Technical Report

UCAM-CL-TR-46
ISSN 1476-2986

Number 46

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Intelligent network interfaces

Nicholas Henry Garnett

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted May 1983 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

Preface

I wish to thank my supervisor, Professor R. M. Needham, for his help and encouragement. I would also like to acknowledge the members of the Systems Research Group for providing a pleasant environment in which to work.

During my research I was supported by a grant from the Science and Engineering Research Council; for which I am grateful.

Except where otherwise stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

I hereby declare that this dissertation is not substantially the same as any I have submitted for a degree or diploma, or any other qualification at any other university. I further state that no part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

Summary

Local Area Networks are now an accepted part of computing research. The technology of the network itself and the hardware to interface it to a computer is standard and in the cases of networks like ETHERNET and the Cambridge Ring is commercially available. The next level up from the hardware is the software interface between the Host computer and the network. This dissertation is concerned with one specific type of interface where the Host is not itself directly connected to the Network, but must access it via a second Network Interface Processor (NIP).

The dissertation begins by describing the design and implementation of two low level interfaces for the Cambridge Ring. The first of these, the Type 2, is machine independant and although based on a simple processor offers some sophisticated facilities to its Host. The second, SPECTRUM, is not so sophisticated, but is customized to interface to just one operating system. The difference between these two approaches is discussed.

We go on to introduce the High Level Interface, which removes all protocol and network related processing from the Host machine. This can benefit both the protocol implementation, by reducing system overheads, and the Host Operating System, by freeing CPU time for other tasks. This is particularly true in the case of time-shared machines which rely on the network for terminal connections. The design and implementation of such an interface are described.

The dissertation concludes by considering the possible roles of the NIP in the areas of security, protection, and reliability. Some thoughts are also given on the design of protocols which exploit the features of a NIP.

Chapter 1

Introduction

There has been, over the past few years, an increasing interest in computer networking. Indeed it has become not only an accepted part of computing systems research but now forms the major part of it. Computer networks may be divided into three classes: Wide Area Networks, Local Area Networks, and Multiprocessor systems. Wide Area Networks are the oldest of the three, finding examples in ARPANET [Heart70] and TYMNET [Tymes71]. Multiprocessors also have a long history, notable examples are: the CDC 6600 [Thornton70], HYDRA [Wulf74] and Cm* [Fuller78, Jones79, Ousterhout79]. Local Area Networks are a relatively recent development and combine features of both these as well as having features uniquely their own.

1.1 Local Area Communication Networks

A Local Area Network (LAN) is one that connect computers, and other devices, within a radius of less than 5Km. This means that all the machines must be in the same building, or in closely adjacent ones. The closeness of this connection gives the LAN several important features. The first is that it is fast, with a point-to-point bandwidth measured in Megabits per second. The second is that its error rate is low, a figure of one bit error in 10^{11} is typical. Local area networks may be divided into three groups based on their topology.

The first group is that of CSMA (Carrier Sense Multi-Access) bus networks. These consist of an un-rooted tree of cables forming a passive shared communication medium or ether. Stations are connected to this with taps at convenient points. Sources transmit data, encapsulated in addressed packets of several Kbits, only when they detect that the ether is idle. Since there is no central control it is possible that two sources will attempt to transmit at the same time. Collision detection, backoff and retry algorithms are implemented in the transmitting computers to handle this. Any packet transmitted on the ether propagates to all receivers; if a receiver recognises its address it copies the data into local memory.

The best known example of this type of network is Ethernet [Metcalfe76].

The second group is that of Ring based networks. Here the stations are all connected sequentially to a looped medium. Since there is a distinct order to the nodes a distributed algorithm can be developed to share the medium without the collision/retry algorithms needed in a CSMA network. A packet from a transmitter passes in one direction around the ring until it is recognised by a receiver, which copies the data. The packet could be deleted here, but most ring networks exploit the topology to return an acknowledgement to the transmitter. Examples of ring systems are the Cambridge Ring [Wilkes79a], PRIMENET [Nelson78], and DCS [Farber75].

The final group of networks are those that do not fit into either of the above topologies, many of these apply wide area network technology to local areas and are either randomly configured or star shaped.

1.2 Network Interfaces

When interfacing a network to a Host computer there are many options open to the designer. The mechanism chosen is often dictated by the requirements of the Host computer, its operating system, or the characteristics of the underlying network.

At the most primitive level the network hardware can be connected directly to the Host and all aspects of network driving handled by the Host Operating System or client software. The suitability of this depends on the relative speeds of the network and Host processor, and the intended use of the network. At Cambridge examples of such interfaces may be found on the many Z80s [Ody81a], a PDP11/45 [Gibbons80a], and a VAX11/750 [Collinson82]. In the first case the Z80 does little more than respond to requests sent via the Ring, so the primitive interface is adequate. In the other two cases the Ring is used infrequently enough for it not to have a detrimental effect. The disadvantage of this type of interface is that the real-time demands of the network may stop all other work during a network transaction.

One way to acquire faster processing, and therefore reduce the real time spent, is to move the network driving software into the processor's microcode. This can have the double advantage of speeding up the network transfers and reducing the load on the Host. The network is still being serviced, however, at the expense of Host programs. For example, the Alto Ethernet interface [Thacker79], uses 16% of the machine during data transfer, and can rise to 20% in the worst case. At Cambridge the CAP

computer also has a micro-coded interface to the Ring. This approach assumes that the Host computer has an easily accessible micro-store, which is often only true if you have designed and built the machine yourself.

A significant improvement can be obtained if a modest amount of hardware is installed in the interface. The data transfer phase of the transaction can be handled automatically and access made via DMA. Since the Host processor is unlikely to use every memory cycle such an interface can have little or no effect on Host performance. Another important task that may be performed by hardware is the calculation of the checksum which most network protocols need. On a processor of modest performance this can take almost as long as the data transfer.

Local Area Networks share some characteristics with devices like discs: there may be a long delay before data is available, but when it is it comes at high speed and has associated real-time requirements. These attributes of discs are catered for by providing an intelligent, or semi-intelligent, controller. It makes sense, therefore, to use a similar approach to network access.

An interesting example of a semi-intelligent network interface may be found in the Local Network Interface (LNI) to the Distributed Computing System [Mockapetris77]. DCS is a ring network in which messages are addressed to processes rather than stations. This means that the LNI must contain an associative name table of all the process names in its Host. The destination address of any message seen passing on the ring is passed through this table and if it matches the data is copied. This is further complicated by the presence of a mask field in both the name table and the message; this allows messages to be broadcast to groups of processes selected by fields in the address. The LNI is a custom LSI chip that is controlled by an alterable Programmed Logic Array, it can therefore be considered a specialised micro-coded computer.

This dissertation is concerned with the class of truly intelligent network interfaces.

1.3 Intelligent Network Interfaces

An intelligent network interface is one that contains an easily programmed computer of some form. The concept of a Front-End network processor is not new and has been employed in Wide Area Networks for some time. Its role in these networks is usually to carry out message routing or maintain virtual circuits. The ideas have also been extended to Local Area Networks [Stack81]. These Front-End processors usually need to be substantial mini-computers, and are of comparable power to the type of machine we expect to comprise most of the Host processors on a LAN. The development of single chip microcomputers and their subsequent increase in performance now makes it possible to build a powerful processor that is both small (one PCB) and cheap (<£500). It is therefore feasible to interface modest mini-computers, and even the larger micro-computers, to a network via their own Network Interface Processor (NIP).

The literature contains many examples of this: FordNet [Biba79], SWAN [Sommer81], CNET [West78], MITRENET [Hopkins81], WELNET [Mark81] and [Carpenter78]. These are all CSMA networks that use the NIP to implement the backoff/retry algorithm the network requires. All network access, including reception and transmission, is performed by the microprocessor with little hardware assistance, so data rates are limited to approximately 1 Mbit/second. Some of these are also limited by the Host/NIP connection, which is often no more spectacular than a serial line. A ring network similar to DCS but using a microprocessor in place of the LNI is described in [Lee78]. Most of these implement only the lowest protocol level: datagram or packet. Some do take advantage of the presence of the NIP to implement flow control and error detection using acknowledgement packets. This is on a node to node basis, not client to client, so it does not constitute a virtual circuit protocol. It was mentioned in the FordNet paper that they intended to implement higher level protocols in the NIP, but there appears to be no published evidence of this.

In most of the above cases the micro-processor node is an integral part of the network, and no other access method is supplied. In other cases the NIP has been used to make the network appear to be some other, more mundane device. An example of this is the Terminal Interface being developed at University College, London for the Cambridge Ring [Rubenstein81]. This is a Z80 based interface that supports a single character terminal protocol on the ring and presents the Host with an interface similar to a terminal multiplexor. Elsewhere on the ring another

Z80 acts as a terminal concentrator and allows a terminal to be connected to any of several Host computers.

Of more interest is the Network Interface Processor that does not hide its true nature behind a pseudo-device interface but supplies explicit network access primitives to the Host. The prime requirement of such a device is that it does its job more efficiently than the Host could itself. Equally important is that the Host is not involved in as much work to use the NIP as it would be in driving the network directly. The first requirement may be met by using a fast processor or giving it special hardware support. The second requires that the NIP have direct access to the Host's memory so it can fetch both data and commands as it requires. Combining these two it is possible to develop hardware that can transfer data from network to Host and vice versa without the intervention of either processor.

The cost of a NIP is an important factor and can be assessed in relation to several criteria. The first, and most important, criterion is its cost compared with the Host machine it is to interface. If the cost of the NIP is more than that of the Host, or is a sizeable fraction of it, it is economically unfeasible to use it unless its advantages are great. Another comparison to be made is between the cost of the NIP and that of a simpler interface, for example a direct connection between the Host and the Network. Clearly the NIP is the more expensive, but again we can only make the comparison in the light of the comparative performance of the interfaces.

The duties of a NIP need not stop at the low level protocols, but may be extended to both more flexible implementations of these and the implementation of higher level protocols. Beyond this the NIP may take on some of the systems and management duties associated with the network. Pushing even further we come back to the case where the presence of a NIP is an integral part of the network access logic. But instead of being a limitation it now supplies all the high level functions a Host requires for communication, freeing it from performing them itself. In addition the NIP can provide a consistent interface and play a part in protection, authentication and security.

1.4 Overview

Chapter 2 describes the Cambridge environment against which much of the work described in this dissertation was done. The informed reader can ignore this chapter.

Chapter 3 describes the Type 2, a high performance NIP for the Cambridge Ring. This only implements the lowest level protocol: Basic Block Protocol, but supplies an extremely flexible interface to this.

Chapter 4 describes the MACE, a machine that has been put to similar use, but is based on a slower processor. The initial interface program for this machine, SPECTRUM, was less flexible than that for the Type 2, but is closer in specification to the Host's requirements.

Chapter 5 presents the concept of a High Level Interface that provides its Host with more powerful protocols and services. Chapters 6 and 7 describe the design and implementation of a High Level Interface on the MACE, called SuperMACE, and show that this approach results in an improved system performance.

Chapter 8 covers further aspects of a High Level Interface including some thoughts on possible hardware improvements and its role in security and authentication.

Chapter 9 examines the influence a NIP may have on the design of protocols, and goes on to describe the architecture of a system that provides a high degree of protection between machines.

Chapter 10 concludes this dissertation.

1.5 Acknowledgements

All the NIP programs described in this dissertation are my own work. The hardware of the Type 2 was designed and implemented by J. J. Gibbons and M. A. Johnson, the MACE was designed and implemented by M. Muller. The Host software for the LSI4 systems and for the SPECTRUM MACE program were written by B. J. Knight. That for the SuperMACE was written by the author.

Chapter 2

Background

This Chapter describes some of the background to the work presented in this dissertation. This material is presented here in preference to introducing it when the subject arises to dispose of it as soon as possible. Those familiar with the following items may skip them in the safe knowledge that they are not missing anything original.

2.1 The Cambridge Ring

The Cambridge Ring is a high performance Local Area Network (LAN) developed at the Cambridge University Computer Laboratory. The ring consists of a loop of two twisted pairs interrupted by repeaters that regenerate the signal. Data is carried in several packets that circulate in a fixed pattern. The number of packets is determined by the length of the ring and its clock rate. Since the ring is unlikely to be an exact number of packets long, the pattern consists of an integral number of packets plus a gap, which is shorter than a packet.

Packets are 38 bits long and consist of an 8 bit source address, an 8 bit destination address, 16 bits of data and six bits for framing, control and error detection. This is shown in figure 2.1.

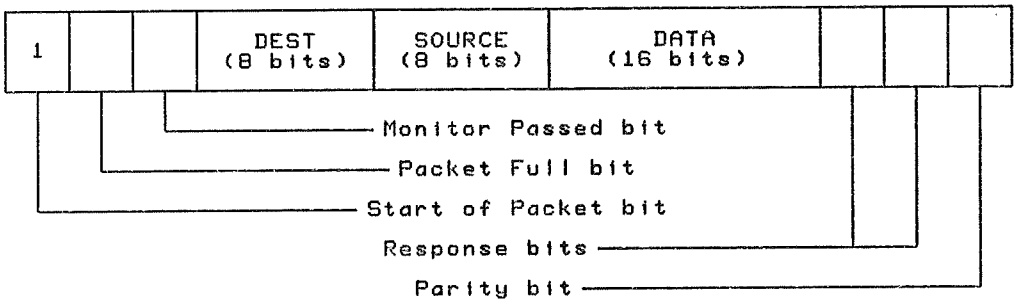


Figure 2.1 Format of a Ring Packet

Computers are connected to the ring via a station, which is in turn connected via a repeater. Each station has an unique address in the range 1 to 254, addresses 0 and 255 are interpreted specially. To the

connected computer the station supplies a full duplex interface, so transmission and reception may be viewed as completely independent devices. This is particularly useful when developing ring driving software since not only may the two halves be tested independently, but may also be tested jointly by transmitting to oneself.

Of all the stations one, the monitor station, is distinguished. This is responsible for initialising the framing structure of the ring at startup and for re-initialising it when it is lost.

2.1.1 Transmission

The primitive transmission operation provided by the station is to send one 16 bit data packet to a selected destination. This is achieved by waiting until an empty packet passes the repeater, claiming this by setting the full bit, and loading into it the destination address, source address (itself) and the 16 bits of data. The transmission logic then waits until the packet has been all the way around the ring. When the packet returns it is compared with the packet sent as an error detection measure, and the response bits are made available to the Host.

There are two response bits, allowing for four responses in total:

Busy The destination recognised its address, but was not ready to receive another packet.

Unselected The destination does not wish to receive packets from our station.

Accepted The destination received the packet.

Ignored No station recognised its address: it either does not exist or is switched off.

Once a transmission has been performed the station is not allowed to refill the empty packet immediately, but must allow it to pass on. This simple mechanism prevents any one station from hogging the ring bandwidth and gives each station an equal chance of claiming packets, even when the ring is heavily used.

An elementary flow control mechanism is supplied to suppress extraneous transmissions. If the response to a transmission is busy the station inhibits the transmission of the next packet for a time. After the first such rejection this is a period of two ring revolutions, and on the second

and subsequent times is sixteen revolutions.

2.1.2 Reception

The primitive reception operation is to receive a single packet from the selected source. The source is controlled by the Source Acceptable Register (SAR). If this is set to a value in the range 1 to 254 then only packets from the station of that address will be accepted; transmissions from elsewhere will be rejected unselected. If the SAR is set to zero no packets at all will be accepted, and if it is set to 255 packets from any station will be accepted, in which case the actual source may be read from another register in the station.

2.2 Protocols Used on the Cambridge Ring

Ring packets as they stand are unsuitable for most applications. There are three reasons for this. First, the overheads of deciding what to do with each packet individually would be prohibitively large, particularly in a program driven interface. Since the hardware supplies a means of selecting packets from one source only, this may be used to allow an entire sequence of packets to be received with greatly reduced overheads.

Second, the hardware only supplies addressing to the grain of a single station. A typical computer will have several processes in it that may be communicating with processes in other machines. When data arrives it is necessary to associate it with a particular process. The only form of identification supplied by the hardware is the source address, which may not uniquely identify it. Some form of extended addressing is required. Installing this in each packet would not only significantly reduce the amount of data that could be transferred, it would not give a large enough range of addresses to be of much help. However, if the data is blocked together an extended address of whatever size was felt necessary could be included in the block format.

