Set Clear Flag  
}  
}  
}
```

```
/*===== DE-SLIP RX DATA =====*/
```

```
Identifies byte sequences and replaces them,i.e <65><66> by <SOM>,  
<65><67> by <EOM>,and <65><68> by <65>.
```

```
-----*/
```

```
void Slip(unchar ch){  
if(ch==Slip1){  
slipflag=TRUE;  
return;  
}  
if((ch==Slip2)&&(slipflag)) {  
slipflag=FALSE;  
ch=SOM;  
}  
if((ch==Slip3)&&(slipflag)){  
slipflag=FALSE;  
ch=EOM;  
}  
if((ch==Slip4)&&(slipflag)){  
slipflag=FALSE;  
ch=Slip1;  
}  
PROCBUF[RxLength++]=ch;  
}  
}
```

```
/*===== WAIT FOR MESSAGES =====*/
```

```
Identify and Strips the SOM and EOM
```

```
-----*/
```

```
void waitForMsg (void ){  
while((rbout^rbin)!=0){  
unchar ch=RXBUF[rbout];
```

```

rbout=++rbout&0X1f;
if(ch==SOM){
  GotSTX=TRUE;
  RxLength=0;
}
else if (ch==EOM){
  GotSTX=FALSE;
  ETXFLAG=TRUE;
}
else if (GotSTX)Slip(ch);
}
}

```

```

/*===== PROCESS MESSAGES =====
Strips the Header and the Footer
-----*/

```

```

void ProcessMsg(unchar c, unchar *Message){
  unchar i=0;           // Reset Counters
  unchar j=0;
  RxCheckSum=0;
  Event=msgNotOk;      // In Case of Error,Eventis Defined
  while(i<c){
    unchar ch=*Message;
    Message++;
    i++;
    switch(Byte){

```

```

/*----- Verify Addresses -----*/
  case waitForAddByte:
/*----- SLAVE 1 Address -----*/
    if((ch>>4)==SLAVE1){
      SLAVE1FLAG=TRUE;
      if((ch&0x0f)==MASTER1){
        MASTER1FLAG=TRUE;
        Byte=waitForSequenceNumAndLength;
      }
    }
    else{
      Byte=endLoop;
      Event=msgNotOk;
    }
  }
/*----- SLAVE 2 Address -----*/
  else if((ch>>4)==SLAVE2){
    SLAVE2FLAG=TRUE;
    if((ch&0x0f)==MASTER1){
      MASTER1FLAG=TRUE;
      Byte=waitForSequenceNumAndLength;

```

```

    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
}
/*----- SLAVE 3 Address -----*/
else if((ch>>4)==SLAVE3){
    SLAVE3FLAG=TRUE;
    if((ch&0x0f)==MASTER1){
        MASTER1FLAG=TRUE;
        Byte=waitForSequenceNumAndLength;
    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
}
/*----- SLAVE 4 Address -----*/
else if((ch>>4)==SLAVE4){
    SLAVE4FLAG=TRUE;
    if((ch&0x0f)==MASTER1){
        MASTER1FLAG=TRUE;
        Byte=waitForSequenceNumAndLength;
    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
}
/*----- SLAVE 5 Address -----*/
else if((ch>>4)==SLAVE5){
    SLAVE5FLAG=TRUE;
    if((ch&0x0f)==MASTER1){
        MASTER1FLAG=TRUE;
        Byte=waitForSequenceNumAndLength;
    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
}
/*----- MASTER 1 Address -----*/
else if((ch>>4)==MASTER2){
    MASTER2FLAG=BUS_ACTIVE=TRUE;
    if((ch&0x0f)==MASTER1){
        MASTER1FLAG=TRUE;
        Byte=waitForSequenceNumAndLength;
    }
}

```

```

    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
}
/*----- Unknown Address -----*/
else{
    Byte=endLoop;
    Event=msgNotOk;
}
break;

/*----- Verify Sequence Number and Length -----*/
case waitForSequenceNumAndLength:
    if((ch>>6==1)||(ch>>6==2)){
/*----- Verify Token Sequence Number and Length -----*/
        if(((ch>>6)==1)&&(MASTER1FLAG)){
            MASTER1FLAG=FALSE;
            VALID_SEQUENCENUM=TRUE;
            if((ch&0x3f)==(c-overheads)){
                Byte=waitForData;
                VALID_LENGTH=TRUE;
            }
/*----- Invalid Length -----*/
        else{
            Byte=endLoop;
            Event=msgNotOk;
        }
}
/*----- Verify Message Sequence Number and Length -----*/
        else if(ch>>6==TxSequenceNum){
            VALID_SEQUENCENUM=TRUE;
            if((ch&0x3f)==(c-overheads)){
                Byte=waitForData;
                VALID_LENGTH=TRUE;
            }
/*----- Invalid Length -----*/
        else{
            Byte=endLoop;
            Event=msgNotOk;
        }
}
}
/*----- Unexpected Sequence Number -----*/
else{
    Byte=endLoop;
    Event=msgNotOk;
}

```

