THE DESIGN AND IMPLEMENTATION OF THE CAMBRIDGE RING PROTOCOLS

by

JULIAN HONG PAK WONG

B.Sc., Washington State University, 1982

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

Department Of Computer Science

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

May 1984

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date 8/5/84

Abstract

Protocol implementation is characterized by various issues such as concurrency, time-critical responses, and asynchronous events. Such issues would be easier to resolve if the underlying operating system provides a hospitable environment that include an efficient interprocess communication facility. Most operating systems, however, do not provide such an environment.

This thesis presents an implementation and discusses what has been learned in implementing the Byte Stream and the Basic Block protocols of the Cambridge Ring on a LSI 11/23 computer running the Unix.V7 operating system.

Table of Contents

List of Figures

Acknowledgement

Many individuals contributed in special ways to the accomplishment of this thesis. I would like to thank the following people :

- Dr. Son Vuong for his supervision and financial support.
- Dr. Samuel Chanson for his guidance and for the reading of this thesis.
- R. Ravindran for his helpful suggestions and the completion of the performance measurements of the BBP.
- Linda Chan and Tan Ghee Hing for their fine editing skills.
- Jean Wu and Rosalyn Lee for their assistance in one way or another.

I am also grateful to the department of Computer Science for the financial support given while preparing this thesis.

And a special "Thank You" to Grace Kim for giving me encouragement and T.L.C. during the difficult times.

I. INTRODUCTION

1.1 Motivation And Objective

Current trends favoring decentralization of computing power and resources gained momentum on the arrival of Local Area Network technology.

A Local Area Network (LAN), as the name implies, refers to an interconnection of computers or devices located in a restricted geographical area. It provides an effective and inexpensive form of data communication among the constituent computers or hosts and devices linked to it. The hosts and devices are sometimes referred under the generic name of nodes.

A LAN, like any other data communication network, basically consists of :

- The physical medium for transmission, e.g. cables, twisted pair and fibre optics.
- The physical control mechanism for the use of the physical medium for transmission.
- The hardware interfaces to the nodes using the physical medium.
- The software residing in the nodes to control the communication process among nodes. This software is commonly known as communication protocols.

Protocols provide a set of rules to facilitate communication among the variety of hosts and devices in the network using the underlying physical medium that is susceptible to errors. The set of protocols implemented in each node can be

conceived as functioning at different layers. Each layer utilizes the services of the layer below it and provides the necessary services for the layer above to function. The lowest layer interacts directly with the hardware and the highest layer provides a transparent service to the user. This layering concept is formalized in the ISO Reference Model for Open Systems Interconnection [7].

The objective of this thesis is to design and implement the Cambridge Ring Byte Stream and Basis Block protocols [5,8] on a LSI 11/23 machine running the UNIX.V7 [13,14] operating system.

The LSI 11/23 machine is connected to a Cambridge Ring Local Area Network [15] of which there is only one other machine currently connected, a TI 990. The above mentioned protocols together provide a comparable transport layer service [7] to this host

Protocol implementation is characterized by the various issues of concurrency, time-critical responses, asynchronous events and robust recovery from errors. Features in the underlying operating system which would aid the programmer tremendously would include shared memory among related processes, good interprocess communication, high speed I/O, access control and synchronization to shared data and the provision of timers. Most of the procedure based operating systems, however, do not provide such features and if they do, it is not uncommon to find the facilities awkward or unsuitable to use. In the case of the quasi standard UNIX operating system, its interprocess communication (IPC) has even been

extended to facilitate protocol implementations [1,6,12]. This approach is undoubtedly effective and would represent an effort of a larger scale where a corresponding amount of human and economic resources is required. In our case, the above mentioned protocols are simply implemented on the existing UNIX.V7 system and the following chapters will enumerate the constraints of the underlying operating system on the final design of the resulting implementation.

1.2 Thesis Outline

Chapter ii begins with a brief overview on the Cambrige Ring and is followed by a brief overview on the set of protocols. Chapter iii describes the UNIX implementation environment and its constraints on the overall design of the set of protocols. The details of each implementation is described in chapters iv and v. Chapter vi concludes this thesis by presenting the final product and by suggesting possible improvements.

Since the implementation is coded in the system language C [9], many of the algorithms or examples given in the chapters are also in C; however, they are simple enough to be understood by a reader with some programming experience.

II. THE CAMBRIDGE RING AND ITS PROTOCOLS

This chapter presents an overview of the Cambridge Ring [15] and its low level protocols, namely the Basic Block and the Byte Stream protocols. This overview is a basis for the description of the implementations that will be described in the later chapters.

2.1 The Cambridge Ring

The Cambridge Ring is a slotted ring with a raw data rate of 10 megabits/second (with current twisted pair technology). The diagram shown below is a typical Local Area Network (LAN) using the Ring.

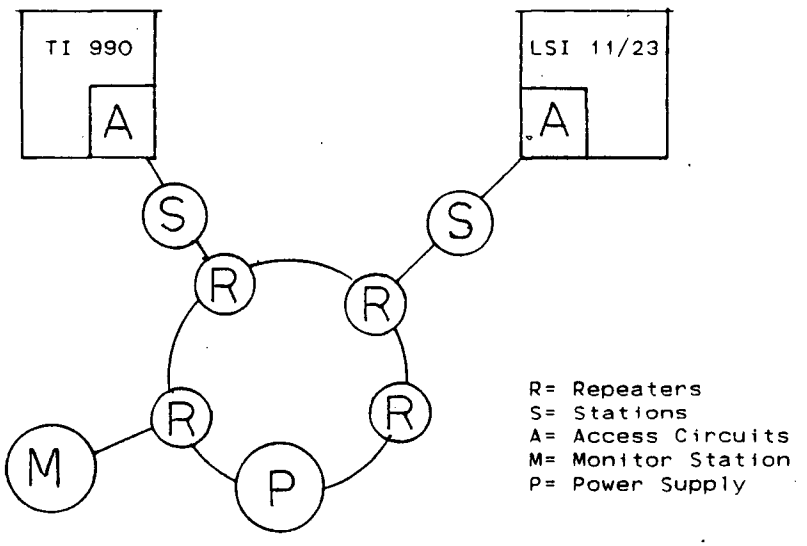

Figure 1 - Cambridge Ring LAN

For any device that is to be attached to the Ring, a repeater is required. Essentially the task of the repeater is to provide signal regeneration and to pass the data to the station.

Data is transmitted round the Ring in minipackets contained in the circulating slots. As shown in Fig. 2, each minipacket consists of two data bytes, a source address byte, a destination address byte, two response bits and four control bits.

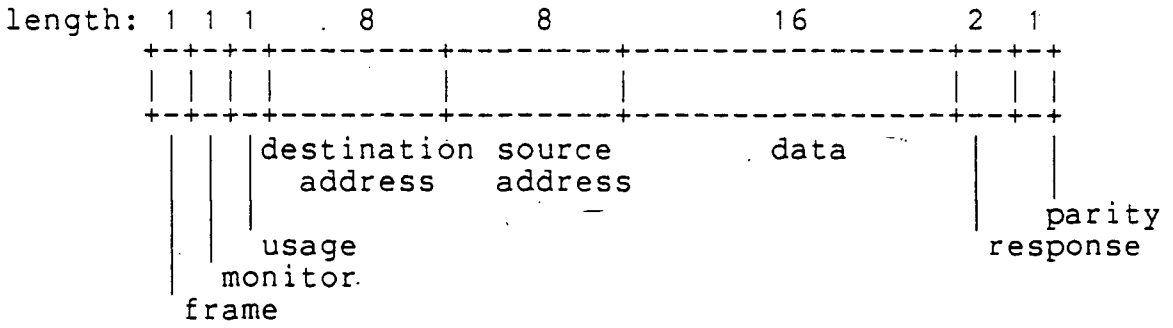

Figure 2 - Format of a Minipacket

Definition of the bits :

- Frame bit is set to one if this were a header packet.
- Monitor bit indicates if the packet had passed the monitor station.
- Usage indicates if the packet is in use.
- Response can take one of four values and indicates the destination stations response.

00 - accepted

01 - busy

10 - rejected

11 - ignored

- Parity is used for detecting transmission errors.

Between the host and the repeater comes the station. Each station has a source select register (SSR) which is used to permit reception from any source, from a nominated source or from no source. The station unit watches the bit stream emitted by the repeater and detects the framing of the minipackets; it provides the mechanism for receiving and transmitting of the minipackets.

When transmitting a minipacket, the station watches for an empty slot; when one arrives, it is marked full. The minipacket to be transmitted is then filled in and sent away. Eventually, this minipacket returns to the sending station after being processed at the destination. The possible responses to the transmitted minipacket are as follows :

Packet Accepted	:	The data and source bytes have been copied into the receiving station
Station Busy	:	The receiving station reception register was full
Packet Rejected	:	The Selection register in the receiving station had been set to exclude reception from this source
Error Packet	:	An error packet is returned

When a minipacket returns to the sending station, it is always marked as empty and passed on. This is an anti-hogging device and ensures that the bandwidth is shared equally among the stations.

The monitor station is a special maintenance station which is set up to monitor the Ring for obscure errors and other functions necessary for a reliable Ring.

The interface between a host and the Ring is provided by the access circuits.

2.2 Overview Of The Basic Block Protocol

The Basic Block Protocol (BBP) is the lowest level protocol which interacts with the Host hardware interface (access circuits) to the Ring. It provides an unreliable datagram service to higher level software. In the ISO protocol framework, this protocol layer is considered as the Data-Link layer. The basic unit of transmission of data in the BBP is the Basic Block; the format of which is as shown below.

