

Real Time Scheduler for Transport Protocols

DATA ENTERED



Acc. No. (VMO) T-1406



DATA ENTERED

By

Samia Sherwani

Supervised by

Dr. Malik Sikander Hayat Khiyal



**Department of Computer Science
Faculty of Applied Sciences
International Islamic University Islamabad
2006**

DATA ENTERED

Doc. No. IT-1106 ★

Data Entered
Jat-2

MS.
004/16
SAR

- 1- Microcontrollers
- 2- Programmable Controllers



Department of Computer Science
International Islamic University, Islamabad

Dated: November 25, 2006

Final Approval

It is certified that we had read the project report submitted by Miss Samia Sherwani, Registration no: 149-CS/MS/2003 and it is our judgment that this project is of sufficient MS standard to warrant its acceptance by International Islamic University, Islamabad for MS degree in Computer Science.

COMMITTEE

External Examiner

Mr. Shaftab Ahmad
Faculty Member,
Bahria University,
Islamabad.



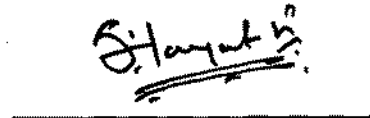
Internal Examiner

Miss Munera Bano
Faculty Member,
Department of Computer Science,
Faculty of Applied Sciences,
International Islamic University, Islamabad



Supervisor

Dr. Malik Sikander Hayat Khiyal
Head,
Department of Computer Science
Faculty of Applied Sciences,
International Islamic University, Islamabad





A dissertation submitted to the
Department of Computer Science,
Faculty of Applied Sciences,
International Islamic University, Islamabad
as a partial fulfillment of the requirement
for the award of the degree of
MS in Computer Science.

Dedication

Dedicated to my Parents

After Allah Almighty, I am completely indebted to my parents for this dissertation, whose affection has always been a source of encouragement for me, and whose prayers always turn out to be a key to my success.

Samia Sherwani

149-CS/MS/2003

Acknowledgment

I bestow all praises, acclamation and appreciation to *Almighty Allah*, The Most Merciful and Compassionate, The most Gracious and Beneficent, Whose bounteous blessings enabled me to pursue and perceive higher ideals of life. All praises for His *Holy Prophet Muhammad (PBUH)* who enabled us to recognize our Lord and Creator and brought to us the real source of knowledge from Allah, *The Qur'an*, and who is a role model for us in every aspect of life.

I would consider it a proud privileged to express my cordial gratitude and deep sense of obligation to my reverend supervisor *Dr. Malik Sikander Hayat Khiyal*, Head, Department Of Computer Sciences, Faculty of Applied Sciences, International Islamic University, Islamabad for his dexterous guidance, inspiring attitude, untiring help and kind behavior throughout the project efforts and presentation of this manuscript.

I will not be out of place to express my profound admiration for my parents and brothers for their prayers for my success. It was mainly due to my family's moral support and financial help during my entire academic career that enabled me to complete my work dedicatedly. I once again would like to admit that I owe all my achievements to my most loving parents, who mean most to me, for their prayers are more precious than any treasure on earth.

Samia Sherwani
149-CS/MS/2003

Declaration

I hereby, declare that I have developed this scheduler for transport protocols and the accompanied report entirely on the basis of my personal efforts made under the guidance of my teachers and supervisor. No portion of the work presented in this report has been submitted in support of any application for any other degree or qualification of this or any other university or institute.

Samia Sherwani
149-CS/MS/2003
samiasherwani@hotmail.com

Project in Brief

Project Title	Real Time Scheduler for Transport Protocols
Objective	Developed multitasking scheduling technique (Collaborative Multitasking) for the processors or microcontrollers which do not have built-in multitasking support.
Undertaken by	Samia Sherwani 149-CS/MS/2003
Supervised by	Dr. Malik Sikander Hayat Khiyal Head of Computer Science Department, Faculty of Applied Sciences, International Islamic University Islamabad.
Starting Date	July 2005
Completion Date	March 2006
Tool Used	Microsoft Visual C++, Keil's Compiler
Operating System	Windows 2000/Windows XP
System Used	Intel Pentium IV

Abstract

Real-Time Operating Systems (RTOS) currently available in industry, for embedded systems, require multitasking support in the targeted processor. The category of such operating systems is known as Pre-emptive Multitasking Kernels. But multitasking support is not provided by all processors. Our aim is to develop multitasking scheduling technique (Collaborative Multitasking) for the processors or microcontrollers which do not have built-in multitasking support like support for context switching, for example, 89C51 Microcontroller, 89C52.

Table of Contents

Chapter	Content	Page No.
1.	Introduction.....	1
1.1	Preemptive Multitasking.....	1
1.2	Collaborative Multitasking.....	2
1.3	Real Time Behavior of the System.....	3
1.4	Aim and Objectives.....	3
1.5	Core Operating System of TCP/IP Stack.....	3
1.6	TCP/IP Application Layer APIs.....	5
1.6.1	Create Socket.....	5
1.6.2	Free Socket.....	5
1.6.3	Create TCP Connection.....	5
1.6.4	Finish TCP Connection.....	6
1.6.5	Send TCP Data.....	6
1.6.6	Receive TCP Data.....	6
1.7	Protocols Implementation.....	7
2.	Communication Protocols.....	8
2.1	Transmission Control Protocol.....	8
2.1.1	TCP Header Format.....	9
2.1.2	Protocol Operation.....	10
2.2	Unreliable Datagram Protocol.....	13
2.2.1	UDP Header Format.....	13
2.2.2	Protocol Operation.....	14
2.3	Internet Protocol (IP).....	15
2.3.1	IP Packet Format.....	15
2.3.2	IP Addressing.....	18
2.3.3	Fragmentation and Reassembly.....	20
2.4	Point to Point Protocol.....	20
2.4.1	PPP Frame Format.....	22
2.4.2	Link Control Protocol.....	23
2.4.3	Network Control Protocols.....	23
2.4.4	General Operation.....	23
2.4.5	Physical Layer Requirements.....	24
3.	Hardware Level Dependencies in Multitasking.....	25
3.1	Scheduling Theory.....	25
3.1.1	Task State Segment.....	25
3.1.2	TSS Descriptor.....	26
3.1.3	Task Register.....	27
3.1.4	Task Gate Descriptor.....	27
3.1.5	Task Switching.....	28
3.1.6	Task Linking.....	30

Chapter	Content	Page No.
	3.1.7 Busy Bit Prevents Loops.....	30
	3.1.8 Modifying Task Linkages.....	31
	3.1.9 Task Address Space.....	31
	3.1.10 Task Linear-to-Physical Space Mapping.....	31
	3.1.11 Task Logical Address Space.....	32
3.2	Hardware Level Dependencies.....	32
	3.2.1 Working with hardware under Windows.....	33
	3.2.2 Serial Channels in C166/C167.....	36
4.	Proposed System.....	37
4.1	Real-Time Programming: Common Practice.....	37
	4.1.1 Preemptive Multitasking.....	37
	4.1.2 Limitations in Preemptive Multitasking.....	38
	4.1.3 Collaborative Multitasking Approach.....	39
	4.1.4 Example.....	42
5.	System Analysis and Design.....	44
5.1	Objectory.....	44
5.2	UML.....	44
5.3	Actors, Use Cases and Sequence Diagrams.....	45
	5.3.1 Use case Model.....	45
	5.3.2 Identifying Actors.....	45
	5.3.3 Identifying Use cases.....	46
	5.3.4 Use Case Diagram.....	47
	5.3.5 Sequence Diagram.....	52
6.	System Design.....	57
7.	Implementation.....	68
	7.1 Main Loop.....	68
	7.2 Initialize TCPIP.....	69
	7.3 COM Procedure.....	69
	7.4 PPP Procedure.....	69
	7.5 IP Procedure.....	70
	7.6 TCP Procedure.....	70
	7.7 UDP Procedure.....	71
	7.8 Application Layer Procedure.....	71
8.	Testing and Conclusion.....	72
	8.1 System testing.....	72
	8.2 Development testing.....	72
	8.3 Integration testing.....	73
	8.3.1 Integration test planning.....	73

Chapter	Content	Page No.
	8.3.2 Test planning activities	74
8.4	Stress testing	74
8.5	Acceptance testing	75
8.6	Performance testing	75
8.7	Reliability testing	76
8.8	Security testing	76
8.9	Testing for Real-Time Scheduler for Transport Protocols	77
8.10	Conclusion	80
	Bibliography	81
Appendix A	User Manual – Real Time Scheduler for Transport Protocols	82
Appendix B	Development Environment	92
Appendix C	Publication	

Chapter 1

Introduction

1. Introduction

Real-Time Operating Systems (RTOS) currently available in industry, for embedded systems, require multitasking support in the targetted processor. The category of such operating systems is known as Pre-emptive Multitasking Kernels. But multitasking support is not provided by all processors. Our aims is to develop multitasking scheduling technique (Collaborative Multitasking) for the processors or microcontrollors which do not have built-in multitasking support (that is, support such as for context switching), for example, 89C51 Microcontroller, 89C52.

As described above, most of embedded system processors and micro-controllors do not have multiprogramming support when used in real time environment, while on the other hand, there are some software routines (for example, TCP/IP) which need very efficient level of multiprogramming in order to execute different tasks, for example, TCP protocol, IP Protocol . So, there is need of a multiprogramming scheduler to hold the burden of such routines in real time environment.

Commonly used Real Time Operating Systems (for example, CMX RTOS, PCOS) use pre-emptive multitasking as scheduling approach to execute multiple tasks simultaneously.

1.1 Preemptive Multitasking

Preemptive multitasking is task in which a computer operating system uses some criteria to decide how long to allocate to any one task before giving another task a turn to use the operating system. The act of taking control of the operating system from one task and giving it to another task is called *preempting*. A common criterion for preempting is simply elapsed time (this kind of system is sometimes called *time sharing* or *time slicing*). In some operating systems, some applications can be given higher priority than other applications, giving the higher priority programs control as soon as they are initiated and perhaps longer time slices.

Restrictions:

- Multitasking support in hardware or processor is needed
- High processing speed required
- High cost

The micro controllers widely used in embedded systems such as 89C51, 89C52, PIC Controller, CMX 851, do not have support for pre-emptive multitasking. Our proposed system is to introduce a **new multitasking approach** called collaborative multitasking in order to execute tasks or routines for transport protocols.

The proposed approach is different from conventional multitasking (Preemptive) in way that it does not require any system resources and support, that is, this approach can be implemented in tiny micro controllers, which are not basically meant for running multiple tasks at same time.

1.2 Collaborative Multitasking

In collaborative multitasking, tasks (any user process running on that controller) collaborate with each other in a way that each task executes a part of its route and then releases system resources voluntarily. In this system, each task is represented by function or routine. A task returns after executing a part of it, saves its state and gives control to other task waiting for resources. This sequence executes in a continuous fashion.

Following is an example of implementation of proposed collaborative multitasking approach. Here, we want to perform three tasks at same time. First Task is Display task whose responsibility is control LCD display. Second task is Comm task whose responsibility is to receive any data from comport and process it, and the third one is KeyPad task which scans the keys and gets any activity of key pressing.

Now, these three tasks will collaborate with each other. When Display function will be called, it will scan all the display memory and will refresh it on the screen in one cycle. After that it will return back and Comm task will be invoked. In a single cycle, Comm will scan its COM port, receive any incoming waiting data and process it. After that it will return back and then finally KeyPad task will be invoked. In a cycle, KeyPad will scan all the keys and will refresh keypad memory indicating any key press event. This sequence will execute continuously.

```

main()
{
    InitSys( );

    While(1)
    {
        Display( );
        Comm( );
        KeyPad( );
    }

    QuitSys( );
}

```

For each layer of TCP/IP, there will be a separate task. All the layers will work in a collaborative fashion. The layers will interact with other layers through job queues. In order to hand over data to other layer, each layer will place the job in queue of next layer.

In a cycle, each layer will process one incoming and one outgoing packet. In total, five simultaneous collections will be supported.

1.3 Real Time Behavior of the System

Timeliness is one of the most important properties of real-time systems. Formal proofs, static analysis and scheduling theory which aim to guarantee timeliness in dependable real-time systems require full knowledge of worst-case execution times, load patterns, task dependencies, and arrival rates of requests. Such information is seldom available, and if those techniques are applied they must often be based on estimations that cannot be guaranteed to be correct.

For example it has become increasingly complex to model a state-of-the-art processor in order to predict timing characteristics of tasks.

1.4 Aim and Objectives

Scheduler is designed keeping in mind following objectives:

- Collaborative in nature, that is, no need of any multitasking support.
- Portable to any platform C166/167, Intel 8051, PIC Controller, CMX 851
- RTOS integration will be supported
- Real time system support
- Five simultaneous connections will be supported
- Modular approach

1.5 Core Operating System of TCP/IP Stack

The core operating system of this TCP/IP stack depends on the processes to regularly relinquish control of the operating system so other processes have the opportunity to gain their time slices. The main thread initializes all the layers, distributes the time slices by calling respective processes.

Each layer works in two directions, that is, it processes data from upper as well as lower layer, as shown in the figure 1.1.

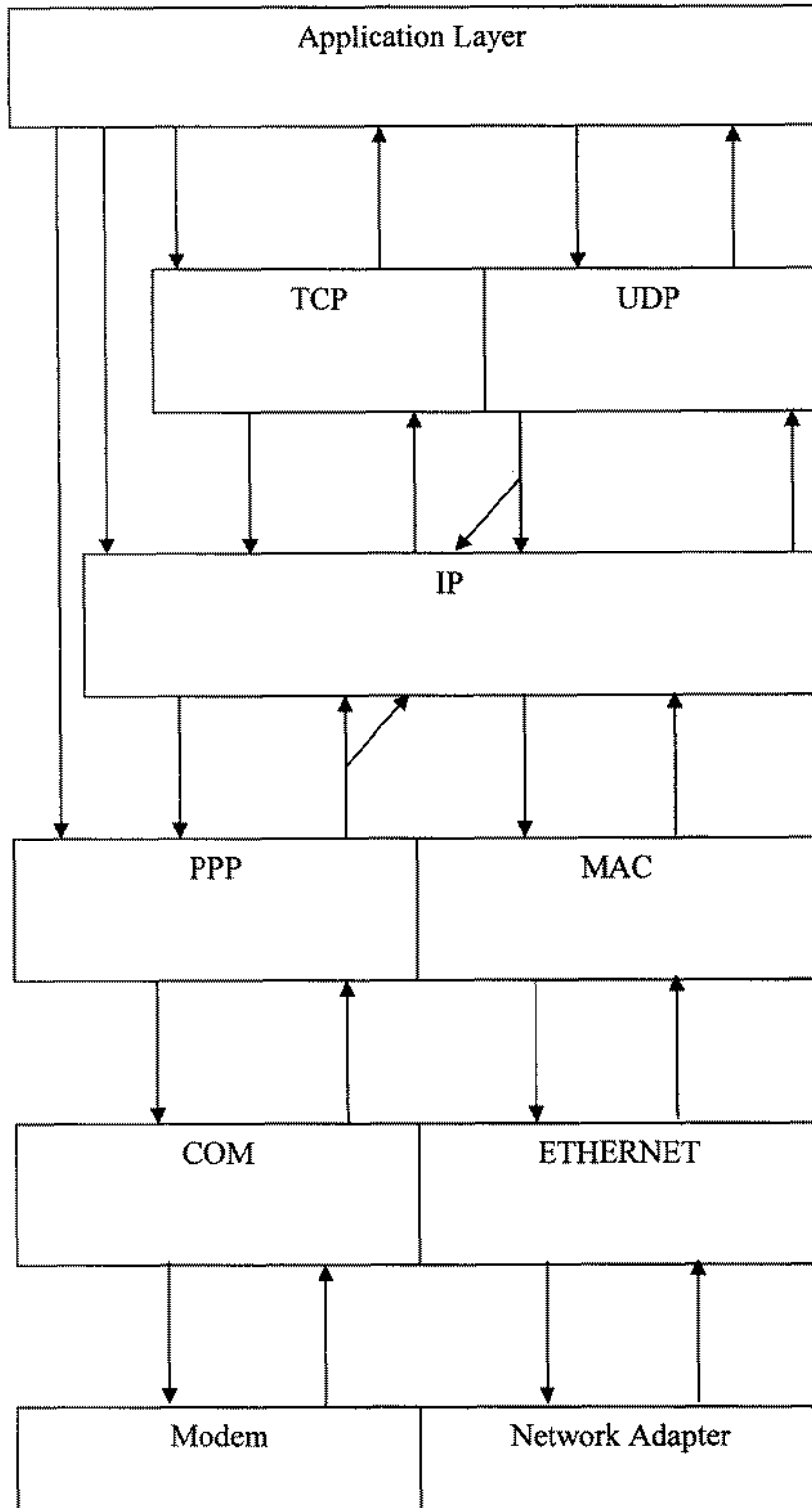


Figure 1.1: Operations performed at different layers

Each layer has two task queues associated with it, one for each direction:

- Layer up task queue, that contains pointer to the buffer received from upper layer and is ready to be processed according to the command associated with the buffer and status of the message flag associated with that direction.
- Layer down task queue, that contains pointer to the buffer received from upper layer. Message flags associated with each process to control sub-processes.

1.6 TCP/IP Application Layer APIs

In this section, the expected application layer APIs will be discussed.

1.6.1 Create Socket

This function Searches for a free socket and returns no of the socket and returns -1 if no socket is available.

1.6.2 Free Socket

- This function marks a currently occupied socket as free. Calling this function makes the currently specified socket as free.
- It takes one parameter, which is identifier of the socket, which is to be freed and returns 1 on successfully freeing the socket and returns 0 on failure.

1.6.3 Create TCP Connection

- This function makes a currently occupied socket connected with remote host using TCP services and allows the application to exchange data.
- This function takes the socket identifier, which is going to be connected as parameter.
- The function returns 1 on successful connection with remote host and returns 0, if connection fails.

- The function returns 1 on successful connection with remote host and returns 0, if connection fails.

1.6.4 Finish TCP Connection

- This function makes a currently occupied socket disconnected with remote host.
- This function takes the socket identifier, which is going to be disconnected as parameter.
- The function returns 1 on successfully disconnection with remote host and returns 0, if disconnection fails.

1.6.5 Send TCP Data

- This function sends a buffer on a currently connected TCP connection and returns 1 on successful transmission of packet and returns 0 on failure.
- This function takes two parameters:
 - a. Socket Identifier
 - b. Buffer number to be transmitted

1.6.6 Receive TCP Data

- This function receives a buffer of data from Currently Active TCP Connection.
- This function halts until a buffer arrives at the socket.
- The function takes socket identifier as parameter and returns the number of buffer received.
- If connection is dropped some other exception occurs then this function returns with invalid (-1) buffer number.

1.7 Protocols Implementation

The following protocols are implemented as per their RFC given below:

TCP	Rfc793
IP	Rfc791
UDP	Rfc768
PPP	Rfc1661
ICMP	Rfc792
SMTP	Rfc821
POP3	Rfc1939

Chapter 2

Communication Protocols

2. Communication Protocols

The Internet protocol suite is a set of communications protocols that implement the protocol stack on which the Internet runs. It is sometimes called the TCP/IP protocol suite, after the two most important protocols in it: the Transmission Control Protocol (TCP) and the Internet Protocol (IP).

The Internet protocol suite can be described by analogy with the OSI model, which describes the layers of a protocol stack, not all of which correspond well with internet practice. In a protocol stack, each layer solves a set of problems involving the transmission of data, and provides a well-defined service to the higher layers. Higher layers are logically closer to the user and deal with more abstract data, relying on lower layers to translate data into forms that can eventually be physically manipulated. [1]

TCP/IP stack is comprised of five layers. Different protocols can be resided at different layers.

1. Application Layer (for example, SMTP, POP3 implemented)
2. Transport Layer (for example, TCP, UDP implemented)
3. Internet Layer (for example, IP, ICMP implemented)
4. Data Link Layer (for example, PPP implemented)
5. Physical Layer (for example, Physical Media, encoding techniques)

2.1 Transmission Control Protocol

Transmission Control Protocol (TCP) is a connection-oriented, reliable-delivery byte-stream transport layer communication protocol. It is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks. The TCP interfaces on one side to user or application processes and on the other side to a lower level protocol such as Internet Protocol. [2]

Applications send streams of 8-bit bytes to TCP for delivery through the network, and TCP divides the byte stream into appropriately sized segments (usually delineated by the maximum transmission unit (MTU) size of the data link layer of the network the computer is attached to). TCP then passes the resulting packets to the Internet Protocol, for delivery through an internet to the TCP module of the entity at the other end. TCP checks to make sure that no packets are lost by giving each byte a sequence number, which is also used to make sure that the data are delivered to the entity at the other end in the correct order. The TCP module at the far end sends back an acknowledgement for bytes which have been successfully received; a timer at the sending TCP will cause a timeout if an acknowledgement is not received within a reasonable round-trip time (or RTT), and the (presumably lost) data will then be re-transmitted. The TCP checks that no bytes are damaged by using a checksum; one is computed at the sender for each block of data before it is sent, and checked at the receiver. Figure 2.1 explains TCP header format.

2.1.1 TCP Header Format

	Bits 0 – 3	4 - 9	10 - 15	16 - 31
+				
0	Source Port			Destination Port
32	Sequence Number			
64	Acknowledgement Number			
96	Data Offset	Reserved	Flags	Window
128	Checksum			Urgent Pointer
160	Options (optional)			
192	Options (cont.)			Padding (to 32)
224	Data			

Figure 2. 1 TCP Header

- Sequence Number: 32 bits**
 The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.
- Acknowledgment Number: 32 bits**
 If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.
- Control Bits: 6 bits (from left to right):**
 URG: Urgent Pointer field significant
 ACK: Acknowledgment field significant
 PSH: Push Function
 RST: Reset the connection
 SYN: Synchronize sequence numbers
 FIN: No more data from sender
- Window: 16 bits**
 The number of data octets beginning with the one indicated in the acknowledgment field, which the sender of this segment is willing to accept.
- Checksum: 16 bits**
 The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header and text. If a segment contains an odd number of header and

text octets to be checked, the last octet is padded on the right with zeros to form a 16-bit word for checksum purposes.

- **Urgent Pointer: 16 bits**

This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only to be interpreted in segments with the URG control bit set.

- **Options: variable**

Options may occupy space at the end of the TCP header and are multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. There are two cases for the format of an option:

Case 1: A single octet of option-kind.

Case 2: An octet of option-kind, an octet of option-length, and the actual option-data octets.

The option-length counts the two octets of option-kind and option-length as well as the option-data octets. Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option must be header padding (i.e., zero). A TCP must implement all options.

Currently defined options include (kind indicated in octal):

Table 2.1 Options in TCP Header

Kind	Length	Meaning
0	-	End of option list.
1	-	No-Operation
2	4	Maximum Segment Size

- **Padding: variable**

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros. [2]

2.1.2 Protocol Operation

TCP connections contain three phases: connection establishment, data transfer and connection termination. A 3-way handshake is used to establish a connection. A four-way handshake is used to disconnect. During connection establishment, parameters such as sequence numbers are initialized to help ensure ordered delivery and robustness.

- **Connection establishment (3-way handshake)**

While it is possible for a pair of end hosts to initiate a connection between them simultaneously, typically one end opens a socket and listens passively for a connection from the other. This is commonly referred to as a passive open, and it designates the server-side of a connection. The client-side of a connection initiates an active open by sending an initial SYN segment to the server as part of the 3-way handshake. The server-side should respond to a valid SYN request with a SYN/ACK. Finally, the client-side should respond to the server with an ACK, completing the 3-way handshake and connection establishment phase.

- **Data transfer**

During the data transfer phase, a number of key mechanisms determine TCP's reliability and robustness. These include using sequence numbers for ordering received TCP segments and detecting duplicate data, checksums for segment error detection, and acknowledgements and timers for detecting and adjusting to loss or delay.

During the TCP connection establishment phase, initial sequence numbers (ISNs) are exchanged between the two TCP speakers. These sequence numbers are used to identify data in the byte stream, and are numbers that identify (and count) application data bytes. There are always pair of sequence numbers included in every TCP segment, which are referred to as the sequence number and the acknowledgement number. A TCP sender refers to its own sequence number simply as the sequence number, while the TCP sender refers to receiver's sequence number as the acknowledgement number. To maintain reliability, a receiver acknowledges TCP segment data by indicating it has received up to some location of contiguous bytes in the stream. An enhancement to TCP, called selective acknowledgement (SACK), allows a TCP receiver to acknowledge out of order blocks.