```

    }
    break;
/*----- Store Message -----*/
case waitForData:
    if(j<c-overheads){
        RxChecksum=RxChecksum^ch;    // Calculates Checksum
        BUF[j]=ch;
        j++;
        if(j==c-overheads)
            Byte=waitForChecksum;
    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
    break;

/*----- Verify Checksum -----*/
case waitForChecksum:
    if(ch==RxChecksum){                // Compare the Checksums
        NO_ERRFLAG=TRUE;
        Byte=waitForAddByte;
    }
    else{
        Byte=endLoop;
        Event=msgNotOk;
    }
    break;
/*----- END LOOP -----*/
case endLoop:
    Byte=endLoop;
    break;
}
}
}

/*===== RX APPLICATION LAYER =====
Data Available For Application
-----*/

void Data_Avail(unchar *Data){
    unchar i=0;
    while(*Data!=0){
        APPLICATION[i++]=*Data;
        Data++;
    }
}
}

```

```
/*===== VERIFY MESSAGE =====  
Verifies the Processed Message  
-----*/
```

```
void Verify_Msg(unchar c, unchar *Message){  
    unchar i;  
    if((NO_ERRFLAG)&&(MASTER1FLAG)  
        &&(VALID_LENGTH)&&(VALID_SEQUENCENUM)){  
        NO_ERRFLAG=MASTER1FLAG=VALID_LENGTH=VALID_SEQUENCENUM=FALSE;  
        for(i=0; i<c; i++){  
            switch(Reply){
```

```
/*----- Waiting For Token (RXMODE) -----*/
```

```
    case TokenReply:  
        if(*Message==TOKENREADY){  
            TOKENREADYFLAG=TRUE;  
            State=Idle;  
            Event=sendToken;  
        }  
        else if(*Message==TOKEN){  
            State=Idle;  
            Event=sendMsg;  
        }  
        else{  
            State=Idle;  
            Event=rcvMsg_SlaveMode;  
        }  
    break;
```

```
/*----- Waiting For Replies (TXMODE) -----*/
```

```
    case TokenAckReply:  
        if(*Message==TACK){  
            TACKFLAG=TRUE;  
            Send_M2=FALSE;  
            M1_Fail=0;  
            State=Idle;  
            Event=sendToken;  
        }  
        else Event=msgNotOk;  
    break;
```

```
    case MsgReply:  
        if(*Message==NACK)  
            Event=msgNotOk;  
        if(*Message==ACK)  
            Event=msgOk;  
        else Event=msgOk;  
    break;  
}
```

```

    DataAvail[i++]=*Message;          // Store Data
    Message++;
}
Data_Avail(&DataAvail);             // Pass Data to Application Layer
}
}

/*===== SEND BYTE AT A TIME =====*/

void sendByte(int c){
    SBUF=c;
    while(!CLEAR){}                // Wait For Successful Transmission of a byte
    CLEAR=FALSE;
}

/*===== SLIP DATA BYTES =====
Add sequences <65><66>if<EOM>,<65><67>if<EOM>&<65><68>if<65> are in data.
-----*/

void sendSlip(unchar c){
    if(c==SOM){
        sendByte(Slip1);
        sendByte(Slip2);
    }
    else if(c==EOM){
        sendByte(Slip1);
        sendByte(Slip3);
    }
    else if(c==Slip1){
        sendByte(Slip1);
        sendByte(Slip4);
    }
    else
        sendByte(c);
}

/*===== TX FRAME =====
Transmit Frame Bytes
-----*/

void sendFrame(unchar c, unchar *Message){
    int i;
    sendByte(SOM);
    for(i=0;i<c;i++){
        sendSlip(*Message);
        Message++;
    }
    sendByte(EOM);
}

```

```
}
```

```
/*===== MESSAGE FRAME =====  
Make Frame
```

```
-----*/
```

```
void makeFrame(unchar c, unchar *Message){  
    unchar i;  
    TxChecksum=fin=0;                // Reset Counters
```

```
    if(Send_S1)TXBUF[fin++]=(MASTER1<<4)|SLAVE1;  
    else if(Send_S2)TXBUF[fin++]=(MASTER1<<4)|SLAVE2;  
    else if(Send_S3)TXBUF[fin++]=(MASTER1<<4)|SLAVE3;  
    else if(Send_S4)TXBUF[fin++]=(MASTER1<<4)|SLAVE4;  
    else if(Send_S5)TXBUF[fin++]=(MASTER1<<4)|SLAVE5;  
    else if(Send_M2)TXBUF[fin++]=(MASTER1<<4)|MASTER1;
```

```
    TXBUF[fin++]=(TxSequenceNum<<6)|c;  
    for (i=0; i<c; i++){  
        TxChecksum=TxChecksum^*Message;  
        TXBUF[fin]=*Message;  
        fin++;  
        Message++;  
    }  
    TXBUF[fin]=TxChecksum;  
}
```