Third, without incurring the same penalties described above, it would not be possible to detect lost or corrupt packets without blocking.

While a strong case in favour of a primitive block protocol has been put, it should be noted that there are applications where a single packet protocol has its advantages. One such is the transmission of digitized voice communications across the ring, where the overheads of constructing blocks intrude on the real-time requirements of the application [Leslie81]. A few lost packets will go un-noticed, and if they are, error recovery can

be dealt with by higher level protocols. Another single packet protocol is the terminal protocol implemented at University College London [Rubinstein81] where a virtual circuit between a terminal and a Host computer is implemented by two streams of packets. The implementors rely on the low error rate of the ring to eliminate the need for any error recovery.

2.2.1 Basic Block Protocol

Basic Block Protocol [Walker78] provides the primitive block protocol demanded above. A basic block consists of a header packet, a route packet, some data packets and a checksum packet. Its structure is shown in figure 2.2.

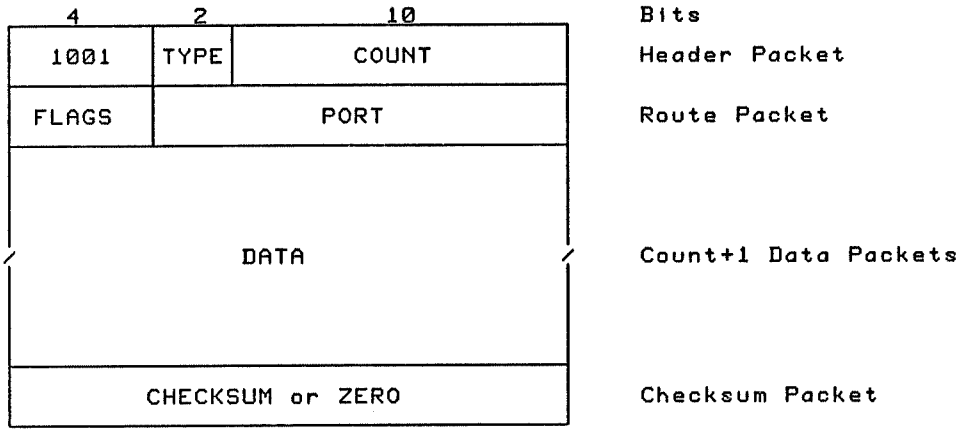


Figure 2.2 Format of a Basic Block

The header packet contains a 4 bit pattern as a weak form of identification, a 2 bit type field and a 10 bit count field, allowing from 1 to 1024 packets of data (the count is really in the range 0 to 1023).

The route packet contains a 12 bit port number, which is used to direct the packet to the right process in the computer. The computer has complete freedom to allocate port numbers as it wishes. The flags field has three of its four bits allocated. The top bit indicates that the block is intended for the computer's ring interface and not for the computer itself; this allows remote debugging and loading of machines and will be covered in more detail in later chapters. The lower two bits of the flags field are used to specify one of four sub-ports, and is used for communication via a ring-ring bridge. The remaining bit is unassigned.

Following the route packet are the data packets. Owing to the way the count is encoded there must be at least one of these.

The content of the checksum packet depends on the value of the type field in the header. If the type is 00 then this packet contains an end-around-carry checksum of all the previous packets in the block. If the type is 01 this packet contains zero (Note that an actual checksum cannot be zero unless all the packets in the block are zero, and the header is never zero). The checksum is present more to detect block framing errors (either because the wrong packet was identified as a header, or transmission was aborted in mid-block and a new one started), than to protect against the corruption of data during transit.

Type code 10 in the header indicates that the count field contains all the data in the block, and the following packets are not sent. Such immediate data packets have only limited applications.

The fourth code (11) has been allocated to indicate the use of a somewhat different block protocol. The count field contains a fixed bit pattern, and the header is followed by an extra packet containing the amount of data in bytes. This is followed by the route, data and checksum packets in the usual way. This variant has been specified with rings that have differently sized packets in mind, and is of only passing interest.

It is an important feature of basic block protocol that the layers of software above it should receive a block in its entirety or not at all. The basic block layer should never pass back incomplete or corrupt blocks.

One perfectly valid means of implementing basic block protocol is to receive all blocks regardless of source and then discard those that are not expected, too long for the available buffering or have bad checksums. This means that a block may be lost without trace, and may result in long delays before the transmitter realises this. A more helpful mechanism is to set the SAR to zero for a brief period of time just after the route packet of an unwanted block is received; the transmitter will see the destination go **unselected** in **block** and by convention will interpret this as meaning that this block is unwanted. This has the advantages that the sender can abort the transmission as soon as it sees this and time is not wasted at both ends in the transfer of a block of data that is not wanted.

Even given the above mechanism it is possible for a block to be lost, and in any case some machines do not implement it. Consequently basic block protocol does not guarantee to deliver a particular block to its destination with anything other than a high probability. Higher level protocols must therefore be capable of recovering from lost blocks.

I will have more to say about the details of basic block protocol in later chapters.

2.2.2 Single Shot Protocol

Single Shot Protocol (SSP) [Ody79] implements a simple Request/Reply protocol similar to a remote procedure call. It is built on top of Basic Block Protocol by reserving the first three data packets for type and control information, and is shown in figure 2.3.

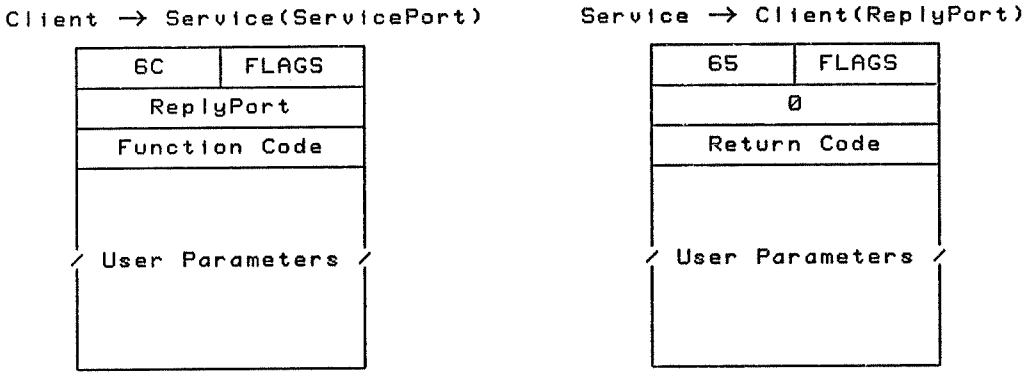


Figure 2.3 Format of SSP Exchange

The only proof the sender has that its request was received is the arrival of the reply. Failure of the reply to arrive is detected with a timeout. However the expiry of this timeout can mean one of two things: that the request was lost, or that the reply was lost. To make it safe to retry the request in both cases, SSP accessed services should be idempotent (i.e. repeatable).

2.2.3 Byte Stream Protocol

Byte Stream Protocol (BSP) [Johnson80] provides a pair of error free, flow controlled channels across the ring. Facilities are provided to push buffered data through a channel, reset the pair to a standard state, and to close the channels down. In some respects this is similar to a Transport Service, and can be upgraded to a full implementation of that protocol [JNT82].

BSP is built on Basic Block Protocol by reserving the first two data packets of a block for control information relating to the two channels, the remainder of the block is used for data. Data is never sent unless requested, giving flow control, and every data block is acknowledged, giving error recovery.

The first data packet sent in a transmitted block contains a command and sequence number relating to the channel being received by the sender. The commands may be: NULL, in which case no action is taken, RDY (ready), in which case the next block of data is being requested, or NOTRDY, meaning that the next block of data should not be sent (but the last one is being acknowledged).

The second data packet contains a command and a sequence number relating to the channel being transmitted by the sender. The command may be NULL as before, DATA, in which case the rest of the block contains the data being sent, or NODATA, meaning the sender has no data to send at present.

The protocol is normally implemented as a finite state machine and is best described by the state transition table given in figure 2.4. This table gives the state transitions for both the receiving and transmitting ends of a channel because in all major respects they are identical.

State E is entered when an Essential element (RDY or DATA) is sent; the other end is expected to reply immediately. State N is entered when a Non-essential element (NOTRDY or NODATA) is sent; the other end is not expected to send anything in return. State I is entered when the other end sends a Non-essential element; the channel remains idle until the other end sends another essential element. For any byte stream there will be four such state machines, one for each end of the two channels.

A Byte stream is initially set up by an OPEN exchange, the format of which is given in figure 2.6. The parameters passed are in two groups: the first relates to the properties of the stream to be created, the second group is defined by the user and relates to the service invoked. Only two BSP parameters are currently in use: the size of the largest block this end will send, and the size of the largest it will receive. The reply port is the port to which all subsequent BSP blocks will be directed.

The reply to an OPEN is an OPENACK. The parameters are grouped in the same way to that in the OPEN, the two BSP parameters consisting of the block sizes that will actually be used. The second word in the block (which is zero in an SSP reply) contains the port to which all subsequent BSP blocks should be routed.

Event	State		
	E	N	I
E[rep] DATA(n-1)/ RDY(n)	Retransmit RDY(n)/DATA(n)	Retransmit NOTRDY(n)/ NODATA(n)	Protocol Error
E[exp] DATA(n)/ RDY(n+1)	Empty/fill buffer n +:= 1 Buffer ready? yes: Transmit RDY(n)/DATA(n) Goto E no: Transmit NOTRDY(n)/NODATA(n) Goto N	Protocol Error	Empty/fill buffer n +:= 1 Buffer ready? yes: Transmit RDY(n)/DATA(n) Goto E no: Transmit NOTRDY(n)/NODATA(n) Goto N
N[exp] NODATA(n)/ NOTRDY(n+1)	Start Idle Handshake Timer Goto I	Protocol Error	No Action
Buffer Ready	No Action	Transmit RDY(n)/DATA(n) Goto E	No Action
Timeout	Retransmit RDY(n)/DATA(n)	No Action	No Action
Idle Handshake Timer Expires	Protocol Error	Protocol Error	Retransmit RDY(n)/DATA(n) Goto E

Notes:
n is the block sequence number (mod 16).

Figure 2.4 BSP State Transition Table

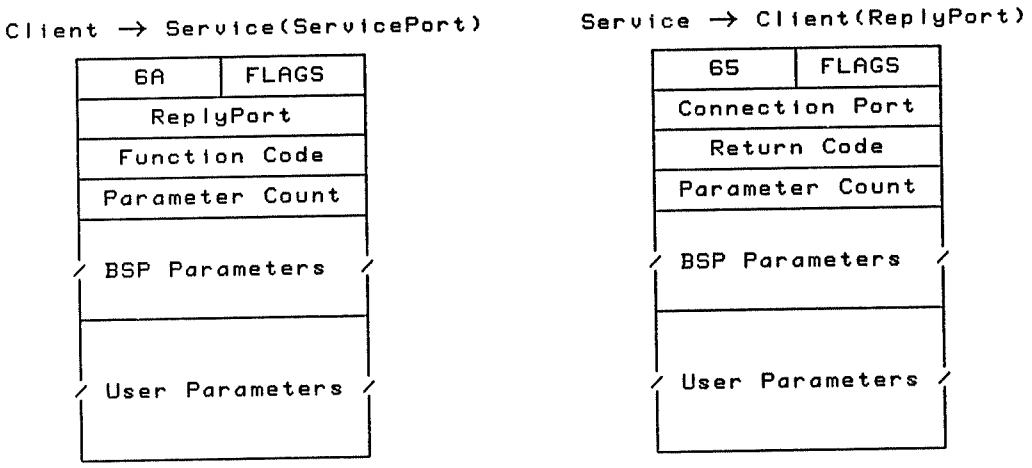


Figure 2.5 Format of BSP OPEN Exchange

2.3 The Cambridge Distributed Computing System

This section describes the structure of the Cambridge Distributed System, the environment in which most of this work was done. I begin with a simple overview of the system and follow with more detailed descriptions of those parts that are considered important, or have special relevance to the work presented here. A full account may be found in [Needham82].

2.3.1 Philosophy

The Cambridge Distributed System is based on the philosophy that it is better to give the user his own personal computer than access to a shared system. This approach becomes increasingly more realistic as the price of computer hardware falls. The advantages of this are that the machine is always to hand, under the user's control and its response is guaranteed to be constant. An isolated personal machine, however, is at a significant disadvantage. The cost of making it into a usable computing system with the addition of peripherals (terminal, hard-copy, permanent storage) is likely to be more than that of the processor itself, and if this has to be repeated several times it soon becomes more feasible to purchase a shared system. Separate machines also make the cooperation of users, and the propagation of software very difficult. The solution to these problems is to connect the computers together with a network; expensive peripherals may be shared, and programs and data moved between machines with ease.

Conventionally such a system is achieved by placing a machine of modest size, with keyboard, display and disc, in every office. While this is perfectly acceptable if the user does not get ambitious about the kind of program he wishes to run, such a machine is clearly not able to support such applications as relational databases, graphics and numerical computations. It is unreasonable to expect a user to leave his office to go in search of a suitable machine each time his computing needs outgrow his personal machine; it is also difficult to justify giving him a larger machine permanently, because he is unlikely to use it to its full capacity. Permanent allocation of a machine to a user also hinders the development and running of distributed algorithms.

One possible solution is to supply on the network a large shared computer to which the user may connect his personal machine as a remote terminal. The user now has access to enough computing power to run his more ambitious programs, but only has access when he needs it. This approach, however, is not really in the spirit of distributed computing, since most application effort would concentrate on the shared machine. At Cambridge

an alternative approach has been adopted.

2.3.2 The Cambridge Distributed System

The Cambridge approach is to give each user a personal computer of modest performance, possibly integrated with his terminal, and without local discs. This machine is capable of only minor duties, one of which is to connect as a remote terminal to some other machine on the network. Most of the computing power of the network resides in the **Processor Bank**: a collection of machines free for any user to claim.

When a user wishes to use a **Processing Server** (as the machines in the processor bank are known) he approaches the resource management system. This attempts to find a suitable machine, load it, and allocate it to the user. Once allocated the machine belongs exclusively to the user for as long as he desires; his personal computer acts simply as a (possibly very intelligent) remote terminal.

Since processing servers may be allocated to any user they may not have any local file storage. Instead all user files are stored on the **Fileserver**, which is accessible from any machine. Consequently a user may access his files regardless of the physical machine he is using.

Once allocated, a user has total control over the processor bank machine he has been given. This includes being able to stop it, start it, load any code into it, and debug it at the hardware level. The degree of sharing of resources here is at a somewhat coarser grain than that supplied by a time-shared system: dealing in whole processors rather than machine cycles. This pre-supposes that there are enough processors in the bank to satisfy demand; in practice this means that there should be slightly more machines than necessary. With the trend towards ever cheaper machines this is not a great price to pay for the advantages.

The composition of the processor bank need not be homogeneous, but may be tailored to suit the requirements of the user community. For example, it might contain a large number of medium sized machines to meet most users needs, plus several more powerful machines for those with greater demands. It may even contain machines with special architectures or hardware features (LISP and P-code machines, floating point hardware etc.). The handling of distributed applications is a simple extension of the basic system to allow a user to request more than one machine, and to allow the machines themselves to request further processors.

In addition to the processor bank the Cambridge Distributed System contains many machines dedicated to providing most of the services conventionally supplied by an operating system. These range from substantial machines controlling substantial peripherals (Fileserver, Laser printer), to small machines controlling simple peripherals (Terminal concentrator, Line printer, Time-of-day clock), to machines that perform the basic management functions of the distributed operating system.

The system does not prevent conventional mainframes and mini-computers from accessing its services, and using the network for their own purposes. The mainframe may use the system to whatever extent it wishes, from simple use of printers and file transfer, to access via remote terminals and keeping its filing system on the Fileserver. An example of converting an operating system to do the latter may be found in [Dellar80].

2.3.3 Implementation Details

The preceding section has given a somewhat idealised view of what the system looks like. In practice users do not have personal machines, but merely a conventional cursor addressed VDU connected to the ring via a terminal concentrator. All user work is done in processor bank machines or in time-shared machines connected to the ring.