Through the use of sequence and acknowledgement numbers, TCP can properly deliver received segments in the correct byte stream order to a receiving application. Sequence numbers are 32-bit, unsigned numbers, which wrap to zero on the next byte in the stream after 2³²-1. One key to maintaining robustness and security for TCP connections is in the selection of the ISN.

A 16-bit checksum, consisting of the one's complement of the one's complement sum of the contents of the TCP segment header and data, is computed by a sender, and included in a segment transmission. (The one's complement sum is used because the end-around carry of that method means that it can be computed in any multiple of that length - 16-bit, 32-bit, 64-bit, etc - and the result, once folded, will be the same.) The TCP receiver recomputes the checksum on the received TCP header and data. The complement was used (above) so that the receiver does not have to zero the checksum field, after saving the checksum value elsewhere; instead, the receiver simply computes the one's complement sum with the checksum, and the result should be -0. If so, the segment is assumed to have arrived intact and without error.

Note that the TCP checksum also covers a 96-bit pseudo header containing the Source Address, the Destination Address, the Protocol, and TCP length. This provides protection against misrouted segments. The TCP checksum is a quite weak check by modern standards. Data Link Layers with a high probability of bit error rates may require additional link error correction/detection capabilities. If TCP were to be redesigned today, it would most probably have a 32-bit cyclic redundancy check specified as an error check instead of the current checksum. The weak checksum is partially compensated for by the common use of a CRC or better integrity check at layer 2, below both TCP and IP, such as is used in PPP or the Ethernet frame. However, this does not mean that the 16-bit TCP checksum is redundant: remarkably, surveys of Internet traffic have shown that software and hardware errors that introduce errors in packets between CRC-protected hops are common, and that the end-to-end 16-bit TCP checksum catches most of these simple errors. This is the end-to-end principle at work. Acknowledgements for data sent, or lack of acknowledgements, are used by senders to implicitly interpret network conditions between the TCP sender and receiver. Coupled with timers, TCP senders and receivers can alter the behavior of the flow of data. This is more generally referred to as flow control, congestion control and/or network congestion avoidance. TCP uses a number of mechanisms to achieve high performance and avoid congesting the network (i.e. send data faster than either the network, or the host on the other end, can utilize it). These mechanisms include the use of a sliding window, the slow-start algorithm, the congestion avoidance algorithm, the fast retransmit and fast recovery algorithms, and more. Enhancing TCP to reliably handle loss, minimize errors, manage congestion and go fast in very high-speed environments are ongoing areas of research and standards development.

- **TCP window size**

The TCP receive window size is the amount of received data (in bytes) that can be buffered during a connection. The sending host can send only that amount of data before it must wait for an acknowledgment and window update from the receiving host. The Windows TCP/IP stack is designed to self-tune itself in most environments, and uses larger default window sizes than earlier versions.

- **Window scaling**

For more efficient use of high bandwidth networks, a larger TCP window size may be used. The TCP window size field controls the flow of data and is limited to 2 bytes, or a window size of 65,535 bytes.

Since the size field cannot be expanded, a scaling factor is used. TCP window scale is an option used to increase the maximum window size from 65,535 bytes to 1 Gigabyte.

The window scale option is used only during the TCP 3-way handshake. The window scale value represents the number of bits to left-shift the 16-bit window size field. The window scale value can be set from 0 (no shift) to 14.

- **Connection termination**

The connection termination phase uses a four-way handshake, with each side of the connection terminating independently. When an endpoint wishes to stop its half of the connection, it transmits a FIN packet, which the other end acknowledges with an ACK.

Therefore, a typical teardown requires a pair of FIN and ACK segments from each TCP end-point.

A connection can be "half-open", in which case one side has terminated its end, but the other has not. The side, which has terminated can no longer send any data into the connection, but the other side can.

- **TCP ports**

TCP uses the notion of port numbers to identify sending and receiving applications. Each side of a TCP connection has an associated 16-bit unsigned port number assigned to the sending or receiving application. Ports are categorized into three basic categories: well known, registered and dynamic/private. The well-known ports are assigned by the Internet Assigned Numbers Authority (IANA) and are typically used by system-level or root processes. Well-known applications running as servers and passively listening for connections, typically use these ports. Some examples include: FTP (21), TELNET (23), SMTP (25) and HTTP (80). Registered ports are typically used by end user applications as ephemeral source ports when contacting servers, but they can also identify named services that have been registered by a third party. Dynamic/private ports can also be used by end user applications, but are less commonly so. Dynamic/private ports do not contain any meaning outside of any particular TCP connection. There are 65535 possible ports officially recognized.

2.2 Unreliable Datagram Protocol

The User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol is used as the underlying protocol. This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. [3] Figure 2.2 explains UDP header format.

2.2.1 UDP Header Format

+	Bits 0 - 15	16 - 31
0	Source Port	Destination Port
32	Length	Checksum
64	Data	

Figure 2.2 UDP Header

- **Source Port** is an optional field, when meaningful, it indicates the port of the sending process, and may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.

- **Destination Port** has a meaning within the context of particular internet destination address.
- **Length** is the length in octets of this user datagram including this header and the data. (This means the minimum value of the length is eight.)
- **Checksum** is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets. The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol, and the UDP length. This information gives protection against misrouted datagrams. This checksum procedure is the same as is used in TCP. If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all zero transmitted checksum value means that the transmitter generated no checksum.

2.2.2 Protocol Operation

The UDP header consists of only 4 header fields of which two are optional. The source and destination port fields are 16-bit fields that identify the sending and receiving process. Since UDP is stateless and a UDP sender may not solicit replies, the source port is optional. If not used, the source port should be set to zero. The port fields are followed by a mandatory length field indicating the length in bytes of the UDP datagram including the data. The minimum value is 8 bytes. The remaining header field is a 16-bit checksum field covering the header and data. The checksum is also optional, but is almost always used in practice.

Lacking reliability, UDP applications must generally be willing to accept some loss, errors or duplication. Some applications such as TFTP may add rudimentary reliability mechanisms into the application layer as needed. Most often, UDP applications do not require reliability mechanisms and may even be hindered by them. Streaming media, real-time multiplayer games and voice over IP (VoIP) are examples of applications that often use UDP. If an application requires a high degree of reliability, a protocol such as the Transmission Control Protocol or erasure codes may be used instead.

Lacking any congestion avoidance and control mechanisms, network-based mechanisms are required to minimize potential congestion collapse effects of uncontrolled, high rate UDP traffic loads. In other words, since UDP senders cannot detect congestion, network-based elements such as routers using packet queueing and dropping techniques will often be the only tool available to slow down excessive UDP traffic. The Datagram Congestion Control Protocol (DCCP) is being designed as a partial solution to this potential problem by adding end host congestion control behavior to high-rate UDP streams such as streaming media.

While the total amount of UDP traffic found on a typical network is often on the order of only a few percent, numerous key applications use UDP, including the Domain Name System (DNS),

the simple network management protocol (SNMP), the Dynamic Host Configuration Protocol (DHCP) and the Routing Information Protocol (RIP), just to name a few. [3]

2.3 Internet Protocol (IP)

The Internet Protocol (IP) is a network-layer (Layer 3) protocol that contains addressing information and some control information that enables packets to be routed. IP is primary network-layer protocol in the Internet protocol suite. Along with the Transmission Control Protocol (TCP), IP represents the heart of the Internet protocols. IP has two primary responsibilities: providing connectionless, best-effort delivery of datagrams through Internet; and providing fragmentation and reassembly of datagrams to support data links with different maximum-transmission unit (MTU) sizes. Figure 2.3 explains IP header format.

2.3.1 IP Packet Format

Fourteen fields comprise an IP packet. The following discussion describes the IP packet fields illustrated in :

- **Version: 4 bits**
The Version field indicates the format of the internet header. This document describes version 4.
- **IHL: 4 bits**
Internet Header Length is the length of the internet header in 32 bit words, and thus points to beginning of the data. Note that the minimum value for a correct header is 5.
- **Type of Service: 8 bits**
The Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important than other traffic (generally by accepting only traffic above a certain precedence at time of high load). The major choice is a three-way tradeoff between low-delay, high-reliability, and high-throughput.

Bits 0-2: Precedence.

Bit 3: 0 = Normal Delay, 1 = Low Delay.

Bits 4: 0 = Normal Throughput, 1 = High Throughput.

Bits 5: 0 = Normal Reliability, 1 = High Reliability.

Bit 6-7: Reserved for Future Use.

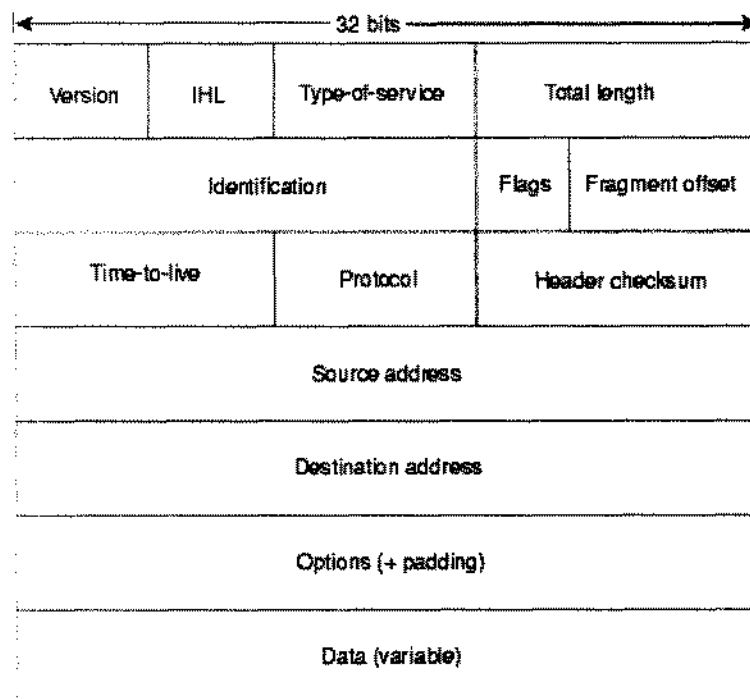


Figure 2.3 IP Header

- **Total Length: 16 bits**

Total Length is the length of the datagram, measured in octets, including internet header and data. This field allows the length of a datagram to be up to 65,535 octets. Such long datagrams are impractical for most hosts and networks. All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments). It is recommended that hosts only send datagrams larger than 576 octets if they have assurance that the destination is prepared to accept the larger datagrams. The number 576 is selected to allow a reasonable sized data block to be transmitted in addition to the required header information. For example, this size allows a data block of 512 octets plus 64 header octets to fit in a datagram. The maximal internet header is 60 octets, and a typical internet header is 20 octets, allowing a margin for headers of higher level protocols.

- **Identification: 16 bits**

An identifying value assigned by the sender to aid in assembling the fragments of a datagram.

- **Flags: 3 bits**

Various control flags are:

Bit 0: reserved, must be zero

Bit 1: (DF) 0 = May Fragment, 1 = Don't Fragment.

Bit 2: (MF) 0 = Last Fragment, 1 = More Fragments.

- **Fragment Offset: 13 bits**
This field indicates where in the datagram this fragment belongs. The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.
- **Time to Live: 8 bits**
This field indicates the maximum time the datagram is allowed to remain in the internet system. If this field contains the value zero, then the datagram must be destroyed. This field is modified in internet header processing. The time is measured in units of seconds, but since every module that processes a datagram must decrease the TTL by at least one even if it process the datagram in less than a second, the TTL must be thought of only as an upper bound on the time a datagram may exist. The intention is to cause undeliverable datagrams to be discarded, and to bound the maximum datagram lifetime.
- **Protocol: 8 bits**
This field indicates the next level protocol used in the data portion of the internet datagram.
- **Header Checksum: 16 bits**
Checksum is on the header only. Since some header fields change (for example, time to live), this is recomputed and verified at each point that the internet header is processed. The checksum algorithm is:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero. This is a simple to compute checksum and experimental evidence indicates it is adequate, but it is provisional and may be replaced by a CRC procedure, depending on further experience.

- **Options: variable**
The options may appear or not in datagrams. They must be implemented by all IP modules (host and gateways). What is optional is their transmission in any particular datagram, not their implementation. In some environments the security option may be required in all datagrams. The option field is variable in length. There may be zero or more options. There are two cases for the format of an option:

Case 1: A single octet of option-type.

Case 2: An option-type octet, an option-length octet, and the actual option-data octets.

The option-length octet counts the option-type octet and the option-length octet as well as the option-data octets. The option-type octet is viewed as having 3 fields:

- 1 bit copied flag,
- 2 bits option class,
- 5 bits option number.

The copied flag indicates that this option is copied into all fragments on fragmentation.

- 0 = not copied
- 1 = copied

The option classes are:

- 0 = control
- 1 = reserved for future use
- 2 = debugging and measurement
- 3 = reserved for future use

The following internet options are defined:

CLASS NUMBER LENGTH DESCRIPTION

CLASS	NUMBER	LENGTH	DESCRIPTION
0	0	-	End of Option list. This option occupies only 1 octet; it has no length octet.
0	1	-	No Operation. This option occupies only 1 octet; it has no length octet.
0	2	11	Security. Used to carry Security, Compartmentation, User Group (TCC), and Handling Restriction Codes compatible with DOD requirements.
0	3	var.	Loose Source Routing. Used to route the internet datagram based on information supplied by the source.
0	9	var.	Strict Source Routing. Used to route the internet datagram based on information supplied by the source.
0	7	var.	Record Route. Used to trace the route an internet datagram takes.
0	8	4	Stream ID. Used to carry the stream identifier.
2	4	var.	Internet Timestamp.

- **Padding: variable**

The internet header padding is used to ensure that the internet header ends on a 32 bit boundary. The padding is zero.

2.3.2 IP Addressing

As with any other network-layer protocol, the IP addressing scheme is integral to the process of routing IP datagrams through an internetwork. Each IP address has specific components and follows a basic format. These IP addresses can be subdivided and used to create addresses for subnetworks, as discussed in more detail later in this chapter.

Each host on a TCP/IP network is assigned a unique 32-bit logical address that is divided into two main parts: the network number and the host number. The network number identifies a network and must be assigned by the Internet Network Information Center (InterNIC) if the network is to be part of the Internet. An Internet Service Provider (ISP) can obtain blocks of network addresses from the InterNIC and can itself assign address space as necessary. The host number identifies a host on a network and is assigned by the local network administrator.

- **IP Address Format**

The 32-bit IP address is grouped eight bits at a time, separated by dots, and represented in decimal format (known as *dotted decimal notation*). Each bit in the octet has a binary weight (128, 64, 32, 16, 8, 4, 2, 1). The minimum value for an octet is 0, and the maximum value for an octet is 255. The figure 2.4 illustrates the basic format of an IP address.

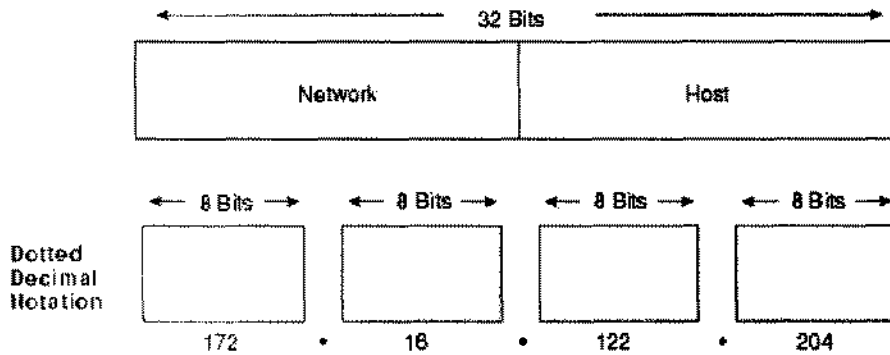


Figure 2. 4 IP Addressing

- **IP Address Classes**

IP addressing supports five different address classes: A, B, C, D, and E. Only classes A, B, and C are available for commercial use. The left-most (high-order) bits indicate the network class. Table 2.2 provides reference information about the five IP address classes.

Table 2. 2 Details of IP Classes

IP Class	Format	Purpose	High-Order Bit(s)	Address Range	No. Bits Network/Host	Max. Hosts
A	N.H.H.H	Few large organizations	0	1.0.0.0 to 126.0.0.0	7/24	$(2^{24} - 2)$
B	N.N.H.H	Medium-size organizations	1, 0	128.1.0.0 to 191.254.0.0	14/16	$(2^{16} - 2)$
C	N.N.N.H	Relatively small organizations	1, 1, 0	192.0.1.0 to 223.255.254.0	22/8	$(2^8 - 2)$
D	N/A	Multicast Groups (RFC 1112)	1, 1, 1, 0	224.0.0.0 to 239.255.255.255	N/A (not for commercial use)	N/A
E	N/A	Experimental	1, 1, 1, 1	240.0.0.0 to 254.255.255.255	N/A	N/A

2.3.3 Fragmentation and Reassembly

The internet identification field (ID) is used together with the source and destination address, and the protocol fields, to identify datagram fragments for reassembly. The More Fragments flag bit (MF) is set if the datagram is not the last fragment. The Fragment Offset field identifies the fragment location, relative to the beginning of the original unfragmented datagram. Fragments are counted in units of 8 octets. The fragmentation strategy is designed so that an unfragmented datagram has all zero fragmentation information (MF = 0, fragment offset = 0). If an internet datagram is fragmented, its data portion must be broken on 8 octet boundaries.

This format allows $2^{13} = 8192$ fragments of 8 octets each for a total of 65,536 octets. Note that this is consistent with the datagram total length field (of course, the header is counted in the total length and not in the fragments). When fragmentation occurs, some options are copied, but others remain with the first fragment only. Every internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet header may be up to 60 octets, and the minimum fragment is 8 octets. Every internet destination must be able to receive a datagram of 576 octets either in one piece or in fragments to be reassembled.

The fields, which may be affected by fragmentation, include:

- options field
- more fragments flag
- fragment offset
- internet header length field
- total length field
- header checksum

If the Don't Fragment flag (DF) bit is set, then internet fragmentation of this datagram is NOT permitted, although it may be discarded. This can be used to prohibit fragmentation in cases where the receiving host does not have sufficient resources to reassemble internet fragments. One example of use of the Don't Fragment feature is to down line load a small host. A small host could have a bootstrap program that accepts a datagram stores it in memory and then executes it. The fragmentation and reassembly procedures are most easily described by examples. The following procedures are example implementations. General notation in the following pseudo programs: " $=<$ " means "less than or equal", " \neq " means "not equal", " $=$ " means "equal", " $<-$ " means "is set to". Also, " x to y " includes x and excludes y ; for example, " 4 to 7 " would include 4, 5, and 6 (but not 7).

2.4 Point to Point Protocol

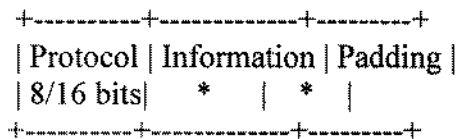
The Point-to-Point Protocol (PPP) originally emerged as an encapsulation protocol for transporting IP traffic over point-to-point links. PPP also established a standard for the assignment and management of IP addresses, asynchronous (start/stop) and bit-oriented synchronous encapsulation, network protocol multiplexing, link configuration, link quality testing, error detection, and option negotiation for such capabilities as network layer address

negotiation and data-compression negotiation. PPP supports these functions by providing an extensible Link Control Protocol (LCP) and a family of Network Control Protocols (NCPs) to negotiate optional configuration parameters and facilities. In addition to IP, PPP supports other protocols, including Novell's Internetwork Packet Exchange (IPX) and DECnet. PPP contains three main components: [4]

- A method for encapsulating multi-protocol datagrams.
- A Link Control Protocol (LCP) for establishing, configuring, and testing the data-link connection.
- A family of Network Control Protocols (NCPs) for establishing and configuring different network-layer protocols.

- **Encapsulation**

The PPP encapsulation provides for multiplexing of different network-layer protocols simultaneously over the same link. The PPP encapsulation has been carefully designed to retain compatibility with most commonly used supporting hardware. Only 8 additional octets are necessary to form the encapsulation when used within the default HDLC-like framing. In environments where bandwidth is at a premium, the encapsulation and framing may be shortened to 2 or 4 octets. To support high-speed implementations, the default encapsulation uses only simple fields, only one of which needs to be examined for demultiplexing. The default header and information fields fall on 32-bit boundaries, and the trailer may be padded to an arbitrary boundary. The PPP encapsulation is used to disambiguate multiprotocol datagrams. This encapsulation requires framing to indicate the beginning and end of the encapsulation. Methods of providing framing are specified in companion documents. A summary of the PPP encapsulation is shown below. The fields are transmitted from left to right.



- **Protocol Field**

The Protocol field is one or two octets, and its value identifies the datagram encapsulated in the Information field of the packet. The field is transmitted and received most significant octet first. The structure of this field is consistent with the ISO 3309 extension mechanism for address fields. All Protocols must be odd; the least significant bit of the least significant octet must equal "1". Also, all Protocols must be assigned such that the least significant bit of the most significant octet equals "0". Frames received, which do not comply with these rules, it must be treated as having an unrecognized Protocol. Protocol field values in the "0***" to "3***" range identify the network-layer protocol of specific packets, and values in the "8****" to "b****" range identify packets belonging to the associated Network Control Protocols (NCPs), if any. Protocol field values in the "4****" to "7****" range are used for protocols with low volume traffic which have no associated NCP. Protocol field values in the "c****" to "f****" range identify packets as link-layer Control Protocols (such as LCP). Up-to-date values of the

Protocol field are specified in the most recent "Assigned Numbers" RFC [2]. This specification reserves the following values:

<u>Value (in hex)</u>	<u>Protocol Name</u>
0001	Padding Protocol
0003 to 001f	reserved (transparency inefficient)
007d	reserved (Control Escape)
00cf	reserved (PPP NLPID)
00ff	reserved (compression inefficient)
8001 to 801f	unused
807d	unused
80cf	unused
80ff	unused
c021	Link Control Protocol
c023	Password Authentication Protocol
c025	Link Quality Report
c223	Challenge Handshake Authentication Protocol

- **Information Field**

The Information field is zero or more octets. The Information field contains the datagram for the protocol specified in the Protocol field. The maximum length for the Information field, including padding, but not including the Protocol field, is termed the Maximum Receive Unit (MRU), which defaults to 1500 octets. By negotiation, consenting PPP implementations may use other values for the MRU.

- **Padding**

On transmission, the Information field MAY be padded with an arbitrary number of octets up to the MRU. It is the responsibility of each protocol to distinguish padding octets from real information.

2.4.1 PPP Frame Format

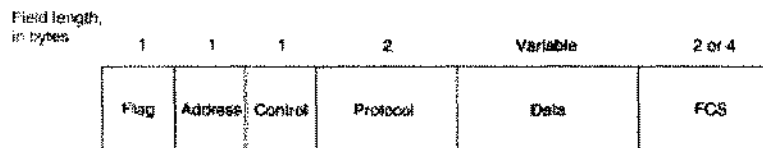


Figure 2. 5 PPP Frame

The following descriptions summarize the PPP frame fields illustrated in figure 2.5:

- **Flag**—A single byte that indicates the beginning or end of a frame. The flag field consists of the binary sequence 01111110.

- **Address**—A single byte that contains the binary sequence 11111111, the standard broadcast address. PPP does not assign individual station addresses.
- **Control**—A single byte that contains the binary sequence 00000011, which calls for transmission of user data in an un-sequenced frame. A connectionless link service similar to that of Logical Link Control (LLC) Type 1 is provided. (For more information about LLC types and frame types, refer to Chapter 16.)
- **Protocol**—Two bytes that identify the protocol encapsulated in the information field of the frame. The most up-to-date values of the protocol field are specified in the most recent Assigned Numbers Request For Comments (RFC).
- **Data**—Zero or more bytes that contain the datagram for the protocol specified in the protocol field. The end of the information field is found by locating the closing flag sequence and allowing 2 bytes for the FCS field. The default maximum length of the information field is 1,500 bytes. By prior agreement, consenting PPP implementations can use other values for the maximum information field length.
- **Frame check sequence (FCS)**—Normally 16 bits (2 bytes). By prior agreement, consenting PPP implementations can use a 32-bit (4-byte) FCS for improved error detection.

2.4.2 Link Control Protocol

In order to be sufficiently versatile to be portable to a wide variety of environments, PPP provides a Link Control Protocol (LCP). The LCP is used to automatically agree upon the encapsulation format options, handle varying limits on sizes of packets, detect a looped-back link and other common misconfiguration errors, and terminate the link. Other optional facilities provided are authentication of the identity of its peer on the link, and determination when a link is functioning properly and when it is failing.