```
/*===== LOAD MESSAGES =====  
Calculates Message Length and Updates Sequence Number
```

```
-----*/
```

```
void loadMsg(unchar c, unchar ch, unchar *Data){  
    Length=0;  
    if((c==1)&&(!S1_Faulty)){Send_S1=TRUE;TxSequenceNum=SN1;}  
    else if((c==2)&&(!S2_Faulty)){Send_S2=TRUE;TxSequenceNum=SN2;}  
    else if((c==3)&&(!S3_Faulty)){Send_S3=TRUE;TxSequenceNum=SN3;}  
    else if((c==4)&&(!S4_Faulty)){Send_S4=TRUE;TxSequenceNum=SN4;}  
    else if((c==5)&&(!S5_Faulty)){Send_S5=TRUE;TxSequenceNum=SN5;}  
    else if((c==7)&&(!M2_Faulty)){Send_M2=TRUE;TxSequenceNum=1;}  
    else Node_Faulty=TRUE;  
    if(ch==9)SENDTOKENFLAG=TRUE;  
    while(*Data!=0){  
        DATABUF[Length++]=*Data;  
        Data++;  
    }  
}
```

```
/*===== TX APPLICATION LAYER =====  
Reads Data from Memory and Pass it LoadMsg function Every Second  
-----*/
```

```
void Data_Ready(void){  
  unchar i=0;  
  unchar j=0;  
  SEN=&R_SensorData;  
  CRW=&C_ReactionWheel;  
  CMT=&C_MagnetoTorquer;  
  while(j<SIZE)DataReady[j++]=0;  
  if(SECONDFLAG){  
    Data_ReadyFlag=TRUE;  
    Sendto++;  
    LastMsg=0;  
    AddNum=Sendto;  
    if(Sendto==4){  
      while(*CRW!=0){  
        DataReady[i++]=*CRW;  
        CRW++;  
      }  
      if(!TENSECONDFLAG){  
        Sendto=0;  
        SECONDFLAG=FALSE;  
      }  
    }  
    else if((TENSECONDFLAG)&&(Sendto==5)){  
      TENSECONDFLAG=SECONDFLAG=FALSE;  
      Sendto=0;  
      LastMsg=9;  
      while(*CMT!=0){  
        DataReady[i++]=*CMT;  
        CMT++;  
      }  
    }  
    else{  
      while(*SEN!=0){  
        DataReady[i++]=*SEN;  
        SEN++;  
      }  
    }  
  }  
}
```

```

/*===== INITIALIZATION =====*/

void Main_Init (void){

TMOD=0x21;           // Timer Mode Select register.
                    // Timer0 in model = 16bit timer.
                    // Gate-control via TR0. Timer-mode
                    // Input from internal system clock.
                    // Gate-control via TR0. Timer-mode
                    // Input from internal system clock.

TH1 =0xFA;          // 9600 baud 11.05920 MHz
TCON=0x50;          // Start Timer0 & Timer1.
SCON=0x50;          // Enable Receiver
IE =0x92;           // Enable interrupts
                    // generation on the uP (IE).

TH0 =0xDC;          // 1's complement of 10ms/[(1/Fosc)*6]
TL0 =0x00;          // =[TH0,TL0].Fosc=11.0592MHz=>[0xDC,0x00]
PCON=0x80;          // Double baud rate

SN1=SN2=SN3=SN4=SN5=1; // Initialize Pointers,Counters & Flags *
Byte=waitForAddByte;
}

/*===== MAIN FUNCTION =====
Master Node Finite State Machine
-----*/

void main(void){
Main_Init ();
State=Idle;
Event=rcvMsg_SlaveMode;

/*===== OPERATIONS =====*/
FOREVER{
switch(State){

/*===== IDLE STATE =====*/
case Idle:
Byte=waitForAddByte;
switch(Event){

/*----- SENDING MESSAGE -----*/
case sendMsg:
RtxCount=0;
Msg_Timer=TXMODE=TRUE;
Token_Timer=FALSE;
TxEnable=TRUE;
Data_Ready();

```

```

if(Data_ReadyFlag){
    Data_ReadyFlag=FALSE;
    loadMsg(AddNum,LastMsg,&DataReady);
    if(Node_Faulty){
        Node_Faulty=FALSE;
        State=Idle;
        if(SENDTOKENFLAG)Event=sendToken;
        else Event=sendMsg;
    }
    else{
        makeFrame(Length,&DATABUF);
        sendFrame(Length+overheads,&TXBUF);
        Reply=MsgReply;
        State=WaitForAck;
        Event=rcvMsg;
    }
}
break;
/*----- TOKEN PASSING -----*/
case sendToken:
    TxEnable=TRUE;
    Msg_Timer=FALSE;
    AddNum=7;
    LastMsg=0;
/*-----TXMODE-----
    Sending Token Available Message
-----*/
if(SENDTOKENFLAG){
    SENDTOKENFLAG=FALSE;
    TOKENBUF[0]=TOKENREADY;
    if(Node_Faulty){
        Node_Faulty=FALSE;
        State=Idle;
        Event=sendMsg;
    }
    else{
        loadMsg(AddNum,LastMsg,&TOKENBUF);
        makeFrame(Length,&DATABUF);
        sendFrame(Length+overheads,&TXBUF);
        Reply=TokenAckReply;
        State=WaitForAck;
        Event=rcvMsg;
    }
}
/*----- Sending Token -----*/
else if(TACKFLAG){
    TACKFLAG=FALSE;
    TOKENBUF[0]=TOKEN;

```