- A Header minipacket
- A Port minipacket
- 1 to 1024 Data minipackets
- A Checksum minipacket

```
+--------+   +-------+   +-------+     +-------+   +--------+
| Header|   | Port |   | Data | - - | Data |   | Chksum|
+--------+   +-------+   +-------+     +-------+   +--------+
                         <-- 1 to 1024 -->
```

Figure 3 - The Basic Block

The Header signifies the beginning of a basic block and contains the length of the Data minipackets in the basic block. A transmitter may send up to 1024 Data minipackets. The format of the Header minipacket is as follows:

```
+---------+----+---------------+
| 1 0 0 1| TY |    LENGTH     |
+---------+----+---------------+
    4       2       10
   bits    bits    bits
```

TY= type of basic block.

Figure 4 - Format of Header

The Port minipacket allows multiplexing of logical channels within a host. The Data minipackets contain the actual message to be transmitted. The Checksum is a 16 bit sum of the basic block, computed modulo $2^{16} - 1$, with end-around carry.

2.2.1 Interface To The Ring

There are 2 types of hardware interfaces which are used in the Ring. The first type is simple programmed interrupt interface and the other is a sophisticated DMA (direct memory access) type interface. [3] The non-intelligent programmed interrrupt interface currently exists between the Ring and the LSI 11/23.

2.2.2 Mode Of Operation

From a programmer's point of view, the Ring can be perceived as a data highway [4]. On transmission the BBP software loads the destination address (8 bits) and the data (16 bits) onto the interface and signals the interface to transmit

this minipacket. The minipacket then goes round the Ring; at the destination interface, the response bits in the minipacket are altered to record the result of this transmission. On arrival at the sender's interface, the result of the transmission is then passed to the software.

The receiving part of the interface consists of a source select register (SSR), a received address register and a 16 bit data register. The SSR allows the receiving interface to be either one of the following states:

- listen to all stations
- listen exclusively to 1 station
- listen to nobody

By setting the contents of the SSR, the BBP could discriminate as to which station it wishes to communicate with. The received address register contains the address of the station which originated the transmission.

2.2.3 Shortcomings

In a programmed interrupt interface, an interrupt is generated for every minipacket received or transmitted. In a high speed Ring, the rate at which interrupts are generated at the interface exceeds the rate at which most operating systems can process them [11]. The problem is amplified if the CPU is also slow (see Fig. 9).

A rapid sender sending continuous material to such a station will likely experience a considerable amount of 'busies'

and will have to retry repeatedly. Consequently, a transmitting host will not be able to exploit the high speed transmission medium. To obtain ideal performance, it is necessary to make a transmission request to the station within about 3.5 microseconds of the return of the previously transmitted packet. Few machines if any can process a program interrupt that fast [11].

Another potential problem with using a programmed interrupt interface is that if the low level protocol interacting with the hardware interface were sufficiently complex, then the frequency of interrupts, coupled with the time spent in handling the interrupts in the protocol will more or less make the processor unavailable to all other processes in the system.

2.2.4 The Cambridge BBP

The following describes the Cambridge BBP [5,11].

Reception

- Set the Source Select Register (SSR) to listen to anybody.
- Any minipacket that arrives which does not conform to a Header is discarded. If a Header arrives, lock this station to listen exclusively to the sending station.
- Wait for the Port. On its arrival, determine whether the port received is active on the host system. If it is not, the Port is rejected and the SSR is set to Reject for a short period of time so as to inform the relevant source to quit its intention about sending this particular basic block.
- Accept the Data minipackets if everything above went on fine.
- Receive the Checksum. If there were no errors, relay the basic block to higher level, otherwise, report the errors. The SSR is then reset to listen to anybody.

Transmission

- Send the Header. If the response from the destination station is 'rejected' or 'busy', try sending the Header again.
- Send the Port. If the response is busy, try again; if unselected, give up.
- Send the Data. Retry on busies; if unselected, give up.
- Send the Checksum under the same rules as in sending the Data.

Both reception and transmission include the use of the Timeout and Retry mechanisms. The Timeout mechanism involves the use of a timer and setting it to a constant time interval. On the expiry of that interval, the corresponding action is abandoned. The Retry mechanism stipulates the number of repeated attempts allowed for an action.

2.3 Overview Of The Byte Stream Protocol

The Byte Stream Protocol (BSP) is the software layer above the BBP and provides the following services to other software layers above it :

- Error Correction
- Flow Control
- Lost Packet Detection
- Duplicate Packet Detection

Coupled with an Initial Connection protocol specification which is not part of the BSP specification itself, but supplementary to it, the BSP essentially provides a Virtual Circuit service.

2.3.1 Format Of Packet

During the data transaction phase, that is immediately after an an Initial Connection, or after a reset, the unit of communication is a Packet which consists of :

- A Reception Command, refering to the reception of data.
- A Transmission Command, refering to the transmission of data.
- The Data itself (if any).

To provide for control information, which refers to the BSP connection as a whole, the Packet may just contain a control command. The two control commands Reset and Close enable the connection to be reset due to unresolved errors as well as to terminate a connection. Provision of flow control is through the use of sequence numbers imbedded in the commands themselves. In this description, the sequence number is denoted as 'n' immediately after the command name; e.g. DATA n.

Two additional control commands arising from the Initial Connection protocol are OPEN and OPENACK ; however the procedure involved in setting up the connection is irrelevant here.

2.3.2 Mode Of Operation

The transactions between two communicating parties alternate between two phases. During data transfer, it is in a data transaction phase and during the exchange of control information, it is in a control transaction phase. We shall first describe the mode of operation during the data transaction phase.

When a transmitting host has data to send, a DATA n is sent; otherwise it sends a NODATA n. When the receiving host is ready to receive data, it sends a RDY n ; otherwise a NOTRDY n is sent. Both reception commands RDY n and NOTRDY n acknowledge

the previous DATA n-1 received from the transmitting host.

Transmission of a DATA n or a RDY n involves setting a timer and expects an acknowledgement before the timer expires. Because it expects an acknowledgement, DATA n and RDY n are sometimes referred to as essential elements. By the same token, NODATA n and NOTRDY n are referred to as non-essential elements as they do not expect an acknowledgement in reply.

When the timer expires or timeouts, the essential element is sent again. The number of repeated attempts or retries made before the attempt is given up is implementation dependent.

2.3.3 BSP State Transition

The control flow of the BSP with the exception of the control commands is expressed in the following state transition diagram :

State \ Event	E_{rep} $DATA_{n-1}/RDY_n$	E_{exp} $DATA_n/RDY_{n+1}$		N_{exp} $NODATA_n/NOTRDY_{n+1}$	Timeout	Buffer ready
E	Retransmit $RDY_n/DATA_n$ E	Empty/fill buffer n+: = 1 Buffer ready? yes: RDY_n $/DATA_n$ E	no: $NOTRDY_n$ $/NODATA_n$ N	— I	Retransmit $RDY_n/DATA_n$ E	— E
N	Retransmit $NOTRDY_n/NODATA_n$ N	Protocol error		Protocol error	— N	Transmit $RDY_n/DATA_n$ E
I	Protocol error	Empty/fill buffer n+: = 1 Buffer ready? yes: RDY_n $/DATA_n$ E	no: $NOTRDY_n$ $/NODATA_n$ N	— I	— I	— I

Figure 5 - State Transition Diagram for the BSP [11]

There are three states and five events.

States

E	Expecting an Acknowledgement. (i.e. a DATA n or RDY n was previously sent)
N	A non-essential element was sent. (i.e. a NODATA n or NOTRDY n was sent)
I	Idle state. A non-essential element was received in reply. (NOTRDY n-1 or NODATA n was received)

Events

E_rep	An essential element which is a repetition of an earlier essential element is received; i.e. a RDY n or DATA n-1 has been received.
E_exp	An essential element with the expected sequence number is received; i.e. a RDY n+1 or DATA n.
N_exp	A non-essential element with the expected sequence number is received; i.e. a NOTRDY n+1 or NODATa n.
Timeout	Expiry of timer while waiting for an acknowledgement.
Buffer Ready	Ability to receive more data or has data to send.

The two control commands Reset and Close pertain to the Byte Stream connection as a whole. The exchange of these control commands constitute the control transaction phase. When a Reset is sent, all received data should be ignored until a Reset is received in reply. An unexpected Reset received elicits a Reset sent in reply. The Close is analogous to the Reset. Both commands sent as a request rather than a reply would employ the Timeout and Retry mechanisms to ensure that the other host has not already terminated the connection and is unavailable to reply.

2.3.4 Initial State

Initially after a connection is made and after a Reset, the receivers should be in state N with sequence number 0, and the transmitter should be in state I with sequence number -1. As soon as the receiver has buffer space for the first Packet (Buffer Ready Event), it will send a RDY 0 and go to state E. The transmitter regards this as event E_exp, increments its sequence number to 0 and transmits either DATA 0 or NODATA 0.

2.3.5 Timers

There are two timers, a Normal timer and an Idle Handshake timer. The Idle Handshake timer is invoked only when the transaction is in state I. This is to prevent a perpetual wait for an E_exp event when the other host is no longer available and is unable to reply. The Normal timer, as the name implies, is invoked at times other than the I state and when an essential element is sent. The time duration for the Normal timer is in the order of one or more seconds, as opposed to one minute for the Idle Handshake timer.