2.4.1 Network Control Protocols

Point-to-Point links tend to exacerbate many problems with the current family of network protocols. For instance, assignment and management of IP addresses, which is a problem even in LAN environments, is especially difficult over circuit-switched point-to-point links (such as dial-up modem servers). These problems are handled by a family of Network Control Protocols (NCPs), which each manage the specific needs required by their respective network-layer protocols. These NCPs are defined in companion documents.

2.4.2 General Operation

To establish communications over a point-to-point link, the originating PPP first sends LCP frames to configure and (optionally) test the data link. After the link has been established and optional facilities have been negotiated as needed by the LCP, the originating PPP sends NCP

frames to choose and configure one or more network layer protocols. When each of the chosen network layer protocols has been configured, packets from each network layer protocol can be sent over the link. The link will remain configured for communications until explicit LCP or NCP frames close the link, or until some external event occurs (for example, an inactivity timer expires or a user intervenes).

2.4.3 Physical Layer Requirements

PPP is capable of operating across any DTE/DCE interface. Examples include EIA/TIA-232-C (formerly RS-232-C), EIA/TIA-422 (formerly RS-422), EIA/TIA-423 (formerly RS-423), and International Telecommunication Union Telecommunication Standardization Sector (ITU-T) (formerly CCITT) V.35. The only absolute requirement imposed by PPP is the provision of a duplex circuit, either dedicated or switched, that can operate in either an asynchronous or synchronous bit-serial mode, transparent to PPP link layer frames. PPP does not impose any restrictions regarding transmission rate other than those imposed by the particular DTE/DCE interface in use.

Chapter 3

Hardware Level Dependencies in Multitasking

3. Hardware Level Dependencies in Multitasking

This chapter explains the terminology and concepts needed to understand the project. It includes concept of multitasking as in higher processors, and analysis of hardware level dependencies, while working under Windows as well as Real-Time Operating System.

3.1 Scheduling Theory

To provide efficient, protected multitasking, 80386 processors employ several special data structures. It does not, however, use special instructions to control multitasking; instead, it interprets ordinary control-transfer instructions differently when they refer to the special data structures. The registers and data structures that support multitasking are:

- Task state segment
- Task state segment descriptor
- Task register
- Task gate descriptor

With these structures the 80386 can rapidly switch execution from one task to another, saving the context of the original task so that the task can be restarted later. In addition to the simple task switch, the 80386 offer two other task-management features: [5]

- Interrupts and exceptions can cause task switches (if needed in the system design). The processor not only switches automatically to the task that handles the interrupt or exception, but it automatically switches back to the interrupted task when the interrupt or exception has been serviced. Interrupt tasks may interrupt lower-priority interrupt tasks to any depth.
- With each switch to another task, the 80386 can also switch to another LDT and to another page directory. Thus each task can have a different logical-to-linear mapping and a different linear-to-physical mapping. This is yet another protection feature, because tasks can be isolated and prevented from interfering with one another.

3.1.1 Task State Segment

All the information the processor needs in order to manage a task is stored in a special type of segment, a task state segment (TSS). The fields of a TSS in 80386 belong to two classes:

- A dynamic set that the processor updates with each switch from the task. This set includes the fields that store:
 - The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI).
 - The segment registers (ES, CS, SS, DS, FS, GS).

- The flags register (EFLAGS).
 - The instruction pointer (EIP).
 - The selector of the TSS of the previously executing task (updated only when a return is expected).
- A static set that the processor reads but does not change. This set includes the fields that store:
- The selector of the task's LDT.
 - The register (PDBR) that contains the base address of the task's page directory (read only when paging is enabled).
 - Pointers to the stacks for privilege levels 0-2.
 - The T-bit (debug trap bit), which causes the processor to raise a debug exception when a task switch occurs.
 - The I/O map base

Task state segments may reside anywhere in the linear space. The only case that requires caution is when the TSS spans a page boundary and the higher-addressed page is not present. In this case, the processor raises an exception if it encounters the not-present page while reading the TSS during a task switch. Such an exception can be avoided by either of two strategies:

- By allocating the TSS so that it does not cross a page boundary.
- By ensuring that both pages are either both-present or both not-present at the time of a task switch. If both pages are not present, then the page-fault handler must make both pages present before restarting the instruction that caused the task switch.

3.1.2 TSS Descriptor

The task state segment, like all other segments, is defined by a descriptor. Format of task segment is as follows:

The B-bit in the type field indicates whether the task is busy. A type code of 9 indicates a non-busy task; a type code of 11 indicates a busy task. Tasks are not reentrant. The B-bit allows the processor to detect an attempt to switch to a task that is already busy.

The BASE, LIMIT, and DPL fields and the G-bit and P-bit have functions similar to their counterparts in data-segment descriptors. The LIMIT field, however, must have a value equal to or greater than 103. An attempt to switch to a task whose TSS descriptor has a limit less than 103 causes an exception. A larger limit is permissible, and a larger limit is required if an I/O permission map is present. A larger limit may also be convenient for systems software if additional data is stored in the same segment as the TSS.

A procedure that has access to a TSS descriptor can cause a task switch. In most systems the DPL fields of TSS descriptors should be set to zero, so that only trusted software has the right to perform task switching.

Having access to a TSS-descriptor does not give a procedure the right to read or modify a TSS. Reading and modification can be accomplished only with another descriptor that redefines the TSS as a data segment. An attempt to load a TSS descriptor into any of the segment registers (CS, SS, DS, ES, FS, GS) causes an exception.

TSS descriptors may reside only in the GDT. An attempt to identify a TSS with a selector that has TI=1 (indicating the current LDT) results in an exception.

3.1.3 Task Register

The task register (TR) identifies the currently executing task by pointing to the TSS. The task register has both a "visible" portion (i.e., can be read and changed by instructions) and an "invisible" portion (maintained by the processor to correspond to the visible portion; cannot be read by any instruction). The selector in the visible portion selects a TSS descriptor in the GDT. The processor uses the invisible portion to cache the base and limit values from the TSS descriptor. Holding the base and limit in a register makes execution of the task more efficient, because the processor does not need to repeatedly fetch these values from memory when it references the TSS of the current task.

The instructions LTR and STR are used to modify and read the visible portion of the task register. Both instructions take one operand, a 16-bit selector located in memory or in a general register.

LTR (Load task register) loads the visible portion of the task register with the selector operand, which must select a TSS descriptor in the GDT. LTR also loads the invisible portion with information from the TSS descriptor selected by the operand. LTR is a privileged instruction; it may be executed only when CPL is zero. LTR is generally used during system initialization to give an initial value to the task register; thereafter, the contents of TR are changed by task switch operations.

STR (Store task register) stores the visible portion of the task register in a general register or memory word. STR is not privileged.

3.1.4 Task Gate Descriptor

A task gate descriptor provides an indirect, protected reference to a TSS.

The SELECTOR field of a task gate must refer to a TSS descriptor. The processor does not use the value of the RPL in this selector. The DPL field of a task gate controls the right to use the descriptor to cause a task switch. A procedure may not select a task gate descriptor unless the

maximum of the selector's RPL and the CPL of the procedure is numerically less than or equal to the DPL of the descriptor. This constraint prevents un-trusted procedures from causing a task switch.

A procedure that has access to a task gate has the power to cause a task switch, just as a procedure that has access to a TSS descriptor. The 80386 has task gates in addition to TSS descriptors to satisfy three needs:

- The need for a task to have a single busy bit. Because the busy-bit is stored in the TSS descriptor, each task should have only one such descriptor. There may, however, be several task gates that select the single TSS descriptor.
- The need to provide selective access to tasks. Task gates fulfill this need, because they can reside in LDTs and can have a DPL that is different from the TSS descriptor's DPL. A procedure that does not have sufficient privilege to use the TSS descriptor in the GDT (which usually has a DPL of 0) can still switch to another task if it has access to a task gate for that task in its LDT. With task gates, systems software can limit the right to cause task switches to specific tasks.
- The need for an interrupt or exception to cause a task switch. Task gates may also reside in the IDT, making it possible for interrupts and exceptions to cause task switching. When interrupt or exception vectors to an IDT entry that contains a task gate, the 80386 switches to the indicated task. Thus, all tasks in the system can benefit from the protection afforded by isolation from interrupt tasks.

3.1.5 Task Switching

The 80386 switches execution to another task in any of four cases:

- The current task executes a JMP or CALL that refers to a TSS descriptor.
- The current task executes a JMP or CALL that refers to a task gate.
- An interrupt or exception vectors to a task gate in the IDT.
- The current task executes an IRET when the NT flag is set.

JMP, CALL, IRET, interrupts, and exceptions are all ordinary mechanisms of the 80386 that can be used in circumstances that do not require a task switch. Either the type of descriptor referenced or the NT (nested task) bit in the flag word distinguishes between the standard mechanism and the variant that causes a task switch.

To cause a task switch, a JMP or CALL instruction can refer either to a TSS descriptor or to a task gate. The effect is the same in either case: the 80386 switches to the indicated task.

An exception or interrupt causes a task switch when it vectors to a task gate in the IDT. If it vectors to an interrupt or trap gate in the IDT, a task switch does not occur. Whether invoked as a task or as a procedure of the interrupted task, an interrupt handler always returns control to the interrupted procedure in the interrupted task. If the NT flag is set, however, the handler is an interrupt task, and the IRET switches back to the interrupted task.

A task switching operation involves these steps:

- Checking that the current task is allowed to switch to the designated task. Data-access privilege rules apply in the case of JMP or CALL instructions. The DPL of the TSS descriptor or task gate must be less than or equal to the maximum of CPL and the RPL of the gate selector.
- Exceptions, interrupts, and IRETs are permitted to switch tasks regardless of the DPL of the target task gate or TSS descriptor.
- Checking that the TSS descriptor of the new task is marked present and has a valid limit. Any errors up to this point occur in the context of the outgoing task. Errors can be handled in a way that is transparent to applications procedures.
- Saving the state of the current task. The processor finds the base address of the current TSS cached in the task register. It copies the registers into the current TSS (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI, ES, CS, SS, DS, FS, GS, and the flag register). The EIP field of the TSS points to the instruction after the one that caused the task switch.
- Loading the task register with the selector of the incoming task's TSS descriptor, marking the incoming task's TSS descriptor as busy, and setting the TS (task switched) bit of the MSW. The selector is either the operand of a control transfer instruction or is taken from a task gate.
- Loading the incoming task's state from its TSS and resuming execution. The registers loaded are the LDT register; the flag register; the general registers EIP, EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; the segment registers ES, CS, SS, DS, FS, and GS; and PDBR. Any errors detected in this step occur in the context of the incoming task. To an exception handler, it appears that the first instruction of the new task has not yet executed. Note that the state of the outgoing task is always saved when a task switch occurs. If execution of that task is resumed, it starts after the instruction that caused the task switch. The registers are restored to the values they held when the task stopped executing.

Every task switch sets the TS (task switched) bit in the MSW (machine status word). The TS flag is useful to systems software when a coprocessor (such as a numeric coprocessor) is present. The TS bit signals that the context of the coprocessor may not correspond to the current 80386 task.

Exception handlers that field task-switch exceptions in the incoming task should be cautious about taking any action that might load the selector that caused the exception. Such an action

will probably cause another exception, unless the exception handler first examines the selector and fixes any potential problem.

The privilege level at which execution resumes in the incoming task is neither restricted nor affected by the privilege level at which the outgoing task was executing. Because the tasks are isolated by their separate address spaces and TSSs and because privilege rules can be used to prevent improper access to a TSS, no privilege rules are needed to constrain the relation between the CPLs of the tasks. The new task begins executing at the privilege level indicated by the RPL of the CS selector value that is loaded from the TSS.

3.1.6 Task Linking

The back-link field of the TSS and the NT (nested task) bit of the flag word together allow the 80386 to automatically return to a task that called another task or was interrupted by another task. When a CALL instruction, an interrupt instruction, an external interrupt, or an exception causes a switch to a new task, the 80386 automatically fills the back-link of the new TSS with the selector of the outgoing task's TSS and, at the same time, sets the NT bit in the new task's flag register. The NT flag indicates whether the back-link field is valid. The new task releases control by executing an IRET instruction. When interpreting an IRET, the 80386 examines the NT flag. If NT is set, the 80386 switches back to the task selected by the back-link field.

3.1.7 Busy Bit Prevents Loops

The B-bit (busy bit) of the TSS descriptor ensures the integrity of the back-link. A chain of back-links may grow to any length as interrupt tasks interrupt other interrupt tasks or as called tasks call other tasks. The busy bit ensures that the CPU can detect any attempt to create a loop. A loop would indicate an attempt to reenter a task that is already busy; however, the TSS is not a re-entrant resource.

The processor uses the busy bit as follows:

- When switching to a task, the processor automatically sets the busy bit of the new task.
- When switching from a task, the processor automatically clears the busy bit of the old task if that task is not to be placed on the back-link chain (i.e., the instruction causing the task switch is JMP or IRET). If the task is placed on the back-link chain, its busy bit remains set.
- When switching to a task, the processor signals an exception if the busy bit of the new task is already set.

By these actions, the processor prevents a task from switching to itself or to any task that is on a back-link chain, thereby preventing invalid reentry into a task.

The busy bit is effective even in multiprocessor configurations, because the processor automatically asserts a bus lock when it sets or clears the busy bit. This action ensures that two processors do not invoke the same task at the same time.

3.1.8 Modifying Task Linkages

Any modification of the linkage order of tasks should be accomplished only by software that can be trusted to correctly update the back-link and the busy-bit. Such changes may be needed to resume an interrupted task before the task that interrupted it. Trusted software that removes a task from the back-link chain must follow one of the following policies:

- First change the back-link field in the TSS of the interrupting task, then clear the busy-bit in the TSS descriptor of the task removed from the list.
- Ensure that no interrupts occur between updating the back-link chain and the busy bit.

3.1.9 Task Address Space

The LDT selector and PDBR fields of the TSS give software Systems Designers flexibility in utilization of segment and page mapping features of the 80386. By appropriate choice of the segment and page mappings for each task, tasks may share address spaces, may have address spaces that are largely distinct from one another, or may have any degree of sharing between these two extremes.

The ability for tasks to have distinct address spaces is an important aspect of 80386 protections. A module in one task cannot interfere with a module in another task if the modules do not have access to the same address spaces. The flexible memory management features of the 80386 allow systems designers to assign areas of shared address space to those modules of different tasks that are designed to cooperate with each other.

3.1.10 Task Linear-to-Physical Space Mapping

The choices for arranging the linear-to-physical mappings of tasks fall into two general classes:

- One linear-to-physical mapping shared among all tasks. When paging is not enabled, this is the only possibility. Without page tables, all linear addresses map to the same physical addresses.
- When paging is enabled, this style of linear-to-physical mapping results from using one page directory for all tasks. The linear space utilized may exceed the physical space available if the operating system also implements page-level virtual memory.

- Several partially overlapping linear-to-physical mappings. This style is implemented by using a different page directory for each task. Because the PDBR (page directory base register) is loaded from the TSS with each task switch, each task may have a different page directory. In theory, the linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share any physical addresses. In practice, some portion of the linear address spaces of all tasks must map to the same physical addresses. The task state segments must lie in a common space so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear space mapped by the GDT should also be mapped to a common physical space; otherwise, the purpose of the GDT is defeated.

3.1.11 Task Logical Address Space

By itself, a common linear-to-physical space mapping does not enable sharing of data among tasks. To share data, tasks must also have a common logical-to-linear space mapping; i.e., they must also have access to descriptors that point into a shared linear address space. There are three ways to create common logical-to-physical address-space mappings:

- Via the GDT. All tasks have access to the descriptors in the GDT. If those descriptors point into a linear-address space that is mapped to a common physical-address space for all tasks, then the tasks can share data and instructions.
- By sharing LDTs. Two or more tasks can use the same LDT if the LDT selectors in their TSSs select the same LDT segment. Those LDT-resident descriptors that point into a linear space that is mapped to a common physical space permit the tasks to share physical memory. This method of sharing is more selective than sharing by the GDT; the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared areas.
- By descriptor aliases in LDTs. It is possible for certain descriptors of different LDTs to point to the same linear address space. If that linear address space is mapped to the same physical space by the page mapping of the tasks involved, these descriptors permit the tasks to share the common space. Such descriptors are commonly called "aliases". This method of sharing is even more selective than the prior two; other descriptors in the LDTs may point to distinct linear addresses or to linear addresses that are not shared.

3.2 Hardware Level Dependencies

Windows, unlike DOS, is a multitasking system, which makes it impossible to allow every application to directly change the hardware settings, as one application may fail to 'know' about

the changes made to the hardware settings by some other application. To create programs working with hardware under Windows one should use API (application programming interface). This interface allows using Windows system services from application programs. API realization is at that entrusted to the drivers. Windows Driver Developer Kit (DDK) is used to create drivers (there is a separate DDK for every Windows OS). Besides API one can use IOCTL codes (this method was widely used in DOS), but we shall deal with API functions only.

3.2.1 Working with hardware under Windows

API standardizes work with hardware. To get access to hardware the following steps are used:

- Get Handler of the device by calling CreateFile with the device name
- To control the device, call an API function for this device or send IOCTL(input - output control), the latter via DeviceIOctl .

In Windows all input/output ports are presented as files, so work with ports is mainly carried out via I/O functions of the file (CreateFile, CloseHandle, ReadFile, ReadFileEx, WriteFile and WriteFileEx). These functions organize the main interface for opening and closing the connection resource descriptor and carrying out read/write operations. API also includes a set of connection functions, which provide access to connection resources.

The usage of the I/O file and connection functions allows the application to perform the following tasks:

- Getting the serial port descriptor.
- Serial port configuration set and request.
- Reading from or writing into the serial port.
- Control of the given events set, which could occur for this serial port.

Sending the executive instructions to the driver of the device connected with the specified serial port; driver call-in is required for extended functions execution. [6]

3.2.1.1 Open and Close Port

Opening a port is actually getting the descriptor of the serial port. Due to API using CreateFile function can do it. This function results in the creation of a file with a reserved name. It is important when getting access to the corresponding port or device. After the descriptor has been obtained the work with the port is carried out the same way it is with files.

3.2.1.2 Function SetupComm

As COM-ports are asynchronous connection devices, buffers for incoming and outgoing data are provided to make the work with the ports more effective. It is connected with the fact that the data bus baud rates greatly varies from the line baud rates, that's why to optimize the work of the system it is advisable to read data from/write data into the port by batches regardless of when they were received. One can also write data into the buffer and only then start the transmission - it is useful when the batch transmission is needed regardless of whether the system is busy. To set the size of the receiving and transmitting buffers SetupComm function is used.

Function syntax:

```
BOOL SetupComm(HANDLE hFile, DWORD dwInQueue, DWORD dwOutQueue);
```

3.2.1.3 Communications Time-outs

Another major thing affecting the work of read and write operations is time-outs. Time-outs have the following effect on read and write operations. If an operation takes longer than the calculated time-out period, the operation is finished. ReadFile, WriteFile, GetOverlappedResult, or WaitForSingleObject returns no error code. All indicators used to monitor the operation show that it finished successfully. The only way to tell that the operation has timed out is that the number of bytes actually transferred are lower than the number of bytes requested. So, if ReadFile returns TRUE, but fewer bytes were read than requested, the operation has timed out. If an overlapped write operation timesout, the overlapped event handle is signaled and WaitForSingleObject returns WAIT_OBJECT_0. GetOverlappedResult returns TRUE, but dwBytesTransferred contains the number of bytes transferred before the time-out. The following code sample shows how to handle this in an overlapped write operation.

3.2.1.4 PurgeComm

Before starting your work with the port it is desirable to clear the buffers; sometimes there's also need to clear the buffers when working with ports. For these purposes PurgeComm function can be used. This function can also stop read and write operations.

Function syntax:

```
BOOL PurgeComm(HANDLE hFile, DWORD dwFlags);
```

3.2.1.5 Work with DCB

The port setting is carried out with the help of the DCB (Device-Control Block) structure. By filling this structure with needed values you can change the connection parameters to those needed at the moment.

To initially create the DCB structure with necessary general settings (baud rates, parity, number of bits, number of stop bits and flow control) is carried out by the BuildCommDCB function.

Function syntax:

```
BOOL BuildCommDCB (LPCTSTR lpDef, LPDCB lpDCB);
```

3.2.1.6 Read and Write Port

As work with the ports in Windows is carried out in the same way as work with files, reading from and writing into the port are carried out with the help of ReadFile and WriteFile functions correspondingly. ReadFile function is used to read the information from the port.

Function syntax:

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD  
nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED  
lpOverlapped);
```

```
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD  
nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED  
lpOverlapped);
```

3.2.1.7 Event

Win32 API provides WaitCommEvent function used to wait for events which can occur for the specified communications device. At that the set of events checked by this function is contained in the events mask connected with the given device.

Function syntax:

```
BOOL WaitCommEvent(HANDLE hFile, LPDWORD lpEvtMask, LPOVERLAPPED  
lpOverlapped).
```

3.2.2 Serial Channels in C166/C167

Serial communication with other micro-controllers, processors, terminals or external peripheral components is provided by two serial interfaces with different functionality, an Asynchronous/Synchronous Serial Channel (ASC0) and a High-Speed Synchronous Serial Channel (SSC).

The ASC0 is upward compatible with the serial ports of the Infineon 8-bit micro-controller families and supports full-duplex asynchronous communication at up to 781 KBaud/1.03 MBaud and half-duplex synchronous communication at up to 3.1/4.1 MBaud (@ 25/33 MHz CPU clock).

A dedicated baud rate generator allows setting up all standard baud rates without oscillator tuning. For transmission, reception and error handling 4 separate interrupt vectors are provided. In asynchronous mode, 8- or 9-bit data frames are transmitted or received, preceded by a start bit and terminated by one or two stop bits. For multiprocessor communication, a mechanism to distinguish address from data bytes has been included (8-bit data plus wake up bit mode).

In synchronous mode, the ASC0 transmits or receives bytes (8 bits) synchronously to a shift clock, which is generated by the ASC0. The ASC0 always shifts the LSB first. A loopback option is available for testing purposes. A number of optional hardware error detection capabilities have been included to increase the reliability of data transfers. A parity bit can automatically be generated on transmission or be checked on reception. Framing error detection allows recognizing data frames with missing stop bits. An overrun error will be generated, if the last character received has not been read out of receive buffer register at the time the reception of a new character is complete.

The SSC supports full-duplex synchronous communication at up to 6.25/8.25 Mbaud (@ 25/33 MHz CPU clock). It may be configured so it interfaces with serially linked peripheral components. A dedicated baud rate generator allows setting up all standard baud rates without oscillator tuning. For transmission, reception and error handling 3 separate interrupt vectors are provided. The SSC transmits or receives characters of 2 ... 16 bits length synchronously to a shift clock, which can be generated by the SSC (master mode) or by an external master (slave mode). The SSC can start shifting with the LSB or with the MSB and allows the selection of shifting and latching clock edges as well as the clock polarity. A number of optional hardware error detection capabilities has been included to increase the reliability of data transfers. Transmit and receive error supervise the correct handling of the data buffer. Phase and baud-rate error detect incorrect serial data. [7]

Chapter 4

Proposed System

4. Proposed System

Software in real-time embedded systems differs fundamentally from its desktop or Internet counterparts. Embedded computing is not simply computation on small devices. In most control applications, for example, embedded software engages the physical world. It reacts to physical and user-interaction events, performs computation on limited and competing resources, and produces results that further impact the environment. Of necessity, it acquires some properties of the physical world, most particularly, time. [8]

4.1 Real-Time Programming: Common Practice

Real-Time systems typically need to perform multiple tasks at the same time. Each invocation of task is a finite amount of computation that requires some resources and takes some time to perform. Tasks may compete for resources such as CPU, I/O access, or network bandwidth, thus a resource manager is needed to allocate resources and schedule task activation. This resource management is a major responsibility of real-time operating systems in common embedded systems. When two eligible tasks are competing for the resources, the operating system must choose to grant resources to one of them, and as a consequence, that task finishes sooner.

The process of choosing a task to grant resources to, that is, CPU time is called real-time scheduling. [8]

4.1.1 Preemptive Multitasking

A typical strategy to implement real-time scheduling is called preemptive multitasking. In preemptive multitasking, operating system uses some criteria to decide how long to allocate to any one task before giving another task a turn to use the operating system. The act of taking control of the operating system from one task and giving it to another task is called preempting. To perform resource management using preemptive multitasking, the resource manager has to perform two duties:

- Context Switching
- Task Scheduling

When a task is in running state and the time slice has been expired, e.g. timer event, then the scheduler is invoked which decides which task deserves to be given next time slice using its scheduling algorithm based on priority system. Finally, it performs context switching, replaces first task by second task and lets the later task to perform its duty.

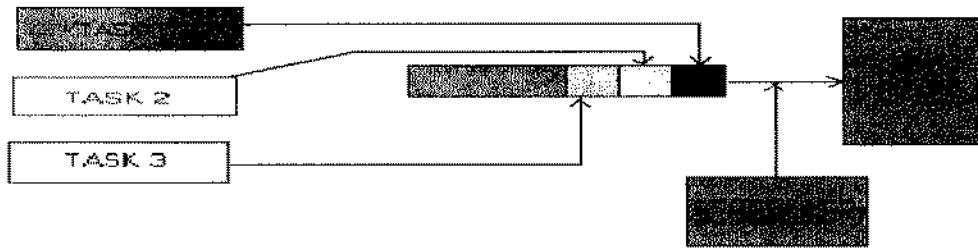


Figure 4.1: Real Time Scheduling

4.1.2 Limitations in Preemptive Multitasking

Preemptive multitasking is implemented on a processor or a micro-controller, which has built in support for context switching and a periodic task trigger on which event scheduler has to be invoked. A common criterion is simply elapsed time, that is, the timer implemented in hardware is programmed to be invoked on expiration of a time slice. The timer generates an interrupt, which initiates an interrupt service routine. In interrupt service routine, scheduling is performed, and it is decided which task is to be granted processor next. The state of currently executing task is saved and the context of the next task is loaded into the CPU. After ISR, the CPU starts executing the newly loaded task. So, to perform task switching, the CPU must have spare context registers, called 'Register Banks'.

High level processors, such as, Intel 8086, Intel 8088, Intel 80386, Intel 80486, Pentium and Pentium Pro support preemption. There are number of micro-controllers that provide built in hardware support for context switching and periodic task trigger, for example Siemens C166, Siemens C167

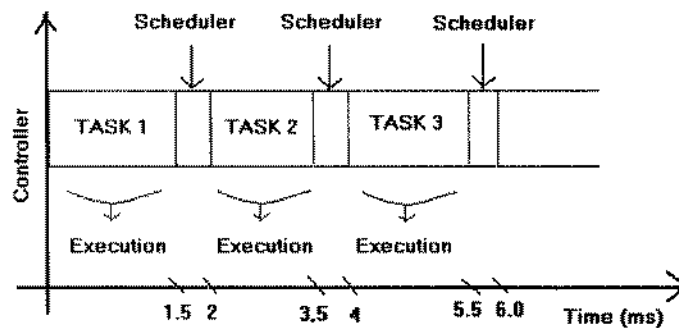


Figure 4.2: Preemptive Multitasking

4.1.3 Collaborative Multitasking Approach

As explained earlier, the multitasking performed by context switching requires very particular hardware support, which is not available in tiny micro-controllers such as Intel 80C51, 80C52

This article gives a programming model to implement multitasking in real-time tasks, for example, running a TCP/IP based application. Collaborative multitasking model gives the idea to address fundamental issues of running preemptive multitasking kernel on tiny micro-controllers.

4.1.3.1 Tasks Collaboration

In collaborative multitasking, tasks (any user process running on that controller) collaborate with each other in a way that each task executes a part of its route, saves its state locally and then releases system resources voluntarily.

In this system, each task is represented by function or routine. In this idea, no task is forced to preempt resources from it. A task returns after executing a part of it, saves its state and gives control to other task waiting for resources. The sequence executes in a continuous fashion.

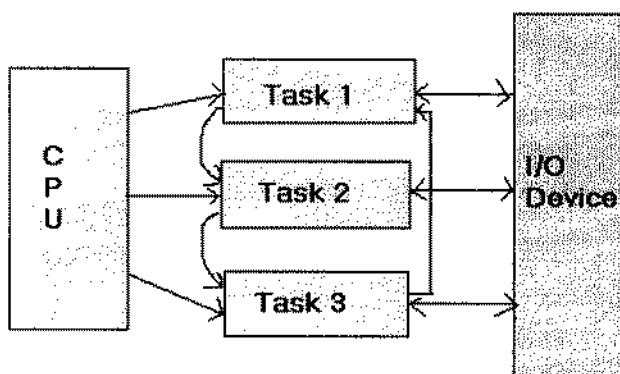


Figure 4.3: Collaborative Multitasking

For example, we have three tasks. First Task is Display task whose responsibility is control LCD display. Second task is Comm task whose responsibility is to receive any data from comport and process it, and the third one is KeyPad task which scans the keys and gets any activity of key pressing.

Now these three tasks will collaborate with each other. When Display function will be called, it will scan all the display memory and will refresh it on the screen in one cycle. After that it will return back and Comm task will be invoked. In a single cycle, Comm will scan its comport, receive any incoming waiting data and process it. After that it will return back and then finally KeyPad task will be invoked. In a cycle, KeyPad will scan all the keys and will refresh keypad memory indicating any key press event. This sequence will execute continuously.

main ()


```

{
    InitSys ( );
    While (1)
    {
    Display ( );
    Comm ( );
    KeyPad ( );
    }
    QuitSys ( );
}

```

The sharing of resources among the tasks is not based on time slices, but sharing is done on work basis or number of instructions. Every task divides its whole work into sub-tasks. Whenever a task is given control of CPU, it executes one of its sub-tasks and returns the control. In next allocation of CPU, it executes next sub-task.

For example, we have an embedded system which has to execute three tasks: Task1, Task2 and Task3 simultaneously. Task1 is further divided into three subtasks: subtask1, subtask2 and subtask3. Task1 completes, as each subtask executes ones.

```

While (1)
{
Task1 ();
Task2 ();
Task3 ();
}

```

```

Task1 ()
{
static int nStat=0;
switch (nStat)
{
case 0: Subtask1(); nStat=1; break;
case 1: Subtask2(); nStat=2; break;
case 2: Subtask3(); nStat=0; break;
}
}
}

```

The scheduler is designed such that every task executes its one sub-task in its turn, and returns back so that next task can be executed. In above example, task1 completes in three iterations. In this way, all the tasks are executed simultaneously because of their collaboration with each other.

4.1.3.2 Queues for Inter-process Communication

Inter-process communication is always an important issue when designing scheduler for real-time embedded systems. In collaborative multitasking programming model, every task has its

incoming and outgoing FIFO, and also there is a shared buffer pool. Whenever a task wants to send data to another task, it acquires a free buffer from buffer pool, copies the data in buffer, and puts the index of buffer in incoming FIFO of their task. Every task polls its incoming FIFO, and processes the data, if present.

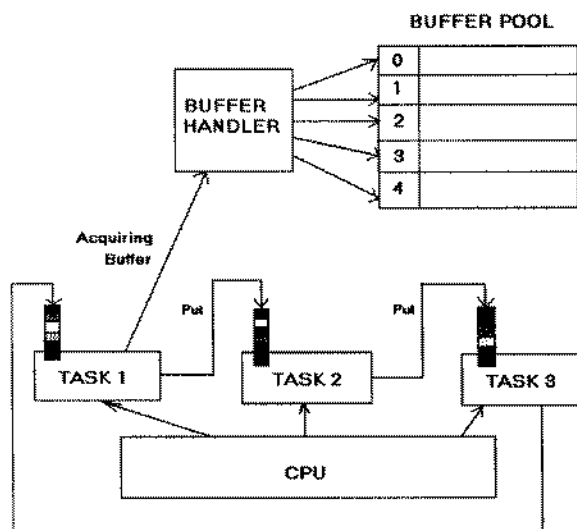


Figure 4.4: Inter-process Communication

4.3.1.3 Task Priorities

Task priority is very important concept in multitasking system. The priority represents the relative importance of a task at run time. When three tasks are running at a time, then the process of determining which task deserves CPU more is called priority.

For example, we have task three that is more important than Task1 and Task2. Then its priority can be implemented, as its iterations can be increased relative to other tasks.

```

While ()
{
Task1 (4);
Task2 (1);
Task3 (2);

I/O Task (1);
}

void Task1 (int nPriority)
{
int nIteration=0;
While (nIteration<nPriority)
{
//execute subtasks

```

```

nIteration++;
}
}

```

The priority of a task can be determined at run-time and it can be set according to the situation.

4.1.4 Example

In this section, we describe the design and development of real-time scheduler for transport protocols, such as TCP/IP. The TCP/IP protocol stack is implemented such that each layer is represented by a task.