```

loadMsg(AddNum,LastMsg,&TOKENBUF);
makeFrame(Length,&DATABUF);
sendFrame(Length+overheads,&TXBUF);
State=Idle;
Event=rcvMsg_SlaveMode;
}
/*----- RXMODE-----
Send Token Request Message
-----*/
else if(TOKENREADYFLAG){
TOKENREADYFLAG=FALSE;
TOKENBUF[0]=TACK;
loadMsg(AddNum,LastMsg,&TOKENBUF);
makeFrame(Length,&DATABUF);
sendFrame(Length+overheads,&TXBUF);
Event=rcvMsg_SlaveMode;
}
break;
/*-----TOKEN GENERATION -----*/
case rcvMsg_SlaveMode:
TxEnable=FALSE;
Msg_Timer=TXMODE=FALSE;
Token_Timer=TRUE;
Reply=TokenReply;
if(BUS_ACTIVE){
Token_Timer=BUS_ACTIVE=FALSE;
TokenCount=0;
}
else if(TokenOut){
Token_Timer=TokenOut=FALSE;
TokenCount=0;
Event=sendMsg;
}
waitForMsg();
if(ETXFLAG){
ETXFLAG=FALSE;
State=ProcMsg;
}
break;
}
break;

/*===== PROCESSING MESSAGE STATE =====*/
case ProcMsg:
Run_Timer=FALSE;
TimerCount=0;
ProcessMsg(RxLength,&PROCBUF);

```

```

Verify_Msg(RxLength-overheads,&BUF);
switch(Event){
/*-----Message Valid-----*/
case msgOk:
    RtxCount=0;                // Reset Retransmission Counter
    State=Idle;
    if(SLAVE1FLAG){
        SLAVE1FLAG=Send_S1=FALSE;    // Clear Flags
        S1_Fail=0;                    // Reset Node Fail Counter
        if(SN1==1)SN1=2;              // Update Sequence Number
        else if(SN1==2)SN1=1;
        Event=sendMsg;
    }
    else if(SLAVE2FLAG){
        SLAVE2FLAG=Send_S2=FALSE;    // Clear Flags
        S2_Fail=0;                    // Reset Node Fail Counter
        if(SN2==1)SN2=2;              // Update Sequence Number
        else if(SN2==2)SN2=1;
        Event=sendMsg;
    }
    else if(SLAVE3FLAG){
        SLAVE3FLAG=Send_S3=FALSE;    // Clear Flags
        S3_Fail=0;                    // Reset Node Fail Counter
        if(SN3==1)SN3=2;              // Update Sequence Number
        else if(SN3==2)SN3=1;
        Event=sendMsg;
    }
    else if(SLAVE4FLAG){
        SLAVE4FLAG=Send_S4=FALSE;    // Clear Flags
        S4_Fail=0;                    // Reset Node Fail Counter
        if(SN4==1)SN4=2;              // Update Sequence Number
        else if(SN4==2)SN4=1;
        Event=sendMsg;
    }
    else if(SLAVE5FLAG){
        SLAVE5FLAG=Send_S5=FALSE;    // Clear Flags
        S5_Fail=0;                    // Reset Node Fail Counter
        if(SN5==1)SN5=2;              // Update Sequence Number
        else if(SN5==2)SN5=1;
        if(SENDTOKENFLAG)Event=sendToken; // Send Token
        else Event=sendMsg;
    }
    break;
/*-----Message Invalid-----*/
case msgNotOk:
    if(!TXMODE){                    // Discard Message
        State=Idle;
        Event=rcvMsg_SlaveMode;
    }

```

```

}
else{
  State=RtxMsg;
  if(RtxCount==2)Event=rtxCountMax;
  else Event=reSendMsg;
}
break;
}
break;

/*===== WAIT FOR ACK STATE =====*/
case WaitForAck:
  TxEnable=FALSE;
  Run_Timer=TRUE;
  switch(Event){
/*----- Receive Message -----*/
  case rcvMsg:
    waitForMsg();
    if(ETXFLAG){
      ETXFLAG=FALSE;
      State=ProcMsg;
    }
    break;
/*----- Time Out -----*/
  case timeOut:
    State=RtxMsg;
    if(RtxCount==2)Event=rtxCountMax;
    else Event=reSendMsg;
    break;
  }
  break;