2.3.4 Processing Servers

At present the processor bank contains two different machine types: Computer Automation LSI4's, and Motorola 60000's. Processing servers differ from most other types of computer in that they have no peripherals other than the Ring. Given this fact it is important that the Ring connection be of high performance. In both cases the connection is made via a second interface processor, which in addition to performing ring access functions for the Host computer is also able to exercise the kind of control over it one expects from the front panel of the machine: loading systems, stopping and starting execution, and debugging. Since these interfaces are the subject of this dissertation I will say no more about them here.

2.3.5 Processor Bank Management

The allocation and loading of processing servers is managed by three machines: the **Resource Manager**, the **Session Manager** and the **Ancilla**.

The Resource Manager is responsible for knowing which machines are allocated, to whom, and for how long. It is also responsible for allocating the machines and getting them loaded with the program or operating system required by the user. Requests to the Resource Manager specify the type of system to be loaded, the length of time it is wanted for, and the attributes the machine should have (i.e. large memory, floppy discs, intimate terminal). The Resource Manager searches its list of free machines for one that has the requested attributes (including any the specified system may demand) and if one is found loads it with the given system. Two times are specified in the request, one gives the maximum time for which the machine will be allocated, and an initial refresh time. Once the machine is loaded it must contact the Resource Manager before the expiry of the refresh time. The purpose of this is so that the machine may discover where it was started from and why it was started. In this message the machine must specify a new refresh time, and must repeat the exchange again before this time expires. The effect of this is to maintain a dead-man's handle between the Resource Manager and the machine and allows crashed machines to recovered. If the refresh time ever expires the Resource Manager may reclaim the machine for re-allocation.

The interface to the Resource Manager is at a low level so requests may be made by other computers for processing servers. For the user at the terminal the Session Manager provides a higher level interface. Initially the user connects his terminal to the Session Manager, which translates his typed commands into low level requests to the Resource Manager. Once the Resource Manager has allocated and loaded a machine the Session Manager drops out of the conversation and the user can communicate directly with the processing server.

The Resource Manager effects the loading of a machine by giving its name plus the name of a file containing the system to be loaded to the Ancilla. This service is responsible for knowing how to load that particular type of machine with a binary load image taken from its own filing system on the Fileserver. The exact activity of the Ancilla may range from simply passing the name of the file on to the machine's ring interface, to relaying the entire image from the Fileserver to the machine itself for less intelligent interfaces. In theory there is a separate Ancilla for each different type of machine but in practice they are sufficiently similar that they may be coalesced into a single machine (although their external interfaces remain logically distinct). The Ancilla also has a role to play in remote debugging.

2.3.6 The Nameserver

To use any of the various services it is necessary to address messages to them. If the system were static these addresses could be fixed and built into programs. This is, however, somewhat inflexible; we want to be able to move services between machines and move machines around the network. To allow this, all services are located by an SSP containing a textual name to the Nameserver, which replies with the address of the service. The Nameserver is distinguished in that its station address is well known and fixed for all time. Relocating a machine or service is therefore a simple matter of changing its entry in the Nameserver.

2.3.7 The Cambridge Authentication System

This is accomplished by at Cambridge with UID-sets. A UID-set consists of four items: a PUID which names an object or user, a TUID which represents an instance of that object, a TPUID which represents the association between the PUID and TUID, and an Authentity (Authority Identity) which names the authority under which this association was made. These all take the form of 64-bit random numbers to protect against forgery, and any of them, particularly the TPUID, may be absent. This association is stored in the Active Object Table (AOT) and is periodically refreshed in the same manner as the Resource Manager's dead-man's handle. The AOT also supplies functions to validate, identify, create, and enhance UID-sets. The reader is referred to [Girling82] for more details.

2.3.8 The Fileserver

The Cambridge Fileserver [Dion81] is implemented on a Computer Automation LSI4/30 computer with 80 Mbyte CDC disc drives. It provides a slightly abstract view of a filing system with two types of object: files and indices. All Fileserver objects are named by a 64-bit Permanent Unique Identifier (PUID) which is composed of 32 bits of object identification and 32 bits of random data to provide a degree of unforgeability.

A file is a vector of 16 bit words and may be any size from zero to about 13.5 Mwords. Space is only allocated on disc for those parts of the file that have actually been written. Consecutive sequences of words may be read or written at any point in the file.

An Index holds references to objects and is simply a list of PUIDs. The only restriction on what references an index may hold is that they must not be for objects on a different disc pack, so that packs may be mounted

and dismounted independently. The structure of references on a single pack, therefore, is a full directed graph or naming network, which may contain loops and multiple references. Each pack has a distinuished root index from which the entire network on that pack descends. The Fileserver guarantees to preserve any object as long as it is reachable from the root index, otherwise its PUID is invalidated and the disc space reclaimed. This is normally achieved by maintaining a count of the number of references that exist in the network to each object; deleting it when this count reaches zero. It is possible, however, for a cyclic structure of indices to become detached from the root system, and for reference counts to become higher than they should (care is taken to ensure they are never lower than necessary). These are taken care of by an asynchronous garbage collector, which is run on behalf of the Fileserver itself in a processor bank machine [Garnett80].

Two types of file are supported by the Fileserver: normal and special. Normal files are intended to be used for the majority of data storage; if a crash occurs while data is being written the Fileserver does not guarantee that the data will be left in a satisfactory state. Special files are intended for storing data that must always be in a self consistent state (for example, filing system directories). Any alteration to a special file will either happen completely, or not at all. Operations on special files are consequently more expensive than on normal files. Indices are treated like special files.

A client of the Fileserver may perform single operations on a file by quoting its PUID in the request, or it may execute a sequence of operations by opening it. In response to an OPEN the Fileserver returns a Temporary Unique Identifier (TUID) for the object which should be quoted in subsequent operations in place of the PUID. The TUID represents an interlock on the file, any attempts to access this file with the PUID will be rejected until the file is closed (and the TUID cancelled) or it times out. Opening a file (or index) has two advantages. The client gains an interlock on the object: if it is open for reading then read requests but no write requests will be granted to other clients, and if it is open for writing no other requests will be allowed on this object at all. If the object is special (or an index) all operations before the final CLOSE constitute an atomic transaction, and will either occur in their entirety or not at all.

Most Fileserver operations are single SSP exchanges with the exception of the READ and WRITE operations. The READ request specifies the object and the amount of data to be read, plus a port to which the data

should be directed. The Fileserver transmits the requested data as fast as possible to the specified port, followed by the standard SSP reply. The WRITE request similarly specifies the object and amount of data to be written to which the Fileserver replies immediately, nominating a port to which the client should send the data as fast as possible. Once the Fileserver has received all the data it sends a second SSP reply to the client reporting on the transaction's success. To make the reading and writing of small quantities of data easier the Fileserver also supplies SSPREAD and SSPWRITE functions, which can transfer up to 256 words of data in one SSP exchange.

There are currently two Fileservers in operation running five disc drives. The filing systems in use include one for the processor bank machines and one for the CAP computer. There are also several smaller private filing systems belonging to servers, notably the Mail and Ancilla filing systems. To allow disc packs to be moved between Fileservers there is a service (Packserver) which, when given a file PUID, will return the name of the Fileserver on which that file is currently resident.

2.3.9 The Filing Machine

Processor bank machines do not in fact use the Fileservers directly, but make all their files accesses through the Filing Machine [Richardson83]. This handles all the housekeeping associated with the filing system so the client machines need only contain a stub which hands all commands on to FM. The Filing Machine is equipped with 1 Mbyte of main memory and can implement an intelligent caching algorithm. The result of this is that the most frequently used files (system commands etc.) remain permanently in the FM's cache, and need never be fetched from disc. This more than makes up for the extra level of indirection the filing machine introduces. The Filing Machine can also implement access controls and accounting.

2.4 TRIPOS

The system usually loaded into processing servers is a variant of the TRIPOS operating system [Richards79a, Knight82]. TRIPOS was originally designed as a portable single user operating system for mini-computers with their own discs and peripherals. It has been successfully enhanced to run in the processor bank in such a way that most of the original software may be run without change.

In the interests of portability the hardware features exploited by TRIPOS are minimal; there is, therefore, no concept of memory management or protection, privileged processor states or priority interrupt levels. The outcome of this is that the TRIPOS operating system is totally unprotected from the user and everything lies in a single global address space. The advantage of this is that objects may be passed by reference to any point in the system, rather than being copied, and TRIPOS gains greatly in simplicity because of this. The main disadvantage is that an aberrant program can cause serious disruption; but since the system is single user, and restarting it should be cheap, this is deemed not to be a problem.

TRIPOS is a multi-tasking operating system, inter-task communications being achieved by message (or packet) passing. Since all addresses are global this is accomplished by switching pointers rather than copying and consequently the system is extremely efficient. Each task has a single packet or work queue, packets sent to the task are appended to the end of the queue. To receive packets a kernel primitive is called which either returns the first packet in the queue or suspends the task until one is available.

The TRIPOS kernel itself only provides support for tasks, devices, message passing and store management. Other services such as a filing system, terminal handling etc. are provided by handler tasks. The kernel and device drivers are written in assembly code while the rest of the system is written in BCPL [Richards79b]. The structure of the tasks and the implementation of the kernel functions as BCPL callable procedures makes TRIPOS an extremely BCPL oriented system, and although other languages have been implemented (Fortran, Pascal, Algol68c, Algol68RS) these see only limited use.

Device drivers are accessed by packet passing in the same way as tasks, but while tasks are addressed by small positive integers, devices are addressed by small negative integers. In general a device is started by the reception of a packet and when it finishes the packet is returned. The shortcomings of this simplistic view of devices will be elaborated in later chapters.

The system normally loaded into a processing server consists of seven tasks and one device driver. The device is, of course, the Ring (or, more accurately, the Ring Interface Processor). The seven tasks are:

The Ring Handler: this is the only task that communicates with the Ring driver and presents a machine independent interface to it along with several ancilliary functions such as port allocation.

The BSP Handler: this implements Byte Stream Protocol using the functions supplied by the Ring Handler. Normally there is only one byte stream open, which is being used by...

The Virtual Terminal Handler: this is responsible for translating the virtual terminal protocol being used over the byte stream to the terminal concentrator into a normal TRIPOS stream. This task is also responsible for maintaining the dead-man's handle to the Resource Manager.

The File Handler: this translates standard TRIPOS file access commands into messages to the Filing Machine.

The Command-Line Interpreter: this runs programs on the users behalf.

The Interactive Debugger: this is always resident and the user may switch the input of his console to it at any time to monitor the machine's activity. It is BCPL oriented in that it knows the structure of a BCPL stack and program, although it may also be used at machine level.

The Ring Services Task: this is responsible for starting up services in response to requests received from the Ring. These services include file transfer and user enquirys. This task is not essential and can be deleted without harm.

A major drawback of this system is that to be able to debug a program four tasks, the Kernel and the Ring device driver all need to be in working order. This can cause significant difficulty when attempting to debug a renegade program that could corrupt vital parts of store. Means to avoid this will be described in later chapters.

Chapter 3

The Type 2

The Type 2 is a high performance machine independent ring interface for connecting 16 bit mini-computers to the Cambridge Ring. By high performance we mean that not only is it capable of transferring data to or from the Host's memory at high speed, but is able to implement at least Basic Block Protocol on the Host's behalf.

In the context of the Cambridge Distributed System it is also required to exert control over the Host to load, stop, start and debug it. At present there are nine LSI4s connected to the Ring via a Type 2 including the Cambridge Fileserver.

3.1 Before the Type 2

The first Ring interface for the LSI4 was based on the Computer Automation picoprocessor, or intelligent cable. This was a simple programmable peripheral controller with DMA access to the LSI4's memory. Its program space was extremely limited, so it was configured just to DMA blocks of data to and from the Ring. Functions were also included to allow the Host to read and write the destination, select, source and status registers in the station. Basic Block Protocol was implemented in a handler task in the Host, which also had to calculate the checksum itself.

When the first Type 2s were made available the first program to be written for them was essentially an emulator of the picoprocessor. This allowed the hardware to be connected to all the machines with a minimum alteration in the Host software. Even with this simple program the Type 2 exhibited a performance improvement over the picoprocessor. This program was christened 'Noddy', due to its extreme simplicity.

3.2 Hardware

This section gives a brief description of the hardware configuration of the Type 2. A full description of the design process and implementation of the Type 2 can be found in [Gibbons80a].

The Type 2 is built around a Signetics 8x300 bipolar microprocessor that can execute any instruction in just 250ns. This speed derives partly from its bipolar technology, partly from its simple instruction set, and partly from the separation of its program and data address spaces, with a separate address bus for each.

The program bus has 12 address bits and 16 data bits and is read only. Each instruction is a single 16 bit word with a three bit op-code, giving just eight possible instructions. These include four standard instructions: MOVE, ADD, AND and XOR which can all work on combinations of registers and data bytes. There are three instructions that affect the course of execution of the program, the simplest of which is an unconditional jump (JMP). The only conditional branch available is to test a register or data byte and branch if it is non zero (NZT). The branch address is specified by a value that replaces the bottom five or eight bits of the program counter; such jumps are therefore limited to pages of 32 or 256 bytes. The number of bits replaced depends on the item tested, eight if it is a register, five if a data byte. The third branch instruction (XEC) specifies an instruction to be executed after it. If the specified instruction is a jump then the jump is taken, otherwise execution after this instruction is resumed after the XEC. The last instruction, XMIT, allows constant data to be inserted into registers or data bytes. Additionally any instruction that accesses a data byte can also specify a shift and mask to be applied to it before being made available. This gives the 8x300 powerful bit manipulation abilities.

The data bus is just eight bits wide and is multiplexed between address and data, thus there can only be 256 bytes of data in the address space. This is alleviated somewhat by duplicating it into two banks (known as Left and Right) which may be accessed independently. The multiplexing of the data bus is performed in part by software. Thus an access to a single byte requires one instruction to set the address in the appropriate address register, plus one to access the byte. Once a address is set, however, the same byte may be accessed repeatedly in just one instruction. Separate addresses may be set up independently for both the left and right banks. In the Type 2 the Left bank contains 256 bytes of fast RAM and the Right contains the mapped control registers of the Ring and DMA

channels. These features make the machine somewhat bizarre to program, and one tends to approach it in the same spirit one approaches microcode.

The Type 2 is connected to the ring by mapping the control registers of the station into bytes in the Right bank. The Type 2 can, therefore, perform operations on the ring simply by reading and writing select locations in its own address space.

The connection to the Host is made via two bi-directional DMA channels and two I/O ports. The DMA channels contain auto-incrementing address registers to assist in vector transfers. The I/O ports present a reasonably machine independent means of establishing communication between the Host and the Type 2.

Physically the Type 2 consists of three circuit boards: one containing the 8x300 plus memory and Ring interface logic, and two identical boards each containing one DMA channel and I/O port. The interface presented by the channel boards is designed to be machine independent so a fourth board is required to slot into the backplane of the Host computer to translate this into signals suitable for the Host's bus. The organisation is shown in figure 3.1.

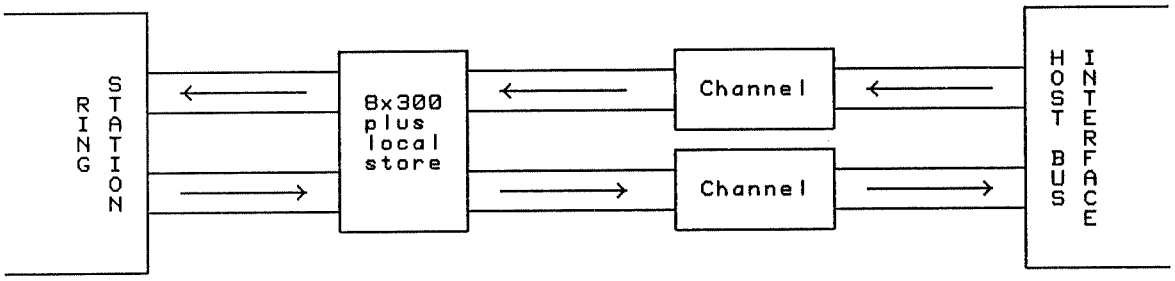


Figure 3.1 Type 2 Hardware Configuration

3.3 Design

This section sets out the required properties of the Type 2 program and the design decisions made to meet them.