III. IMPLEMENTATION ENVIRONMENT

3.1 Introduction

One common issue which arises in protocol implementations is where the protocol should reside [2]. The protocol may either reside in the kernel of the operating system itself, in a process provided by the operating system or both. The decision made would be strongly influenced by the machine architecture and the operating system design.

UNIX assumes two modes of operation: user and kernel modes. The relatively small resident UNIX kernel is a priviledged and protected section of code. Designed to be simplistic, it provides the essential services of process management, clock management, interprocess communication (IPC) and a general purpose input/output mechanism. The kernel interacts directly with the hardware and provides the abstraction and administration of the hardware resources to higher layer software above it.

Device drivers in UNIX are easily accessed through the UNIX I/O system. The I/O system uses the same clean interface of the file system. There are only two types of files in UNIX: Ordinary files and Special Files. Special files represent devices and are used to contain the device drivers.

The Basic Block Protocol (BBP) which interacts directly with the Ring hardware interface is thus implemented as a device

driver using the following file system interface :

- Open
- Read
- Write
- Close
- Ioctl

For protocol efficiency reasons, it would also be desirable for the BSP to reside in the kernel. However, the UNIX kernel environment prohibits this.

The UNIX kernel, both by the design and the physical limitations of the LSI 11/23 (address space of 64K bytes), is limited to the most basic of functions. The UNIX kernel in the LSI 11/23 machine incorporates the overlay concept. The most important and unmodifiable body of code and the data areas are always resident in memory; the rest is contained in overlays of size 8K bytes each. At any one time, there is only one overlay residing in memory. In using overlays, the objective is to minimize overlay swapping by placing related code together in one overlay. Calling for services which is located in other overlays involves swapping of the current overlay with the desired one in backing store.

Due to its requirement for buffers of size 2K bytes each, the BSP would not fit into the tightly constrained space left in the reserved data areas of the kernel. Although one may reduce the size of buffers to accommodate them into the data areas of the kernel, the BSP text would nevertheless require at least two overlays (unless we could ridiculously presume the BSP to be of

size 8K). Thus the execution of the BSP itself involves swapping. If the BBP is not placed in any of the overlays occupied by the BSP, then frequent swapping of overlays will occur since the BSP frequently requires the services of the BBP. In addition both the BBP and the BSP require services from other kernel routines, such as copying of data from kernel to user space, clock routines and processor priority manipulation routines. Consequently, there will be overlay swapping; the swapping frequency, however, could be minimized by a careful rearrangement of the contents of the overlays. Communication protocols which are time-critical in nature may not function correctly if its performance degrades dramatically as a result of the inordinate amount of disk activities arising from swapping.

Besides the constraints, there are other disavantages in implementing the BSP in the kernel. One major disadvantage is that since the kernel is a large body of code, it would take a considerable number of attempts to recompile the operating system in which the new kernel code is to be tested. For the LSI 11/23 machine, it took an average of five minutes to set up and perform a file system check. Secondly, the nature of complexity in the resulting implementation code for the BSP renounces the very principle of the UNIX kernel design of simplicity. Lastly, the kernel code is one large monolithic piece of code which is affected by the errors of any component part of it.

The above deterrents more than justify the placement of the

BSP code at the user level. However, placing the code at the user level implies that process scheduling can be a significant source of real time delay and the inadequate interprocess communication (IPC) facility of UNIX would hinder the development and efficiency of the protocol.

For efficiency reasons, it would be ideal to have the BSP running immediately after system booting. This may be realized in UNIX by making it a daemon process, that is a process owned by the system. This is desirable as basic blocks may arrive from other stations on the Ring even though there is no active port currently available. Another apparent efficiency is that any user wishing to use the network can access the BSP services without much delay. Is this concept feasible in UNIX ?

It is possible in UNIX if the MPX files are working correctly. MPX files are multiplexed files which enable 1 to n number of processes to communicate with one another through the owner of the MPX file. The owner of the MPX file has to perform the necessary multiplexing of messages to the various processes. The communicating processes need not be "related" (see section on "Forks and Pipes") in order to communicate. However MPX files are an experimental part of the UNIX system and is not working as specified in the version that we have. The fact that independent or non-related processes cannot communicate with one another in a simple and efficient way drastically affects the structure of the BSP.

3.2 Interprocess Communication In UNIX

The nature of communication protocols dictates a structure of cooperating processes with a main process acting as a coordinator or Server. Each process performs a function independently and reports to the Server of the completion or arrival of an event. We may think of such a model as the interaction of a Server process and a number of Client or Worker processes. The Server processes requests/services from its Client/Worker processes (see Fig. 10). The underlying motivation for such a model stems from the need to promote efficiency through concurrency. For example, we could have the Server delegate the tasks of reading and transmitting of basic blocks to two Worker processes such that the Server is free to perform other functions.

Such a model clearly reflects the nature of protocols which are characterized by issues such as asynchrony, concurrency and time-critical responses. We shall now see how UNIX supports such a requirement of cooperating processes.

3.2.1 Forks And Pipes

Creating a process in UNIX involves making a copy of itself by issuing the system call fork. The new process is denoted the child process and the creator of the new process, the parent. There is an explicit relationship between parent and child, and this information is recorded into the system tables. The child process has its own unique 16 bit process-id and inherits from the parent all the files that were opened before the fork. It

has its own address space, implying its own data and stack areas. The text may be shared since text are placed in read-only segments. The shared opened files are the only common link through which the IPC in UNIX is founded. The example below illustrates the creation of multiple processes.

```
if ( (id= fork()) == -1 ) {
    perror();
    return;
}
if ( id == 0 ) {
   /* child process's section of code */
}
else {
   /* parent process's section of code */
}
```

The fork system call returns -1 if there was an error. The error could be caused by various reasons such as the depletion of system resources or the number of processes active under this user exceeds the maximum allowed. "perror" is a library routine which prints out the error message. The process-id returned from the fork within the child process is 0 and within the parent process, the new child process-id, which is never zero. Extreme care should be taken after the fork since the return of the call is in the same location for both the child and parent. This is because the child is an exact copy of the parent.

Communication between two processes is often done through <u>pipes</u>. A <u>pipe</u> is a special kind of file that provides a one way means of communication. One end of the pipe reads and the other end writes. To provide a dialogue, two pipes are therefore needed.

An example:

```
main() {
  int pip[2];
  char buf1[10], *string;

   if ( (pipe(pip)) == -1 ) {
      perror();
      return;
   }
   if ( (id= fork()) == -1 ) {
      perror();
      return;
   }
   if ( id == 0 ) {
      /* code for child */

      close( pip[1] );     /* close writing end of pipe */
      read(pip[0],buf1,10);
      exit();
   }
   else {
      /* code for parent */

      close(pip[0]);       /* close reading end of pipe */
      string = "Go Home  ";
      write(pip[1],string,10);
   }
}
```

The pipe routine requires as parameter a two component integer array. It returns the file descriptors for reading and writing into the array in respective locations. File descriptors in UNIX are integers used to identify files during file operations. In the example above, the process reading from the pipe has to close the writing end of the pipe; similarly the process writing to the pipe closes its reading end. The closing of the writing end is mandatory for the reading process. This is because reads are blocked until there is something to read or an end-of-file occurs. Not having closed its own writing end of

the pipe would cause the read to be perpetually blocked as the end-of-file will never be read.

Pipes provide flow control and a buffer of size 4096 bytes for messages arranged in a FIFO queue. Writing to a pipe would not be blocked unless the pipe is full.

The pipe mechanism may be extended to provide for many Worker processes to communicate with the Server process through a single pipe. The reading ends of the pipe for all the Worker processes are closed while the writing end of the pipe for the Server process is closed. Additional pipes may be provided to designated Worker processes to facilitate two way communication with the Server. Since the pipe queues the requests, multiple events inherent in protocol implementation could be realized in a chronological fashion.

3.2.2 Locked Files And Signals

Ordinary files may also be used by processes for intercommunication. To provide synchronization when using files, "locks" may be placed on files so as to ensure mutual exclusion of shared data. A process wishing to access the shared file would only be allowed if the file is "unlocked". During access, the file would appear locked to all other processes attempting to access it.

The last available means of interprocess communication is through "signals". Signals are software interrupts which could be used to inform a process that an event has happened. The event is represented as an arbitrary integer, usually possessing

significance only to the communicating parties concerned. Signals mainly provide synchronization since it could only hold an integer size worth of information.

A process utilizing signals must provide a signal handling routine. When a signal arrives, control is passed to this signal handling routine. If no such routine exists, the process is terminated. The concept is similar to the interrupt handling routines used for I/O devices. Occasionally, signals have the undesirable effect of aborting a partially completed system call. Thus, usage of signals involves an incorporation of a checking mechanism when system calls are used. This is to determine whether a failure of a system call is genuine or a consequence of the signal arrival.

3.3 Design Overview

The UNIX environment thus dictates the fundamental design structure of the BSP. Its inadequacies, besides its impact on the protocol efficiency, is manifested chiefly in the following aspects of the final design :

- Multiplexing of users
- User Interface
- Administration of IPC

Before we dwell on the inadequacies of UNIX, a brief description on the BSP design is appropriate here. The final design of the BSP involves two Client/Worker processes, namely the User and the Reader, interacting with the Server (see Fig. 6). The User --symbolizing the user-- processes the requests from the user and the services rendered from the Server. The Reader performs the relay of the basic blocks read from the Ring driver to the Server. Although the transmission of basic blocks may also be performed by another Worker process, this is not done; we shall see the justification for this in chapter v.