/*===== RETRANSMIT MESSAGE STATE =====*/
case RtxMsg:
  switch(Event){
/*----- Resend Message -----*/
  case reSendMsg:
    TxEnable=TRUE;           // Enable Driver
    Msg_Timer=Run_Timer=FALSE; // Stop Timers
    sendFrame(Length+overheads,&TXBUF);
    RtxCount++;           // Increment Retransmission Counter
    State=WaitForAck;
    Event=rcvMsg;
    if(Send_S1){
      S1_Fail++;           // Increment Node Fail Counter
      if(S1_Fail==10)S1_Faulty=TRUE; // Slave 1 Failed
    }
    else if(Send_S2){

```

```

S2_Fail++; // Increment Node Fail Counter
if(S2_Fail==10)S2_Faulty=TRUE; // Slave 2 Failed
}
else if(Send_S3){ // Increment Node Fail Counter
S3_Fail++; // Slave 3 Failed
if(S3_Fail==10)S3_Faulty=TRUE;
}
else if(Send_S4){ // Increment Node Fail Counter
S4_Fail++; // Slave 4 Failed
if(S4_Fail==10)S4_Faulty=TRUE;
}
else if(Send_S5){ // Increment Node Fail Counter
S5_Fail++; // Slave 5 Failed
if(S5_Fail==10)S5_Faulty=TRUE;
}
else if(Send_M2){ // Increment Node Fail Counter
M1_Fail++; // Master 1 Failed
if(M1_Fail==10)M2_Faulty=TRUE;
}
break;
/*-----Maximum Retransmission -----*/
case rtxCountMax:
Msg_Timer=Run_Timer=FALSE; // Clear Timers
RtxCount=TimerCount=0; // Reset Counter
Send_S1=Send_S2=Send_S3=FALSE; // Clear Flags
Send_S4=Send_S5=Send_M2=FALSE;
State=Idle;
if(SENDTOKENFLAG)Event=sendToken; // Send Token
else Event=sendMsg;
break;
}
break;
}
/*===== END OF STATES =====*/
}
}
/*===== END OF PROGRAM =====*/

```

```
/* Name: (SLAVE1.C), Serial Communication on RS485 Bus Network
Descr: The protocol software code for the Slave Nodes.
```

```
-----*/
```

```
#include <At89x52.h>          /* SFR declarations for AT89x52*/
```

```
/*----- Define Constants -----*/
```

```
#define SOM  254             /* Start of Frame */
#define EOM  255             /* End of Frame  */
#define MASTER1 1           /* Master1 Address */
#define MASTER2 2           /* Master2 Address */
#define SLAVE1  3           /* SLAVE1 Address */
#define Slip1  65            /* Caps A */
#define Slip2  66            /* Caps B */
#define Slip3  67            /* Caps C */
#define Slip4  68            /* Caps D */
#define ACK  0x06            /* Acknowledgement */
#define NACK 0x15            /* Negative Acknowledgement */
#define SIZE 65              /* Define Buffer Length */
#define overheads 3         /* Define Overheads Length */
```

```
#define FALSE 0
```

```
#define TRUE  1
```

```
#define TxEnable P1_2       /* Driver Transmit Enable Pin */
```

```
#define Test P1_3          /* Test Pin */
```

```
#define unchar unsigned char
```

```
#define FOREVER for (;;)   /* Endless loop */
```

```
/*----- Declarations -----*/
```

```
unchar xdata BUF[SIZE];
unchar xdata RXBUF[SIZE];
unchar xdata PROCBUF[SIZE];
unchar xdata DATABUF[SIZE];
unchar xdata TXBUF[SIZE];
unchar xdata DataReady[SIZE];
unchar xdata DataAvail[SIZE];
unchar xdata APPLICATION[SIZE];
```

```
/*----- Simulate Data -----*/
```

```
unchar Magnetometer[]={ "MM Data" };
```

```
unchar *Data_Pointer;
```

```
/*----- Flags -----*/
```

```
bit MASTER1FLAG,MASTER2FLAG,SLAVE1FLAG,STORE;
```

```
bit NO_ERRFLAG,VALID_LENGTH,VALID_SEQUENCENUM;
```

```

bit CLEAR,slipflag,GotSTX,Time_Out,Reply_Timer;
/*----- Counters -----*/

uchar Length,RxLength,TimerCount,RtxCount,rbin,rbout;
uchar TxChecksum,RxChecksum,StoreRxSequenceNum,RxSequenceNum;

/*----- Define States & Events -----*/

enum States{Idle,ProcMsg,RtxMsg};
enum Events {noEvent,RcvMsg,sendAck,sendNack,timeOut,sameMsg,rtxLastAck};
enum Bytes {waitForAddByte,waitForSequenceNumAndLength,waitForData,
            waitForChecksum,endLoop};

enum Bytes Byte;
enum States State;
enum Events Event;

/*===== TIMER INTERRUPT =====*/

void Timer_Int (void) interrupt TF0_VECTOR {
TR0 = FALSE;           // Stop Timer0.
TH0 = 0xDC;           // 1's complement of 10ms/[(1/Fosc)*6]
TL0 = 0x00;           // =[TH0,TL0].Fosc=11.0592MHz=>[0xDC,0x00]
TR0 = TRUE;           // Restart Timer0.