3.3.1 Requirements

The primary requirement is of course that the Type 2 should implement Basic Block Protocol on the Host's behalf. This means that it must preface the user's data with the header and port packets, and add the checksum packet at the end, calculating the checksum itself. Similarly the

header and port should be stripped on reception, and the checksum validated. The Type 2's limitations mean that it cannot buffer the user's data, which must therefore be transfered directly from the Host's memory to the Ring (or vice versa). This means that the checksums must be calculated on-the-fly. This fact has been a major influence on the design of this, and later, intelligent interfaces.

That the protocol should only be BBP rather than any higher protocols is dictated by the Type 2's insufficiency of program and data space to implement anything more complex. Additionally it was decided that the Host would gain nothing from direct access to Ring packets.

In the context of the Cambridge Distributed System the Type 2 not only has duties to perform on behalf of the Host, it also has to implement various control functions initiated from the Ring. These functions include loading the initial system into the Host, setting it going and debugging it when it fails. The Type 2 must always be able to perform these functions, so it must protect its integrity against any possible misbehaviour by the Host or by any other machine on the Ring.

3.3.2 Host Commands

The principal commands given by the Host to the Type 2 are concerned with driving Basic Block Protocol. A typical command might be:

> "Transmit n words of buffer b as a basic block to port p on station s."

Such a command must occupy several 16 bit words and a method must be supplied to transfer them all to the Type 2. Three such methods can readily be devised. First, they can be passed one after another through an I/O port. To prevent the Host and Type 2 getting out of step some form of protocol would be necessary to synchronise the exchange; results would have to be returned the same way. This is not in keeping with the our desire to the minimise the Host's work when using the Ring since the exchange protocol could become quite complicated. The second method parcels the entire command up into a codeword, or command vector. The address of this is passed to the Type 2 via the I/O port which can then read the codeword via a DMA channel. When the command is finished the results can be DMA'ed back into the codeword and its address returned via the I/O port. The third method is a variation of the second, a fixed command vector is used and the Host need only signal the Type 2 when a command is ready, since the address is already known. Results are

returned via a second vector. A minor amount of protocol is necessary to prevent, for example, the Host overwriting the vector with a new command before the Type 2 has read the previous one. This can be made independent of the choice of location of the command vector by supplying the address to the Type 2 during initial loading.

The second method was implemented since the current hardware lent itself most naturally to it. To implement basic block protocol three commands are necessary. The general form of the transmit command has been given. A reception command looks like this:

> "If a Basic Block arrives on port p from station s and contains not more than n words of data, put it in buffer b."

The station number s may specify either a specific station, or may be a value denoting that any station will do. The form of this command raises two important points about the design of the reception system in the Type 2. The first is that it must be capable of holding several reception commands at once for different port/station combinations. The second is that requests need not necessarily be returned in the order they were submitted, it depends on the order in which Blocks arrive from the Ring. This is no problem with the adopted communication method, the address of the codeword uniquely identifies it; if either of the other two methods had been used, however, it would be necessary to pass an identifier to the Type 2 with the command to be returned with the reply.

The third command cancels reception requests. The Type 2 has no concept of the passage of real time, so reception requests will remain with the Type 2 until either satisfied, or cancelled. It is the Host's responsibility to maintain reception timeouts, and when a timeout expires the relevant request must be cancelled. The Cancel command simply specifies the address of the codeword to be cancelled. Cancelling the request is simple if it is dormant; if, however, it is in the process of being satisfied there are two possible courses of action: to fail the cancellation and allow the reception to complete in the normal way, or to abort the reception of the block and satisfy the cancellation. Since the Host is cancelling the request it can be assumed that it is no longer interested in seeing it satisfied, so the second course of action is the correct one. It is still possible for a cancel command to be issued while the satisfied request is in the pipeline back to the Host and the cancel command will fail since the specified request can no longer be found. The Host must, therefore, be ready to handle the case of cancelled reception requests being returned.

3.3.3 Buffer Chaining

The commands described above are suitable for transmitting or receiving one basic block from (or into) just one buffer. There are cases where it would be useful to give the Type 2 a buffer larger than the maximum Basic Block size and have it either transmitted as a sequence of blocks, or not returned until it has been filled by several separate blocks. The requirement for this comes in part from the observation that Fileserver Reads and Writes require exactly these primitives. For transmission no change is required to the command format. The Type 2 can trivially detect when the supplied buffer size is greater than a Basic Block and split the transmission up into several maximum sized blocks plus one to carry the remainder.

For reception the case is somewhat more complicated. It is only useful to wait until an entire buffer has been filled if the amount of data to be received is known; for example, a Fileserver read. At other times the amount of data, and the size of blocks, will not be known and the Host will want notification on a per-block basis. The Host will here want to submit a buffer of at least the maximum size of block it expects, to be returned as soon as a block is received into it. The difference between these two lies in the action taken after a block is received: in one the codeword is returned immediately, in the other the codeword is returned only if the buffer is full, otherwise it is made ready for the reception of a further block. The Host can specify these two different actions with a little as one bit in the request codeword.

Given that splitting a single buffer into several basic block is a useful thing to do, what about splitting a basic block across several buffers? When transmitting or receiving higher level protocols (SSP, BSP) the block will be divided into two or more sections by the protocol. For example: an SSP request has three words of protocol specific header followed by user data. Using the simple basic block functions any SSP protocol implementation would either demand that the user leave three words free at the front of each buffer he submits, or would have to copy it into a buffer of its own to add the protocol data. If the Type 2 undertook to join buffers together, however, the SSP package could simply prepend the user's buffer with a three word buffer of its own. The user can lay out his buffer without worrying about leaving space for the protocol package, and when it is finally transmitted it is included in the block by indirection rather than by copying. A more suitable example it that of a long read from the Fileserver; here the data which is coming from the Fileserver is to be installed in several file handler cache blocks. The data is unlikely to

arrive in blocks of the correct size, and the Host will have to engage in some elaborate shuffling to get it to fit correctly. If the Type 2 were to automatically insert the blocks into the buffers in the correct order the Host would be saved much trouble. It can therefore be declared that a general buffer/block splitting facility would be a useful attribute of the Type 2.

Dealing with reception first: the required Host specification can be achieved by adding a link field to the codeword and chaining further codewords on to it for the same port and station. When a block comes in the data is simply inserted into successive buffers which the Type 2 finds by following the chain. A problem arises when the amount of buffering provided in the chain is less than the size of the block. The conventional response to an otherwise valid block that is too large for the buffer is to go unselected as early in the block as possible. An early response can be obtained if the Type 2 were to follow the chain, totalling up the buffer sizes and comparing it with the size of the block before deciding whether to accept it. Unfortunately, since the Type 2 has insufficient memory to allow it to keep a slave copy of the chain, it would have to follow the real chain in the Host's memory using a DMA channel. Potentially this can take a long time if there are many small buffers, possibly more than the critical time during which the next packet must be accepted. A lazier alternative is to follow the chain only when it is necessary to get another buffer. This has the advantage that there is little overhead, but the lack of buffer space may not be detected until some distance into the block, resulting in the pointless transfer of data.

Chained reception presents further options regarding the the return of completed codewords. Normally when receiving a block into a chain of buffers the Host is not interested in being told about the completion of intermediate buffers, but will want to be told about the success of the entire transfer once, at the end. This implies that a third option is required in addition to the two previously described, namely: do not return the codeword at all, just drop it. The complete set of options can be specified by two bits in the reception request, called the Block bit and the Buffer bit. If only the block bit is set the codeword will be returned only if the basic block finished while that buffer was being filled; this correspond to the normal one-block-only option previously. If the buffer bit is set the codeword will be returned only when that buffer is completely filled; this is the buffer filling option. If neither bit is set the buffer will be filled but will simply be dropped when it is finished and the Host will not be told. If both bits are set either event will cause the codeword to return.

An envisaged Host to the Type 2 is the Fileserver which has its own particular needs regarding the use of buffers. When the Fileserver is performing a WRITE for a client it is unlikely to have enough buffers in main memory to contain all the data. It must therefore write the contents of a filled buffer out to disc and return it to the Type 2 to be re-filled. Since the data may be arriving in blocks of any size, buffer filling is required to make maximum use of disc bandwidth. To avoid communication overheads the Fileserver should be able to simply add the recycled buffer to the end of the existing chain of codewords, where the Type 2 will pick it up in due course. Unfortunately if the Fileserver is just a little late in adding the new codeword to the end of the chain the Type 2 may examine the link, find it empty, and assume the chain has ended, with dire consequences for any proceeding reception. This can be avoided by including an explicit end of chain marker in the real last codeword of the chain. If the Type 2 comes across a codeword with an empty link but without an end of chain mark it knows that the Host will add a new codeword to this chain soon and can wait for it. If the Type 2 is seeking a new buffer to continue the reception of an incoming block this wait has a critical time of about a millisecond, so such a feature is only useful in making allowance for the Host occasionally being slow, where it is usually able to add codewords at full speed.

The introduction of chaining alters the way in which cancellation works. The Type 2 is at any time only aware of the head codeword of a chain. If any blocks have been received on that chain this may not be the codeword originally submitted by the Host, so an attempt to cancel that will fail. The entire chain has to be cancelled by successively submitting cancel request for codewords in the chain until one succeeds. However, since cancellations are relatively rare, this will not be unduly expensive.

Chaining on transmission has much the same form as that for reception. An important difference is that the lazy chain following scheme described above cannot be used. This is because the header packet of a basic block must contain a count of the data packets within it. The only way to determine this is for the Type 2 to follow the chain totalling up the buffer sizes. For the same reason given above (insufficient workspace) this is not possible, so a restricted form must be used. While transmit codewords may be submitted in chains in the same way as reception codewords, no attempt is made to overlap basic blocks across buffer boundaries. Each buffer is therefore transmitted as an integral number of full sized basic blocks plus a smaller filler. This is acceptable, because in those cases where a large quantity of data is to be transmitted it is most

likely that the destination will have a Type 2 or something similar and will be splitting the blocks up on reception anyway. Since the buffer boundaries are not time-critical the wait-for-link facility used for reception is not necessary. Space was found for a small amount of codewords queueing to allow several codewords (or, rather, codeword chains) to be submitted in quick succession. Since there is no analogous concept to "receive one block" only one option bit is passed with the codeword to determine whether the codeword is to be returned or dropped on completion.

3.3.4 The Ring Interface

In addition to presenting an interface to its Host the Type 2 must also present an interface to remote machines on the Ring. This is demanded by the Cambridge Distributed System since it must be possible to exert full control over a processing server from anywhere on the Ring. These control functions fall into two separate, but related, categories: Loading and Debugging.

3.3.4.1 Loading

Once a processing server has been allocated to a user it must be loaded with the operating system image of his choice and started. Since the machine will have no peripherals except the Type 2 the system must be inserted from here. To load a system correctly the Type 2 must also be able to halt, reset and start the Host; this can be achieved by the simple expedient of giving it access to the control lines of the Host.

The required system is normally to be found in a file in the Fileserver. For the reasons given above it is not possible for the Type 2 to engage in Fileserver transactions, and in any case the load file is likely to be in some format that would have to be decoded. The best the Type 2 can do is to accept basic blocks of data and insert them into the Host's memory at a given address. To convert the load file format into Type 2 load format a special server is required. It was primarily for this purpose that the Ancilla was devised.

3.3.4.2 Debugging

During the development of a new program it is inevitable that it will crash and need debugging. Most of the time the underlying operating system will survive so the examination can be carried out using conventional debugging facilities (e.g. the Tripos debug task). This is not possible, however, if the program takes the rest of the system down with

it (easy to do on Tripos), or it is the system itself that needs debugging. In these cases what is required is the ability to debug the entire processing server from without. Like loading, the Type 2 provides the obvious place to do this. The Type 2, however, is supposed to be machine independent, and is certainly not capable of containing a full debugger. The Type 2 should therefore supply a set of primitive , machine independent, debugging operations with which a more sophisticated program running elsewhere on the Ring can build all the usual debug facilities.

What should these primitives be? The prime requirement is to be able to examine the Host's memory, and if repairs are to be effected to write to it. If the data structures to be examined are to be in a consistent state the Host processor should be halted while the examination is taking place, and allowed to continue when it is finished. There are also times when the debugger (or some other program) is merely required to perform a monitoring function and the processor should not be stopped. The minimum set of primitives are, therefore, commands to read and write a single word of store, and commands to halt, run and reset the processor.

Another reason for selecting as small a set of primitives as possible is that most of the debugging software is unable to make use of already existing code. This is because the bulk of the code is written to transfer blocks of data from the Host's memory to the Ring and vice versa under the control of codewords. The debug commands require that the the block be seen by the Type 2 itself and not passed on to the Host. It is true that the reception of debug data to be written to the Host, and the transmission of that read, could be performed by the main code, but only at the expense of further complicating an already complex piece of code. This last consideration virtually excludes the possibility of implementing vector read and write primitives for debugging (unless the entire basic block code is to be repeated), so they are reduced to single word read and write, which can be implemented in fixed sized basic blocks.

A problem that arises with both loading and debugging is how to differentiate between basic blocks sent to the Host and commands sent to the Type 2. The only possible means of differentiation is is that debug commands should be sent to a special port number. It is not, however, acceptable for the Type 2 to arbitrarily take a port number out of the set allowed, the Host may have a legitimate desire to use that port number itself. The port number must therefore come from outside the normal range. Examination of the route packet of a basic block reveals four bits that were formerly unused. Allocating one of these bits to indicate that

the block is intended for the interface processor not only gives the Type 2 an un-ambiguous means of identifying commands, it gives it a space of port numbers equal to that enjoyed by the Host.

3.3.4.3 Protection and Authentication

The Type 2 Ring interface gives the remote user complete control over the Host processor. Unfortunately without any form of authentication any other user can also exert these same controls, at the least annoying the legitimate user, and at worst breaching security. The only item of information in a basic block that cannot be forged is its source address; the authenticity of the blocks must therefore be proved using this alone. The only legal source of loading commands is the Ancilla; but without resorting to the ethically questionable practice of binding the address of the Ancilla into the program, the Type 2 has no way of authenticating load requests. It is allowable to write the address of the Nameserver into the program, and Nameserver interactions are sufficiently simple that the Type 2 could manage to look up a name using fixed sized blocks. What name should it look up? Simply looking up "ANCILLA" is no good, since there may be several Ancillae for several different machine types. Since the Type 2 is meant to be machine independent looking up a name of the form "ANCILLA-<machine type>" is clearly not allowed. A possible solution is to introduce a new Nameserver function "MYANCILLA" which would return the Ring address of the correct Ancilla. This would require the Nameserver to keep a table listing processing servers versus Ancillae.

The adopted solution was none of the above. The Type 2 is fitted with a program readable coding plug that was originally intended to give the Ring address of the Nameserver to make the code independent of that too. Instead the Ring address of the relevant Ancilla is supplied here and it is a simple matter to check the source of loading commands against this for authentication.

The case of debugging is different as the identity of the source of debug commands is not fixed (unless the Ancilla undertakes to indirect all debug requests). In general the debugger will be some other machine on the Ring running an interactive debug program; the Type 2 should only accept debug commands from that source during the debug session. The Type 2's trust in the Ancilla can be used to set up this association; all that is required is a command from the Ancilla saying "allow yourself to be debugged from station s". In this way the Type 2 will obey debug commands from station s and nowhere else, except the Ancilla. At any time the Ancilla can close the debug session by setting the debug machine address to zero. The authentication procedure engaged in by a debugger to

persuade the Ancilla to let it have access to a particular machine may be made as complex as necessary since the Ancilla is not victim to the same strictures of space as the Type 2.