3.3.1 Multiplexing Of Users

The assignment of Port numbers to facilitate identification and multiplexing of users has to be done at the kernel level. This necessity arises from the inability of non-related processes to communicate with one another. That is to say the

prospect of a single Server process interacting with one or more independent User processes is not feasible. Each user therefore has its own BSP process. The administration of ports is provided by means of two system calls:

- creatport()
- relport()

- creatport(reader-id) returns a port number, otherwise it returns -1. The parameter passed is the process-id of the Reader.
- relport(portno) makes available the portno to be reused.

To record the assignment of ports to processes and to administrate the available port numbers, a system table, <u>port-procid</u> is created. The port numbers and the corresponding process-ids are utilized by the BBP to pass the arrived basic block to the appropriate Reader of the BSP Server. When the Ring device driver is finally closed, the entries in this system table are erased. The figure below graphically depicts the final design of the BSP and the underlying BBP.

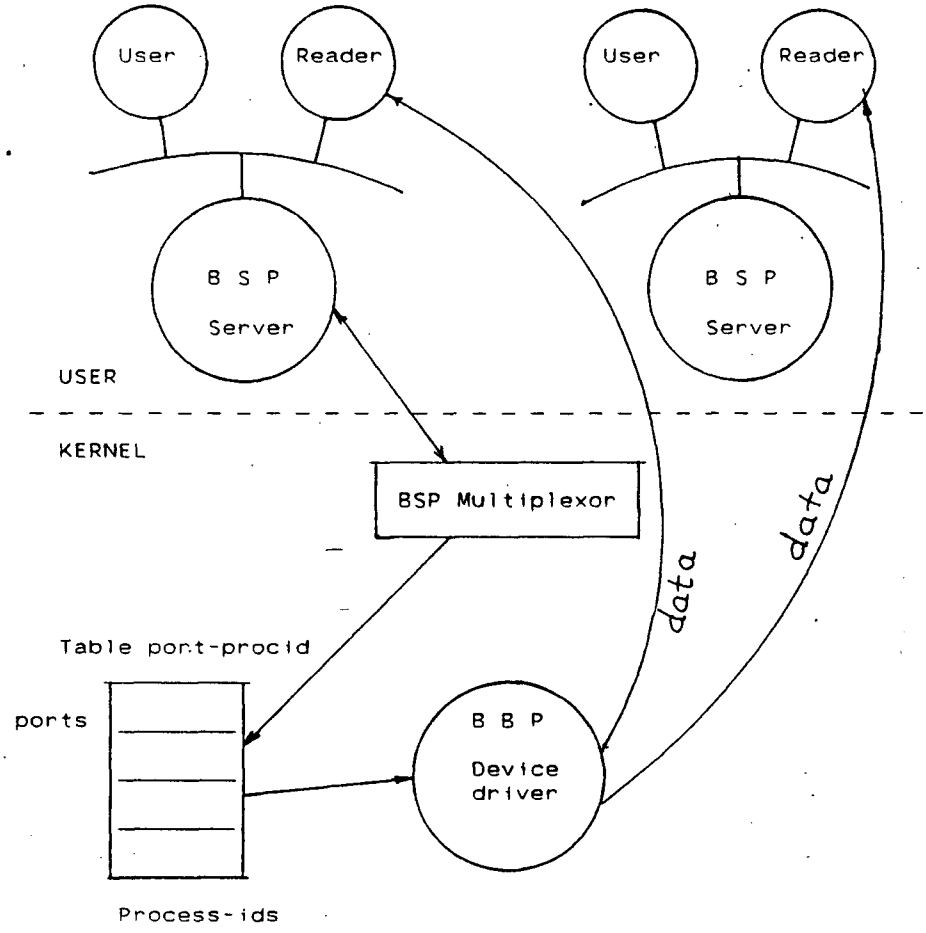

Figure 6 - Overall Design of BSP and BBP

3.3.2 User Interface

Since unrelated processes may only communicate through files, the interface for users to access the services of BSP may be done using a locked file; enabling an awkward, slow and limited means of interaction. Its limitations rest on the synchronization and identification of the reads and writes from both the actual user and the User process. Server. This interaction with the user is not part of the BSP and is consequently a protocol itself, utilizing the services of the BSP and resides in a process owned by the BSP through a fork (i.e. this protocol will reside in the User process). With such a constraint, the services of the Cambridge Ring may only be utilized in a simple file transfer service; this is because other types of services may only be realized in more difficult ways.

3.3.3 Administration Of IPC

The final design involving the Server interacting with the User and Reader processes is realized through the use of a single pipe multiplexing both Client processes. Care is taken to ensure that the appropriate ends of the pipe are closed and a message format is designed to facilitate identification and multiplexing of the Client processes. The message is made up of a control part and a data part. The control part, as the name implies, contains information that identifies the process as well as the type of requests. If there is data attached, the control part also contains the length of data to be read in.

The format of the buffer used between the BSP and the Ring device driver and the format used here are unavoidably different. Consequently there is copying of data to overcome different formats of buffer used.

IV. BASIC BLOCK PROTOCOL IMPLEMENTATION

4.1 Existing BBP Implementation

There is an existing implementation of the BBP on a TI 990. It is a simple implementation which deviates slightly from the standard BBP proposed by Cambridge. To enable communication with this BBP, the BBP implemented on the LSI 11/23 is made compatible. What is described below may not fully represent the original implementor's argument for the design and is partially this author's analysis for such a design. This is because full documentation was not available. The major deviations and their justifications are :

- There is no minipacket for the Port . The port number is imbedded into the Header packet. This is to fully utilize the vacant space in the Header packet, to simplify the BBP algorithm and overall to improve the throughput. Hence for the recognition of the Header, the two extra control bits in the 40 bit minipacket is set to signify that the minipacket is a Header. The layout of the modified basic block is as shown.

```
+---------------+  +---------------+ +---------------+   ------+
|type |0|length|  | data  | data | | data  | data | - |data |
+---------------+  +---------------+ +---------------+   ------+
  4       11         8       8        8       8           8
 bits    bits       bits    bits     bits    bits        bits
```

type = client protocol type (0 - 15)

0 = Reserved
1 = Virtual Wire Protocol
2 = Byte Stream Protocol
3-14 = Unassigned
15 = Reserved

length = data length - 1 (0 - 2047) bytes
data = client's data (1 - 2048) bytes

Figure 7 - Modified Basic Block

- The Checksum is not calculated nor sent; this is justified by the fact that the rate of corruption of bits in the Ring is extremely low. Rates of corruption of one bit in 10^{10} or one in 10^{11} are common. Furthermore, the calculation of checksum is generally known to be a major usage of critical processor time during the reception and transmission of a basic block. Measurements were conducted and the results were that without the Checksum packet, the throughput was increased by as much as 10% per basic block.

The layout of the block of data describing the interface between the BBP and higher level software is as shown:

```
+--------+-----+-----+--------+--------+---
| station| 0000| type|  data  |  data  |   .......
+--------+-----+-----+--------+--------+---
    8      4     4       8        8
   bits   bits  bits    bits     bits
```

station = destination/source station address
type = client protocol type
data = 1 - 2048 client data bytes

Figure 8 - Interface block between Ring driver and BSP

4.2 Algorithms

To illustrate clearly the structure of the BBP, the following reception and transmission protocol algorithms are listed below.

RECEPTION

```
LOOP:   Set SSR to 255              (listen to anybody)

        If HEADER arrives
          then
           begin
              LOCK on to the transmitting station.
              start TIMER for basic block.

              For I = 1 to Length-basic-block DO
                 Copy DATA into internal buffer.

              Inform Higher Level & goto LOOP
           end
          else
           begin
                Discard minipacket.
                Goto LOOP
           end

TIMEOUT:
         Discard whatever that has been received so far.
         Goto LOOP.
```

NOTE: When the TIMER expires, the clock would interrupt and execute the statements after label TIMEOUT.

TRANSMISSION

```
        Start TIMER
        RETRIES = 0
LOOP1:  Txmit HEADER   (including imbedded PORT information)

        Case RESPONSE:

          BUSY, REJECTED:   If RETRIES < MAX-HEADER-RETRY
                              then
                                begin
                                  RETRIES= RETRIES + 1
                                  Goto LOOP1
                                end
                              else
                                return (ERROR)
          IGNORED, ERROR-PACKET:  return (ERROR)
        end Case

        FOR Rest of Basic Block DO
          BEGIN

                RETRIES= 0
LOOP2:          Txmit DATA

                Case RESPONSE:

                   BUSY: If RETRIES < MAX-DATA-RETRY
                           then
                            begin
                                RETRIES= RETRIES + 1
                                Goto LOOP2
                            end
                           else
                                return (ERROR)
                   REJECTED,
                   IGNORED,
                   ERROR-PACKET:  return(ERROR)

                end Case
          END

TIMEOUT:  return (ERROR)
```

4.3 Unix I/O

Devices running under the Unix operating system are also considered as files. There are essentially two types of files in Unix, namely Ordinary files and Special files. For a device connected to Unix, a special file for it is made. The system calls that apply to Ordinary files also apply to Special files. Consequently the Ring device driver in our case is made invisible to the user : the reader simply reads and writes to/from the Ring as if it were accessing a file.