/*----- 9ms Reply Timer -----*/

if(Reply_Timer){      // Start Reply Timer
TimerCount++;        // Increment Timer Counter
if(TimerCount==10){
TimerCount=0;        // Reset Timer Counter
Time_Out=TRUE;      // Set time-out flag
}
}
}

/*===== SERIAL INTERRUPT =====*/

void Serial_Int (void) interrupt SIO_VECTOR {
if(RI){
RI=FALSE;           // Clear RI
if ((rbin+1^rbout)!=0){ // Buffer Empty
RXBUF[rbin]=SBUF;   // Store Received Byte
rbin = ++rbin & 0X1f; // Reset Buffer
}
}
if(TI){
TI=FALSE;           // Clear TI
CLEAR=TRUE;         // Set Clear Flag
}
}

```

```
}  
}
```

```
/*===== RX APPLICATION LAYER =====  
Data Available For Application  
-----*/
```

```
void Data_Avail(unchar *Data){  
unchar i=0;  
while(*Data!=0){  
APPLICATION[i++]=*Data;  
Data++;  
}  
}
```

```
/*===== DE-SLIP RX DATA =====  
Identifies sequences and replaces them, i.e <65><66>=<SOM>,<65><67>=<EOM>,  
and <65><68><65>.  
-----*/
```

```
void Slip(unchar ch){  
if(ch==Slip1){  
slipflag=TRUE;  
return;  
}  
if((ch==Slip2)&&(slipflag)) {  
slipflag=FALSE;  
ch=SOM;  
}  
if((ch==Slip3)&&(slipflag)){  
slipflag=FALSE;  
ch=EOM;  
}  
if((ch==Slip4)&&(slipflag)){  
slipflag=FALSE;  
ch=Slip1;  
}  
PROCBUF[RxLength++]=ch;  
}
```

```
/*===== WAIT FOR MESSAGES =====  
Identify and Strips the SOM and EOM  
-----*/
```

```
void waitForMsg (void ){  
while((rbout^rbin)!=0){  
unchar ch=RXBUF[rbout];  
if(ch==SOM){
```

```

GotSTX=TRUE;
RxLength=0;
}
else if (ch==EOM){
GotSTX=FALSE;
Event=RcvMsg;
}
else if (GotSTX)Slip(ch);
rbout=++rbout&0X1f;
}
}

```

```

/*===== PROCESS MESSAGES =====*/
Strips the Header and the Footer
-----*/

```

```

void ProcessMsg(unchar c, unchar *Message){
unchar i=0; // Reset Counters
unchar j=0;
RxChecksum=0;
Event=noEvent; // In Case of Error,Eventis Defined
while(i<c){
unchar ch=*Message;
Message++;
i++;
switch(Byte){
/*----- Verify Addresses -----*/
case waitForAddByte:
/*----- MASTER 1 Address -----*/
if((ch>>4)==MASTER1){
MASTER1FLAG=TRUE;
MASTER2FLAG=FALSE;
if((ch&0x0f)==SLAVE1){
SLAVE1FLAG=TRUE;
Byte=waitForSequenceNumAndLength;
}
else{
if(((ch&0x0f)==MASTER1)||((ch&0x0f)==MASTER2))
STORE=FALSE;
Byte=endLoop;
State=Idle;
Event=noEvent;
}
}
/*----- MASTER 2 Address -----*/
else if((ch>>4)==MASTER2){
MASTER2FLAG=TRUE;

```

```

MASTER1FLAG=FALSE;
if((ch&0x0f)==SLAVE1){
    SLAVE1FLAG=TRUE;
    Byte=waitForSequenceNumAndLength;
}
else{
    if(((ch&0x0f)==MASTER1)||((ch&0x0f)==MASTER2))
        STORE=FALSE;
    Byte=endLoop;
    State=Idle;
    Event=noEvent;
}
}
/*----- Other Slave or Unknown Address -----*/
else{
    Byte=endLoop;
    State=Idle;
    Event=noEvent;
}
break;
/*----- Verify Sequence Number and Length -----*/
case waitForSequenceNumAndLength:
    if((ch>>6==1)||(ch>>6==2)){
/*----- Verify Message Sequence Number and Length -----*/
        if(STORE&&(ch>>6!=RxSequenceNum)){
            VALID_SEQUENCENUM=TRUE;
            StoreRxSequenceNum=ch>>6;
            if((ch&0x3f==(c-overheads)){
                Byte=waitForData;
                VALID_LENGTH=TRUE;
            }
}
/*----- Invalid Length -----*/
else{
    Byte=endLoop;
    Event=sendNack;
}
}
/*----- Previous Message -----*/
else if(STORE&&(ch>>6==RxSequenceNum)){
    Event=sameMsg;
    Byte=endLoop;
    break;
}
/*----- First Message -----*/
else if(!STORE){
    VALID_SEQUENCENUM=TRUE;
    StoreRxSequenceNum=ch>>6;
    RxSequenceNum=ch>>6;
}

```