```

INIT_STACK;
WHILE (nQUIT)
{
APP_TASK;
TCP_TASK;
UDP_TASK;
IP_TASK;
PPP_TASK;
COM_TASK;
}

```

The main thread initializes all the layers, distributes the time slices by calling respective processes. Each layer works in two directions, that is, it processes data from upper as well as lower layer. A separate buffer bank is reserved for data to be processed, in the form of two-dimensional array. Each buffer has following associated attributes:

- Name of the buffer (Free, Temporary, PPP Down, PPP Up, IP Down, IP Up, UDP Up, UDP Down, TCP Up, TCP Down, Application Down, COM Up)
- Command (No command, dial, ping, valid IP frame)

Message flags associated with each process control sub-processes. Each layer has two Data Queues associated with it, one for each direction: Layer up Job Queue, that contains pointer to the buffer received from upper layer and is ready to be processed according to the command associated with the buffer and status of the message flag associated with that direction, and Layer-down Job Queue, for data received from down layer. These Job Queues are responsible for inter-process communication.

Whenever an application wants to perform a TCP/IP related task, it gets a buffer from buffer bank, adds data to buffer associates a command with buffer which indicates what has to be done with the data in buffer, and passes buffer reference to the Layer Down Job Queue of the lower layer.

On turn of task associated with the next layer, the incoming job queue is checked, and the buffer is processed according to the command, flags are set and the buffer reference is added to Layer Down Job Queue of the next layer. Next layer behaves in same way, until data reaches COM layer, and is written to COM port.

It is not necessary for a task to complete its job in single iteration. So, each task has to maintain its state, so that it can continue from the same point in next iteration. For that, each layer performs part of its task, saves its state in buffer, and keeps the track of previous work with the help of flags associated with each task.

The system behaves in same way for opposite direction and use Layer up Data Queues and UP message flags.

5 System Analysis and Design

In this chapter, the system under discussion is analyzed.

5.1 Objectory

The methodology that is used for Object Oriented Analysis and Design is Objectory or Object Oriented Software Engineering approach (Jacobson Method).

The Object Oriented Software Engineering has following four models:

- The Requirements Model
- The Analysis Model
- The Design Model
- The Implementations Model

5.2 UML

The UML is a new standard for the modeling notations using diagrams of different types. The UML allows people to develop several different types of visual diagrams that represent various aspects of the system. The following are seven types of diagrams that are in UML:

- Use Case Diagram
- Sequence Diagram
- Collaboration Diagram
- Class Diagram
- State Transition Diagram
- Component Diagram
- Deployment Diagram

5.3 Actors, Use Cases and Sequence Diagrams

5.3.1 Use case Model

The Use case Model uses actors and use cases. These concepts are simply an aid to defining what exists outside the system (Actors) and what should be performed by the system (Use case).

In Use case model the actors are identified, Use cases are identified and a use case model is constructed.

5.3.2 Identifying Actors

In our project, there are three actors as shown in Figure 5.1:

- Terminal Application, that uses the system to communicate on Internet.
- Internet Service Provider Server, that provides dial up connection.
- Internet Application, that exchanges data with terminal application.

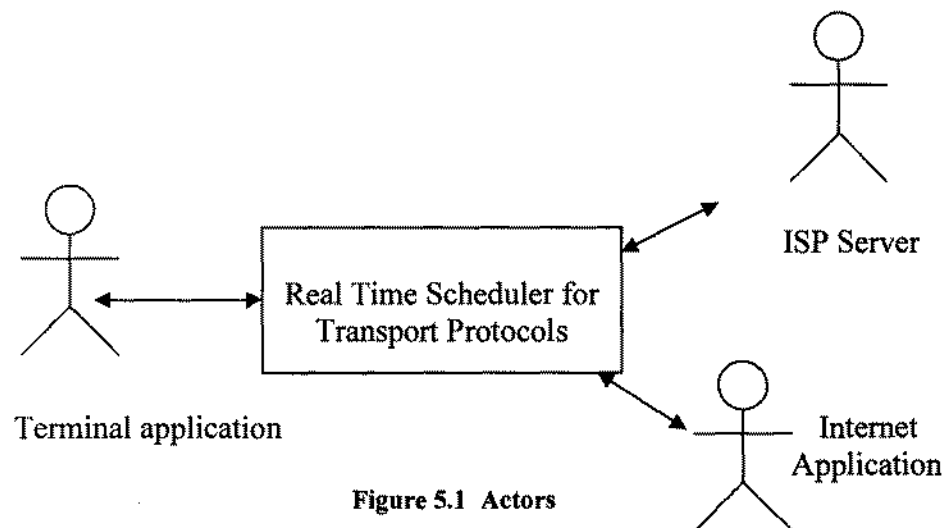


Figure 5.1 Actors

5.3.3 Identifying Use cases

Following are the use cases that could be initiated by the terminal application as in Table 5.1.

Table 5.1 Client Use cases

No	Use case Name		
1	Connect to Internet Service Provider	1	Terminal Application, ISP Server
2	Ping Internet Service Provider	2	Terminal Application, ISP Server
3	Establish TCP Connection	3	Terminal Application, Internet Application
4	Exchange TCP Data	4	Terminal Application, Internet Application
5	End TCP Connection	5	Terminal Application, Internet Application

The Priorities identifies the sequence of the Design and implementation of the use cases.

Each use case will be analyzed, designed, coded and implemented in a single iteration and the use cases having greater priorities will be implemented first.

5.3.4 Use Case Diagram

Figure 5.2 gives Use Case diagram for the system.

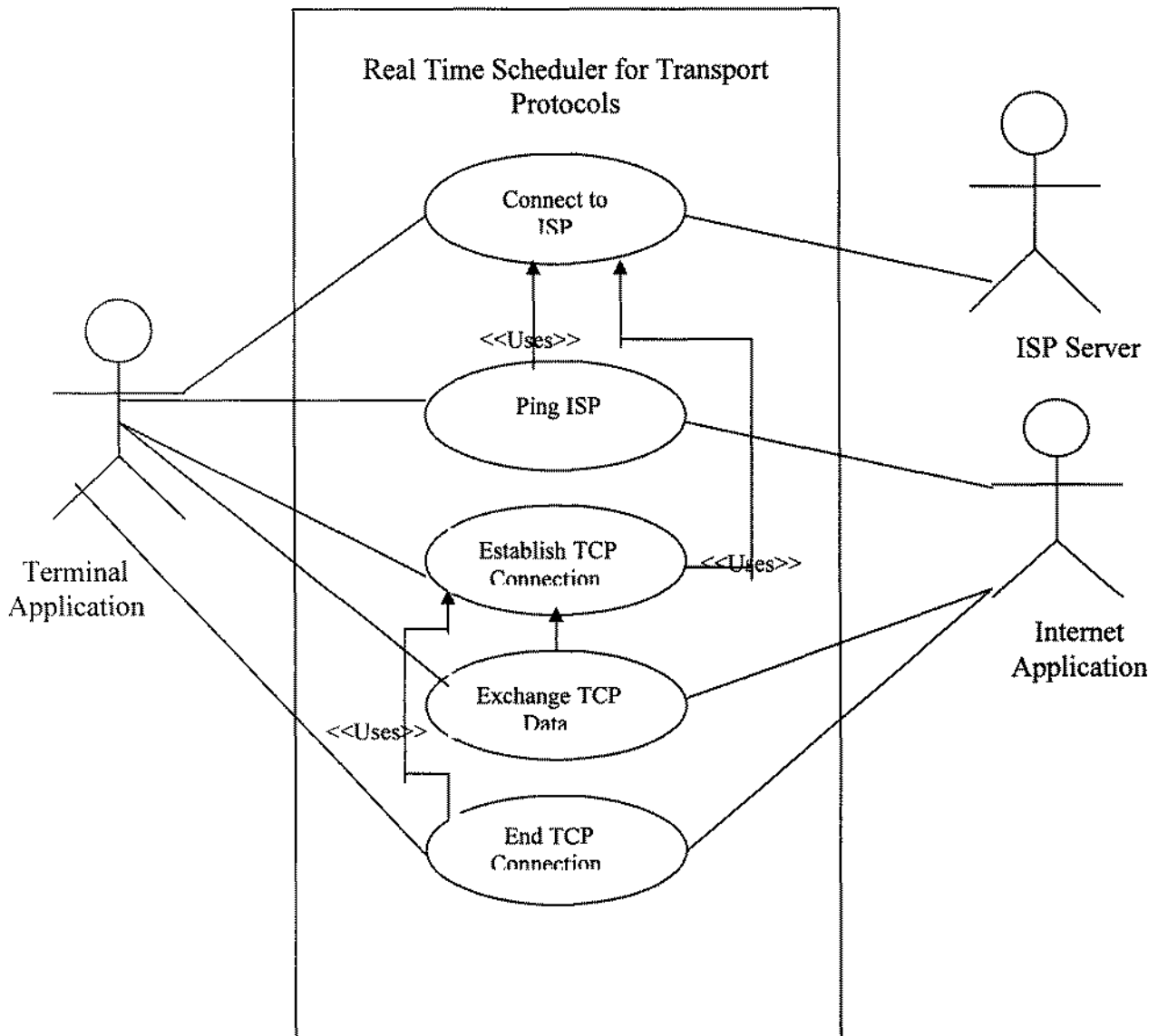


Figure 5.2 UseCase Diagram

5.3.4.1 Connect to ISP

Actor:

Terminal Application,
ISP Server

Overview:

Terminal Application requests to dial Internet Service Provider: providing telephone number, user name and password. The system processes the request and as soon as it gets positive response from Internet Service Provider, the connection establishes.

Pre-Condition:

No pre-condition.

Post-Condition:

Terminal Application connects to Internet.

Success Scenario:

1. Terminal Application requests to dial Internet Service Provider, providing telephone number, user name and password.	2. The request reaches application layer, which forwards the request to PPP layer. 3. PPP layer transfers the command for dialing to COM layer, set its timer and starts waiting for connection. 4. COM layer writes haze command for dialing to modem. 5. COM Layer gives response of dial to PPP layer. 6. The PPP layer confirms that the connection with remote PC has been established by searching 'CD' or 'CONNECT' string in the response. 7. PPP layer makes LCP configuration, i.e. baud rate, parity, etc. and transmits LCP configuration request to COM Layer.
9. ISP server sends LCP response.	8. PPP starts waiting for LCP response. 10. COM layer writes received data to modem. 11. If PPP receives LCP Configuration Acknowledgement, It transmits PAP configuration request for user authentication to COM Layer.
14. ISP Server sends PAP acknowledgement and IPCP configuration request for IP Layer configuration.	12. PPP starts waiting for PAP response. 13. COM layer writes received data to modem. 15. PPP transmits IPCP configuration request for user authentication to COM Layer and connection establishes. 16. PPP starts waiting for PAP response. 17. COM layer writes received data to modem.

Failure Scenario:

- 6a. No response reached and PPP declares timeout.
 - The request is discarded.
 - Terminal Application is asked to initiate the request again
- 6b. PPP layer receives negative response.
 - The request is discarded.
 - Terminal Application is asked to initiate the request again.

5.3.4.2 Ping ISP**Actor:**

Terminal Application,
ISP Server

Overview:

Terminal application requests to ping ISP. The system sends the request through com port and reply is received.

Pre-Condition:

Terminal application must be connected to internet.

Post-Condition:

Ping reply is received.

Success Scenario:

Actor Action	System Response
1. Terminal application requests to ping ISP, providing IP address of ISP.	2. The COM Layer hands over the request to IP layer.
	3. At IP layer. ICMP request is generated, which is handed over to COM layer.
	4. COM layer writes request to modem.
	6. COM layer hands over reply to IP layer.
	7. Terminal application is informed about ICMP Reply by application layer.
5. Server sends ping reply.	

5.3.4.3 Establish TCP Connection**Actor:**

Terminal Application, Internet Application

Overview:

Terminal Application requests for TCP connection with some internet application. After negotiation, the connection establishes.

Pre-Condition:

Terminal Application must be connected to ISP.

Post-Condition:

TCP connection is established.

Success Scenario:

Actor Action	System Response
1. Terminal application requests for TCP connection providing IP address or the name of destination, in case of name server. 6. Internet application sends reply.	2. Application layer creates TCP, binds the local and remote port to connection, and also bind TCP connection to local IP address. 3. TCP Layer makes the TCP header and starts waiting for acknowledgement of the synchronization request. 4. IP Layer makes IP header and TCP Checksum. 5. COM layer writes data to modem. 7. The packet is checked for validity at PPP, IP and TCP layers. If SYN+ACK is received, then connection is established. 8. TCP layer sends ACK to the Internet Application. 9. The Application Layer informs terminal application.

Failure Scenario:

7a. If NACK is received

-- Notify terminal application that connection cannot be established.

7b. If Time Out occurs.

Notify terminal application that connection cannot be established.

5.3.4.4 Exchange TCP Data**Actor:**

Terminal Application, Internet Application

Overview:

After TCP connection is established, the terminal application sends and receives data from Internet application.

Pre-Condition:

TCP Connection must be present.

Post-Condition:

The applications exchange TCP Data.

Success Scenario:

Actor Action	System Response
1. Terminal application requests for sending data to the internet application.	2. Application layer hands over the request to TCP layer. 3. TCP Layer makes the TCP header and adds sequence number, which is acknowledgement of the previous data packet sent. 4. IP Layer makes IP header and TCP Checksum. 5. PPP layer makes PPP frame. 6. COM layer writes data to modem.
7. Internet application sends reply.	8. The packet is checked for validity at PPP, IP and TCP layers. If ACK and Data are received, then send ACK for the highest packet received.

Failure Scenario:

- 8a.If no ACK is received after 15 seconds.
-- Resend the request.

5.3.4.5 End TCP Connection**Actor:**

Terminal Application, Internet Application

Overview:

After sending and receiving of TCP data is completed, the TCP connection is gracefully closed.

Pre-Condition: TCP Connection must be present.

Post-Condition: The TCP connection ends up.

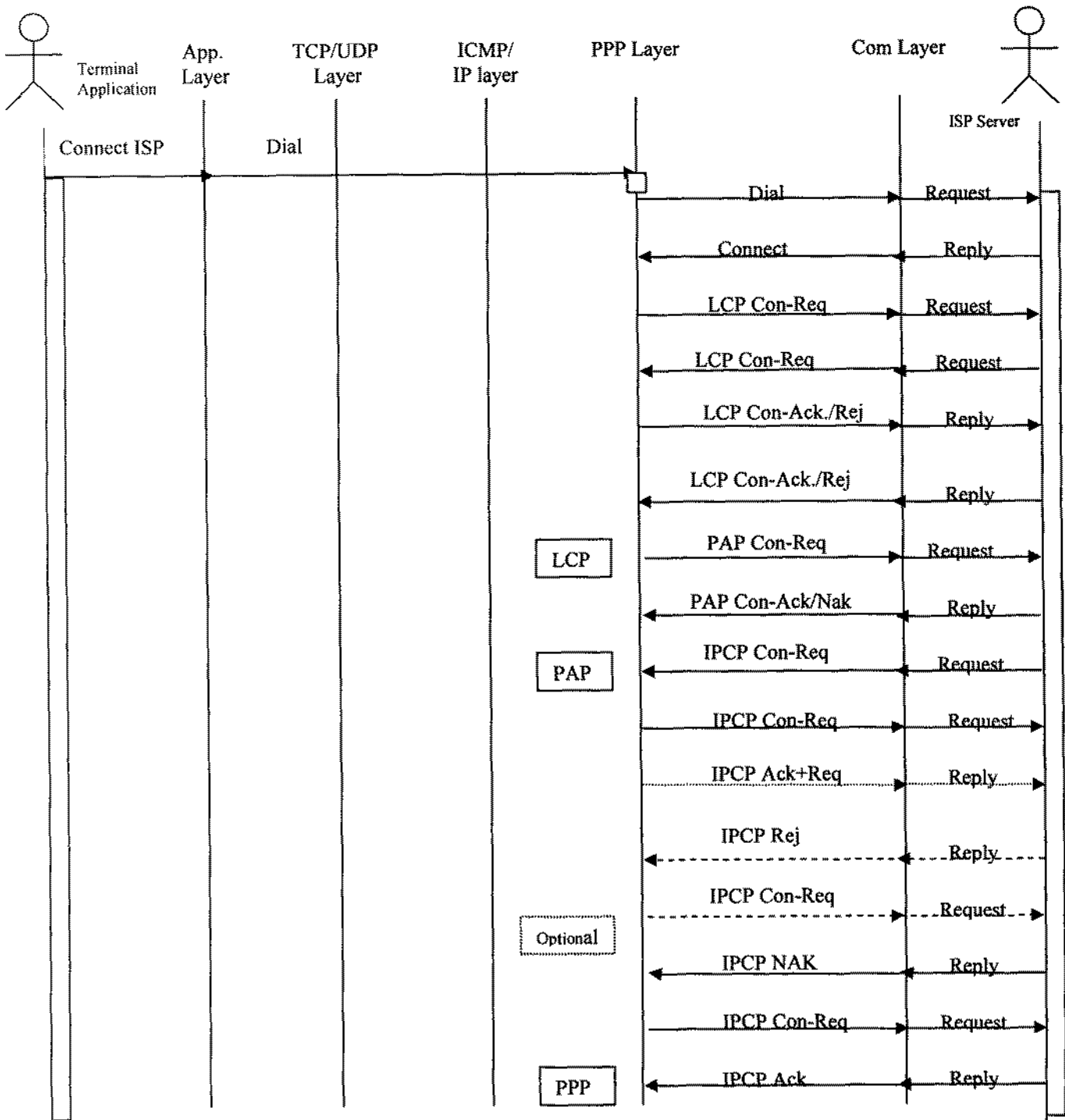
Success Scenario:

Actor Action	System Response
1. Terminal application requests for ending up the TCP connection.	2. Application layer hands over the request to TCP layer. 3. TCP Layer creates FIN request. 4. IP Layer makes IP frame. 5. PPP layer makes PPP frame. 6. COM layer writes data to modem.
7. Internet application sends reply.	8. The packet is checked for validity at PPP, IP and TCP layers. If FIN ACK is received, then TCP layer makes ACK. 9. IP Layer makes IP frame. 10. PPP layer makes PPP frame. 11. COM layer writes data to modem. 12. The connection is closed gracefully.

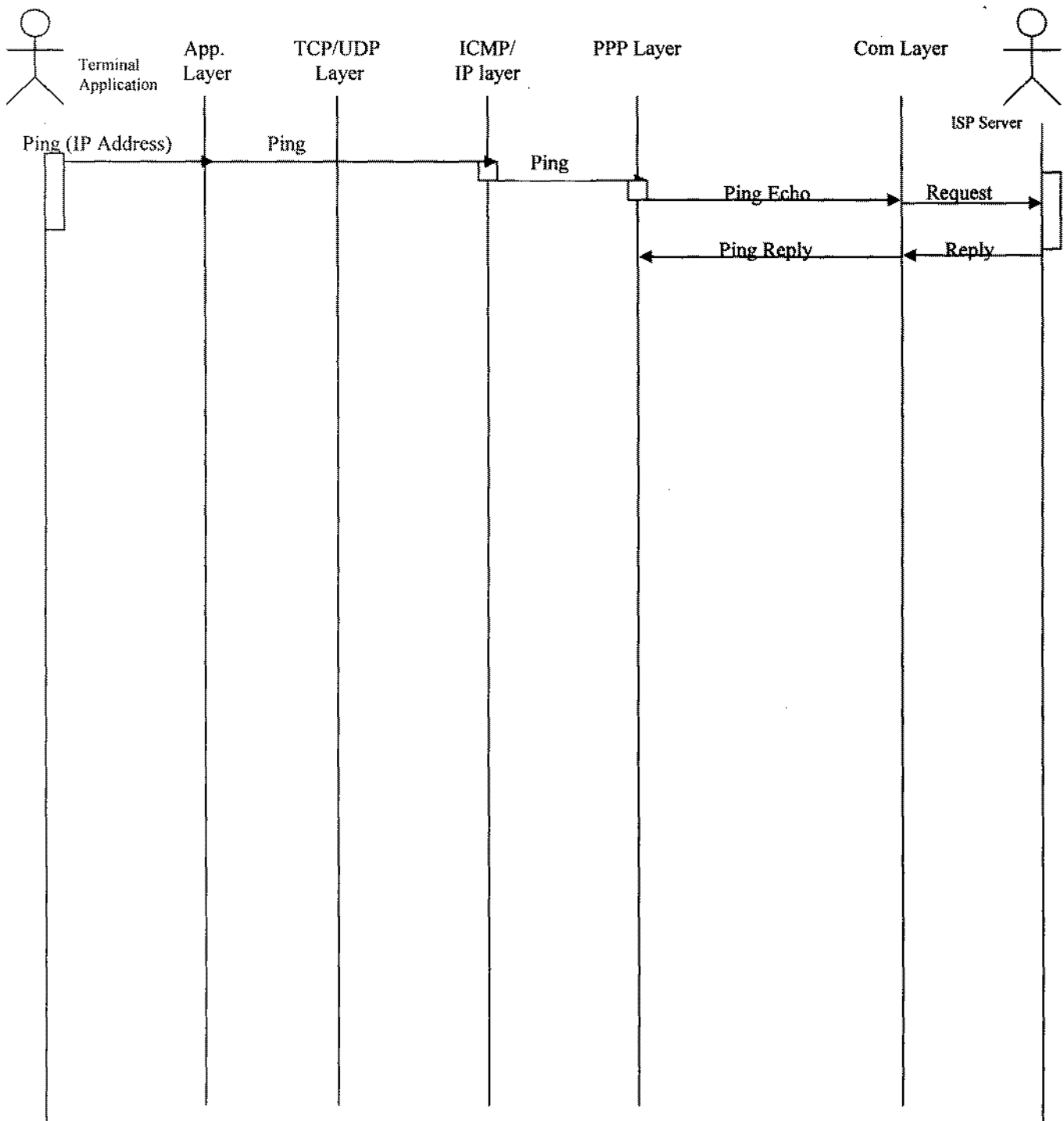
5.3.5 Sequence Diagram

The sequence diagrams for above defined Use Cases are:

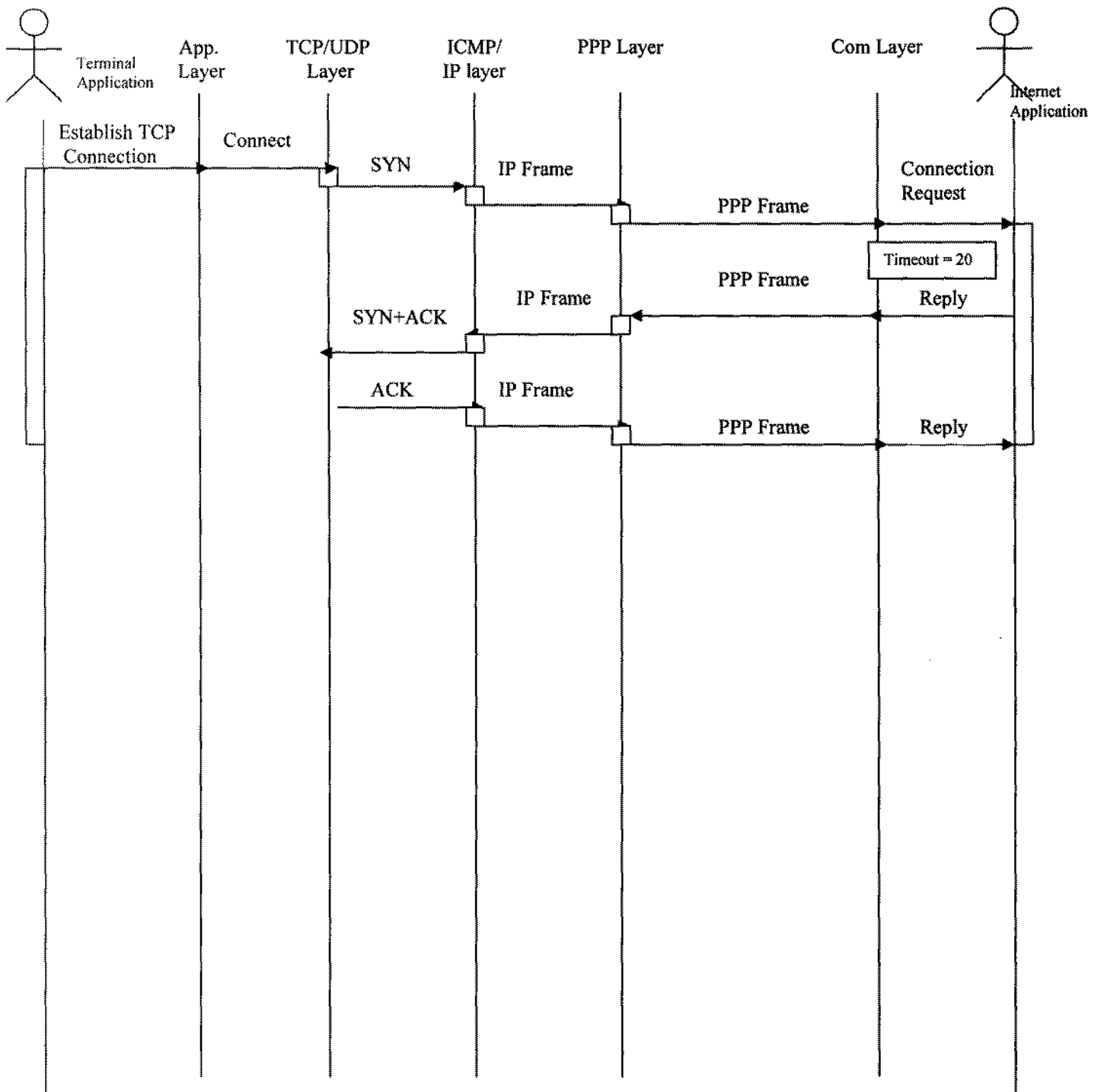
5.3.5.1 Connect to ISP



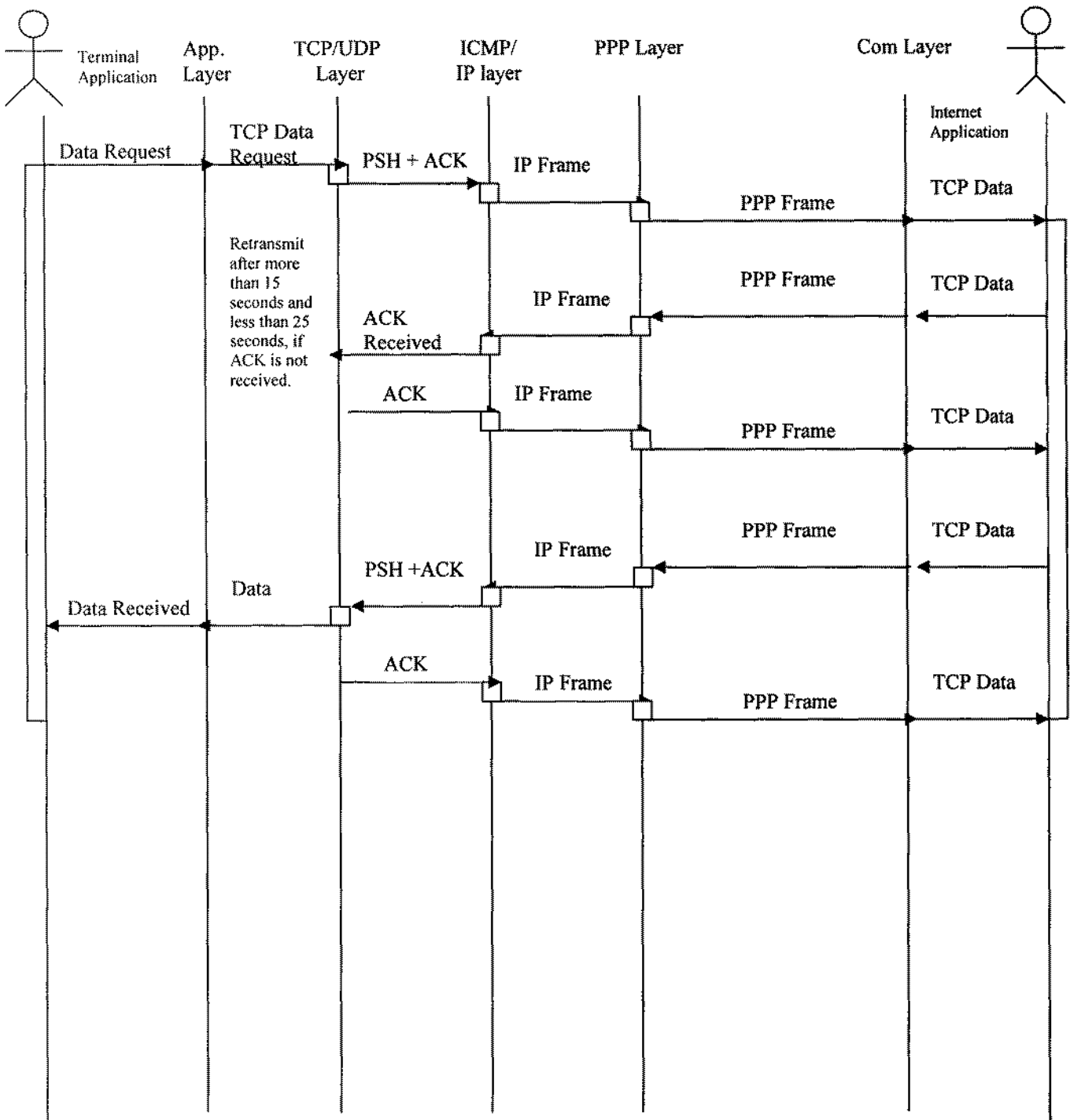
5.3.5.2 Ping ISP



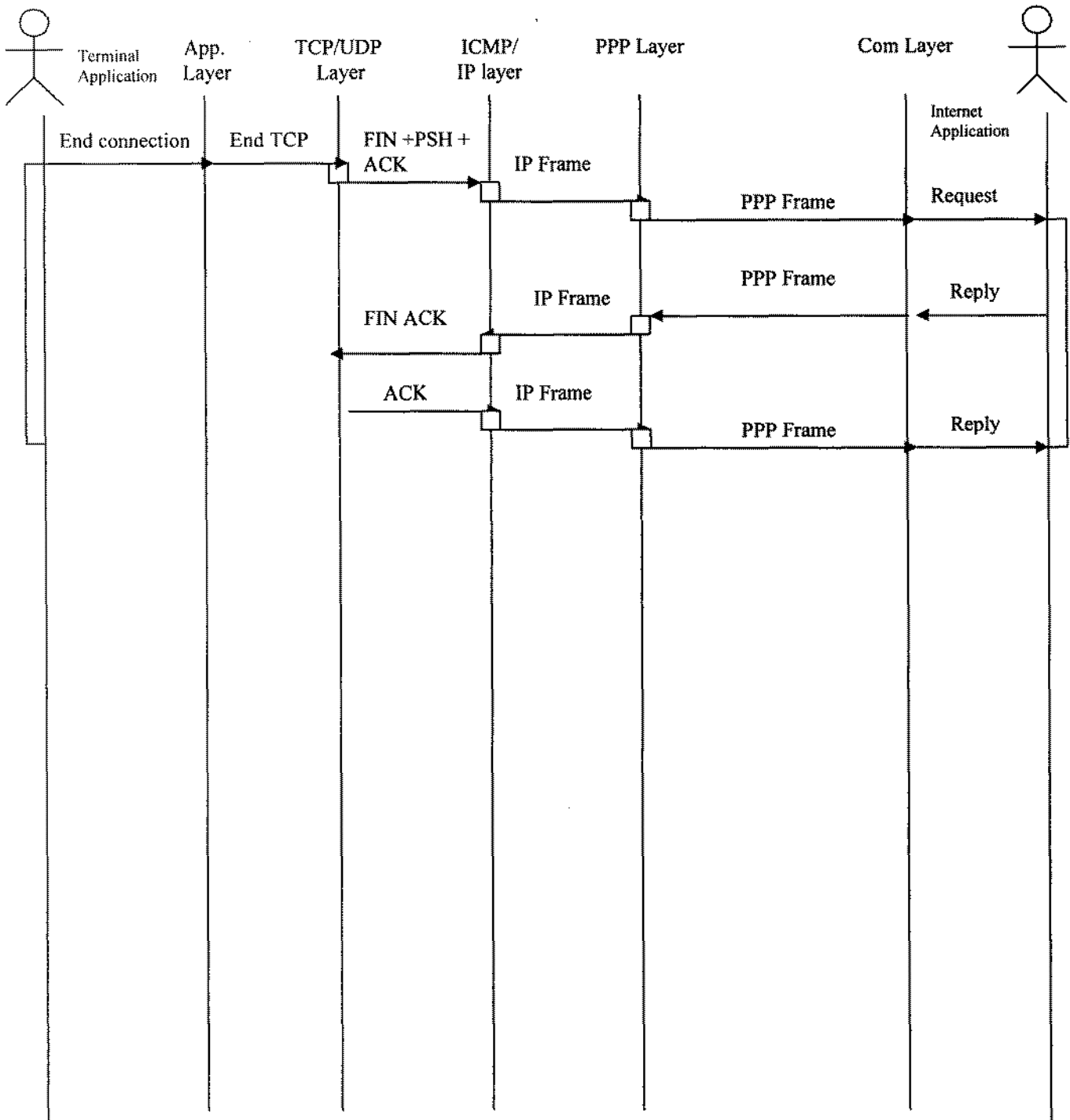
5.3.5.3 Establish TCP Connection



5.3.5.4 Exchange TCP Data



5.3.5.5 End TCP Connection



Chapter 6

System Design

6. System Design

In this section, we describe the design and development of real-time scheduler for transport protocols, such as TCP/IP. The TCP/IP protocol stack is implemented such that each layer is represented by a task.

6.1 Data Structure of TCP/IP Stack

The main thread can be represented as:

```
INIT_STACK;
WHILE (nQUIT)
{
  APP_TASK;
  TCP_TASK;
  UDP_TASK;
  IP_TASK;
  PPP_TASK;
  COM_TASK;
}
```

It initializes all the layers, distributes the time slices by calling respective processes. Each layer works in two directions, that is, it processes data from upper as well as lower layer.

A separate buffer bank is reserved for data to be processed, in the form of two-dimensional array. Each buffer has following associated attributes:

6.1.1 Buffers

A two-dimensional array of unsigned char of 80(Max. No. of Buffers) by 2000 (Size of Buffers). Each Buffer has its associated following attributes:

1. Buffer Name

FREE	not occupied.
TEMPORARY	Occupied but not to any specific to layer
COM Up	Physical layer
PPP Down	PPP layer down ¹
PPP Up	PPP layer up ²
IP DOWN	IP layer down ¹
IP UP	IP layer up ²

¹ Down: Means process of received data by the lower layer to upper layer

² Up: Means process of data to be sent by the upper layer is passed to upper layer

UDP Down	Udp layer down ¹
UDP Up	Udp layer up ²
TCP Down	Tcp layer down ¹
TCP Up	Tcp layer up ²
APP Down	Application layer down ¹

2. Buffer Commands

FREE: No command

TMP: Temporary Command

CONNECTTCP: Not Used

DIAL: Checked by PPP layer for dialing to Internet by Application layer

PING: Checked by IP and PPP layer for (ICMP) Ping Message to another host on Network. Set by Application layer for IP layer. Set by IP layer for PPP layer

IPFRAME: Set by IP, MAC and PPP layer for PPP layer as valid IP frame

PPFFRAME: Checked by PPP layer as valid PPP frame. Set by Physical and PPP layer for PPP layer as valid PPP frame

COMFRAME: Set by PPP layer for Physical layer as valid Physical frame

UDPFRAME: Checked by IP and PPP layer as valid UDP frame Set by Application layer Set by UDP layer for Application layer

TCPFRAME: Checked by IP and PPP layer as valid TCP frame Set by Application layer Set by TCP layer for Application layer

CONNECT TCP: Checked by TCP layer for establishment of connection Set by Application layer

APPFRAME: Checked by TCP layer as a valid Application Frame. Set by Application layer

TCPTRNSD: Set by PPP/MAC layer when a TCP frame has been transed.

TCPFRAMEN: Checked by IP and PPP layer as valid TCP frame Set by Application layer. Set by TCP layer for Application layer. TCP frame is not re-transmitted in case of this command.

FINTCP: Checked by TCP layer to set Control Bits. Set by Application layer

UDPFRAMEN: Checked by IP and PPP layer as valid UDP frame. Set by Application layer. Set by UDP layer for Application layer

TCPFRAMER: Checked by IP and PPP layer as valid TCP frame Retransmission. Set by TCP layer.

3. Offset
4. Current Pointer
5. Timers
6. Counters
7. Connection (used for TCP)
8. Frame Sequence Number
9. Remote Ip higher word
10. Remote Ip lower word

6.1.2 Task Queues

Message flags associated with each process control sub-processes. Each layer has two TASK QUEUES (fifos) associated with it, one for each direction: Layer up TASK QUEUE, that contains pointer to the buffer received from upper layer and is ready to be processed according to the command associated with the buffer and status of the message flag associated with that direction, and Layer down TASK QUEUE, for data received from down layer. These TASK QUEUES are responsible for inter-process communication. Each TASK QUEUE has its associated following attributes:

1. **Name:**

FREE: Not occupied

TMP: Occupied but not to any specific to layer

TCPTra: Trans frames Fifo associated with TCP Connection

TCPreC: Receive frames Fifo associated with TCP Connection

TCPApp: Application frames Fifo associated with TCP Connection

UDPApp: Application frames Fifo associated with UDP Connection

TCPUp : Tcp layer up²

TCPDown: Tcp layer down¹

UDPUp: Udp layer up²

UDPDown: Udp layer down¹

IPUp: IP layer up²

IPDown: IP layer down¹

MACUP: Mac layer up²

PPPUp: PPP layer up²

COMUp: Physical layer up²

PPPDown: PPP layer down¹

IPFragr: Used for IP Fragmentation Reassembly (Temp)

IPFragt: Used for IP Fragmentation Reassembly

APPSYSCALL: Used for the TCP/IP System Calls

MAILAPP: Used for the Mail Application System Calls

2. Head
3. Tail
4. Count
5. Size

6.2 Working of TCP/IP Stack

The stack works as follows:

Scenario 1:

Buffer Command: Dial

APP_TASK:

1. Gets unused buffer from buffer bank.
2. Adds dial up information to buffer.
3. Sets buffer name=PPP Up, command=DIAL
4. Adds buffer pointer to PPP up TASK QUEUE.

PPP_TASK:

1. Extracts buffer from its PPP up TASK QUEUE.
2. Sets UP message flag to start dial sub-process in next time slice and exits.
3. DIALUP:
 - a. Sets buffer name= COM Up
 - b. Adds the buffer to COM up TASK QUEUE.
 - c. Resets UP message flag= 'Wait for PPP connection'

COM_TASK:

1. Extracts buffer from its COM up TASK QUEUE.
2. Sets message flag to 'Write Com' to start write comport sub-process in next time slice and exits
3. WRITE_COM:
 - a. Writes data to modem
 - b. Sets UP message flag ='Extract data from UP TASK QUEUE'

COM_TASK: (Data IN)

1. Gets unused buffer from buffer bank.
2. Reads COM data into the buffer
3. Sets buffer name=PPP Down.
4. Adds buffer pointer to PPP down TASK QUEUE.

PPP_TASK:

1. Extracts buffer from its PPP down TASK QUEUE.
2. If CD or CONNECT found in buffer:
 - a. Sets 'Connection' flag.
 - b. Sets frame type to LCP.
 - c. If LCP ACK is received, sets frame type to PAP.
 - d. Sets DOWN message flag= Process PPP frame in next time slice and exits.
 Else 'Discard buffer and exit.'

3. If PAP ACK is received:
 - a. Sets 'Connection Established' flag.
 - b. Initialize UP and DOWN message flags and return.
4. PROCESS_PPP_FRAME:
 - a. Checks frame type, and processes frame accordingly.
 - b. Resets UP message flag= 'Wait for <Frame Type> connection'
 - c. Adds buffer to COM up TASK QUEUE.

Scenario 2:

Buffer Command: Ping

APP_TASK:

1. Gets unused buffer from buffer bank.
2. Creates connection, binds the connection to local IP address.
3. Gets the remote IP to ping the buffer.
4. Sets buffer name=IP Up, command=PING
5. Adds buffer pointer to IP up TASK QUEUE.

IP_TASK:

1. Extracts buffer from its IP up TASK QUEUE.
2. Sets UP message flag to start ping IP sub-process in next time slice and exits.
3. PING_IP:
 - a. Makes ICMP and IP header.
 - b. Adds the buffer in PPP up TASK QUEUE.
 - c. Initialize UP message flag.

PPP_TASK:

1. Extracts buffer from its PPP up TASK QUEUE.
2. Sets UP message flag to start ping PPP sub-process in next time slice and exits.
3. PING_PPP:
 - a. Sets buffer name= COM Up
 - b. Adds the buffer to COM up TASK QUEUE.
 - c. Resets UP message flag= 'Wait for PPP connection'

COM_TASK:

1. Extracts buffer from its COM up TASK QUEUE.
2. Sets message flag to 'Write Com' to start write comport sub-process in next time slice and exits
3. WRITE_COM:
 - a. Writes data to modem
 - b. Sets UP message flag ='Extract data from UP TASK QUEUE'

COM_TASK: (Data IN)

1. Gets unused buffer from buffer bank.

2. Reads COM data into the buffer
3. Sets buffer name=PPP Down.
4. Adds buffer pointer to PPP down TASK QUEUE.

PPP_TASK (Data IN)

1. Extracts buffer from its PPP down TASK QUEUE.
2. If Protocol type= 0X21:
 - a. Sets buffer name= IP Down.
 - b. Adds buffer pointer to IP Down TASK QUEUE.

IP_TASK (Data IN)

1. Extracts buffer from its IP Down TASK QUEUE.
2. If Protocol type= 0X01 for IP Layer, its ICMP frame.
 - a. If frame type= 0, discards and returns.
 - b. If frame type= 8, PING Reply is acknowledged.

Scenario 3:

Buffer Command: TCP Connection (SYNCHRONIZATION)

APP_TASK: (Connection UP)

1. Creates TCP Connection.
2. Gets unused buffer from buffer bank.
3. Sets buffer name= TEMPORARY, buffer offset=100
4. Associate the buffer to newly created connection.
5. Sets window size= 1024.
6. Binds TCP Connection to local IP address.
7. Sets command= CONNECT TCP, buffer name = TCP Up.
8. Adds buffer to TCP up TASK QUEUE.

TCP_TASK: (Connection UP)

1. Extracts buffer from its TCP up TASK QUEUE.
2. Sets UP message flag to start CONN_TCP sub-process in next time slice and exits.
3. CONN_TCP:
 - a. Makes TCP header.
 - b. Adds the buffer pointer in IP up TASK QUEUE.
 - c. Sets buffer name =IP Up, command = TCPFRAMEN
 - d. Sets UP message flag='Wait for SYN acknowledgement' to start WAIT_SYNC sub-process in the next time slice.

IP_TASK: (Connection UP)

1. Extracts buffer from its IP up TASK QUEUE.
2. Sets UP message flag to start TCPFRAME_IP sub-process in next time slice and exits.

3. TCPFRAME_IP:
 - a. Makes IP header.
 - b. Makes TCP checksum.
 - c. Sets buffer name =PPP Up
 - d. Adds buffer pointer to PPP Up TASK QUEUE.

PPP_TASK: (Connection UP)

1. Extracts buffer from its PPP up TASK QUEUE.
2. Sets UP message flag to start TCPFRAME_PPP sub-process in next time slice and exits.
3. TCPFRAME_PPP:
 - a. Sets protocol byte=0X21 for IP frame.
 - b. Sets CRC.
 - c. Makes PPP Frame with lower byte padding.
 - d. Sets buffer name = COM Up, command = TCP FRAME (For re-acknowledgement purpose)
 - e. Resets UP message flag= 'Extract data from UP TASK QUEUE'.
 - f. Adds the buffer to COM up TASK QUEUE.

COM_TASK: (Connection UP)

1. Extracts buffer from its COM up TASK QUEUE.
2. Sets message flag to 'Write Com' to start write comport sub-process in next time slice and exits
3. WRITE_COM:
 - a. Writes data to modem
 - b. Sets UP message flag = 'Extract data from UP TASK QUEUE'

COM_TASK: (Connection Down)

1. Gets unused buffer from buffer bank.
2. Reads COM data into the buffer
3. Sets buffer name=PPP Down.
4. Adds buffer pointer to PPP down TASK QUEUE.

PPP_TASK (Connection Down)

1. Extracts buffer from its PPP down TASK QUEUE.
2. If Protocol type= 0X21:
 - a. Sets buffer name= IP Down.
 - b. Adds buffer pointer to IP Down TASK QUEUE.

IP_TASK (Connection Down)

1. Extracts buffer from its IP Down TASK QUEUE.
2. If Protocol type= 0X06 for IP Layer, its TCP frame.
 - a. Sets remote IP, frame sequence number.
 - b. Sets buffer name= TCP Down, command = IP FRAME
 - c. Adds buffer pointer to TCP down TASK QUEUE.

TCP_TASK: (Connection Down)

1. Extracts buffer from its TCP Down TASK QUEUE.
2. Functions acknowledgement / any request from remote TCP.
3. Checks overflow of frame sequence number and local acknowledgement number.
4. Sets TCP Acknowledgement counter.
5. WAIT_SYNC:
 - a. Waits for TCP Acknowledgement counter to be 12.
 - b. If TCP Acknowledgement counter NOT EQUAL to 12 in 20 seconds timestamp, buffer is freed.

Scenario 4:**Buffer Command: TCP Connection (PUSH)****APP_TASK: (Connection UP)**

If TCP connection exists:

1. Gets unused buffer from buffer bank.
2. Sets offset=100, command=APP FRAME, connection= existing connection.
3. Sets buffer name = TCP UP.
4. Adds buffer pointer to TCP up TASK QUEUE.

TCP_TASK: (Connection UP)

1. Extracts buffer from its TCP up TASK QUEUE.
2. Sets UP message flag to start APP_FRAME sub-process in next time slice and exits.
3. APP_FRAME:
 - a. Makes TCP header.
 - b. Adds the buffer pointer in IP up TASK QUEUE.
 - c. Sets buffer name =IP Up, command = TCP FRAME
 - d. Sets UP message flag='Extract data from UP Task Queue' to start GetUpFrameTCP sub-process in the next time slice.

IP_TASK: (Connection UP)

1. Extracts buffer from its IP up TASK QUEUE.
2. Sets UP message flag to start TCPFRAME_IP sub-process in next time slice and exits.
3. TCPFRAME_IP:
 - a. Makes IP header.
 - b. Makes TCP checksum.
 - c. Sets buffer name =PPP Up
 - d. Adds buffer pointer to PPP Up TASK QUEUE.

PPP_TASK: (Connection UP)

1. Extracts buffer from its PPP up TASK QUEUE.
2. Sets UP message flag to start TCPFRAME_PPP sub-process in next time slice and exits.

3. TCPFRAME_PPP:
 - a. Sets protocol byte=0X21 for IP frame.
 - b. Sets CRC.
 - c. Makes PPP Frame with lower byte padding.
4. If command = TCP FRAME:
 - a. Removes PPP and IP headers to set offset and current pointer.
 - b. Adds buffer of changed offset and pointer to connection trans c_traTask Queue.
 - c. Sets buffer name = TCP UP, command = TCPTRNSD.
 - d. Reset timer of buffer.
5. If command = TCPFRAMER:
 - a. Removes PPP and IP headers to set offset and current pointer.
 - b. Sets buffer name = TCP UP, command = TCPTRNSD.
 - c. Reset timer of buffer.

COM_TASK: (Connection UP)

4. Extracts buffer from its COM up TASK QUEUE.
5. Sets message flag to 'Write Com' to start write comport sub-process in next time slice and exits
6. WRITE_COM:
 - a. Writes data to modem
 - b. Sets UP message flag ='Extract data from UP TASK QUEUE'

COM_TASK: (Connection Down)

1. Gets unused buffer from buffer bank.
2. Reads COM data into the buffer
3. Sets buffer name=PPP Down.
4. Adds buffer pointer to PPP down TASK QUEUE.

PPP_TASK (Connection Down)

1. Extracts buffer from its PPP down TASK QUEUE.
2. If Protocol type= 0X21:
 - a. Sets buffer name= IP Down.
 - b. Adds buffer pointer to IP Down TASK QUEUE.

IP_TASK (Connection Down)

1. Extracts buffer from its IP Down TASK QUEUE.
2. If Protocol type= 0X06 for IP Layer, its TCP frame.
 - a. Sets remote IP, frame sequence number.
 - b. Sets buffer name= TCP Down, command = IP FRAME
 - c. Adds buffer pointer to TCP down TASK QUEUE.

TCP_TASK: (Connection Down)

1. Extracts buffer from its TCP Down TASK QUEUE.
2. If two buffers are not unused: returns.
3. If time elapsed after reception of frame exceeds 20 seconds:

- a. Rejects buffer.
 - b. Sets connection timer = 4.5 seconds for auto acknowledgement purpose.
- Else verifies length.
4. If SYN + ACK received:
 - a. Sets remote acknowledgement number.
 - b. Initiates local sequence number.
 - c. Sends acknowledgement.
 - d. Frees received buffer and returns.
 5. If RST received: Frees received buffer and returns.
 6. If type is (NORMAL OR ACK): Check overflow of frame sequence number.
 7. If FIN received:
 - a. Make TCP header.
 - b. Add buffer pointer to IP Don TASK QUEUE.
 8. If out of sequence packets received:
 - a. Place packets in c_RecvTask Queue.
 - b. If skipped packets received in 20 seconds:
 - i. Stack packets.
 - ii. Send ACK of highest packet.
 - Else:
 - i. Clear c_RecvTask Queue.
 - ii. Use AUTO_REACK.
 9. If TCP connection exists, port = 80, timer > 5 seconds:
 - a. Sends acknowledgement.
 - b. Resets connection timer.
 Else: Retransmission required.

Scenario 5:

Buffer Command: UDP Transmission

IP_TASK: (Transmission UP)

1. Extracts buffer from its IP up TASK QUEUE.
2. As command = UDP FRAME, set message flag to start UDP_FRAME_IP in the next time slice and exit.
3. UDP_FRAME_IP:
 - a. Makes IP header.
 - b. Makes UDP checksum.
 - c. Sets buffer name =PPP Up
 - d. Adds buffer pointer to PPP Up TASK QUEUE.

PPP_TASK: (Transmission UP)

1. Extracts buffer from its PPP up TASK QUEUE.
2. As command = UDP FRAME, Sets UP message flag to start UDP_FRAME sub-process in next time slice and exits.
3. UDP_FRAME:
 - a. Sets protocol byte=0X21 for IP frame.

- b. Sets CRC.
- c. Makes PPP Frame with lower byte padding.
- d. Set name= COM UP, command = UDP FRAME.
- e. Sets UP message flag ='Extract data from UP TASK QUEUE'

COM_TASK: (Transmission UP)

1. Extracts buffer from its COM up TASK QUEUE.
2. Sets message flag to 'Write Com' to start write comport sub-process in next time slice and exits
3. WRITE_COM:
 - a. Writes data to modem
 - b. Sets UP message flag ='Extract data from UP TASK QUEUE'

COM_TASK: (Transmission Down)

1. Gets unused buffer from buffer bank.
2. Reads COM data into the buffer
3. Sets buffer name=PPP Down.
4. Adds buffer pointer to PPP down TASK QUEUE.

PPP_TASK (Transmission Down)

1. Extracts buffer from its PPP down TASK QUEUE.
2. If Protocol type= 0X21:
 - c. Sets buffer name= IP Down.
 - d. Adds buffer pointer to IP Down TASK QUEUE.

IP_TASK: (Transmission Down)

1. Extracts buffer from its IP Down TASK QUEUE.
2. If Protocol type is of UDP frame.
Sets buffer name= UDP Down, command = IP FRAME
3. Adds buffer pointer to UDP down TASK QUEUE.

UDP_TASK: (Transmission Down)

1. Extracts buffer from its UDP up TASK QUEUE.
2. Sets valid connection by checking local port number. Else verifies the length.
3. Adds the buffer in application task queue of the connection.

Chapter 7

Implementation

7. Implementation

The TCP IP Scheduler is designed in C++ and the mail application is designed in Visual C++. The scheduler is implemented as follows:

7.1 Main Loop

The main loop of TCP IP Scheduler is as follows:

```
inits_tcpip(); //Initializes FIFOs and message flags.
while(1)      {
    if(g_nQuit==1) return; if(exitth==1) break;
    Sleep(10);
    if(exitth==1) break;
    com_proc();if(exitth==1) break;
    ppp_proc(); if(exitth==1) break;
    ip_proc();  if(exitth==1) break;
    tcp_proc(); if(exitth==1) break;
    udp_proc(); if(exitth==1) break;
    app_proc(); if(exitth==1) break;
    udp_proc(); if(exitth==1) break;
    tcp_proc(); if(exitth==1) break;
    ip_proc();  if(exitth==1) break;
    ppp_proc(); if(exitth==1) break;
    app_proc(); if(exitth==1) break;
    cnt++;
}
```

For each layer of TCP/IP stack, a separate procedure is defined. All these procedures are called in a sequence.

7.2 Initialize TCPIP

```

void inits_tcpip();
void init_com()
{
    curucom=-1;
    curdcom=-1;

    msgucom=M_GETUFRM_COM;
    msgdcom=M_GETDFRM_COM;
}

```

7.3 COM Procedure

```

void com_proc()
{
    switch(msgucom)
    {
        case M_GETUFRM_COM : getufrm_com(); break;
        case M_WRITE_COM   : write_com(); break;
    }

    switch(msgdcom)
    {
        case M_GETDFRM_COM : getdfrm_com(); break;
    }
}

```

7.4 PPP Procedure

```

void ppp_proc()
{
    // process upper frames
    switch(msgufrm)
    {
        case M_GETUFRM_PP : getufrm_pp(); break; // get upper frame
        case M_DIAL_PP    : dial();          break; // dialing request - dial number
        case M_WTCON_PP   : wtcon();         break; // wait for modem connection after dial
        case M_WTACKPAP_PP : wt_ackpap();    break; // wait for PAP acknowledgement
        case M_WTACKIPCP_PP : wt_ackipcp();  break; // wait for IPCP acknowledgement
        case M_TCPFRM_PP  : tcpfrm();        break; // Process TCP Frame
        case M_UDPFrm_PP  : udpfrm();        break; // Process IP Frame
    }
}

```



```

case M_PING_PP          : ping_pp();  break; // transparently trans request
}

// process lower frames
switch(msgdfrm)
{
case M_GETDFRM_PP      : getdfm_pp();  break; // get lower frame
case M_CNDFRM_PP       : cnfdfrm_pp();  break; // confirm frame
case M_PROCDFRM_PP     : procdfrm_pp();  break; // process frame and ACK
}
}

```

7.5 IP Procedure

```

void ip_proc()
{
    // process upper frames
    switch(msgu_ip)
    {
        case M_GETUFRM_IP: getufrm_ip(); break; // get frame
        case M_TCPFRM_IP : tcpfrm_ip(); break; // its frame from tcp
        case M_UDPFrm_IP : udpfrm_ip(); break; // its frame from udp
        case M_PING_IP   : ping_ip(); break;   // its ping command
    }
    // process lower frames
    switch(msgd_ip)
    {
        case M_GETDFRM_IP : getdfm_ip(); break; // get frame
    }

    // Check fragmentation
    ck_frag();
}

```

7.6 TCP Procedure

```

void tcp_proc()
{
    // process upper frames
    switch(msgu_tcp)
    {
        case M_GETUFRM_TCP: getufrm_tcp(); break; // Process the out going TCP frame
    }
}

```

```

case M_CONNTCP : conn_tcp(); break;           // Connect TCP
case M_APPFRM  : appfrm(); break;            // its application frame
case M_WTSYNC  : wt_sync(); break;          // wait for sync
case M_FINTCP  : fintcp(); break;           // finish TCP connection
}
// process lower frames
switch(msgd_tcp)
{
case M_GETDFRM_TCP: getdfrm_tcp();break;// Process the incoming frame
}
recfifos_tcp();//Rearrange Data
clear_tfrms();
a_reack(); //Re-ACK
}

```

7.7 UDP Procedure

```

void udp_proc()
{
    getufirm_udp();
    getdfrm_udp();
//    Initialize UDP fifos
}

```

7.8 Application Layer Procedure