There are two types of Special files which reflect the two major kinds of I/O. They are block and character. The block interface is suitable for devices like disks, tapes which can work with addressable 512 byte-blocks. Those devices that do not belong to this category fall into the character device class. The following system calls used to access the files are :

- fd= open(filename,flag)
- close(fd)
- bytes-read= read(fd, buffer, numbytes)
- write(fd, buffer, numbytes)
- ioctl(fd, command, data)

where fd = file descriptor used to identify the file.

When a new device driver is to be added onto the system,

the following has to be done :

- The Interrupt Vector table in file /sys/conf/l.s has to be modified to enable hardware interrupts to be trapped to the appropriate interrupt routines.
- The Character Device Table (cdevsw) in file /sys/conf/c.c has to be modified to specify the interface routines present for the device driver.
- A Special file representing it has to be created in the directory /dev.

The device driver itself should consist of the five routines which map onto the five standard system calls used to access a file as mentioned above.

In our Ring device driver, the corresponding routines are :

- ringopen()
- ringclose()
- ringwrite()
- ringread()

The ioctl system call is not used. In addition the 2 corresponding interrupt handlers for the Ring hardware interface, rnrint() and rnxint() are placed in this driver.

4.4 Performance Of BBP Implementations

Three design schemes for the BBP, namely Conventional I/O , Read-Ahead I/O and Signalling were implemented and their performances, in terms of throughput, were measured. The differences in the three designs rest on the buffering of data and the rate at which data is passed to the user from the Ring device driver.

Although the use of more than one buffer would smooth out the peaks in basic block arrivals and increase the throughput, the Ring device driver currently has only one buffer for reception. This is because of the decision for protocol simplicity and a lack of available space in the reserved data areas of the kernel.

4.4.1 Conventional I/O

A programmer new to the implementation of a device driver in Unix would naturally implement the BBP according to the idea that lay before him in terms of the Read system call. In order to receive data from the Ring, a user would execute a Read call; the Read call gets blocked until the data arrives from the Ring. This driver has the undesirable effect of ignoring data from the Ring until after a Read call is issued.

Measurements were done and the resulting throughput relative to the speed of the Ring was significantly low. Neverthless, the objective of this simple driver implementation was to familiarize the programmer with the speed disparity between the high speed Ring and the system and to get something

working before embarking on a somewhat more sophisticated driver.

To alleviate the problem of this speed disparity and to provide for data flow, the other two schemes were tried.

4.4.2 Read-Ahead I/O

In this Read-Ahead I/O scheme, any data that arrives from the Ring is accepted unless the driver input buffer is full. The input buffer is always kept as full as possible, so that there is data pending in the driver whenever a Read system call is executed. After the Read call, the input buffer is marked empty and is available for use again.

The major drawback with this scheme is that the user process does not know when any data has arrived from the Ring, hence it has to execute the Read in an infinite loop. Each Read system call executed has to be processed by the system, irrespective of whether there is pending data from the Ring; consequently tying up the system unnecessarily when no data is coming from the Ring at all.

4.4.3 Signalling

This scheme is quite similar to the Read-Ahead I/O scheme, except that this time the user process knows when to execute the Read system call. Each time the Ring device driver successfully receives a block of data, it sends a software interrupt (signal) to the user process which opened this device driver. The signal interrupt handling routine of the affected process would then

execute the Read call to obtain the basic block. However, the signal interrupt handling routine can only be executed when the affected user process is active.

The major drawback with this scheme is that the effect of a software interrupt never takes place immediately. It may occur after only some slight delay if the affected process is currently running or possibly after a considerable delay if the rejected process is suspended and has been swapped out. [10]

4.4.4 Measurements

Measurements were conducted for the above three schemes to determine their relative performances; this is as shown in Fig. 9. The experimental setup consists only of the LSI 11/23 acting as both transmitter and receiver, i.e., the Ring originated and terminated on the same machine (a loop-back mode). Since the reader and transmitter are in the same machine, there existed only a pseudo form of asynchronism. The results using a single buffer each for reception and transmission were tabulated (see Appendix A). The graphs in Fig. 9 show the superior performance of the Read-Ahead I/O scheme over the other two schemes.

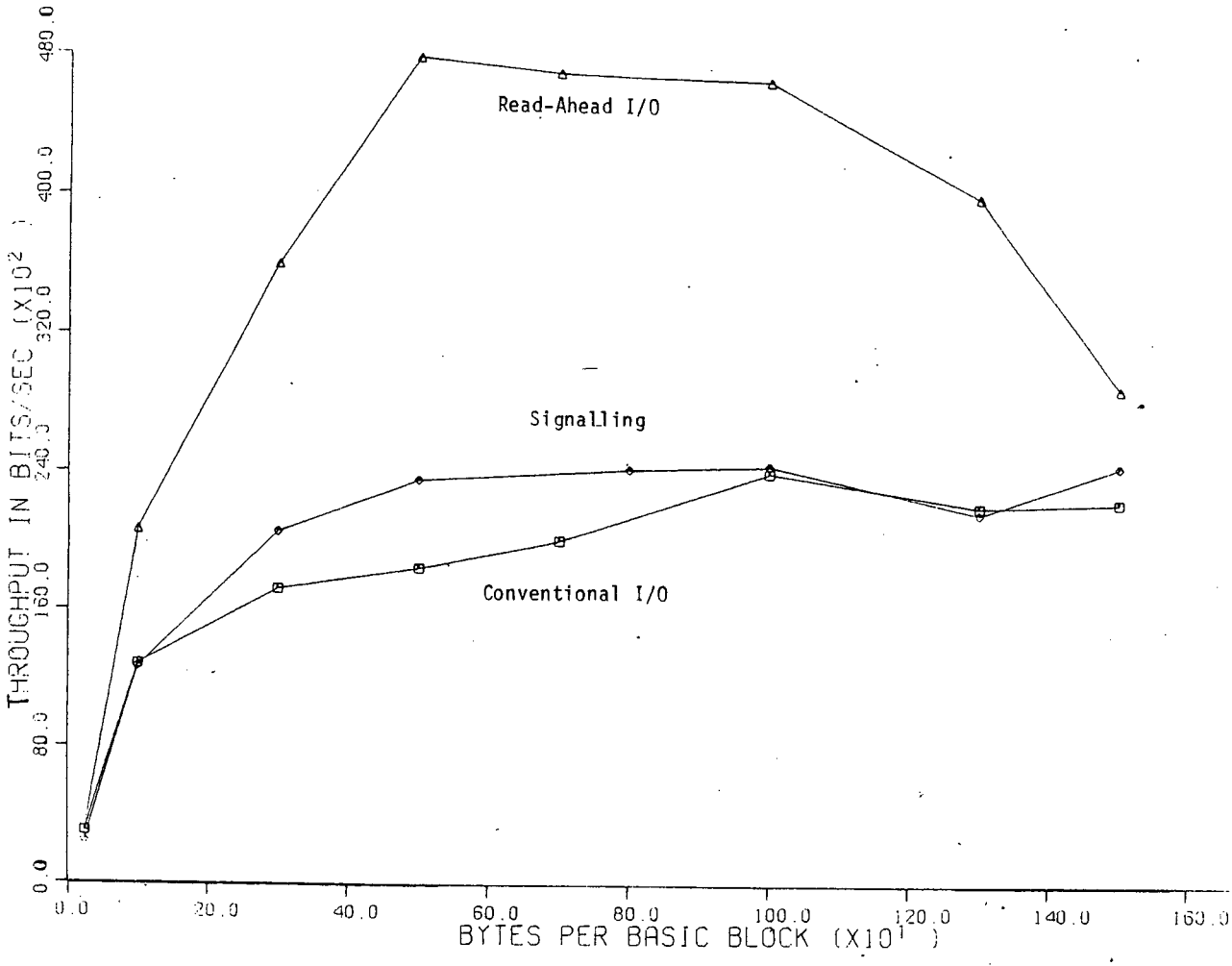

Figure 9 - Performances of the implementations

V. BYTE STREAM PROTOCOL IMPLEMENTATION

This chapter describes the translation of the description of the BSP into a working implementation of it. The BSP program is written in the high level system language C. In developing this system, the following classic design process was closely adhered to :

- Specify the problem
- Specify the data structures
- Define format of data structures
- Specify the algorithms
- Simplify and modularize the program module into smaller modules
- Repeat for each program module

The design process is iterative in nature, that is each step may be repeated to refine the process.

As mentioned earlier in chapter iii, the final conceptual design of the BSP involves the BSP Server interacting only with the User and Reader. Multiple instances of the BSP is supported by the BSP multiplexor in the kernel. The BSP multiplexor is an abstract entity and is realized as the interaction of the two system calls : creatport and relport, with the BBP through the use of the port-procid table (see Fig. 6).

In the following sections, we shall concentrate on the BSP implementation residing at the user level of the Unix system, as

depicted in fig. 10.