Use of the Type 2 may not necessarily be confined to processing servers, but may extend to standalone machines (an example is the Fileserver). Such machines will have their own bootstrap mechanism and will not need loading from an Ancilla. Similarly such machines will not need remote debugging, and in some cases it must be positively discouraged. On the other hand when commissioning new hardware, or testing for faults, it is convenient to be able to load and debug the machine from anywhere without appealing to the Ancilla. Some simple conventions for the value presented on the coding plug allows the three different modes of working to be selected. The value from the coding plug is used to set the Ring select register when the Type 2 is waiting to be loaded, and is tested against the source of debug requests at other times. Normally this value is a ring address between 1 and 254. If it is set to zero, however, the Type 2 will de-select everyone when waiting to be loaded, and will accept no debug commands, having the desired effect of disabling loading and debugging. If the coding plug is set to 255 the Type 2 should accept loading request from anyone, and by testing the plug for 255 when debugging requests arrive, allow these from anyone as well.

3.4 Implementation

This section briefly describes the implementation of the Type 2 program, highlighting some features that have an effect on its performance. Some concepts are also introduced here that will have a bearing on the design of subsequent Ring interfaces.

The transmit and receive basic block drivers are implemented as independent state machines. The 8x300 does not have interrupts, so all external events must be tested for explicitly by polling status bits. To allow full duplex working it is necessary to poll for both reception and transmission events simultaneously. This means that a single polling loop is used which, on detecting an event, enters the correct state machine and is returned to when the action is complete. Since the 8x300 cannot support subroutines the normal means of achieving this cannot be used. Instead each state machine must maintain a state number; when it is to be entered this byte is used in an XEC instruction to index into a jump table. In this way each state machine may be written as a set of sequential routines. To further simplify the state machines, and to avoid the duplication of code,

the polling loop also performs low level transmission timeouts and retrys, reception timeouts and deselection of unwanted blocks. The I/O ports are also polled in this loop primarily because in the LSI4 implementation the Host processor is halted until the Type 2 takes the data, which it must therefore do as fast as possible. The DMA channels are not polled in this loop for two reasons: first, DMA transfers are expected to be fast, and second, the addition of this to the polling loop would result in the further complication of an already complex piece of code.

The DMA channels and I/O ports are logically split into two sets, one for transmission and one for reception. This can lead to some contention for the DMA channels between the transmission or reception of a block and the reading of a codeword. This is easily avoided in the transmission case by only reading a codeword immediately before obeying it. Reception is not so simple since the reception request should be made active as soon as possible. A codeword may be submitted by the Host at any time, and in particular this may be while the Type 2 is busy receiving a basic block. To avoid interfering with the reception of the block the codeword address could be saved until after it has finished and then read. To allow for the Host submitting several requests in rapid succession the addresses would have to be queued. Since RAM space in the Type 2 is limited this is not possible, and the somewhat less acceptable practice of reading the codeword as soon as it is submitted has been adopted. If a codeword is submitted while a block is being received the DMA channel will be in use. Further, it may still be busy with a transfer since the writing of a word of data is performed in parallel with the wait for the next packet[1] . This last is no real problem since the Type 2 can simply wait for the DMA to finish. The real problem is that the DMA channel address register will contain a pointer into the buffer; DMAing the codeword in will cause this address to be lost, and since the register cannot be read it cannot be saved. To restore the DMA address register to its correct value the address must be recalculated. The Type 2 keeps track of the number of packets still to be received by storing it as a negative number that is incremented towards zero. By adding this to the address of the <u>end</u> of the buffer the current buffer address can be produced and reinserted into the DMA register, ready for the next packet.

For the purposes of monitoring the remote debugging reception and transmission routines must co-exist with the normal basic block system. This required that they use the main polling loop to allow full duplex working. To achieve this the debug basic block state machines are added on

1 This is particularly true of the Fileserver, since the disc controller may reserve the bus for longer than the packet inter-arrival time.

as an extra set of states to the normal set for both transmission and reception. Movement into the debug sub-machines is prompted in the case of reception by the detection of an 'interface bit' in the route packet of a basic block, and by the setting of a 'debug transmission pending' bit in the case of transmission. The 'debug transmission bit' is set when a debug command requires a reply, which has to be done this way because debug commands are obeyed in the reception state machine, and it is not known there what state the transmitter is in.

Loading is performed in a totally separate mode which is entered when the Type 2 is reset and exited into the normal mode only when the system has been loaded. In loading mode the Type 2 keeps the select register set to the value on the coding plug, thus accepting packets from that source only. Since this value may be zero, selecting no-one, the Type 2 must not only poll for ring receptions, but must poll the I/O ports too. This is to allow for machines that do not need loading from the Ring; activity on the I/O port implies that the machine is alive and causes the Type 2 to immediately enter normal mode. Another consequence of the need to allow for standalone machines is the meaning of the reset command. The Type 2 is able to reset the Host and conversely if the Host is reset the Type 2 should be too. The result of this is that the Type 2 can expect the order to reset the Host to reset itself as well. This is also a convenient way for the Type 2 to destroy any outstanding state in might have.

The only existing implementation of a Host software interface to the Type 2 is that for Tripos in an LSI4 which was written by B. J. Knight [Knight82]. This is examined briefly here because it is the object of some comments later in the chapter and for comparison with equivalent interfaces described in later chapters. The code is divided into three parts, two ring device drivers plus a Ring Handler task. The two device drivers are each associated with one of the the two I/O ports and handle transmission and reception repectively. Normally the only task to send packets to them is the Ring Handler.

The transmission device driver is the simplest. Each Tripos packet directed to it contains a codeword. This is submitted to the Type 2 and the driver waits until it returns before returning the packets and processing the next one on its input queue. Note that it does not use any of the chaining or multiple submission facilities of the Type 2.

The reception driver is somewhat more complex owing to the fact that the Type 2 must be given all requests immediately, and the codewords will not be returned in a predictable order. As soon as a packet arrives at the driver the codeword it contains is submitted to the Type 2 and the packet appended to an internal chain in the driver. When the codeword returns from the Type 2 the corresponding packet is retrieved from the chain and returned to its origin. This device also handles the cancellation requests: the codeword is submitted to the Type 2 and if the cancellation succeeds the relevant packet is also dropped from the chain.

The Ring Handler task presents a simple basic block level interface to the Ring, using only the single block and buffer filling facilities of the Type 2 and none of the chaining. The Ring Handler also manages the allocation of port numbers.

3.5 Performance Measurements

As soon as the Type 2 program was ready to go into service the interim 'Noddy' programs were scrapped and it was installed in all existing Type 2s. This meant, unfortunately, that a full set of comparative performance tests could not be made. The only such measurements available were made by comparing the performance of the Type 2 development system with the 'Noddy' interface on an LSI4/10 and an LSI4/30. The measurement made was to record the time taken to transmit 100 full sized basic blocks to the same destination. The destinations were: SINK, a Ring station that accepts all packets, plus the three source machines running a program that accepts all blocks on a given port.

The figure 3.2 gives the results, which are the averages of several readings in each case. The figures are in seconds.

Source	Destination			
	SINK	LSI4/10 Noddy	LSI4/30 Noddy	LSI4/10 Type2
LSI4/10/Noddy	7.32 (a)	-	7.14 (b)	7.25 (c)
LSI4/30/Noddy	4.57 (d)	10.61 (e)	-	4.62 (f)
LSI4/10.Type2	4.04 (g)	9.53 (h)	5.19 (i)	-

Figure 3.2 Transmission Times for 100 Basic Blocks

The first point to note about this table is that readings (a), (b) and (c) give somewhat anomalous results, since it is expected that transmission to SINK is the fastest operation in all cases. These three average values are all within the range of values recorded for each instance, so for the current purpose should be considered equal.

Comparison of the figures (particularly (b)-(c), (d)-(g), (e)-(h)) leads to the conclusion that an LSI4/10 with a Type 2 is equivalent to an LSI4/30 using Noddy. Later measurements show that an LSI4/30 with a Type 2 interface took about 3.80 seconds to transmit to SINK (the only measurement that could be repeated), showing that while the difference in processor speed between the LSI4/10 and the LSI4/30 apparent above (compare (a) with (d) and (h) with (i)) was still present it was much less significant. The times for transmitting from one Type 2 to another are all between 4 and 4.20 seconds, regardless of processor type, showing that the Type 2 can normally receive as fast as it can transmit. These figures show that the Type 2 is capable of transferring data at a rate of about 400K bits per second, compared with the theoretical Ring maximum point-to-point bandwidth of about 800K bits/second.

The above figures were taken when the Ring had three slots and a small gap. At the time of writing the Ring has four slots and a gap of nearly a packet size. The result of this is that the theoretical point-to-point bandwidth has dropped to about 600K bits/second. The transmission rate of the Type 2, however, has only dropped to 370K bits/second. This reduction is due entirely to the increase in ring revolution time and contention for Ring slots since the greater size of the ring is caused by more stations rather than more wire.

The Type 2s currently in service run at a quarter of their true speed, executing one instruction every microsecond. This is largely because of the great expense of suitable high speed PROMs needed to store the program. If the Type 2 were to run at full speed it is expected that it will be able to make maximum use of the theoretical bandwidth, even on a single packet Ring.

From the Host's point of view the Type 2 is a significant improvement. While the speed of ring transactions has improved slightly for an LSI4/30 and by nearly a factor of two for an LSI4/10 the real improvement derives from the introduction of parallelism. The Host is now relieved of the task of constructing the basic blocks and calculating checksums (a task that can take as long as the transmission itself). Formerly heavy use of the Ring halted all other activity in the machine, and the Ring Handler

was a major consumer of CPU time. This is no longer the case, and the Ring Handler runs for an insignificant amount of time.

A unfortunate consequence of introducing the Type 2 has been an increase in the size of the Ring drivers and the Ring Handler over the original Noddy version. I will explain why this is so in the next section.

3.6 Discussion and Conclusions

In this section I wish to examine where the hardware, design and software of the Type 2 met their requirements, and where they failed.

The hardware of the Type 2 has largely met its goal of a simple basic block level ring interface. The original design was conceived with the type of program it was to run in mind and the program produced has been influenced by this. The major drawback of the hardware is the micro-processor around which it is built. The 8x300, while possibly suitable for controlling vending machines and electric cookers, is too limited for this application. Since the 8x300 has eight bit data paths, and all the data items processed are sixteen bit quantities, the number of instructions required for most operations is at least doubled. This is further aggravated by the extreme simplicity of the instruction set which makes the simplest operation into a major programming exercise. For example, just to calculate the checksum requires a sequence of fifteen instructions to be executed for every ring packet transmitted or received. The restriction to 256 bytes of RAM is another failing that forces the programmer to be extremely frugal. The need to keep a chain of outstanding reception requests and a queue of waiting transmission codeword addresses means that all this memory is used.

The original requirement to produce a general basic block interface was, in retrospect, not perfectly met. The buffer chaining facilities have never been used in practice and the rest of the code would probably benefit from their removal. The principal reason for this is the high level Tripos ring interface that is presented by the Ring Handler (and pre-dates the Type 2) has no support for chaining, and it would be incompatible with other Tripos Ring interfaces to change it. It should also be noted that most application programs would not benefit from such facilities in any case. The only machine that could use the chaining is the Fileserver, but due to lack of manpower, and the unavailability of the Fileserver for program testing this has not been done.

While the general concept of chaining is correct the implementation here leaves much that is incomplete or un-defined. This is because the hardware limitations have resulted in some corners being cut and some essential functions being left out. It is probably a good thing that the chaining has not been used.

A disturbing aspect of the addition of the Type 2 to Tripos was the growth, rather than reduction, in the size of the ring driving software in the Host. There are two factors that contribute to this. First, the Type 2 does not match the model of devices expected by Tripos. In particular, the possibility of packets sent to the reception ring device driver being kept for a long time and being returned in a random order, makes this driver much more complex than normal. The second factor is paradoxical in that the Type 2 is both too intelligent, and not intelligent enough. It is not intelligent enough because the Host must still do most of the housekeeping associated with ring use: maintain timeouts, retry transmissions, sequence cancellations and manage port number allocation. It is too intelligent because its modes of failure are more complex than those of a simple device and require correspondingly more code to handle the many cases.

On the plus side the Type 2 has relieved the Host of a considerable burden in driving basic block protocol, and the increase in performance is marked. It should be noted that this improvement is due entirely to moving the implementation of basic block protocol into the Type 2, and does not come from the Type 2's inherent speed. The object of comparison, the Noddy interface, is itself a Type 2 running a different program, and since it does not checksum, or work in full duplex, its raw data rate is greater than that of the Type 2 program described here. The conclusion we must draw from this is that speed alone does not lead to greater performance, but that increased functionality is as important, if not more so.

Chapter 4

The MACE

In the summer of 1981 the Cambridge Computer Laboratory acquired several computers based around the Motorola MC68000 microprocessor to supplement the LSI4s in the processor bank. These machines were purpose built to serve the needs of a processing server and the particular feature of interest here is the presence of an interface processor of significantly more power than the 8x300: the MACE[1]. This chapter serves as an introduction to the MACE and the particular problems it presents to the programmer. The format of this chapter follows that of chapter 3.

4.1 Hardware

Physically the system consists of three printed circuit boards: the 68000 processor, its memory card, and the MACE. All three slot into a common backplane and the Ring is connected to the MACE via a pair of sockets on the front of the card. Of the three boards the MACE is by far the most complex.

The 68000 is a reasonably standard configuration, and is not designed specifically for the application. The board has a socket for a memory management unit although at present none have been fitted and the socket is occupied by a header that connects the processors address lines directly to the backplane. Devices are mapped in the normal way into several pre-defined I/O pages; in the processor bank systems there is only one mapped device: an interrupt line to the MACE. The 68000 also has a 50Hz clock interrupt available to it. The memory card can contain up to 512K bytes of dynamic RAM and, although it can be fitted with less, this is the standard configuration. Some systems have been given extra cards to boost them to 1 or 1.5 Mbytes.

1 This is rumoured to stand for Multi-Access Control Equipment but the true origin of this name has been obscured by the mists of time and its inventor.

The MACE is built around a Motorola MC6809 microprocessor. In architectural terms the 6809 lies somewhere between the earlier 8-bit processors and the full 16- or 32-bit configuration of the 68000. While it is externally an eight bit machine most internal registers are sixteen bits wide and most instructions are equally applicable to eight or sixteen bit operands. The 6809 also has a rich set of addressing modes that make it suitable for high-level languages, and make assembly language programming easier. The processor has a cycle time of 500ns, and since the average instruction takes six cycles this makes it a 1/3 Mips machine (the 68000 has a rating of about 3/4 Mips)[2].

The memory map of the MACE consists of 56K bytes of RAM, 4K bytes of mapped I/O space, and 4K bytes of PROM. The full complement of RAM is the most important point since it allows large programs to be written for it.

The Ring is interfaced to the 6809 in a conventional way by means of memory mapped registers. Additionally, however, there is a direct path for data from the Ring to the DMA hardware and vice versa. Unfortunately this is not vector transfer logic, and the 6809 must participate in the movement of data. Neither is there automatic checksum hardware (although the necessary connections exist to allow it to be added as an extra board), so the 6809 must calculate these itself.

The DMA Hardware is built around a DMA controller that supports four DMA channels each with its own auto-incrementing address register and counter. The four channels are allocated by the hardware to specific functions. One channel is dedicated to performing direct transfers from the Ring to the Host's memory and another for transfers in the opposite direction; these channels are uni-directional. The other two channels are bi-directional and are uncommitted in the hardware. All four channels may be operated by the 6809 directly. There are two minor problems associated with the use of this particular DMA controller. The first is that it is designed to work in an address space of just 64K bytes, so both the address registers and counters are only 16 bits wide. The 68000 has an address range of 24 bits so to enable the 6809 access to the Host's entire memory four extra 8-bit latches are provided, one for each channel,

2 The 6809 figure is based on the most commonly executed instruction: a 16-bit load (or store) into a CPU register from a constant 8-bit offset from an index register. The 68000 figure is based on the equivalent 68000 instruction: a 16-bit load into a data register from a constant 16-bit offset from an address register. In practice the 68000s are run as 32-bit machines and never use 16-bit operands, the equivalent 32-bit operation gives the 68000 a rating of only 1/2 Mips.

to supply the upper portion of the address. This introduces problems of its own since these latches are not incremented automatically and special action has to be taken whenever a DMA transfer crosses a 64K page. The second problem is that the DMA controller is really designed to DMA into the same memory space as it is controlled from, and can assume that its control registers will not be accessed while it is in the middle of a DMA cycle. In the MACE this is not true and the 6809 may well access the control registers at an inconvenient moment for the DMA controller. To avoid this the contention is detected and the 6809 halted until the DMA transfer is complete.