```

switch (syscall)
{
    case SYSCALL_DIAL: Dial_PPP(); break;
    case SYSCALL_DISCONNDIAL: Disconn_PPP(); break;
}
else
{
    switch (SOCK_Cmnd[syscall])
    {
        case SYSCALL_CONNTCP : Create_TCP_Conn(syscall); break;
        case SYSCALL_DISCONNTCP: Finish_TCP_Conn(syscall);break;
        case SYSCALL_SENDTCP: Send_TCP_Data(syscall);break;
        case SYSCALL_RCVTCP: Rev_TCP_Data(syscall);break;
        case SYSCALL_CONNUDP :break;
        default break;
    }
}
}
}
}

```

Chapter 8
Testing and Conclusion

8. Testing and Conclusion

Testing of a program is used to check whether it produces the same results as are expected from it and how does it handles in the situation where an exception of error occurs. Testing has following types:

8.1 System testing

System Testing is to test the system as a whole to validate that it meets its specification and the objectives of its users. System Testing focuses on testing the system as an entity. Generally, it is the responsibility of a group, which is separate from the system development team.

It is generally good practice for system testing to be an independent activity as the testers are not themselves stakeholders in the system development. If developers are involved at this stage, they may be reluctant to design tests, which reveal problems in the developed system, as this is an implicit criticism of the quality of their work.

8.2 Development testing

- Hardware and software components should be tested as they are developed and as sub-systems are created. These testing activities include:
 - Unit testing.
 - Module testing
 - Sub-system testing
- However, these tests cannot cover:
 - Interactions between components or sub-systems where the interaction causes the system to behave in an unexpected way
 - The **emergent** properties of the system

As part of the development process, each component that has been developed should be

tested either by its developer or by a separate testing group. The objective of this testing process is to find defects in that component. These defects should then be removed before the component is delivered for integration.

However, these tests can only be based on the component specification (if it exists) along with knowledge about the structure of the component. There may be component errors which are not discovered because these relate to the interaction of the component with other components in the system. The emergent properties of a system are those properties, which apply to the system as a whole rather than to particular components in the system. While some assessment can be made, e.g. of individual component reliability, unit and module testing be used to assess the overall reliability or performance of the whole system.

8.3 Integration testing

The major activity in the integration process is integration testing where the developer of the system carries out a series of tests as the system is put together from its components.

Integration testing should be concerned with tests, which cannot be executed on individual system components or sub-systems. Interface testing is concerned with designing tests which will validate the interactions between components and property testing is concerned with testing the emergent system properties such as reliability, performance etc. As these do not emerge until the system exists as a single entity, it is clearly impossible to test them earlier in the process.

8.3.1 Integration test planning

A separate group should always be responsible for test planning for two reasons:

1. It means that test planning can be carried out at the same time as system development
2. It removes a potential conflict of interest from the development team - is their responsibility to develop software or to test (and potentially find faults with) that software. Developers may, consciously or unconsciously; design tests, which they know, avoid

problems in the system. For large, complex systems, integration test planning may involve hardware and software engineers and human factors specialists.

8.3.2 Test planning activities

Wherever possible (and this is really not easy) the integration test planning team should identify individual system increments, which can be tested and should design tests for these increments. These decisions may be made using the delivery schedules for the different sub-systems (it makes sense to stagger delivery - getting everything on the same day is an integration nightmare) but schedule changes may mean that increments aren't available when required.

Testing tools such as tools to compare test outputs, tools to automatically run tests from files of test data, simulators for hardware which is not available may have to be developed before system testing is possible. The development of these tools goes on in parallel with systems development and often represents a significant fraction of the overall system development costs.

8.4 Stress testing

Stress testing is particularly important for large, multi-user systems where the load on the system varies dramatically from time to time. In essence, you estimate the maximum load that the system is likely to have to handle then test it with more than that load. What should happen is graceful failure where the level of service offered to all users is reduced. What often happens is catastrophic failure where the system moves from working reasonably for all users to a complete loss of service.

Building up the load on the system is not just a test of system performance. Because there is so much stress on the system, defects, which can be corrected automatically in other situations, come to light during stress testing. For example, say a screen is not properly updated but the normal use calls for this screen to be replaced quite quickly in normal use.

The error may never be discovered. Stress testing slows the system down and may reveal this kind of defect.

This is not too important but stress testing can also reveal defects, which are caused by built-in timing assumptions in real-time systems.

8.5 Acceptance testing

Acceptance testing may take place after a system has been installed but often it takes place at the developer's premises using customer-supplied data. The customer observes the system tests to check if the system meets the specified requirements.

It's important to understand that the decision on whether or not to accept a system does not necessarily depend on the system meeting every requirement and successfully executing every test supplied by the customer. The customer needs the system (presumably) so they may be willing to accept an imperfect system for installation. The problems identified are noted and the contractor may have to agree to fix these problems in the first new release after the system has been delivered.

There may also be disagreement between the customer and the contractor at this stage about what requirements actually mean. The customer may have one interpretation of the requirements and the contractor a different interpretation. Therefore, when there is a problem with an acceptance test, some negotiation is necessary to decide whether the customer or the developer has the right interpretation. Often, the result will be that some system changes have to be made and the customer has to pay for some or all of the costs of these changes.

8.6 Performance testing

It may be possible to use data for stress testing for performance testing as critical performance problems are most likely to occur when the system is heavily loaded.

The major problem with performance testing is that there are rarely explicit performance requirements, which are specified in a measurable way. Furthermore, there may be serious

conflicts between e.g. security and performance requirements and the only way to fix the performance problems might be to weaken system security.

The perceived performance of a system is important (if it is an interactive system) whereby the performance is as much to do with expectations as it is with actual figures. If users use a system with a specific performance level, they will expect a new system to at least match that level, even if it offers much greater functionality. This has to be taken into account when setting performance criteria.

8.7 Reliability testing

The problem with reliability is that it isn't an absolute but depends on the context of use of the system. Two different patterns of system use can result in different perceived system reliability.

For this reason, it is very important to get the operational profile right i.e. the predicted pattern of inputs which will be presented to the system. This is possible for some classes of system (where reliability testing is very mature) such as telephone switches where the actual usage of an existing system can be logged and used as the basis of an operational profile.

It is much harder to predict an operational profile when a completely new system or process is introduced - no one really knows how users will adapt to the change and what inputs will be generated.

Reliability testing must take into account the seriousness of system errors. For example, an error in an air traffic control system where a display was pink rather than red is much less serious than error in the same system where the height of the aircraft was wrongly computed.

8.8 Security testing

This is an unusual form of testing because it can't really be planned in the same way. While it is possible to pre-conceive some simple security tests, effective security testing can only really be interactive and, arguably, can only be carried out once the system is in use.

Interactive testing is necessary because security problems may not have a single cause. A user may detect a potential weakness in the system and then exploit this in some other way to gain access to protected parts of the system. It is almost impossible to anticipate this in advance

The argument that security testing cannot be effective until the system is in use comes from the fact that many security problems are due to the way in which a system is used such as insecure passwords, use of over-general permission vectors, etc. These can't really be tested in a pre-production version of the system.

8.9 Testing for Real-Time Scheduler for Transport Protocols

- **Dial ISP- Successful Connection**

Input
User name: sbc10060077 Password: **** Phone Number: 13111333
Output
ISP Connected Successfully.

- **Dial ISP- No LCP Reply**

Input
User name: sbc10060077 Password: **** Phone Number: 13111333
Output
LCP Timeout

- **Dial ISP- No PAP Reply**

Input
User name: sbc10060077 Password: **** Phone Number: 13111333
Output
PAP Timeout

- **Send Email- Successful Connection**

Input
To: samiasherwani@hotmail.com SMTP Server: 210.56.8.10
Output
Email Sent Successfully.

- **Send Email- Server not running**

Input
To: samiasherwani@hotmail.com SMTP Server: 210.56.8.11
Output
Unable to connect to server.

- **Send Email- Invalid Email Address**

Input
To: samia SMTP Server: 210.56.8.10
Output
No user

- **Receive Email- Successful Connection**

Input
User Name: samia07@isb.comsats.net.pk Password: **** SMTP Server: 210.56.8.10
Output
Emails Received

- **Receive Email- Server not running**

Input
User Name: samia07@isb.comsats.net.pk Password: **** SMTP Server: 210.56.8.11
Output
Unable to connect to server.

- **Receive Email- Incorrect Username or password**

Input
User Name: samia123@isb.comsats.net.pk Password: **** SMTP Server: 210.56.8.10
Output
Unable to connect to server.

- **Debugging using Cross Compiler**

Input
The TCPIP Scheduler was compiled using the Keil's Cross Compiler.
Output
Compilation Successful, Debugging Successful

8.10 Conclusion

Collaborative Multitasking is proposed for the processors or micro-controllers, which do not have built-in multitasking support. The TCP/IP stack developed by the use of this technique is tested with SMTP/POP application and also compiled on Keil's cross compiler for embedded system support. The results are positive. As the whole stack works in one thread, so it does not require multitasking support in target processor. Hence the research is successful.

Bibliography

Bibliography

- [1] <http://en.wikipedia.org/wiki/TCP/IP>
- [2] RFC 793, Transmission Control Protocol, by Information Sciences Institute University of Southern California, 4676 Admiralty Way Marina del Rey, California 90291
- [3] RFC 768, User Datagram Protocol, by J. Postel ISI, 28 August 1980
- [4] RFC 1661, Point to point protocol, W. Simpson, Editor Daydreamer, July 1994
- [5] Scheduling theory: www.online.ee/i80386
- [6] <http://www.hw-server.com/docs>
- [7] C167CR/ C167SR - 16-Bit Single-Chip Microcontroller, Data Sheet, V3.1, Apr. 2000
- [8] Liu, J., Lee, E.A.: Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine* 23 (2003) 65--75
- [9] Andr'e C. N'acul and Tony Givargis. Lightweight Multitasking Support for Embedded Systems using the Phantom Serializing Compiler. *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2* (2005) 742 – 747.
- [10] Tensilica Inc. <http://www.tensilica.com>.
- [11] Microchip Inc. <http://www.microchip.com>.
- [12] Phillips Inc. <http://www.philips.com>.

Appendix A

User Manual

A. User Manual – Real Time Scheduler for Transport Protocols

A1. Data Tracing View

This screen shows the current activities at different layers of TCP/IP stack.

The screenshot displays the 'Data Tracing View' window of the Real Time Scheduler for Transport Protocols. The window title is 'Untitled - RWS'. It features a menu bar (File, Edit, View, Help) and a toolbar with icons for file operations and simulation control. On the left side, there is a 'Layers To Analyse' section with checkboxes for 'Analyse TCP Layer Data', 'Analyse UDP Layer Data', 'Analyse IP Layer Data', 'Analyse PPP Layer Data', and 'Analyse COM Layer Data'. Below this are 'Check All' and 'Uncheck All' buttons, an 'Auto Scroll' section with checkboxes for each layer, and another 'Check All' and 'Uncheck All' pair. A 'Clear Data' section contains buttons to clear data for each layer and a 'Clear All Layers Data' button. At the bottom, there are three tabs: 'Data Tracing View' (selected), 'Simulation View', and 'Log View'. The main area contains five data trace tables:

- TCP Layer Data Trace:**

Serial #	Action	Date	Data
0	Write	20/07/2002 13:...	FF 03 80 21 01 3C 00 10 02 06 00 2
1	Read	20/07/2002 13:...	21 45 00 09 2C 00 00 40 00 3E 06 6
2	Write	20/07/2002 13:...	FF 03 80 21 01 05 00 0A 03 06 C8
3	Read	20/07/2002 13:...	
- UDP Layer Data Trace:**

Serial #	Action	Date	Data
0	Read	20/07/2002 13:...	21 45 00 02 01 00 00 40 00 26
- IP Layer Data Trace:**

Serial #	Action	Date	Data
0	Write	20/07/2002 13:...	FF 03 80 21 01 3C 00 10 02 06 00 20 FF 00 03 06 D2 38 08 BF 79 EB EB 7E 00 00 00 DE 06 7E 00 00 86 E7 7D 20
1	Read	20/07/2002 13:...	21 45 00 00 2C 00 00 40 00 3E 06 6A 47 D2 38 08 BA CB 7C 2C C8 00 19 26
2	Write	20/07/2002 13:...	FF 03 80 21 01 05 00 0A 03 06 CB 7C 2C C6 EB E0 06 D2 38 08 BF 1C 1D 7E 38 DE 92 5F 23 7E 00 00 00 86 E7 7E
3	Read	20/07/2002 13:...	21 45 00 02 01 00 00 40 00 26 11 7D C9 DD D3 FF 0A CB 7C 2C C6 80 01 04 02 01 ED 07 23 04 00 28 00 10 00 00
- PPP Layer Data Trace:**

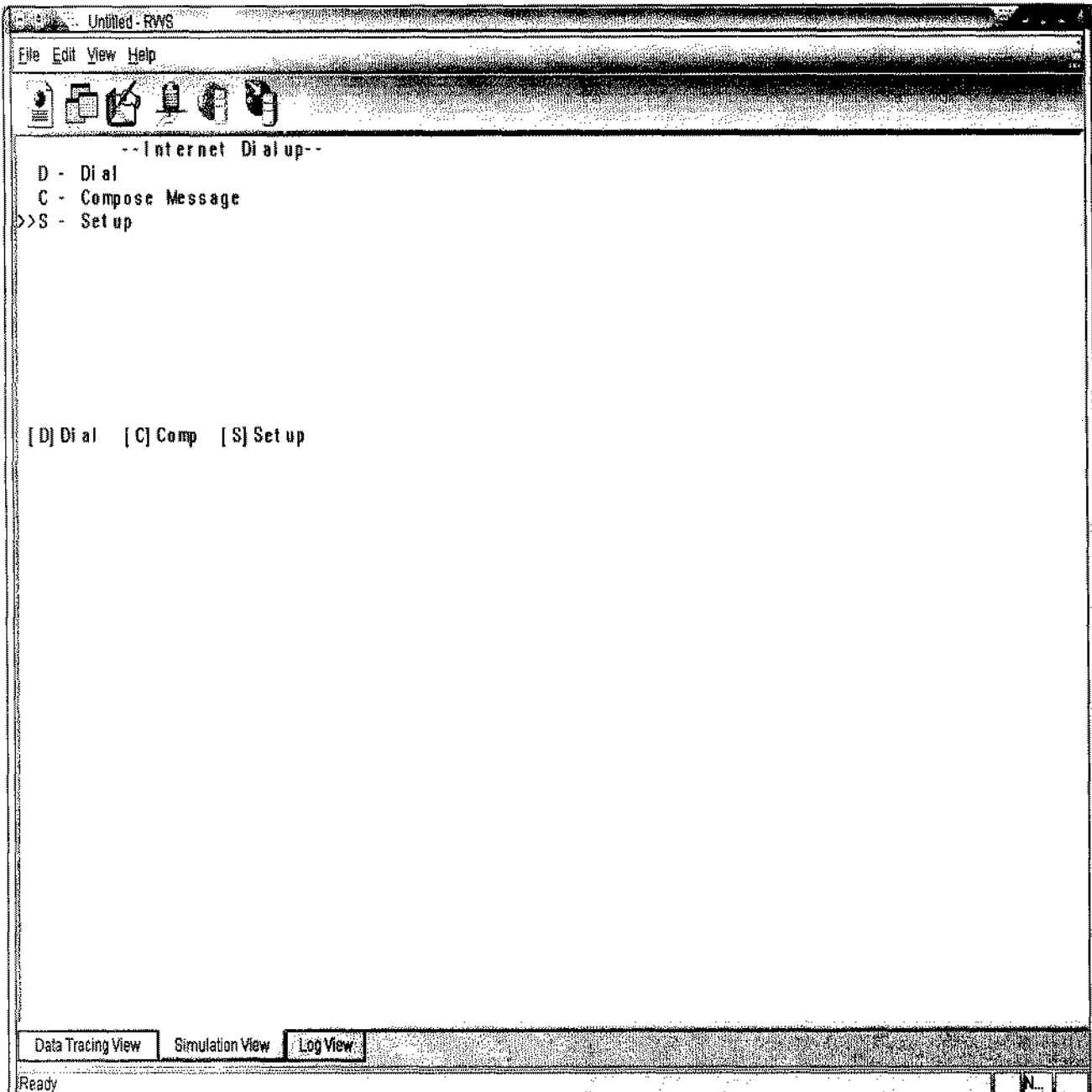
Serial #	Action	Date	Data
0	Write	20/07/2002 13:...	61 74 26 66 64 74 31 33 31 31 31 33 33 34 00
1	Write	20/07/2002 13:...	7E FF 7D 23 C0 21 7D 21 7D 22 7D 20 7D 34 7D 22 7D 26 7D 20 7D 20 7D 20 7D 20 7D 25 7D 26 2E FD 7D 20
2	Write	20/07/2002 13:...	
3	Write	20/07/2002 13:...	7E FF 03 C0 23 01 03 00 19 08 73 62 63 31 30 30 33 39 31 32 36 08 78 33 79 68 37 66 64 36 D8 1F 7E
4	Write	20/07/2002 13:...	7E FF 03 80 21 01 04 00 1C 03 06 00 00 00 00 01 06 00 00 00 00 83 06 00 00 00 00 00 04 06 00 00 00 86 E7 7E
- COM Layer Data Trace:**

Serial #	Action	Date	Data
0	Write	20/07/2002 13:...	61 74 26 66 64 74 31 33 31 31 31 33 33 34 00
1	Write	20/07/2002 13:...	00 DA 43 4F 4E 4E 45 43 54 20 34 35 33 33 33 2F 41 52 51 00 0A 00 0A 00 0A 55 73 65 72 20 41 63 63 65 73 7
2	Read	20/07/2002 13:...	7E FF 7D 23 C0 21 7D 21 7D 25 7D 20 28 7D 22 7D 26 7D 20 7D 2A 7D 20 7D 20 7D 23 7D 24 C0 23 7D 25 7D
3	Read	20/07/2002 13:...	7E FF 7D 23 C0 21 7D 22 7D 22 7D 20 7D 34 7D 22 7D 26 7D 20 7D 20 7D 20 7D 20 7D 25 7D 26 2E FD 7D 20
4	Read	20/07/2002 13:...	7E FF 03 C0 23 02 03 00 05 00 FD 02 7E
5	Read	20/07/2002 13:...	7E FF 03 80 21 01 02 00 10 02 06 00 20 0F 00 03 06 D2 38 08 BF D8 ED 7E
6	Read	20/07/2002 13:...	7E FF 03 80 FD 01 02 00 0A 12 06 00 00 00 01 A2 65 7E

A2. Simulation View

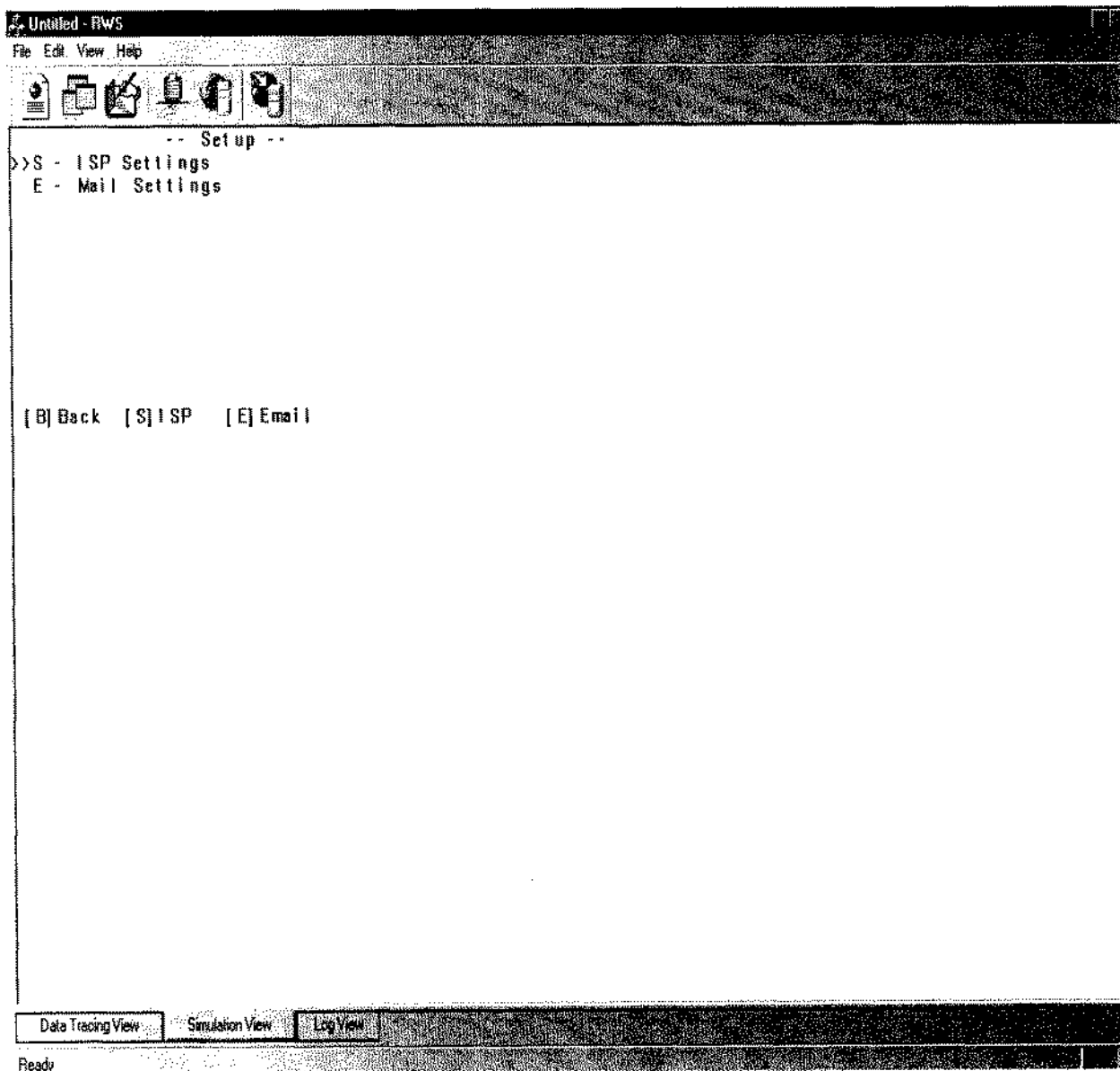
- **Internet Dialup Screen**

The main screen of simulation view provides the facility to connect to an ISP by the use of 'Dial option' (CTRL +D), and to change ISP and email server information by the use of 'Setup Option' (CTRL+S).



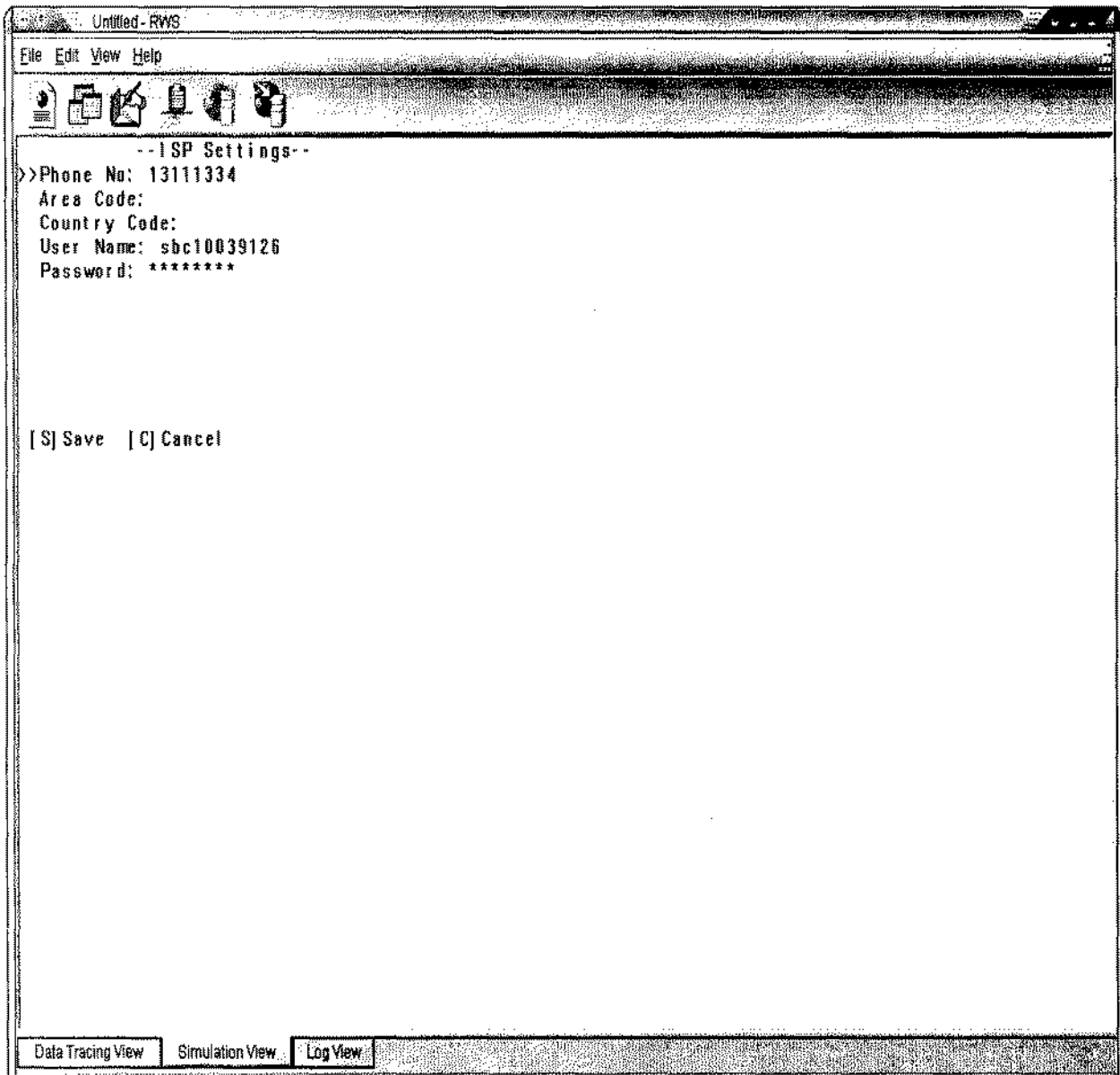
- **Setup Option**

Using the setup option, the ISP and e-mail server settings can be altered.



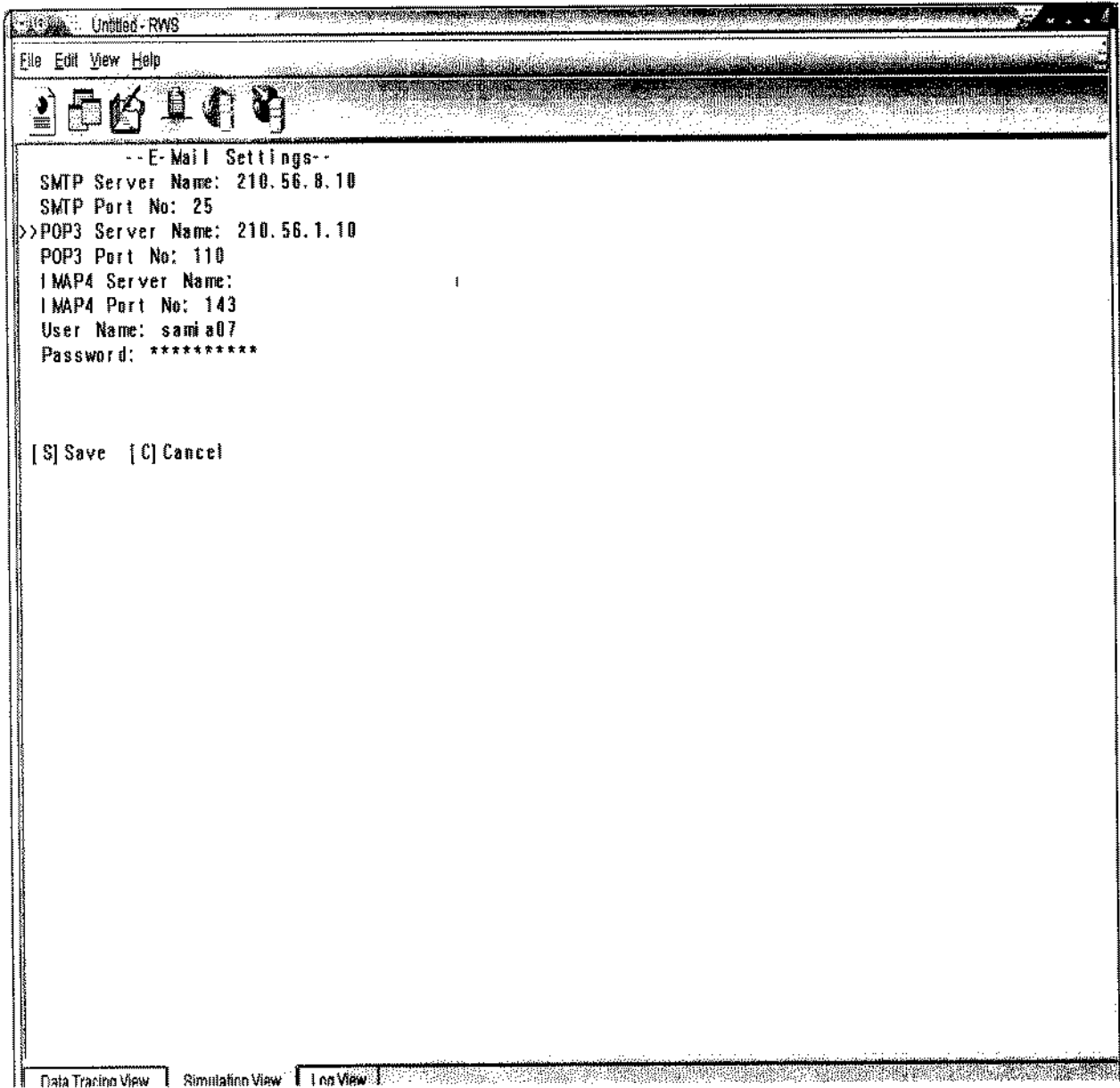
- **ISP Settings**

Here information such as ISP Phone Number, User name and Password is provided.



- Mail Settings

Here information such as SMTP/POP Server IP address, POP User name and Password is provided.



- **Dial Option**

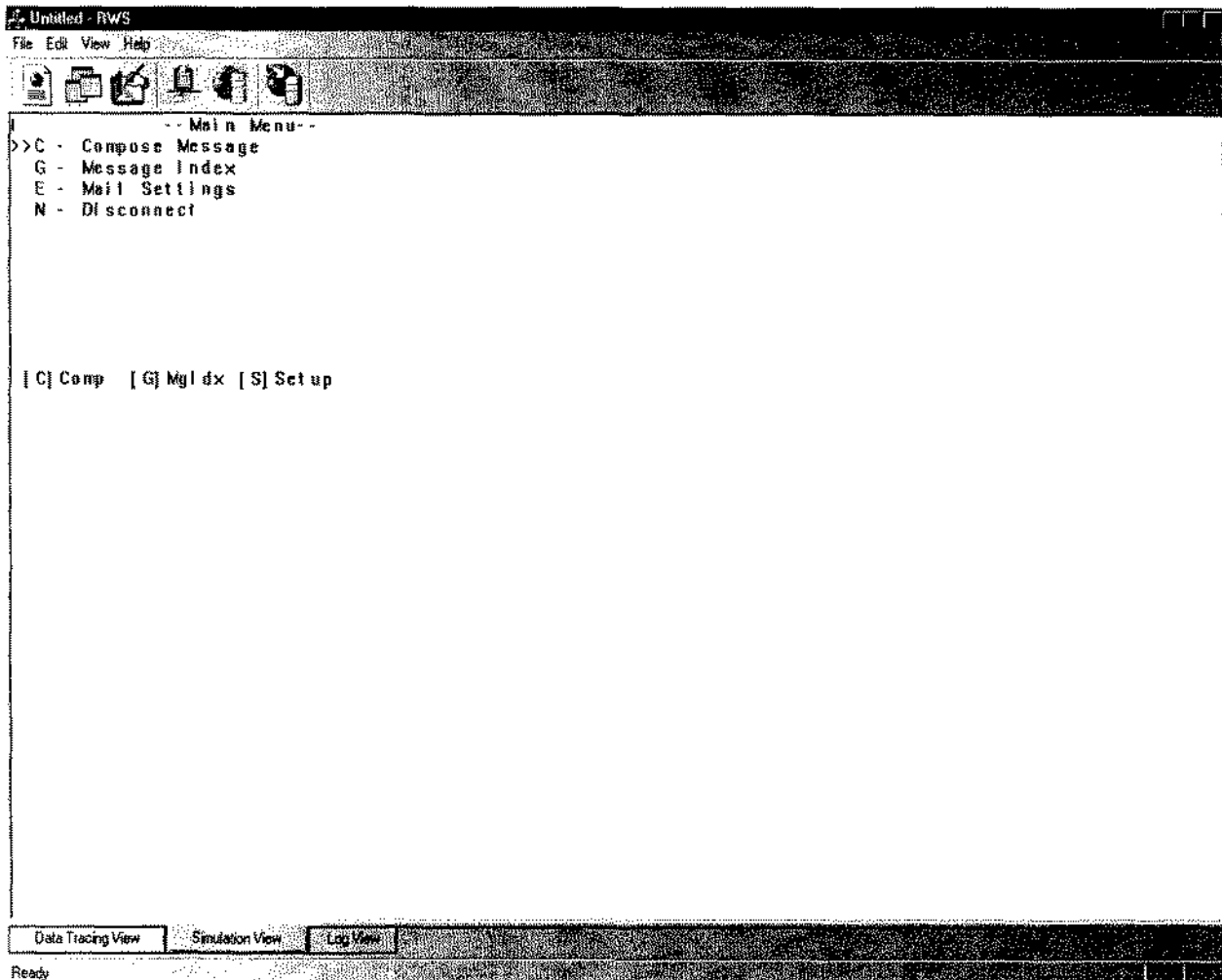
The 'Dial option' (CTRL +D) present on the Internet Dialup screen is used to connect the application to an ISP using Real Time Scheduler for TCP/IP.



• Email Menu

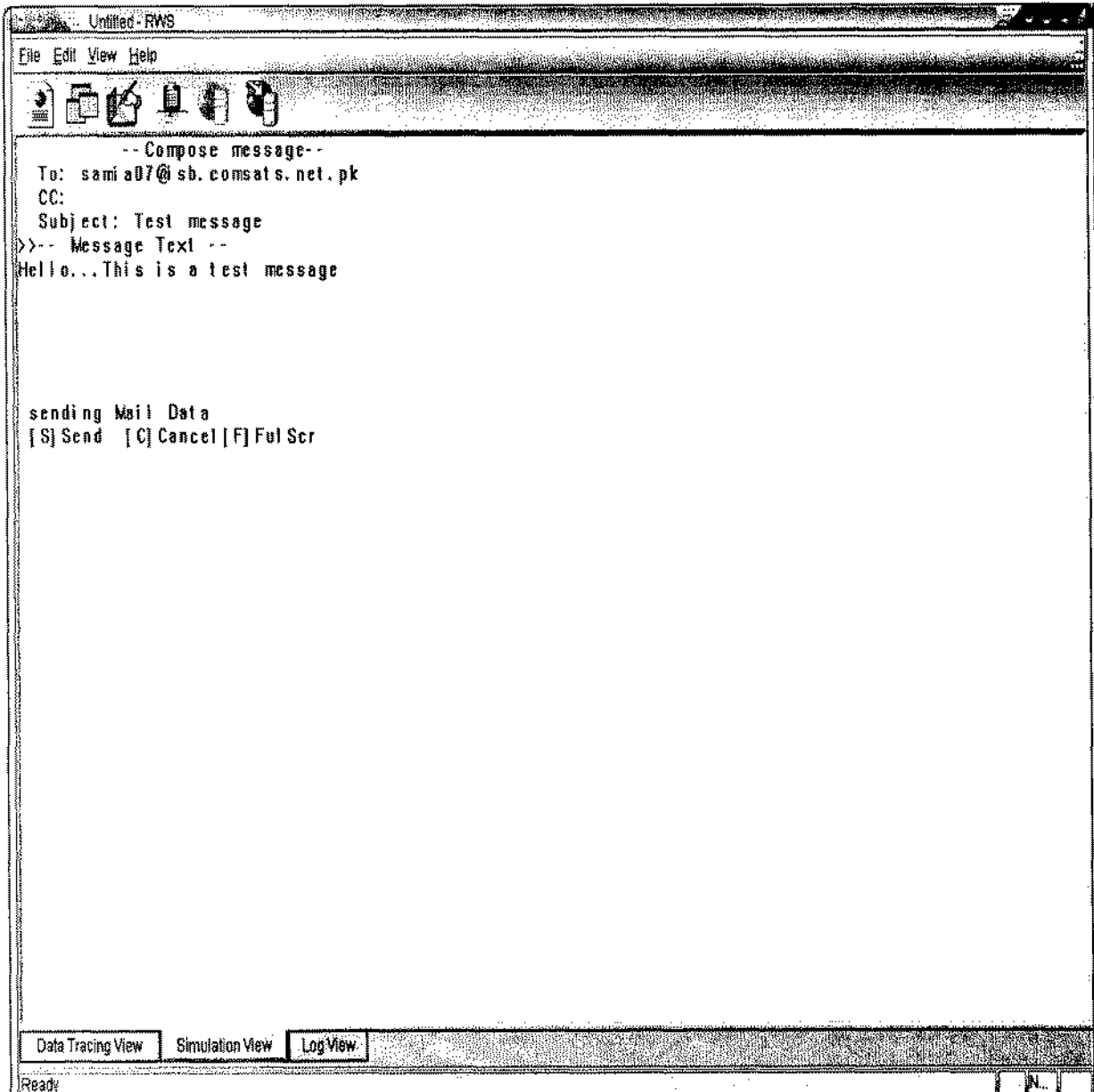
As soon as the application connects to the ISP, the Email menu displays on the screen. With the help of this menu, email messages can be composed and the emails can be retrieved.

1. Compose message option is used to compose and send an email.
2. Message index displays the list of messages present in inbox.
3. Mail settings option is used to change default email settings.
4. Disconnect option is used to exit.



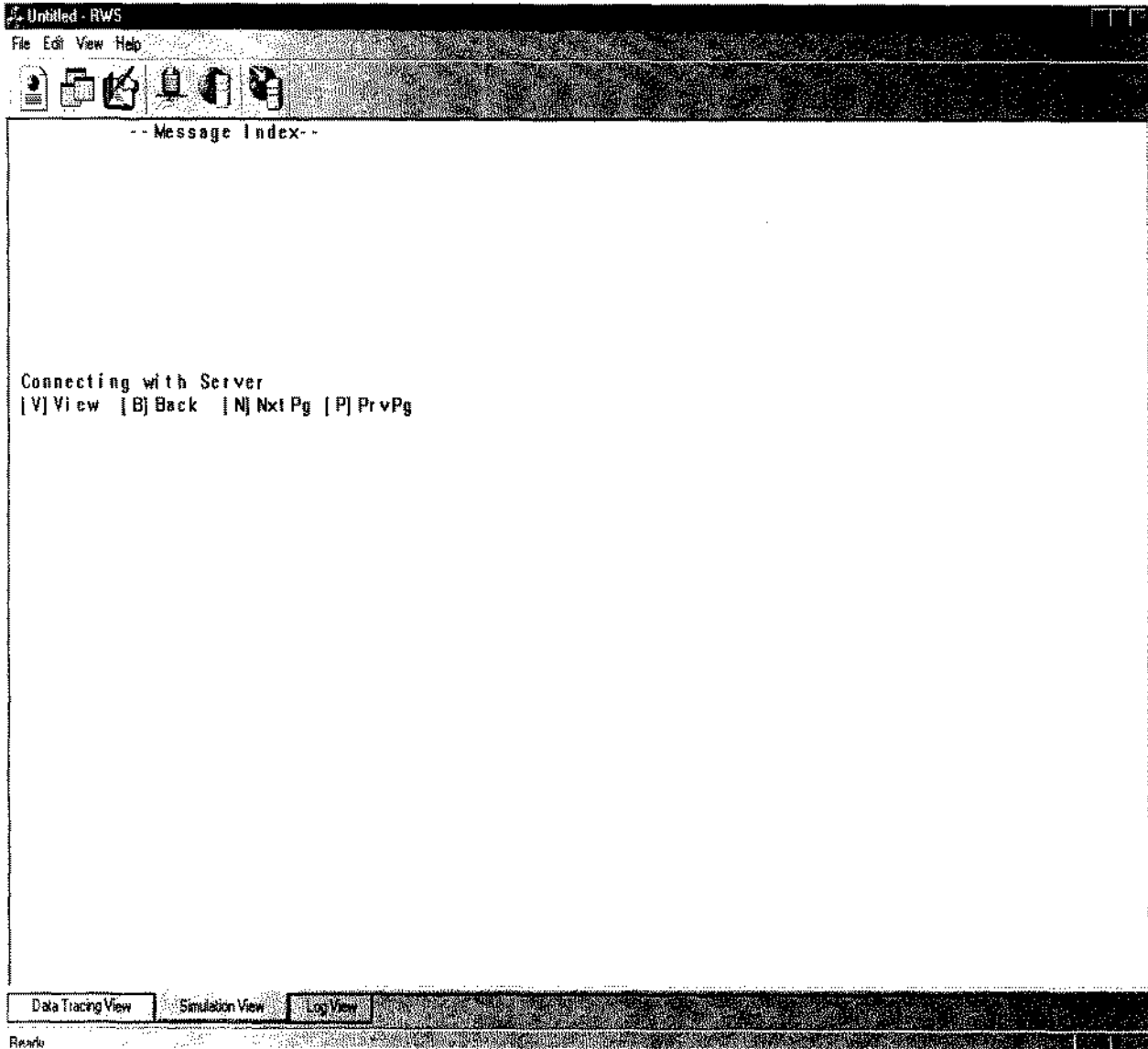
- **Compose Message**

This option is used to send an email.



- **Message Index**

This option is used to retrieve incoming email messages.



A3. Log View

It shows the activities related to Task Queues in the scheduler.