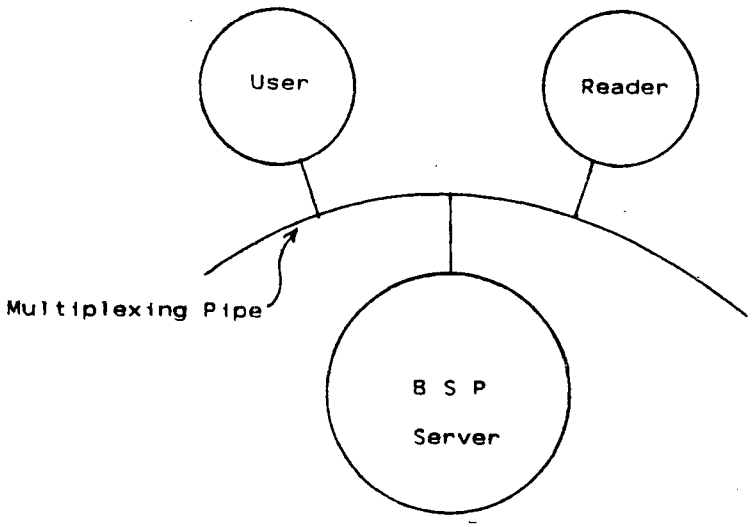

Figure 10 - Server/Client Model of BSP

In essence, most of the implementation is straightforward, but two program modules, namely the Timeout mechanism and the Writer, whose function is to transmit basic blocks, caused the author to mull over their designs. In the case of the Timeout mechanism, difficulties emerged from the need to have two timers evoked concurrently; one each for reception and transmission. In the transmission of basic blocks, the issue of how to integrate the Writer arises: should it be part of the BSP Server or a Worker process itself.

To enlighten and not to bore the reader with details, the

approach taken to present this implementation is to describe only the representations of the following constituent elements which epitomize the BSP design :

- Server
- Reader
- Processing of basic blocks
- Timer
- Writer

These representations embody the program modules and the data structures that are operated on.

5.1 Server Representation

The Server is the main process which coordinates the activities of the Reader and User. It provides a virtual circuit service to the User through the following requests that correspond to the BSP commands :

- REQ_OPEN
- REQ_READ
- REQ_WRITE
- REQ_RESET
- REQ_CLOSE

The REQ_READ and REQ_WRITE are comparable to the "Buffer Ready" events in the BSP state transition diagram. All the requests from the User are blocked pending reply from the

Server. Requests from the Reader take the form of a notification of a basic block arrival, BB_ARRIVE and an unresolvable failure, READ_FAIL in the Ring driver (BBP). The following figure below illustrates the main procedure of the Server.

```
main() {
  USER_NODE *u_node, *get_request();
  int request;

  if ( (init_station()) == ERROR )
        return(ERROR);

  for (;;) {
     u_node= get_request(&request);
     switch( request ) {

      case REQ_OPEN:  open_req(u_node);
                      break;

      case REQ_CLOSE: close_req(u_node);
                      break;

      case REQ_WRITE: write_req(u_node);
                      break;

      case REQ_RESET: reset_req(u_node);
                      break;

      case BB_ARRIVE: frame_arrival(u_node);
                      break;

      case READ_FAIL: p_error(u_node->user_num,READ_FAIL);
                      exit(ERROR);

      default:        p_error(u_node->user_num,BAD_REQUEST);
                      break;
     }
  }
}
```

Figure 11 - Main Procedure of Server

The task of the Server is to initialize the communication link and then go into an infinite loop processing requests. The

initialization involves creating the User and Reader processes, setting up the necessary pipes and initializing the data structures used. Requests are obtained from the multiplexing pipe by a call to the function get request. The task of get_request is to simply identify the type and source of the request. Messages in the multiplexing pipe take on a record structure of type PIPE FRAME as shown below :

```
typedef struct {
  int from;
  int request;
  int count;
  char mesg[];
} PIPE_FRAME;
```

Definitions:

from:	source of request.
request:	request type.
count:	size of data.
mesg:	buffer for data if any.

Figure 12 - Format of PIPE_FRAME

Except for the p error function processing fatal errors, the rest of the functions delegated to perform their appropriate tasks require the pointer u node to access pertinent information for its correct functioning.

The status of the BSP communication link is maintained in three data structures of types SERVER STATE, USER NODE and timenode.

```
Format:                    Definitions:

typedef struct {
  int id;                  process-id of Server.
  USER_NODE *usr;          pointer to USER_NODE.
  TIMERPTR timehead;       pointer to first timer node in list.
  TIMERPTR timetail;       pointer to last timer node in list.
  int user_id;             process-id of User.
  int read_id;             process-id of Reader.
  int bus[2];              the multiplexing pipe.
  int bbfd;                File descriptor for Ring driver.
  int sigflg;              Flag used to mark signal interrupt.
} SERVER_STATE;
```

Figure 13 - Format of SERVER_STATE

The SERVER_STATE as shown above, contains global information that identifies the Client processes, the multiplexing pipe, the Ring driver, and the pointers to the other two data structures which contain information that reflect the actual state of the communication link. An instance of a BSP link contains two buffers, one each for reception and transmission. These buffers are represented in the data structure BUFFER, and are accessed by pointers in the USER_NODE. The composition of the data structures will be elaborated on in the relevant sections. For the present, a sketch on the various data structures and their accessiblilty from the SERVER_STATE is depicted overleaf.

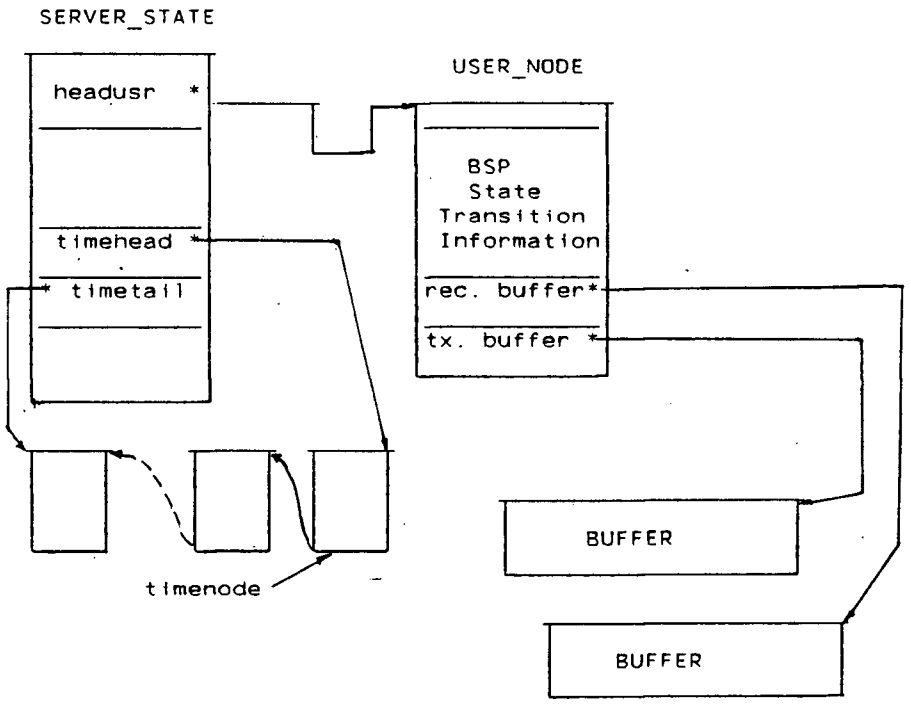

Figure 14 - BSP Data Structures

5.2 Reader Representation

The task of the Reader is to read in the basic blocks arriving from the Ring driver and relay them to the Server to be processed. It consists of two functions, namely the signal handling routine, sighandlr and the main routine, reader proc.

The reader_proc function simply waits for the signal to arrive from the Ring driver by executing the pause call. The system function pause enables a process to relinquish control of the CPU and regain control whenever a signal arrives. Arrival

of the signal causes the invocation of the sighandlr function which then reads in the basic block or the error message from the Ring driver.

On having successfully read in the basic block, the Reader notifies the Server by the BB_ARRIVE request. In the event of a failure, the READ_FAIL request is sent instead. The figure below depicts the Reader and illustrates the use of the pause and signalling concepts.

```
static PIPE_FRAME frame;
static int fd;
static unsigned *int_ptr;
static int len, done;

sighandlr( sign )
   /* signal handling routine */

  signal(sign,sighandlr);  /* reset signal */
  if ((len= read(server.bbfd,frame.mesg,SIZE_MAX_BUF)) > 0 ) {
          /* relay basic block to Server */

    int_ptr= frame.mesg;
    set_frame(&frame,READER,int_ptr->lobyte&0377,BB_ARRIVE,len);
    msg_to_server(&frame,(CNTRLEN+frame.count));
  }
  if ( len == ERROR ) {
          /* return error message to Server */

    set_frame(&frame,READER,0,READ_FAIL,0);
    msg_to_server(&frame,CNTRLEN);
    done= TRUE;
  }
}

reader_proc() {
    done= FALSE;
    while ( !done )    /* while not done */
      pause();         /* relinquish control of CPU */
}
```

Figure 15 - Representation of Reader

5.3 Processing Request BB_ARRIVE

The essence of the BSP lies in the processing of the basic blocks arrived from the Reader. It comprises a group of related secondary functions coordinated by the primary or main function frame_arrival as shown in Fig. 16.