The conceptual hardware configuration is shown in figure 4.1.

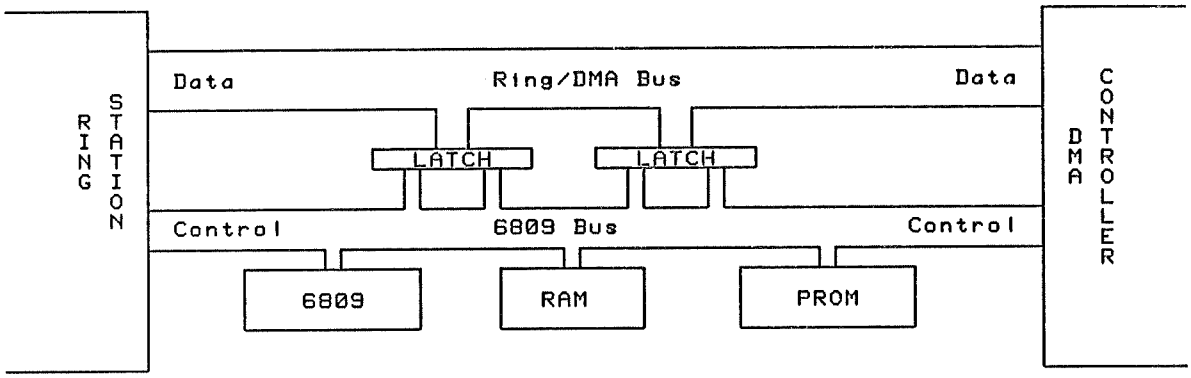


Figure 4.1 Hardware Configuration of the MACE

There are currently twelve of these 68000 system in use, eight are for general use in the processor bank, one has 1 Mbyte of store and runs the Filing Machine, one is the basis of a high resolution display cum workstation and one is the controller of a laser printer. The last has been fitted with two MACE boards and is used as a Ring-Ring bridge, with the MACEs cross-connected so that they receive from one ring and transmit onto the other [Leslie83].

4.2 Design

It was never intended that the MACE have just one program in the way that the Type 2 does. Instead there may be several different programs for different purposes and Host applications. To allow this the code in the PROM has been deliberately kept simple and limited to just loading and debugging the MACE itself. Initially the protocols used here were private, later, however, it was changed to use the same protocols as the existing Z80 systems [Ody81b]. This was done largely so that the MACEs may be

loaded automatically from the Bootserver; additionally it enabled existing debug programs to be used (with slight conversions).

Consequently the initial service program, named SPECTRUM after its multi-coloured Hosts, was not written to be general purpose but tailored to the requirements of Tripos. Again the design may be split into two parts.

4.2.1 The Host Interface

The only means of communication between the 68000 and the MACE is a pair of mutual interrupts and the DMA channels controlled by the MACE. This makes inter-machine communication slightly more complicated than in the Type 2. The mechanism used is the request/reply vector one described briefly in chapter 3. This requires that the Host place its request in memory at an address previously agreed with the MACE and then interrupt it. The MACE can now read the request from the vector and act on it. When a reply is generated the MACE writes the results into another fixed vector and interrupts the 68000. To prevent a new request or reply being written before a previous one has been read both machines test a bit in the first word of the vector they write. If this is set the vector can be written, which has the effect of clearing the bit. To match requests with replies an identifier of some sort must be included.

SPECTRUM supplies only three functions to the Host which correspond closely to those presented by the Tripos Ring Handler task. Specifically these are: 1) transmit a buffer of data, splitting it into blocks as necessary, 2) set up a reception request either for a single block or a buffer full, and 3) cancel a reception request. Unlike the Type 2 the cancel function specifies a port and station number and causes exactly one request (if any) to be cancelled. This still requires the Host to repeatedly submit requests until a cancel fails. Cancelled receptions are returned with a suitable return code. The Tripos Ring Handler deals with buffers that begin on long-word boundaries (32-bit) and are an integral number of dibytes (16-bits) long. SPECTRUM expects all buffer addresses and sizes to be expressed in byte quantities.

A major difference between SPECTRUM and the Type 2 is that no block chaining is attempted. Since Tripos never uses this, it is no great loss. An improvement over the Type 2 is that the Host can include a retry count in the transmission request, and a timeout in receptions. Both of these reduce considerably the amount of work needed by the Ring Handler.

4.2.2 The Ring Interface

To remote machines, particularly the Ancilla, SPECTRUM presents a remote debug and load interface similar to that of the Type 2. The major difference is that there is no separate loading mode, the same commands are used for both purposes. The principal commands allow arbitrarily large portions of the of the Host's memory to be read or written. These are used for interactive debugging and memory dumping as well a initial loading. In addition commands are available to reset, start and halt the machine similar to those in the Type 2. A function is also available to set the addresses of the request and reply vectors and allows them to be changed depending on the system loaded. None of these commands are protected.

4.3 Implementation

This section gives a brief overview of the SPECTRUM program, highlighting some areas that will be important later.

The basic structure follows the Type 2 implementation in consisting of two state machines. These are traversed not by means of state numbers but by altering the contents of the interrupt vectors. Internally a ring transaction is represented by a control block that contains the station and port numbers, retry count or timeout, base address and size of the buffer plus the address of a routine to be called when the transfer has finished. There is also a flag byte that specifies, among other things, whether the buffer is in local or Host memory. These last two features allow both debugging and Host requested ring transactions to be treated in exactly the same way by the ring driving code and only differentiated at the end when the the action routine is called to reply to the Host or complete the debug request.

A significant disadvantage of the MACE is that only one interrupt vector is used for all interrupts. To make the various devices independent a piece of code is attached to this that decodes the identity of the interrupt from the status registers and takes an indirect jump through one of several software defined vectors. This is exacerbated by all the hardware interrupt vectors being in PROM address space and to make them alterable by loaded programs an indirection must also be included in the PROM. The result of this is that the fastest time between the interrupt being raised and the handling routine being entered is 42 cycles, or 21 microseconds[3].

3 This is a ring reception interrupt, the times on the others are: ring transmission 47 cycles, Host interrupts 51 cycles, and clock ticks 61 cycles. DMA interrupts are not used.

Since the time for a ring revolution is less than twenty microseconds, driving the program entirely on a per-packet interrupt basis resulted in a severe performance degradation from this factor alone. In order to avoid this a test was added at the end of each interrupt routine to see if there was another interrupt pending. If there was the routine looped back to beginning to service it. To prevent hogging this was only allowed to happen sixteen times in succession before a return from the interrupt had to be made. The number sixteen was chosen because a study of ring traffic suggested that the vast majority of basic blocks were less than this number of packets long [Ody81a].

Encouraged by this the entire program was converted to polling with a subsequent improvement. The reason for this is very simple: the decoding performed on each interrupt is exactly the same code as the substance of a polling loop, converting from interrupts to polling merely changes an interrupt return and immediate reentry into a jump back to the beginning of the loop.

Another consequence of the hardware design is that certain operations on the ring station use the Ring/DMA bus and in particular destroy the content of the latches connecting it to the processor bus. These latches are the source and destination of 6809 initiated DMA transfers, so all requests must be read and results written to completion with interrupts off. This prevents these activities being carried out as a background task asynchronously with the ring transfers. This also prevents pre-fetching and 'post-putting' of words for the ring transfers. Ring DMA transfers cannot be asynchronous in any case since the completion of the DMA transfer is the only indication available that the packet contents is ready for adding to the checksum.

The software to run in the Host is much simpler than that for the Type 2. It consists of a single Ring device driver and a Ring Handler Task. The device driver copies certain fields of any packet sent it into the request vector, and copies the results back into the packet when a reply is generated. It uses the address of the requesting packet as the unique identifier needed in the request, allowing it to find the packet easily when the reply comes back. The Ring Handler Task simply passes all request straight on to the device driver after re-arranging the arguments into a format closer to what the MACE expects. It also implements the cancel retry loop and manages port number allocation.

4.4 Performance

The performance characteristics of the MACE are best demonstrated by comparison with the Type 2. The comparisons are made between a 68000/MACE and an LSI4/30/Type2. The test used was the same as that used before, namely the time to transmit 100 full sized basic blocks to SINK and to eachother. The results are shown in figure 4.2, the readings are in Kbits/second.

Source	Destination		
	SINK	68000/MACE	LSI4/Type2
68000/MACE	341	321	341
LSI4/Type2	367	353	365

Figure 4.2 Comparison of Raw Throughput in KBits/second

The consistent figures in the top row show that the MACE is limited by the speed at which it can transmit, although the figure in the centre of the lower row indicates that it can do a little better on reception.

4.5 Conclusions

The figures show that while the MACE is slower than the Type2 it can manage a roughly comparable throughput. The MACE suffers from the same disadvantages as the Type 2: having to involve itself with every packet and calculating checksums in software. That it is a slower processor accounts for some of the difference although the more powerful instruction set of the 6809 makes up for some of this; for example, only four instructions are needed to calculate the checksum.

By tailoring the MACE program to the requirements of the Host it has been possible to reduce the Host software to a minimum. It would have been possible to reduce it even further if the MACE took buffer addresses and sizes in the same units as the ring handler and the semantics of the cancel command were changed. The actual increase in functionality of SPECTRUM over the Type 2 is quite small, only transmission retries and reception timeouts being significant, so the consequent effect is all the more remarkable.

The MACE played an important role in the development and testing of the Tripos kernel for the 68000. Since the 68000 has no other peripherals all debugging had to be performed via the Ring. This included not only remote examination of the Host's memory but the operation of simple single character input and output by polling certain locations in the 68000's memory via the MACE. This allowed one to communicate with the standalone Tripos debugger in the Host from another machine. The result of this was that the Ring Interface of SPECTRUM was implemented and in use long before the Host interface was installed.

One can conclude, then, that the MACE is a ring interface of moderate performance. While it cannot compete with the Type 2 for throughput its most important feature is an easily programmed microprocessor and large quantities of RAM. The exploitation of these features is the subject of later chapters.

Chapter 5

A High-Level Intelligent Interface

5.1 Introduction

The previous two chapters have described the designs of two low level intelligent interfaces. It has been shown that the major gains are derived from the parallelism introduced by the separate processor, and from an increase in functional specification of that processor. It is suggested, therefore, that full advantage should be taken of the presence of this extra machine to offload yet more of the Ring driving workload; thereby increasing the performance of the entire system to its maximum. The result of this is the development of a High-Level Interface.

A High-Level Interface is one that is independent of the details of the underlying network and protocols. This means that the Host can deal with an abstract view of the world and that it may be changed without any alteration being needed in the Host. A primary objective of the High Level Interface is to implement most of the higher level protocols in a more suitable environment and consequently more efficiently. The activities of the High-Level Interface may, however, pass beyond the purely quantitative domain of improving protocol performance to the qualitative domain of systems and management aspects associated with network communications.

In the following discussion examples will be drawn from the Cambridge environment and from the Xerox Internet Specification [Xerox81a]. The comments made, however, are believed to be relevant to all Local Area Networks.

5.2 The Implementation of Protocols

To demonstrate that various protocols will benefit by being moved to an interface processor it will be instructive to examine the conventional techniques used to implement them. Some unconventional implementations will also be examined. When any protocol implementation is moved out of the Host processor into a NIP there are two areas that should be examined for possible benefits and disadvantages. The first is the effect it has on the Host machine: whether the Host is involved in more or less work to use the protocol, and whether the gains are significant relative to the Host's use of the network. The second is the effect it has on the protocol implementation itself: whether it gains anything by moving into a more benign environment, and whether it loses access to useful facilities in the Host.

Recognising that no one protocol implementation is exactly like any other (even if they are of the same protocol) it is possible to identify three broad classes of protocol and implementation.

5.2.1 Packet Protocols

This is the most primitive protocol made available to any client of the network and constitutes the foundation upon which any higher level protocols are built. At Cambridge this is Basic Block Protocol, and in the Xerox Internet is the Internet Datagram. At its upper interface this class of protocol deals with buffers of data and network addresses, at its lower interface it deals with the raw hardware of the network. Because all network traffic is dependant on this protocol an efficient implementation is necessary to gain maximum throughput. To avoid software overheads such protocols are usually implemented at as low a level as possible. So we find implementations in hardware (HDLC chips), in microcode (Alto ETHERNET interface [Thacker79], CAP Ring interface), or in machine code in the device drivers (RSX Ring Interface [Gibbons80a], VAX UNIX Ring Interface [Collinson82][1]).

The previous two chapters have already documented the results of moving this form of protocol out of the Host. These interfaces were for the Cambridge Ring which is a special case as far as driving programs are concerned. This is caused by the unusually high rate of interrupts when a basic block is being transmitted or received. To cope comfortably with a rate of one interrupt every fifteen microseconds or so, either requires a

1 This is actually written in the language C, no UNIX device drivers are written in assembler.

fast processor, or hardware assistance. It therefore makes good sense to remove the basic protocol driving from the Host and let a second processor cope with this high event rate.

This is not necessarily the case with other network architectures where the hardware and software packet sizes are identical. In this case hardware assistance for transferring data between Host memory and the network is mandatory since it arrives at line transmission speed. Here, the processing load on the Host is unlikely to be particularly large and may not be a significant factor in the time taken to transmit or receive a packet. We can say, however, that for any network, while the implementation of this level of protocol in an interface processor may not be significantly better than that in the Host, it will certainly not be any worse. It should of course be pointed out that if any higher level protocols are to be implemented in the NIP then this level of protocol must be present; so there is really no choice.

5.2.2 Connectionless Protocols

This is a general term for those protocols that do not maintain any state between invocations by the client; each use by the client being a totally separate instance. These protocols can usually be implemented within, or synchronously with, the client process and typically by library procedures or operating system calls. These protocols are often based on simple packet exchanges and make few guarantees about performing the task with total success; it is usually expected that the client will deal with failures by re-trying the interaction several times. Examples in the Cambridge environment are SSP, File Server protocol, Remote Debug and Load protocols for LSI4's and 68000's. These protocols are usually simple to define, and equally easy to implement; so if the supplied implementation does not do exactly what is required, is thought to be too generalised for efficiency, or the programmer needs a slightly different protocol for his own purpose, he is quite ready to produce his own ad-hoc version. This results in a profusion of divergent protocols and implementations, which all have to be altered if the specification of the base protocol changes (or not, if nobody minds the divergence getting even greater).

The principal advantage that any high level protocol can obtain from being implemented in a NIP is that of closer co-operation with the lower packet level protocol. It has already been mentioned that many high level protocols would benefit from being able to split a single packet between several buffers for both transmission and reception, enabling it to

assemble or decode the packet more easily. It would be even more useful if the protocol package were allowed to exert some control over the lower levels as the packet is being transmitted or received; for example to decompose the components of the protocol 'on-the-fly'. An attempt to allow something of this sort was made in the Type 2 with its buffer chaining facilities, with not entirely successful results. The problem with the Type 2, and with a conventional packet protocol implementation (where it is usually separated from the client by the width of the Operating System), is that it is difficult to know exactly what information is required by the lower level to make the correct decisions on behalf of the upper level in all cases. What we really want is for the lower level to ask advice of the upper level at the appropriate moments; the upper level is in a better position to know what the structure of this packet is, and can take decisions in a wider context. This up-call[2] mechanism is clearly something that cannot be implemented across a machine/machine boundary, or from a device driver to a client process with any degree of efficiency. If both levels, however, were implemented in the same machine, and the operating system of that machine were designed to allow such activity upwards referral of this kind would be possible.

Returning to the case of connectionless protocols: it would certainly be possible to move the equivalent of the library procedures or operating system primitives into the NIP in a transparent manner. This still leaves the ad-hoc implementations of these and other protocols to be dealt with. The argument that the supplied implementation is inefficient should now be spurious; no implementation in the Host should be able to match that in the NIP. While it may also be possible to improve the flexibility of the protocol interface while moving it, it is not possible to anticipate every demand that will be made and there will probably remain a need for highly specialised applications. The arguments that the supplied implementation does not do what is required can therefore remain valid. Many of these differences may be for good functional reasons, for example the Fileserver READ and WRITE protocols differ from SSP for a good reason. The most suitable solution for these protocols is to legitimise them and implement them in the NIP. Other protocols differ merely at the whim of their implementor and could easily be converted into a legitimate form; examples are the LSI4 Debug protocol, which could be converted to SSP, and the load protocol, which could be converted to the Fileserver protocol.