```

number 0 is free buf
number 1 is free buf
number 2 is free buf
number 3 is free buf
number 4 is free buf
number 5 is free buf
number 6 is free buf
number 7 is free buf
number 8 is free buf
number 9 is free buf

[ _name[0]=F_TCPUP:      set_fifoix(0,2);
[ _name[1]=F_TCPDN:     set_fifoix(1,2);
[ _name[2]=F_UDPUP:     set_fifoix(2,2);
[ _name[3]=F_UDPDN:    set_fifoix(3,2);
[ _name[4]=F_JPPUP:    set_fifoix(4,2);
[ _name[5]=F_JPCDN:    set_fifoix(5,2);
[ _name[6]=F_MACUP:    set_fifoix(6,2);
[ _name[7]=F_PPFUP:    set_fifoix(7,2);
[ _name[8]=F_CDDUP:    set_fifoix(8,2);

```

Osa Tracing View Simulation View **Log View**

Ready

Appendix B

Development Environment

B. Development Environment

B-1. Hardware Specification

1. Pentium 4 Processor
2. 56 K External Serial Modem
3. Telephone line and a valid ISP Connection

B-2. Software Specification

1. Windows 2000/Xp for Simulation Environment
2. C and Assembly for layer implementation
3. C++ for interface application
4. MFC for simulation GUI
5. Keil's C compiler for embedded system support

Appendix C

Publication

Real-Time Scheduler for Transport Protocols

Samia Aslam Sherwani and Malik Sikander Hayat Khiyal
Department of Computer Science, Faculty of Applied Sciences,
International Islamic University, Islamabad, Pakistan

Abstract: Real-Time Operating Systems (RTOS) currently available in industry, for embedded systems, require multitasking support in the targeted processor. The category of such operating systems is known as Pre-emptive Multitasking Kernels. However multitasking support is not provided by all processors. We have developed multitasking scheduling technique (Collaborative Multitasking) for the processors or microcontrollers which do not have built-in multitasking support such as support for context switching, for example, 89C51 Microcontroller and 89C52.

Key words: Collaborative multitasking, context switching, embedded systems, microcontrollers, real time operating system, real time scheduling, TCPIP suite

INTRODUCTION

In the modern age, the intelligence of computing power has been integrated into every device and gadget, resulting in embedded systems. An embedded system refers to a device with computer logic on a chip inside it, typically consists of a single-board microcomputer with software in ROM designed to perform a dedicated function. Such systems are structured in a different manner as compared to high performance desktop systems. Their designing issues include: Low cost, predictability, responsiveness (Seo *et al.*, 1998) and temporal accuracy (Kopetz and Oetsenreiter, 1987).

Embedded systems normally exist as part of a bigger system and are constructed with the least powerful processors that can meet the basic functional and performance requirements so that the manufacturing cost of the equipment can be lowered. As discussed by Agarwal and Bhatt (2004), due to absence of general features and extremely tight design constraints, unlike in conventional systems, the developers of embedded systems have to work with complex algorithms to manage resources in the most optimized manner.

Scheduling is a mechanism that determines which job has to be executed from the pool of jobs in system on the basis of the scheduling algorithm implemented. Whenever multiple tasks share common processing resources, they require their states to be stored at the time of process switching, so that these can be restored afterwards. The state includes all the registers that the process may be using, especially the program counter, plus any other

operating system specific data that may be necessary. Often, all the data that is necessary for state is stored in one data structure, called process control block. According to Naoul and Givargis (2005), in order to support multitasking on a system, an operating system layer is needed, which it is not commonly available in embedded systems due to lack of sufficient memory. Examples are PIC (Huang, 2005) by Microchip and 8051 (Calcutt *et al.*, 1998) by Philips. These microcontrollers are cheap enough to give cost effective devices. If these micro-controllers are being planned to be utilized for handling complex multitasking scenarios, it is only possible if handled programmatically within embedded software design.

REAL-TIME SCHEDULER: COMMON PRACTICE

Donald Gillies defined real-time system as follows:

- A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced.

Real-time systems perform a number of tasks at a time. The Resource Manager allocates processor time to each task according to the schedule in such a way that the tasks appear to be parallel. The process of choosing a task to grant resources to, that is, Central Processing Unit time is called Real-Time Scheduling as defined by Liu and Lee (2003).

Corresponding Author: Samia Aslam Sherwani, Department of Computer Science, Faculty of Applied Sciences,
International Islamic University, Islamabad, Pakistan Tel: (92) 03345046539

Preemptive multitasking: A typical strategy to implement real-time scheduling is called preemptive multitasking. In preemptive multitasking, operating system uses some criteria to decide how long to allocate to any one task before giving another task a turn to use the operating system. The act of taking control of the operating system from one task and giving it to another task is called preempting.

To perform resource management using preemptive multitasking, the resource manager has to perform two duties:

- Context Switching
- Task Scheduling

When a task is in running state and the time slice has been expired such as timer event, the scheduler is invoked which decides which task deserves to be given next time slice using its scheduling algorithm based on priority system. Finally, it performs context switching, replaces first task by second task and lets the later task to perform its duty. Figure 1 explains the scenario.

Limitations in preemptive multitasking: Preemptive multitasking is implemented on a processor or a micro-controller, which has built in support for context switching and a periodic task trigger on which event scheduler has to be invoked. A common criterion is simply elapsed time: the timer implemented in hardware is programmed to be invoked on expiration of a time slice. The timer generates an interrupt, which initiates an interrupt service routine. In interrupt service routine, scheduling is performed and it is decided which task is to be granted processor next. The state of currently executing task is saved and the context of the next task is loaded into the CPU. After ISR, the CPU starts executing the newly loaded task. So, to perform task switching, the CPU must have spare context registers, called 'Register Banks' as shown in Fig. 2.

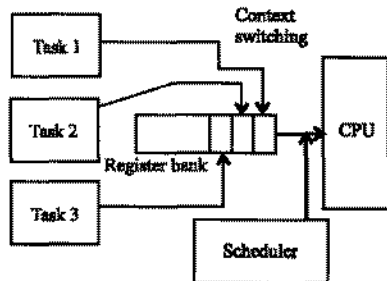


Fig. 1: Real time scheduling

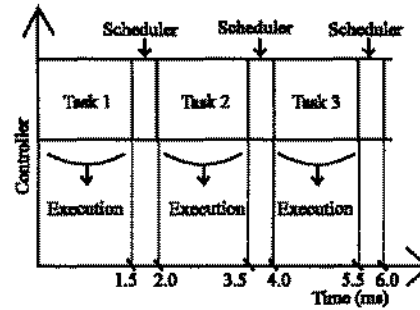


Fig. 2: Preemptive multitasking

High level processors, such as, Intel 8086, Intel 8088, Intel 80386, Intel 80486, Pentium and Pentium Pro support pre-emption. There are number of micro-controllers that provide built in hardware support for context switching and periodic task trigger, for example C166 and C167 by Siemens (2000).

COLLABORATIVE MULTITASKING APPROACH

As explained earlier, the multitasking performed by context switching requires very particular hardware support, which is not available in tiny micro-controllers such as Intel 80C51 and 80C52.

This article gives a programming model to implement multitasking in real-time tasks, for example, running a TCP/IP based application. Collaborative multitasking model gives the idea to address fundamental issues of running preemptive multitasking kernel on tiny micro-controllers.

Tasks collaboration: In collaborative multitasking, tasks (any user process running on that controller) collaborate with each other in a way that each task executes a part of its route, saves its state locally and then releases system resources voluntarily.

In this system, each task is represented by function or routine. In this idea, no task is forced to preempt resources from it. A task returns after executing a part of it, saves its state and gives control to other task waiting for resources as shown in Fig. 3. The sequence executes in a continuous fashion.

For example, we have three tasks. First Task is Display task whose responsibility is control LCD display. Second task is Comm task whose responsibility is to receive any data from comport and process it and the third one is KeyPad task which scans the keys and gets any activity of key pressing. Now these three tasks will collaborate with each other. When Display function will be called, it will scan all the display memory and will refresh it on the screen in one cycle. After that it will

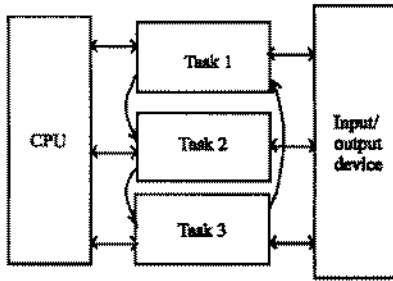


Fig. 3: Collaborative multitasking

return back and Comm task will be invoked. In a single cycle, Comm will scan its comport, receive any incoming waiting data and process it.

After that it will return back and then finally KeyPad task will be invoked. In a cycle, KeyPad will scan all the keys and will refresh keypad memory indicating any key press event. This sequence will execute continuously.

```
main ()
{ InitSys ();
  While (1)
  {   Display ();
      Comm ();
      KeyPad ();
  }
  QuitSys ();
}
```

The sharing of resources among the tasks is not based on time slices, but sharing is done on work basis or number of instructions. Every task divides its whole work into sub-tasks. Whenever a task is given control of CPU, it executes one of its sub-tasks and returns the control. In next allocation of CPU, it executes next sub-task.

For example, we have an embedded system which has to execute three tasks: Task 1, Task 2 and Task 3 simultaneously. Task 1 is further divided into three subtasks: subtask 1, subtask 2 and subtask 3. Task 1 completes, as each subtask executes ones.

```
While (1)
{   Task1 ();
    Task2 ();
    Task3 ();
}
Task1 ()
{   static int nStat=0;
    switch (nStat)
    { case 0: Subtask1(); nStat=1; break;
      case 1: Subtask2(); nStat=2; break;
      case 2: Subtask3(); nStat=0; break;
    }
}
```

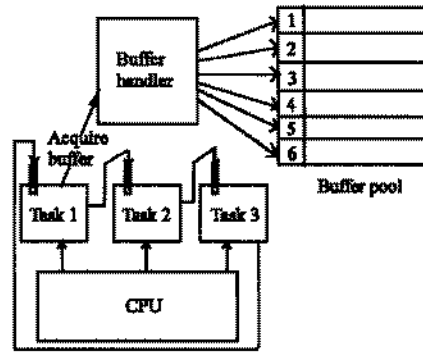


Fig. 4: Inter-process communication

The scheduler is designed such that every task executes its one sub-task in its turn and returns back so that next task can be executed. In above example, Task 1 completes in three iterations. In this way, all the tasks are executed simultaneously because of their collaboration with each other.

Queues for inter-process communication: Inter-process communication is always an important issue when designing scheduler for real-time embedded systems. In collaborative multitasking programming model, every task has its incoming and outgoing FIFO and also there is a shared buffer pool. Whenever a task wants to send data to another task, it acquires a free buffer from buffer pool, copies the data in buffer and puts the index of buffer in incoming FIFO of their task. Every task polls its incoming FIFO and processes the data, if present, as shown in Fig. 4.

Task priorities: Task priority is very important concept in multitasking system. The priority represents the relative importance of a task at run time. When three tasks are running at a time, then the process of determining which task deserves CPU more is called priority.

For example, we have Task 3 that is more important than Task 1 and Task 2. Then its priority can be implemented, as its iterations can be increased relative to other tasks.

```
While ()
{   Task1 (4);
    Task2 (1);
    Task3 (2);
    I/O Task (1);
}
void Task1 (int nPriority)
{
```

```
int nIteration=0;
While (nIteration<nPriority)
{
//execute subtasks
nIteration++;
}
}
```

The priority of a task can be determined at run-time and it can be set according to the situation.

Example: Here, we describe the design and development of real-time scheduler for transport protocols, such as TCP/IP. The TCP/IP protocol stack is implemented such that each layer is represented by a task.

```
INIT_STACK;
WHILE (nQUIT)
{
APP_TASK;
TCP_TASK;
UDP_TASK;
IP_TASK;
PPP_TASK;
COM_TASK;
}
```

The main thread initializes all the layers, distributes the time slices by calling respective processes. Each layer works in two directions, that is, it processes data from upper as well as lower layer. A separate buffer bank is reserved for data to be processed, in the form of two-dimensional array. Each buffer has following associated attributes:

- Name of the buffer (Free, Temporary, PPP Down, PPP Up, IP Down, IP Up, UDP Up, UDP Down, TCP Up, TCP Down, Application Down, COM Up)
- Command (No command, dial, ping, valid IP frame, etc)

Message flags associated with each process control sub-processes. Each layer has two Data Queues associated with it, one for each direction: Layer up Job Queue, that contains pointer to the buffer received from upper layer and is ready to be processed according to the command associated with the buffer and status of the message flag associated with that direction and Layer down Job Queue, for data received from down layer. These Job Queues are responsible for inter-process communication.

Whenever an application wants to perform a TCP/IP related task, it gets a buffer from buffer bank, adds data to buffer, associates a command with buffer which

indicates what has to be done with the data in buffer and passes buffer reference to the Layer Up Job Queue of the lower layer.

On turn of task associated with the next layer, the incoming job queue is checked and the buffer is processed according to the command, flags are set and the buffer reference is added to Layer Up Job Queue of the next layer. Next layer behaves in same way, until data reaches COM layer and is written to COM port.

It is not necessary for a task to complete its job in single iteration. So, each task has to maintain its state, so that it can continue from the same point in next iteration. For that, each layer performs part of its task, saves its state in buffer and keeps the track of previous work with the help of flags associated with each task.

CONCLUSIONS

This TCP/IP stack is tested with SMTP/POP3 application and also compiled on Keil Cross compiler for embedded system support. As the whole stack works in one thread, so it does not require multitasking support in target processor. Hence the research is successful.

REFERENCES

- Agrawal, S. and P. Bhatt, 2004. Real-time Embedded Software Systems: An Introduction. Technology Review.
- Calcutt, D., F. Cowan and H. Parchizadeh, 1998. 8051 Microcontrollers: Hardware, Software and Applications. John Wiley and Sons, New York, USA.
- Huang, H., 2005. PIC Microcontroller: An Introduction to Software and Hardware Interfacing. Delmar Learning, New York, USA.
- Kopetz, H. and W. Ochseneiter, 1987. Clock synchronization in distributed-real-time systems. IEEE Transactions on Computers, C-36: 933-939.
- Liu, J. and E.A. Lee, 2003. Timed multitasking for real-time embedded software. IEEE Control Systems Magazine, 23: 65-75.
- Nacul, A.C. and T. Givargis, 2005. Lightweight Multitasking Support for Embedded Systems Using the Phantom Serializing Compiler. Proceedings of the Conference on Design, Automation and Test in Europe, 2: 742-747.
- Seo, Y., J. Park and S. Hong, 1998. Efficient User-Level I/O in the ARX Real-Time Operating System, Lecture Notes in Computer Science, Vol. 1474.
- Siemens, 2000. C167CR/C167SR 16-Bit Single Chip Microcontroller, Data Sheet, V3.1.