The control flow or algorithm for all the secondary functions is based on the state transition of the BSP accessible from the pointer u_node. The format of the structure USER_NODE pointed to by u_node is shown in Fig. 17. To give a sketch on the processing of basic blocks, the following subsection describes the processing of data transaction commands (as opposed to control commands).

```
frame_arrival( u_node )
 USER_NODE *u_node;
{
 BUFFER *bufptr;

 bufptr= u_node->r_que;
 switch( bufptr->first_cmd ) {

   case RESET_COM:    rec_reset(u_node,bufptr);
                      break;

   case CLOSE_COM:    rec_close(u_node,bufptr);
                      break;

   case OPEN_COM:     rec_open(u_node,bufptr);
                      break;

   case OPENACK_COM:  openack_rec(u_node,bufptr);
                      break;

   default:           data_frame(u_node,bufptr);
                      break;
 }
}
```

Figure 16 - Diagram of function frame_arrival

```
typedef struct unode {
  int user_num;
  int r_state, r_seq, r_tries, r_max_size, r_port;
  int x_state, x_seq, x_tries, x_max_size, x_port, x_addr;
  int rbufstat;
  int read_pending;
  int writ_pending;
  int proc_status;
  BUFFER *r_que, *x_que;
} USER_NODE;
```

Figure 17 - Format of USER_NODE

Description of fields in USER NODE

- user_num is the port number.
- r_state is the state of the BSP, which could either take one of the following states:

  a) E - Expecting acknowledgement
  b) N - Not expecting an acknowledgement
  c) I - Idle State

- r_seq is the reception sequence number.
- r_tries is a variable to keep track of the number of attempts made for an action.
- r_max_size keeps track of the size of the buffer to hold the basic block for the User.
- x_state, x_seq, x_tries and x_max_size similarly applies to the transmission.
- x_port refers to the destination port this port is communicating with.
- x_addr refers to the destination station address.
- rbufstat refers to the status of the reception buffer; denoting whether it is filled or empty.
- read_pending is a boolean value denoting whether a READ_REQ was made by the User.
- writ_pending is a boolean value denoting whether a WRITE_REQ was made by the User.

- proc_status refers to the status of the BSP during the control transaction phase of the communication.
- *r_que and *x_que refer to the buffers used to contain the basic blocks during reception and transmission.

5.3.1 Data Transactions

The function data frame processes data transaction commands and its mode of operation corresponds to activities of the BSP state transition model. Essentially the operation is to process the commands in the basic block stored in the reception buffer and update the basic block in the transmission buffer. If data has arrived and the processing is correct, the data is transferred to the User, provided the request REQ_READ was previously sent; otherwise a flag is set marking the reception buffer as full --providing mono-buffering--. The mode of operation is as follows:

- If the BSP status is in the data transaction phase, then continue, otherwise return.
- Initialize the contents of the basic block in the transmission buffer.
- Process the reception and transmission commands by delegating the tasks to the subordinate functions. The values stored in tosend1 and tosend2 returned from the respective calls to the functions processing the reception and transmission commands signify whether the transmission buffer is to be sent or an error has occurred.
- Set the timer and transmit the basic block stored in the transmission buffer according to the values returned in tosend1 and tosend2.

The algorithm is shown in the following figure.

```
data_frame( u_node, bufptr )
 USER_NODE *u_node;
 BUFFER *bufptr;
{
 int tosend1, tosend2, timeout();

 if ( ! (u_node->proc_status & NORMALTX )
      return;
 tosend1= tosend2= FALSE;
 set_station_port(u_node,u_node->x_que);
 switch( get_cmd(bufptr->first_cmd)&0360 ) {

   case RDY_COM:     stop_timer(u_node->user_num,W);
                     tosend1= rdy_cmd(unode,bufptr);
                     break;

   case NOTRDY_COM:  stop_timer(u_node->user_num,W);
                     tosend1= nonrdy_cmd(u_node,bufptr);
                     break;

   case NULL_COM:    break;

   default:          p_error(u_node->user_num,BAD_COMMAND);
                     break;
 }

 switch( get_cmd(bufptr->xmit_cmd)&0360 ) {

   case RDY_COM:     stop_timer(u_node->user_num,R);
                     tosend2= data_cmd(unode,bufptr);
                     break;

   case NOTRDY_COM:  stop_timer(u_node->user_num,R);
                     tosend2= nondata_cmd(u_node,bufptr);
                     break;

   case NULL_COM:    break;

   default:          p_error(u_node->user_num,BAD_COMMAND);
                     break;
 }
  if ((tosend1 != ERROR) && (tosend2 != ERROR)) {
    if ( tosend1 )
        set_timer(u_node->user_num,W,ALARM_TIME,timeout);
    if ( tosend2 )
        set_timer(u_node->user_num,R,ALARM_TIME,timeout);
    if ( tosend1 || tosend2 )
        sendtoring(u_node);
  }
}
```

Figure 18 - Diagram of function data_frame

5.3.2 Format Of Buffer

The reader would of course be right when he assumes that the format of the BUFFER corresponds to the format of the basic block. The format, in addition, maps onto the interface block used between the Ring driver and the BSP. The format is as shown below.

```
typedef struct buf {
  int len_of_blk;
  int message[];
  char ptype,station;
  char rec_flg, first_cmd;
  char xmit_flg, xmit_cmd;
  char msg[SIZE_MAX_BUF];
} BUFFER;
```

Definitions :

len_of_blk:	length of interface block to be sent.
message:	a dummy; used as a mask.
ptype:	protocol type (see interface block in BBP).
station:	destination station address.

The rest of the definitions correspond to the format of the pac used in the BSP.

Note: In the LSI 11/23 architecture, the low order and high order bytes are represented in the reversed order.

5.4 Timer Design

It is characteristic of protocol implementations to incorporate the Timeout mechanism as a process which sleeps (i.e. relinquishes control of the CPU) for the Timeout interval and on awakening (i.e. when the time interval expires), informs the Server of this Timeout event. The Server would then invoke the appropriate function, namely the Timeout routine, to handle

this event. The ease in which this could be done depends on the facilities provided by the operating system.

Unix provides two functions, namely the pause and the alarm which enable a primitive form of the Timeout mechanism to be realized. By invoking the pause function, a process is caused to sleep and a call to the alarm function with a parameter specifying the time interval allows a process to be interrupted after the time interval has expired. This alarm feature employs the concept of signaling as described previously. Thus a simple timer is easily implemented by having a process set the alarm before sleeping.

However, the alarm feature of Unix does not queue calls. There is at most one outstanding alarm and a subsequent call to it before the expiry of the current alarm supersedes the current one. The BSP requirement for two timers apparently involves two extra processes acting as timers. The implication of such a timer implementation manifests itself in the inefficiency and awkwardness of the Server/Client interaction. The basis of its awkwardness and inefficiency stems from :

- The increase in the total number of processes required for the Server to multiplex inevitably degrade its performance.
- Determination of the identity of the timers, i.e., whether it is for reception or transmission.
- A pipe has to be linked from the Server to each timer process to provide a two way communication necessary for

the setting and killing, i.e., revocation, of the timers. A protocol has to be established between the Server and the timer process in order to distinguish the type of request wanted by the Server : the setting and killing of timers. Setting a timer involves having the Server transmit the Timeout interval to the timer process.

- Additional information has to be kept in state tables to ascertain which Timeout interval has expired (Normal or Idle Handshake) and the appropriate Timeout routines to be invoked.

5.4.1 General Timer

To overcome such inadequacies and awkwardness, a general Timeout mechanism is designed. It obviates the need for two extra timer processes and provides a clean interface for the setting and stopping of timers. Furthermore, two or more timers can be set without any significant increase in cost, in terms of space and computation requirements. To set a timer, the function set timer is called, passing to it the Timeout interval and the Timeout routine. In order to distinguish the identity of the timer, whether it is for reception or transmission, the parameter mode is also defined. To kill a timer, the function stop timer is invoked, passing to it the identity of the timer to be killed.

The principle behind this general Timeout mechanism lies in

the use of a queue of timer nodes , i.e., a data structure of type timenode, to represent the list of timers being set. This queue is ordered according to the expiry time of each timer (see Fig. 20). The alarm function is then invoked for the first node of the queue. On the expiry of the timer, the function alrm handlr is invoked and its task is to update the queue by removing the first node; call the appropriate Timeout routine and finally invoke the alarm again for the new first node in the queue; provided there is one. The expiry time for a timer is computed as the sum of the time the timer was invoked and the Timeout interval. The information contained in the timer node includes the identity of the timer, the Timeout interval, and the Timeout routine. The format of the data structure timenode is as shown below.

```
struct timenode {
  int userno;
  int mode;
  int interval;
  long int expiry;
  int (*handlr)();
  struct timenode *next;
}
```

Figure 19 - Format of structure timenode

Definitions:

userno:	port number.
mode:	defines type of timer.
interval:	time interval to be set.
handlr():	address of Timeout routine.
next:	pointer to next timenode.

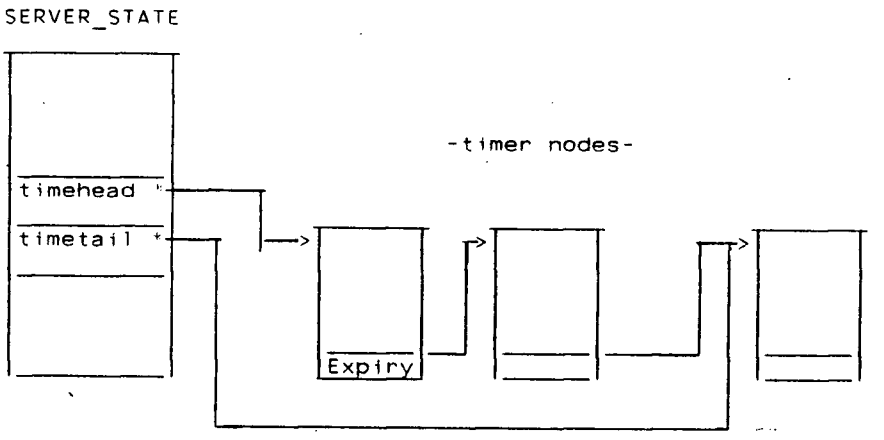

Figure 20 - Representation of Timers

5.4.2 Mutual Exclusion Of Shared Queue

Interrupts precipitated by the alarm signal arrival may cause inconsistency in the queue if both the alrm_handlr and the routine being interrupted update the links of the queue.