2 We owe this term to D. D. Clark [Clark82b], which lends dignity to an otherwise sordid practice. I will have more to say on this subject in later chapters.

Merely converting or legitimising the existing protocols does not solve the problem: as new applications arise, new ad-hoc protocols arise. There are two approaches to this. The first is to strongly urge implementors to use the existing protocols wherever possible, and to legitimise those new protocols that prove themselves to be necessary. The second approach is to take a hard line and totally outlaw any possibility of new protocol implementations. This can be easily done by denying the Host access to the packet protocol. In conventional systems access to the lowest level protocol is essential to allow the higher protocols to be implemented. Now that these are all implemented in the NIP there is no longer any need for the Host to have such access.

This restriction is perfectly acceptable, and even desirable, in a service environment where the users are not concerned with the details of the network or protocols. It is not acceptable, however, in a research or development environment where there may be a need for new protocols to be tried out, particularly if they are candidates for removal to the NIP. If the two environments are disjoint then all that are required are two versions of the NIP program: one that allows access to the packet protocol and one that does not. If the environments are not disjoint, and the same machine may be used for service or development, then it may be necessary for the Host system to authenticate itself in some way to the NIP before being allowed to use the packet protocol interface.

5.2.3 Virtual Circuit Protocols

This class of protocol attempts to supply a higher level of service to the client than the previously described protocols. This usually takes the form of guaranteeing the delivery of data to the other end free from errors and under flow control. The protocol therefore undertakes to maintain a connection between the two ends along which data flows. The example of this from the Cambridge environment is BSP, and in the Xerox Internet is the Synchronous Packet Protocol. Such protocol implementations are expected to be able to respond to asynchronous events from the network, the clock and the client, often within a crisis time, without impairing their ability to respond to the others. It is not usually possible, therefore, to implement the protocol in the client process because while the client program is executing the protocol is unable to respond to events. Thus the protocol must be implemented as a separate process, and the client must use the protocol via an inter-process communication mechanism. Because of the time-critical nature of some protocol events it may be necessary to run the protocol handler process at a high priority, and in swapped systems it may need to be permanently

resident in main memory to meet its crisis times. The latter is often only possible if the process is made part of the operating system (e.g. the VAX UNIX BSP Handler). Because of their complexity and unusual requirements one seldom finds an ad-hoc implementation of such protocols.

From the Host's point of view the removal of this class of protocol to a NIP is extremely beneficial. It results in the deletion of an entire process from the system, one that was likely to consume large amounts of processing time. Since the protocol process is likely to run at a high priority, any event in the protocol will cause the current process to be preempted in its favour. The protocol process itself will probably run for only a short period before suspending. This causes the operating system to engage in a large amount of process switching. With the protocol removed the only events left are those caused by, or directed at, the client, which are relatively fewer, and need not always cause a process switch to take place.

Moving the protocol to the NIP now gives it the same advantages as the previous class of protocol: closer cooperation with the lower level protocols. Because the amount of processing needed for any single protocol event is often very small a Host implementation is likely to suffer from large system overheads; a process switch (which may involve a change of protection domain too) can take a long time and inter-process communications mechanisms are often slow. In a NIP the system overheads can be reduced to virtually zero by carefully tailoring the operating system to the application.

To see a further advantage of a NIP based implementation of this protocol class we must examine the applications to which they are put. These fall into two broad areas. The first is that of File Transfer, Remote Job Entry and other relatively short-lived applications. The protocol is being used here largely for its ability to deliver data in an error free manner. Any one instance of the protocol will only last a few minutes, or if it persists will be re-used by different applications in short bursts. Such uses of the protocol will benefit only slightly from its movement into a NIP; that a file transfer takes five seconds instead of twenty is of little real significance.

The second area of application is that of Remote Terminal Access, where the protocol is being used to define a session, and although the error

correcting characteristics are useful they are of secondary importance[3]. Applications of this sort are characterised by the long time the virtual circuit is open, and the continuous use to which it is put during that time. It is widely becoming the practice to connect all of the terminals on a site to the network and allow access to the various Host computers only via these remote terminals. This has important consequences for the operating system of the Host machine. In a conventionally configured system, with intimately connected terminals, console interaction is simple, consisting of merely transferring a byte to or from a buffer on each interrupt. To turn this into a network connection one must replace this simple console driver by the virtual circuit protocol driver. The terminal driver is replaced not only by the protocol handling process described above, but a virtual terminal protocol implemented on top of this. This turns a previously negligible consumer of resources into a major one and involves the operating system in some considerable complexity to turn the low level concept of a terminal into the high level concept of a virtual terminal, particularly if it is to be done in a transparent way.

The advantage to this sort of application of moving the virtual circuit protocol into a NIP is that it reduces the complexity of the Host operating system. This is because the NIP can be accessed at a lower level than the former protocol implementation since it presents a device level interface. Admittedly the NIP is a more complex device than a terminal, or terminal multiplexor, but suitable design of its interface should allow it to be driven by even the simplest code. Having moved the virtual circuit out of the Host it is tempting to consider moving the Virtual Terminal Protocol too. An advantage of the NIP is that it should be able to transfer data from the network straight into the Host's memory with minimal intervention. Most VTP protocols require that the data be filtered for commands and escape sequences, so the NIP would have to examine every byte itself, negating this advantage. A VTP is, in any case, a relatively simple protocol, and is usually better implemented at a suitable place in the Host operating system.

3 The error characteristics of LANs are such that lightweight protocols may be designed on the assumption of no transmission errors, and drastic action (like closing the connection) can be taken on the rare occasions when they do occur. Such an approach, and protocol, is discussed in [Rubinstein81].

5.2.4 Alternative Implementations

The above classes are, of course, generalisations, and many examples may be found that do not fall into them. In the Cambridge environment one finds the Basic Block protocol implemented in a process (the early Tripos Ring Handler), and BSP implemented in the client process.

This latter example is an experimental implementation for Tripos by D.D.Clark [Clark82a]. It contains several interesting features, and the principles on which it is based had an effect on the author's own work. This implementation of BSP is an attempt to eliminate some of the system overheads, specifically task switching and message passing, incurred by the conventional BSP handler task. This is achieved by moving the protocol implementation into the client task as a pair of coroutines plus several procedures. The coroutines handle the asynchronous events of the protocol and it is while these are running that most of the work of the protocol is performed. The client program is implemented as one or more further coroutines and all the coroutines are managed by a special coroutine scheduler. The client accesses the protocol by calling its external procedures. These are generally simple, setting flags and causing the coroutines to be scheduled; the client does not, for example, supply data at this time. When the protocol coroutines run they will decide whether a basic block is to be sent to the other end of the connection, and at this point will ask the client to supply data to fill the block with an up-call. Similarly the protocol will hand data to the client with an up-call when a block is received.

The advantages of this are first, that the up-call mechanism allows the protocol package to ask advice of the client at the relevant moment, enabling it to tailor its actions to suit the application. For example if, when the up-call to ask for data occurs, the client has no data to hand but expects some to be available soon (within milliseconds) it can tell the protocol to wait a short time before sending the packet. Thus by only asking the client for data immediately before transmission, a packet may be shared more effectively between layers. The protocol package is now solely concerned with the job of implementing the protocol, and any peripheral concerns can now be left to the client. For example, in a conventional implementation the protocol package would have to implement an extremely general buffer management strategy to meet all possible applications. This task now belong to the client, which can use an algorithm more appropriate to the application, with a consequent simplification. Finally the up-call mechanism more faithfully echoes the structure of network packets. This is because the packets are built up

from the front, the first part belongs to the lowest level protocol, the second part to the next level and so on. A sequence of calls up through the layers with each layer inserting its protocol information into the buffer before passing it on is a natural way of composing the packet. The same structure also holds for the decomposition of received packets.

There are, of course, some disadvantages to this implementation. The major one is that the protocol package does not interface to any conventionally written programs. Only applications that can accept the up-calls and will not interfere with or circumvent the coroutine scheduler can be allowed. At present there is only one of these, a Virtual Terminal Protocol handler. The described structure does not fit into any known language or operating system structure, so an implementation of this sort may not always be possible. This can always be circumvented by designing an operating system or a language (or both) which allows such structures, and this opens up some interesting possibilities. Because the up-call mechanism works by presenting the client with a buffer to fill, it is not possible to use any form of multiple buffering, which can serve to improve the throughput of the stream. Instead the client is forced to copy his data into the buffer in the critical path, just before transmission. The final disadvantage of this implementation is one of performance. While the reasons for moving to a coroutine based single process system are sound, the advantage gained in comparison with the conventional implementation is small. This is because the resulting systems overheads have not changed much. The highly optimised machine code task scheduler of the Tripos kernel has been exchanged for a somewhat complicated coroutine scheduler written in BCPL. This comparison is somewhat unfair, and the improvement would undoubtedly be more marked in an operating system that was more protected, and had higher task switching overheads than Tripos.

Another BSP implementation worth mentioning is that written by J. J. Gibbons and D. W. .Singer for a PDP11/45 under the RSX operating system [Gibbons80b]. This also runs in the clients process, but presents a conventional stream interface. This is again configured as two coroutines, but unlike Tripos no attempt is made to make the byte stream transparent to clients and only purpose written programs use it. It is interesting to note that this implementation also contains an up-call mechanism, which is used only in erroneous or exceptional circumstances.

5.3 Systems Aspects

Moving the implementation of protocols from the Host to the NIP is only a beginning. Looking further we can see that the NIP can take over some of the systems and management functions of the Host where they relate to the network. Some of these are necessary, and obvious, consequences of moving the protocols, others could be performed by the Host but are more aptly implemented in the NIP given the movement of the protocols. I do not wish to say too much about these matters here, since they are covered in greater detail in later chapters. I will therefore limit myself to a brief discussion of the possibilities.

If the protocols are to be implemented efficiently the NIP should also have control over network addressing. This is best achieved by retaining the real address in the NIP and passing the Host a virtual address to use in network transactions. This has the advantage of hiding the exact format of a network address from the Host, allowing it to be changed. It also provides a degree of protection, since the Host will now only be able to access those remote machines for which the NIP holds addresses. This implies that the NIP must also control the mechanisms by which the Host obtains addresses; two such mechanisms exist. The first is by mapping a well known name, usually a character string, into an address. This can be controlled by supplying functions to perform the mapping in the NIP and pass back a virtual address. The other source is addresses that appear as data in messages from elsewhere. Sometimes these are necessary, for example the reply address for SSP, and so must be allowed for. Those that are necessary will be included in the protocol specification, and can be identified and converted into virtual addresses to be passed back to the Host; any others cannot be identified and will be passed on to the Host unaltered. If the NIP is to rigidly maintain its protection the Host can be prevented from using these addresses by defining everything to use virtual addresses; the Host consequently has nothing it can do with a real address if it gets one (except pass it on to someone else!). Alternatively, if protection is not a major issue, the NIP can supply a function to convert any real address given it by the Host into a virtual one.

The inverse to the above is protection of the Host from external malice. Performing a preliminary acceptance check on any received messages will prevent troubling the Host with badly formatted or erroneous blocks, and can even extend to checking for acceptable function codes etc. The Host may even request that the NIP ignores all messages from a particular source if it is causing trouble.

Allied to protection are the issues of security and authentication. Once data has passed out of the Host on to the network some guarantee may be required that is has been delivered to the required destination and no-one else. Authentication is the mechanism by which the communicating parties can satisfy themselves that the other is genuine and not an imposter. Once this has been achieved it is necessary to transfer data, and if the data is sensitive it may need to be delivered securely, this usually means that it must be encrypted. If the Host is prepared to trust its NIP then it can play a large part in both these areas. The NIP can automatically authenticate any incoming messages and not bother the Host with any that prove to be bogus; similarly it can add authenticating information to outgoing blocks. If the encryption and decryption of data is left to the NIP the Host need never deal with the encrypted form of the data at all.

Further areas of interest are Remote Loading and Debugging of the Host, the use of stable storage in the NIP to preserve state information beyond machine crashes, and the customizing of a NIP to perform specific activities on the behalf of specialised Hosts.

5.4 Conclusions

This chapter has discussed the possible advantages of moving more than just the low-level protocols to an interface processor. There are advantages to be gained both by the Host in losing much software, and by the protocols in moving to an environment more suited to their needs. Beyond this it has been suggested that the NIP should take on some of the systems and management functions associated with the network.

There is always a danger of putting too much into the NIP. From the purely quantitative point of view there will come a point at which any attempt to load more work onto the NIP will result in a degradation in performance. Of all the tasks a NIP could perform we must only select those that are suitable. Protocol support should stop at ISO OSI level 3 [ISO79], since above this the protocols require that the data be scanned for commands and escapes, eliminating the advantage the NIP has in avoiding copying.

It would be possible, for example, to move most of the Host operating system into the NIP, particularly if the Host were a processing server with only the Network to connect it to the rest of the world. This would probably need a NIP of the same size and power as the Host, which is not the intended equation, and is really a different research topic [Dellar80].

There remains, of course, the need to supply some 'system' functions in the NIP when the Host is a processing server, to allow the machine to be debugged and reloaded remotely. The functions that should go into a NIP are those that are directly related to the use of the Network, any others should be treated with the utmost suspicion.

Chapter 6

The Design of a High-Level Interface

The previous chapter described the concept of a High-Level Interface and gave some reasons why it would be a good idea. In this chapter I will describe the pragmatic design of such an interface on the MACE for Tripos. The next chapter will deal with some of the implementation issues. With this I intend to show that the claims made in chapter five are true.

6.1 Host Interface

The Host interface defines both the physical form of the communications between the two machines and the semantics of the commands exchanged.

6.1.1 Inter-Machine Communication

The hardware of the MACE/68000 system merely allows the two machines to interrupt one another and the MACE to have DMA access to the 68000's memory. This resulted in the use of two fixed buffers in the SPECTRUM program to contain requests and replies. This arrangement, while perfectly adequate, is somewhat unsatisfactory since it involves the Host in copying requests into the vector and results out.

If the MACE is to present a higher level of interface it should be able to communicate with its clients at their level, particularly if we want it to take over some of the existing functions transparently. The standard mechanism for communication between tasks, and between tasks and devices in Tripos is the packet. This means that the MACE must be able to both receive requests and reply using Tripos packets. A device driver could be written that copied the contents of any packet sent it into the request vector, and copied the results back when a reply was returned. This is a somewhat wasteful activity however, and it would make much more sense if the MACE obtained its arguments directly from the requesting packet, avoiding the copy. The device driver need now simply transmit the address of the packet to the MACE (using the old request vector), and return the reply to the sender when the reply is to be generated.

It could be argued that this makes the MACE operating system dependant. It should be noted, however, that the requirements of the internal interfaces of the MACE are different from the requirements of the Host interface. The Host dependencies can therefore be concentrated into just one module, which translates the requests into internal actions, the rest can be made completely machine independent. It should also be noted that the MACE does not interpret the packets it is given as inter-process messages. As far as it is concerned it is given a pointer to a vector of store that contains a function code and a set of arguments at certain pre-determined offsets. What the structure of the vector is outside the few parts the MACE examines it does not need to know. It could be argued that the MACE is really driven by codewords, which in the current implementation can be conveniently used as Tripos packets. The only differences between these and the codewords used in the Type 2 is that they cannot be chained together, and the argument encoding is somewhat sparser than is strictly necessary. The former is no great loss, and as will be seen later is largely unnecessary. The latter will simply cause the MACE to DMA a few redundant bytes when reading its arguments, but since the major cost of a DMA transfer is in the setup this should not be significant. This does involve the MACE in two DMA transfers, one to get the packet address and one to get the packet itself. This is offset, however, by avoiding any copying in the Host, and while the request vector may need to be of a fixed size, the packet may have any number of arguments.