```

    if((ch&0x3f)==(c-overheads)){
        Byte=waitForData;
        VALID_LENGTH=TRUE;
    }
/*----- Invalid Length -----*/
    else{
        Byte=endLoop;
        Event=sendNack;
    }
}
/*----- Unexpected Sequence Number -----*/
    else{
        Byte=endLoop;
        State=Idle;
        Event=noEvent;
    }
break;
/*----- Store Message -----*/
case waitForData:
    if(j<c-overheads){
        RxChecksum=RxChecksum^ch;        // Calculates Checksum on Rx Bytes
        BUF[j]=ch;
        j++;
        if(j==c-overheads)
            Byte=waitForChecksum;
    }
    else{
        Byte=endLoop;
        Event=sendNack;
    }
break;

/*----- Verify Checksum -----*/
case waitForChecksum:
    if(ch==RxChecksum){                // Compare the Checksums
        NO_ERRFLAG=TRUE;
        Byte=waitForAddByte;
    }
    else{
        Byte=endLoop;
        Event=sendNack;
    }
break;
/*----- END LOOP -----*/
case endLoop:
    VALID_LENGTH=VALID_SEQUENCENUM=NO_ERRFLAG=FALSE;
    Byte=endLoop;

```

```

    break;
}
}
}

/*===== VERIFY MESSAGE =====
Verifies the Processed Message
-----*/

void Verify_Msg(unchar c, unchar *Message){
    unchar i=0;
    if((NO_ERRFLAG)&&(SLAVE1FLAG)
    &&(VALID_LENGTH)&&(VALID_SEQUENCENUM)){
        NO_ERRFLAG=SLAVE1FLAG=VALID_LENGTH=VALID_SEQUENCENUM=FALSE;
        Event=sendAck;
        STORE=TRUE;
        RxSequenceNum=StoreRxSequenceNum;
        while(i<c){
            DataAvail[i++]=*Message;
            Message++;
        }
        Data_Avail(&DataAvail);
    }
}

/*===== SEND BYTE AT A TIME =====*/

void sendByte(int c){
    SBUF=c;
    while(!CLEAR){}
    CLEAR=FALSE;
}

/*===== SLIP DATA BYTES =====
Add sequences <65><66>if<EOM>,<65><67>if<EOM>&<65><68>if<65> are in data.
-----*/

void sendSlip(unchar c){
    if(c==SOM){
        sendByte(Slip1);
        sendByte(Slip2);
    }
    else if(c==EOM){
        sendByte(Slip1);
        sendByte(Slip3);
    }
    else if(c==Slip1){
        sendByte(Slip1);
    }
}

```

```

sendByte(Slip4);
}
else
sendByte(c);
}

```

```

/*===== TX FRAME =====
Transmit Frame Bytes
-----*/

```

```

void sendFrame(unchar c, unchar *Message){
int i;
sendByte(SOM);
for(i=0;i<c;i++){
sendSlip(*Message);
Message++;
}
sendByte(EOM);
}

```

```

/*===== MESSAGE FRAME =====
Make Frame
-----*/

```

```

void makeFrame(unchar c, unchar *Message){
unchar i;
unchar fin=0; // Reset Counters
TxChecksum=0;
if(MASTER1FLAG){
MASTER1FLAG=FALSE;
TXBUF[fin++]=(SLAVE1<<4)|MASTER1;
}
else if(MASTER2FLAG){
MASTER2FLAG=FALSE;
TXBUF[fin++]=(SLAVE1<<4)|MASTER2;
}
TXBUF[fin++]=(RxSequenceNum<<6)c;
for(i=0;i<c;i++){
TxChecksum=TxChecksum^*Message; // Calculates Checksum on Tx Bytes
TXBUF[fin++]=*Message;
Message++;
}
TXBUF[fin]=TxChecksum;
}

```

```

/*===== LOAD MESSAGES =====
Calculates Message Length
-----*/

```

```

void loadMsg(unchar *Data){
Length=0;
while(*Data!=0){
  DATABUF[Length++]=*Data;
  Data++;
}
}

```

```

/*===== TX APPLICATION LAYER =====
Reads Data from Memory and Pass it LoadMsg function per Request
-----*/

```

```

void Data_Ready(void){
unchar i=0;
unchar j=0;
Data_Pointer=&Magnetometer;
while(j<SIZE)DataReady[j++]=0;
while(*Data_Pointer!=0){
  DataReady[i++]=*Data_Pointer;
  Data_Pointer++;
}
}

```

```

/*===== INITIALIZATION =====*/

```

```

void Main_Init (void){

TMOD=0x21;           // Timer Mode Select register.
                    // Timer0 in mode1 = 16bit timer.
                    // Gate-control via TR0. Timer-mode
                    // Input from internal system clock.
                    // Timer1 in mode1 = 16bit timer.
                    // Gate-control via TR0. Timer-mode
                    // Input from internal system clock.