Indivisibility of the update operation is provided by always killing the current alarm, if there were any, and resetting the alarm again once the updating is done.

5.4.3 Granularity Of Clock

The granularity or resolution of the clock available to processes by the function time is one second and this has an undesirable impact on the functioning of the Timeout mechanism. First, the recommended Normal Timeout interval of one second is not feasible and secondly, it would not be unusual to find the expiry time for timers set at different times to be the same, causing Timeout events to occur in succession. In brief, the accuracy of the Timeout mechanism could only be as precise as the resolution of the clock.

5.5 Writer Design

The Writer is a term used to denote the function which handles the transmission of the basic block. Its specific task is to execute a write to the Ring driver and process the value returned from the call. Errors resulting from the write are stored in the global variable errno by the system. Only three error types from the Ring driver permit the write attempt to be repeated (see Fig. 21); the rest are unresolvable and forces the Server to abandon the communication link. The function sendtoring representing the Writer is as shown below.

```
sendtoring( u_node )
  USER_NODE *u_node;;
{
  BUFFER *bptr;
  int retries, done, len;

  bptr= u_node->x_que;
  len= bptr->len_of_blk;
  retries= MAX_TRIES;
  done= FALSE;

  do {
    if ( (write(server.bbfd,bptr->message,len)) == ERROR ) {
       /* To process the error message */

      switch( errno ) {
       case REJECT:
       case IGNORED:
       case BAD_BLOCK:  p_error(u_node->user_num,errno);
                        return(ERROR);
       case TXMITERR:
       case TIMEOUT:
       case COUNT_OUT:  if ( (--retries) < 0 ) {
                          p_error(u_node->user_num,errno);
                          return(ERROR);
                        }
                        break;

       default:         perror("sendtoring");
                        return(ERROR);
      }
  }  while ( !done );
  return(FALSE);
}
```

Figure 21 - Representation of Writer

It should be clear by now that the Writer does not reside in a process; acting as a slave or Worker process to the Server. The issue of whether the Writer should be represented as a Worker process or as part of the Server itself is influenced by the conflicting needs of overcoming the inherent deadlock problem and the desire for an elegant and conceptually efficient

design if the Worker is represented as an independent process.

Having the Writer as a Worker process requires another pipe between it and the Server to facilitate a two way communication. This pipe will be used by the Server to pass the basic block for it to be transmitted by the Writer. On having successfully transmitted a basic block, the Writer would inform the Server of this event through the multiplexing pipe. If at any time both pipes are filled and both Server and Writer are blocked trying to write to one another, then the classical deadlock problem ensues.

The deadlock problem could be overcome with added complexity to the design of the Server; however the goal of simplicity is adopted here by making the Writer part of the Server process. Although one may argue that the gains implicit in the simplicity of the Server design does not offset the elegance and apparent efficiency gained by having the Writer as a Worker process, we contend here that the apparent efficiency is debatable unless performance measurements are actually conducted.

Increasing the number of Client processes by one would increase the number of instances a Worker process has to wait for the use of the multiplexing pipe. This is because the buffer size for a pipe is only 4096 bytes and the sizes of the basic blocks travelling within the multiplexing pipe is often 2048 bytes in length --the default size of the basic block-- and copying of basic blocks among the Server, Reader, User and the Writer occurs quite frequently. Compounding this deterioration

is that pipes are essentially files and communication using pipes are therefore inherently slow.

On the other hand, having the Writer as part of the Server simply means that transmitting the basic block to the Ring driver is done directly and the Server is blocked pending the completion of the system call write. The interval of time the Server is suspended is highly dependent on the scheduling policies of the system and not on the high speed Ring hardware interface. Therefore one may expect the performance of the BSP to deteriorate when there are many users logged on.

VI. CONCLUSIONS

Given the time constraint, an inadequate operating system, a slow LSI 11/23 processor, and the size of the project itself, a simple and "working" system is finally achieved.

6.1 Reflections On The Implementations

The implementation of the BBP took approximately three months to complete. Two months were dedicated to the actual implementation and one month to the understanding of the Unix kernel and other miscellaneous details. These three months stated, however spanned over a four month period as the LSI 11/23 machine was not always available for system development. The two months of actual implementation time can be attributed to the testing phase and the recompilation time, rather than the actual coding. This is because the code is only about 250 lines in length. The final size of the BBP code occupies 5068 bytes; decomposed into 934 bytes for the text segment and 4134 bytes for the data segment.

The BSP implementation took about three months; presently, the testing of the protocol in the loop-back fashion indicates it is working properly. This testing is facilitated by extending the design by having two User processes instead of one in the Server/Client model of the BSP (see fig. 10). The size of the BSP code occupies 16036 bytes, decomposed into 13156 bytes of text and 2880 bytes of data.

Despite the inadequacy of the IPC of UNIX, the code for the BSP is nevertheless modular and well structured. It would therefore be easier to maintain, and in the future, would alleviate the task of modifications to improve or debug this evolving software.

Nothing much was said about the performance of the BSP. This is partly due to a lack of a suitable configuration for testing. However, copying of basic blocks among the Server, Reader and the User processes via the multiplexing pipe undoubtedly indicates the kind of low performance expected.

6.2 Suggestions For Possible Improvements

The recent testing involving both the LSI 11/23 and the TI 990 on the network suffered some trivial inconsistencies which were not previously perceived. Such inconsistencies must be ironed out. In addition, the BSP multiplexor has been implemented but has yet to be integrated with the rest of the code. Hopefully, these apparently minor tasks could be eradicated without much delay.

Further rigorous testing of the set of protocols involving a more comprehensive set of simulations of errors is also desirable to promote confidence and faith in the use of this evolved product.

BIBLIOGRAPHY

1. Blair, G.S. et al, "A Practical Extension to UNIX for Interprocess Communication", Vol. 13, 1982, pp. 45-58.

2. Clark, D.D., "Modularity and Efficiency in Protocol Implementation", RFC: 817, MIT Lab. for Computer Science, Computer Systems and Communications Group, July 1982.

3. Clark, D.D. et al, "An Introduction to Local Area Networks", Proc. IEEE, Nov. 1978, pp. 1497-1517.

4. Collinson, R.P., "The Cambridge Ring and UNIX", Software-Practice and Experience, Vol. 12, 1982, pp. 24-39.

5. Dallas, I.N., "Transport Service Byte Stream Protocol", Report No. 1, Revision 3, UKC Computing Lab., The University of Canterbury, Kent, Canterbury, August 1981.

6. Haverty, J.F. and Rettburg, R.D., "Interprocess communication for a server in UNIX", Proc. IEEE Computer Society International Conf. on Computer Communications Networks, September 1978, pp. 312-315.

7. ISO/TC97/SC16, "Data Processing - Open Systems Interconnection - Basic Reference Model", Document N537 Revised, December 1980.

8. Johnson, M.A., "Ring byte stream protocol specification", Systems Research Group Paper, University of Cambridge Computing Laboratory, April 1980.

9. Kernighan, B.W. and Ritchie, D.M., "The C Programming Language", Prentice-Hall, 1978.

10. Lions, J., "A Commentary on the UNIX operating system", Dept. of Computer Science, University of New South Wales, 1977.

11. Needham, R.M. and Herbert, A.J., "The Cambridge Distributed Computing System", Addison-Wesley, 1982, pp. 24-39.

12. Rashid, R.F., "An interprocess communication facility for UNIX", Technical Report CMU-CS-80-124, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, revised June 1980.

13. Ritchie, D.M. and Thompson, K., "The UNIX time-sharing system", Bell Sys. Tech. J., Vol. 57, No. 6, July 1978, pp. 1905-1929.

14. Thompson, K., "UNIX time-sharing system: UNIX Implementation", Bell Sys. Tech. J., Vol. 57, No. 6, July 1978, pp. 1931-1946.

15. Wilkes, M.V. and Wheeler, D.J., "The Cambridge Digital Communication Ring", Proc. Local Area Communication Network Symposium, Mitre Corp., May 1979.

APPENDIX A - TABULATIONS OF PERFORMANCES OF BBP

i. Convention I/O

Number of bytes transffered = 48075 bytes
1 Clock Tick in UNIX = 1/60 sec.

Bytes per Basic Blk.	Clock Ticks	Writes issued	Writes successful	Throughput bits/sec.
25	7491	1942	1923	3080
100	1795	572	481	12855
300	1341	246	161	17208
500	1255	171	97	18387
700	1153	110	69	20013
1000	964	74	49	23937
1300	1049	51	37	21998
1500	1036	51	33	22274

ii. Read-Ahead I/O

Bytes per Basic Blk.	Clock Ticks	Writes issued	Writes successful	Throughput bits/sec.
25	7503	1931	1923	3075
100	1117	484	481	20659
300	640	162	161	36056
500	483	97	97	47776
700	492	69	69	46902
1000	497	49	49	46431
1300	579	37	37	39855
1500	800	33	33	28845

iii. Signalling

Number of bytes transffered = 17868 bytes
1 Clock Tick in UNIX = 1/60 sec.

Bytes per Basic Blk.	Clock Ticks	Writes issued	Writes successful	Throughput bits/sec.
25	3450	719	715	2486
100	677	181	179	12669
200	507	92	90	16916
300	418	60	60	20518
500	365	36	36	23498
800	355	23	23	24160
1000	352	18	18	24365
1300	397	14	14	21604
1500	352	12	12	24365