TH1 =0xFA;          // 9600 baud 11.05920 MHz
TCON=0x50;         // Start Timer0 & Timer1.
SCON=0x50;         // Enable Receiver
IE =0x92;          // Enable interrupts
                    // generation on the uP (IE).

TH0 =0xDC;         // 1's complement of 10ms/[(1/Fosc)*6]
TL0 =0x00;         // =[TH0,TL0].Fosc=11.0592MHz=>[0xDC,0x00]
PCON=0x80;         // Double baud rate

```

```
}
```

```
/*===== MAIN FUNCTION =====*/  
Slave Node Finite State Machine  
-----*/
```

```
void main(void){  
Main_Init();  
State=Idle;  
Event=noEvent;
```

```
/*===== OPERATIONS =====*/  
FOREVER{  
switch(State){
```

```
/*===== IDLE STATE =====*/  
case Idle:  
TxEnable=FALSE; // Disable Driver  
Byte=waitForAddByte;  
waitForMsg();  
if(Event==RcvMsg)State=ProcMsg;  
break;
```

```
/*===== PROCESSING MESSAGE STATE =====*/  
  
case ProcMsg:  
Reply_Timer=TRUE;  
ProcessMsg(RxLength,&PROCBUF);  
Verify_Msg(RxLength-overheads,&BUF);  
switch(Event){
```

```
/*----- SENDING REPLY MESSAGE -----*/  
case sendAck:  
Reply_Timer=FALSE; // Stop Timer  
if(Time_Out)Event=timeOut;  
else{  
TxEnable=TRUE; // Disable Driver  
Data_Ready();  
loadMsg(&DataReady);  
makeFrame(Length,&DATABUF);  
sendFrame(Length+overheads,&TXBUF);  
State=Idle;  
Event=noEvent;  
}  
break;
```

```
/*----- SENDING NEGATIVE ACKNOWLEDGEMENT -----*/  
case sendNack:  
Reply_Timer=FALSE; // Stop Timer
```

```

if(Time_Out)Event=timeOut;
else{
    TxEnable=TRUE;                // Enable Driver
    DataReady[0]=NACK;
    RxSequenceNum=StoreRxSequenceNum;
    loadMsg(DataReady);
    makeFrame(Length,&ATABUF);
    sendFrame(Length+overheads,&TXBUF);
    State=Idle;
    Event=noEvent;
}
break;
/*----- Time Out -----*/
case timeOut:
    Time_Out=FALSE;                // Clear Time-Out Flag
    State=Idle;
    Event=noEvent;
break;
/*----- Previous Message Received -----*/
case sameMsg:
    State=RtxMsg;
break;
}
break;

/*===== RETRANSMIT MESSAGE STATE =====*/
case RtxMsg:
    Reply_Timer=FALSE;                // Stop Timer
    if(Time_Out)Event=timeOut;
    else{
        TxEnable=TRUE;                // Enable Driver
        sendFrame(Length+overheads,&TXBUF);
        State=Idle;
        Event=noEvent;
    }
break;
}
/*===== END OF STATES =====*/
}
}
/*===== END OF PROGRAM =====*/

```

Bibliography

- [Hal96] Fred Halsall. *Data Communications, Computer Networks and Open Systems*. Addison-Wesley Publishers Ltd, Fourth edition, 1996.
- [Holz91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc, 1991.
- [Kop97] Hermann Kopetz. *Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997
- [Lap97] Philip A. Laplante. *Real-time Systems Design and Analysis: An Engineer's Handbook*. IEEE Press, second edition, 1997.
- [Sch98] Thomas W. Schultz. *C and the 8031*. Prentice-Hall, Inc, second edition, 1998.
- [Sla98] Robert Slater. *Distributed Dependability*. 18-849b Dependable Embedded System. Carnegie Melon University, 1998.
- [Sta00] William Stalling. *Data and Computer Communications*. Prentice-Hall, Inc, sixth edition, 2000.
- [Tan81] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, Inc, 1981.
- [UK94] Bhargav P. Upender and Philip J. Koopman, Jr. *Embedded System Programming*. 7(11), November 1994

- [WL99] James R. Wertz and Wiley J. Larson. *Space Mission Analysis and Design*. Third edition, Microcosm Press and Kluwer Academic Publishers, 1999.
- [Zim80] Hubert Zimmermann. *OSI Reference Model - The ISO Model Architecture for Open System Connection*. IEEE Trans. On Communications, Vol.COM-28 No. 4, April 1998.



magano_design_2001