6.1.2 Replacing the Ring Handler

All Tripos Ring interactions are normally directed to the Ring Handler Task, which directs a request to the Ring device driver, which itself submits a request to the MACE. If we are to eventually move all protocol handling into the MACE, it seems reasonable as a first step to take over the existing functions of the Ring Handler. The only real change needed is the specification of the interface, which now becomes that of the Ring Handler. The only function not performed in some way by the original SPECTRUM program was that of port management. This has to be moved to the MACE for it to implement higher level protocols, so a Ring Handler-like interface can easily be provided. This move has some important ramifications for Tripos, which will be covered in the section on Host Software in chapter seven.

The Ring Handler functions constitute exactly those lower level protocol functions whose use, it was decided in chapter 5, was to be discouraged. Unfortunately so many programs in Tripos use the Ring Handler directly

that it would not be possible to do away with them immediately. We must therefore consider their presence as a purely temporary measure to aid in the change, and their use will be prohibited or severely restricted in the future. The rest of this design proceeds on the assumption that these functions are not present or accessible.

6.1.3 Single Shot Protocol

This is the major connectionless protocol in the Cambridge environment although I will also deal here with the other protocols of this type. For SSP we must supply not only the client end of the protocol (which is all most conventional implementations do), but the service end as well if we are to present a full implementation.

The client end of SSP is simple: given the service to be called, a buffer containing the arguments and a buffer to contain the reply, the MACE can construct a basic block of the required format for the request and submit a reception for the reply. The service end, on the other hand, requires two separate MACE functions. The first simply takes a buffer for the expected request and submits a reception request to await the arrival of a block. This is more than a simple basic block reception because the MACE will strip the protocol header off the block and install the rest in the Host's buffer. Knowing this is supposed to be an SSP it can also make various validity checks on the incoming blocks. The second function required at the service end is one to construct the SSP reply and transmit it back to the client.

The BSP OPEN protocol is similar to SSP in many way, so similar that the two can easily be implemented by the same piece of code. It is unlikely, however, that their Host level interfaces will be shared. This is because while there are no side-effects of a successful SSP transaction, there is one of a successful BSP OPEN exchange: a byte stream. The meaning of an OPEN block has been broadened recently to allow it to initiate protocols other than byte streams, it now means that more blocks than just the request and reply will be flowing between the two ends. This change was necessary to ensure that Ring-Ring bridges kept the virtual circuit open for these blocks. This is obviously the case for BSP, but it is also true of other protocols, for example the Fileserver protocol. Since any other OPEN exchanges will be part of other protocols, which should themselves be implemented in the MACE, there is no need to supply a general interface to the OPEN function.

Fileserver protocol is a different case. Only a bona-fide Fileserver will ever need to implement the service end of this protocol, so the MACE need only supply the client functions. At present, however, the 68000 systems do not use the Fileserver directly but access the Tripos filing system via the Filing Machine [Richardson83]. Here it would be more useful to implement the protocol used between this and its clients. Again, most of the 68000s need only have the client functions, but the Filing Machine, which is itself a 68000, must implement the server end of the protocol, and also use the Fileserver protocols.

6.1.4 Naming and Addressing

The previous section deliberately avoided several important questions. Among these are: how does the Host specify the service to be called in an SSP request, and how is an SSP reply directed back to the client. These are specific issues in the general area of naming and addressing.

Services are normally located by mapping a service name into a service description via the Nameserver. This service description comprises a machine address, a port number, a function code and set of flag bits. The flag bits specify such attributes of the service as protocol type, the expected response time, and the basic block type to be used. All four items of this service description must be conveyed to an implementation of SSP to make a call on the service. While this is possible using an indirection in a single machine, it represents a lot of data (six bytes if tightly packed) to pass repeatedly across a machine/machine boundary. A scheme whereby the client supplied all the service information is also open to misuse. This latter could always be prevented by forcing the client to supply the service name with each request, which the MACE could use to obtain a service description itself from the Nameserver. By caching those service descriptions it obtains against the name it could avoid a further Nameserver interaction on later calls to the same service. Since the name must be included in the requesting packet with a pointer, it would have to be read into the MACE via a separate DMA transfer to check against the cache. The service name is likely to be longer than the corresponding service description, so the cost of the interaction is likely to be larger than if the service description were supplied directly.

Instead of the implicit caching described above we use an explicit caching mechanism that not only reduces the quantity of data passed across the interface to a minimum, but provides the protection against abuse we desire. The mechanism is explicit because the Host specifies which services are to be cached with a function that takes a service name and

maps it into a service description via the Nameserver. The Host does not receive this description back, instead it is stored in the MACE as a Service Descriptor (Descriptor from now on), the Host in its place receives a Descriptor Identifier, which need only be a sixteen bit quantity. It is the descriptor identifier that the Host supplies to the MACE when it wishes to send an SSP to the service.

Descriptor identifiers are the virtual addresses mentioned in chapter 5. An analogy may be drawn between these descriptors and capabilities in that they are unforgeable, and possession of a descriptor for a service allows the Host to attempt a call to it. Unlike capabilities, descriptors do not guarantee the service's accessibility in the same way, for example, that possession of a capability for a store segment in CAP [Wilkes79b] guarantees the existance of that segment.

With a descriptor the MACE can trust the service description it is given and can use it to improve the quality of service it provides. The flag bits can be examined and from these it can decide which function code to use, how large a timeout to give the reception request, and even what type of basic block protocol to use. This also constitutes a validity check since it will not allow an SSP to be performed to a service whose description specifies some other protocol. It is possible, however, for the information contained in a descriptor to become out of date. This may be because the service machine has crashed, or in a multiple ring system because a bridge has crashed or deleted the connection through under use. Whatever the reason, some form of recovery action should be attempted. Normally this is the job of the client, which would look the name up again in the Nameserver and retry the request. It is now possible for the MACE to do this transparently on the Host's behalf when it looks likely that the interaction has failed for this reason.

Descriptors also solve the problem of associating an SSP reply with the original request. It is merely necessary for the SSP service function to pass back a descriptor identifier when a request arrives. The only fields of such a descriptor that will be valid are the machine address and the port number. To prevent this descriptor being used for purposes other than generating replies, and to prevent service descriptors being used for the purpose, descriptors can be typed. There are three types of descriptor that can be identified. Request type descriptors specify a service supplied on another machine on the network. Service type descriptors specify a service supplied by this machine, and Reply descriptors represent a return link to the caller of a service. Request and service descriptors are both obtained from the Nameserver and can be

differentiated by comparing the service's machine address with that of the local machine. These types allow a further check to be made by the MACE since it knows what type of descriptor each SSP function will take. There is a problem here if a process wishes to call a service that is implemented by another process on the same machine. Any attempt by it to look up the name will result in a descriptor of service type and not request type. The MACE should therefore allow service type descriptors to be used in the SSP request function, since the only harm the Host could do is to itself.

Beyond the simple lookup function the MACE should supply several further Nameserver interaction functions. It turns out that it is necessary to implement only two extra functions in the MACE to interact with the Nameserver. The first is Reverse Trace which, given a descriptor, returns the name of the machine it refers to, possibly following the path back through several Ring-Ring bridges. The second is Own Name, which simply returns the name of the local machine. This can be turned into a descriptor by using the Lookup function on the name. The remaining Nameserver functions are purely informative and may be accessed via the SSP request function. Finally an information function is supplied to translate a descriptor into its constituent parts. This is no more than the Host could do itself by sending an SSP to the Nameserver, and there is certainly no way that it could use this information to subvert the MACE's protection (remember the Host is denied access to basic block protocol)[1] .

No function is supplied to delete a descriptor to avoid some of the problems associated with the name/descriptor caching. For example, if two separate processes look up the same name they will both receive the same descriptor identifier, and thus access the same descriptor. If one process were to delete the descriptor the other would be left with an invalid identifier. Rather than allow deletion, all descriptors have an associated timeout which is refreshed each time the descriptor is used. The timeout of Request and Service descriptors is quite long, approximately twenty minutes, while that on Reply descriptors is much shorter, about two minutes. The only descriptors that are explicitly deleted are Reply Descriptors after a reply has been sent to prevent them being re-used.

1 The implementation also includes a function to make a descriptor of limited attributes, this was used in testing and is not expected to be made available in a 'production' system.

6.1.5 Byte Stream Protocol

It is from the implementation of this protocol that we expect the major gains to be derived. The interface must therefore be designed to combine maximum throughput with simplicity and flexibility. There are three phases in the life of a byte stream connection: setup, data transfer and closedown. I will consider these in turn.

A byte stream is normally created as a result of an OPEN exchange. It has already been noted that such an exchange is similar to SSP. There are several differences, in particular in the number of arguments required. A BSP OPEN includes a set of BSP-specific parameters between the protocol header and the user data. Just two such parameters are defined at present, which specify the maximum block sizes the client is prepared to send and receive. The OPENACK contains an equivalent pair, which are the sizes to actually be used in the exchange, being in each case the smaller of the values specified by the client and the service. These parameters are really the concern of the stream implementation and are of little or no interest to the Host. There is no need here to expect the Host to supply them or ever be told about them and the interface to BSP can be designed so that it is independent of this consideration. Another side-effect of the OPEN exchange is the communication of port numbers between the two ends to be used by the resulting byte stream.

The ultimate result of the OPEN exchange is the creation of a byte stream. This should only be done if the return code received in the OPENACK, or supplied to the MACE function which generates it, is zero. Any other value indicates that the service has rejected the connection attempt. A byte stream is represented by a 16 bit integer or Stream Identifier which must be quoted by the Host in any operations on it, and will be quoted by the MACE in any communications regarding the stream that it generates.

BSP will be initially incorporated into Tripos by transparently replacing the original BSP handler with an interface to the MACE. Unfortunately Tripos has a different convention for setting up a byte stream to that described above. The BSP Handler Task does not carry out the OPEN exchange, this being the responsibility of the client program. The Handler is only called when the exchange is successful and is handed all the necessary parameters to create the stream. To allow this a second stream creation function is supplied with a similar specification. Once all Host programs are using the MACE OPEN function this could be withdrawn.

The data transfer phase is the most important part of the stream interface and the area where the style of the interface is most important to achieve maximum throughput. Most byte stream implementations expect the client to supply or accept data in buffers of the same size as the negotiated block sizes. Many supply their own buffers of the given size. The reason for this is simple: when the stream gets a buffer from the client it constitutes a 'Buffer Ready' event in the state machine and a RDY or DATA can be sent without further consideration. This is often inconvenient for the client since the final choice of block sizes may have been by the client at the other end of the stream which may have chosen them too small. It may also involve it in copying from its own buffers into the stream buffers and vice versa.

For an example of this consider the case of a program that is reading data from a file and sending it down a byte stream. Filing system transfers are often in quantities of several Kbytes for efficiency. Even if the filing system were able to insert the data into the stream buffers it is more than likely that they will be too small. The client program therefore has no recourse but to maintain two sets of buffers, one set for the filing system and one set for the stream and copy data between them. Some increase in speed can be obtained by multi-buffering, but this really does little more than mask this inefficiency. We really want the stream implementation to accept buffers of any size and split them up if necessary.

That buffers larger than the negotiated block size should be submitted is clear, it is also true that buffers smaller than the block size should be submitted. An example of where this would be useful may be found in a VTP handler. This would normally use a single fixed-format line request that needs to be prepended to any client data to be transmitted, or sent on its own if none was available. This is usually achieved by copying the line request into a stream buffer followed by the client data and dispatching the whole thing, usually with a stream push to force it through to the other end. It would be much simpler if the line request were to permanently occupy a small buffer of its own that was handed to the stream when necessary. The client data, with a push, could likewise be handed to the stream in its original buffer. The MACE will not transmit any data until it has enough to fill a block of the negotiated size, or until a push is performed. The line request on its own will not cause transmission, only when the client data is submitted will the two be composed together in a block and transmitted to their destination (allowing for the constraints of flow control). By similar argument we can say that the input side of the stream will benefit from such a scheme.

Output may therefore be performed by a single command that specifys a buffer of any size containing the data, plus some flag bits. The meaning of these flag bits correspond closely to the bits sent in a DATA command. A reply is only produced when all the data in the buffer has been transmitted, or a higher level event like a stream reset occurs.

There is a slight infelicity in this mechanism that will become apparent if we return to the earlier example of a VTP handler. If it turns out that there is no client data to send it is still necessary to transmit the line request on its own. The buffer has been submitted without the push bit set to prevent it being sent prematurely. Some means must be made available to push any buffered data even when there is no more data to be sent. The simplest way of doing this is to submit a buffer of zero size with the push bit set.

Input can be defined in much the same way, the Host submits buffers of variable size which are returned when filled. A buffer may also be returned partly filled if the last block received into it had the push bit set.

The final phase in the life of a byte stream is the closedown phase. I also include here the Reset exchange that may occur during the data transfer phase. This is because the Close and Reset functions have much in common and are treated in the same way by the protocol package. The following discussion will be based on Reset since this is the more important case. Close is identical with the exception that the stream does not persist beyond the last part of the interaction.

There are two possible sources of a Reset, the client and the service at the other end of the stream[2]. Dealing with the simplest case first: local resets may be brought about by a simple command to the MACE. The stream is reset and the reply returned only when this has been successfully done.

Remote resets are somewhat more complex and highlight an important deficiency in the use of codewords and in the Tripos view of devices. A remote reset may occur at any time and must be communicated to the Host immediately. How should this be done? It could be communicated in the return codes of any buffers that the MACE is holding, and must now return. But the MACE may not have any buffers to return, indeed, the client may be waiting for the reset before submitting any. This mechanism

2 There is a third source, the protocol package itself, but as far as the Host/MACE interface is concerned this is the same as a remote reset.

cannot, therefore, be relied upon. The MACE should be able to generate a message of its own at this point and have it directed to the client. This is where the limitations of codewords become apparent, because they tend to favour a request/reply style of interface. While under most circumstances this is exactly what we need, on occasions such as this it is a definite disadvantage. A request/reply vector interface does not suffer this same limitation since the MACE can generate messages of its own. Here, however, we come across a deficiency in Tripos (and many other operating systems). Once an asynchronous message has been generated by the MACE it must be sent to the correct client task. Under Tripos device drivers are second class citizens and are not expected to generate packets themselves, but merely respond to those sent them by tasks. This simply moves the original problem from the MACE into the Host machine.

The solution to this bears some relation to the up-call mechanism mentioned in chapter five. At stream creation time the client supplies the MACE with a packet that is set up to find its way back to the client. The MACE saves this packet until a remote reset occurs, when it generates a reply to send this packet back to source. The client acknowledges receipt of the reset by returning this same packet to the MACE, restoring things to their normal state. Only one such packet is needed because the MACE will not send the return reset to the other end until the client has replied, any further reset received can only be retries. This depends on the client returning the packet quickly to meet the real-time constraints of the reset exchange. To prevent the stream being hung up on this the MACE can apply a timeout to the Host, in the same way as it applies one to the remote end of the stream, and close it if the packet is not returned.

Identical mechanisms to this can be applied to closing the stream. The principal difference is that once the exchange has completed the remote reset and close packets must be returned to the client so it can safely release the memory they occupy.

Whenever a reset or close occurs all the buffers being held for that stream by the MACE must be returned. Each buffer is returned with a return code indicating the reason. It is intended that these return codes are purely informative, and should only be used to determine the fate of the buffer and not that of the entire stream. The Host should not use detection of such a return code to, for example, reset or close its representation of the stream. It should only do these things when the reset or close packet, or the reply to its request arrives. The MACE undertakes to return all buffers before this packet is sent, so the Host can process it in a consistent state. It is possible for a buffer to be on

its way to the MACE when a remote reset occurs and will arrive when the MACE is waiting for the client to acknowledge the reset. This buffer cannot be treated as normal because the stream is in the wrong state, so it is returned immediately with a return code to this effect. Similar conditions can arise during a remote close sequence, but here the buffer can be returned with an invalid stream identifier indication.

6.2 The Ring Interface

SuperMACE supports exactly the same loading and debug commands as the SPECTRUM program. This is purely for compatability reasons as it allows the Host to be loaded and debugged by exactly the same mechanisms as before. These functions are, however, totally unprotected and do not conform to any standard protocol. If the MACE is to impose strict rules of protocol on its Host it should conform to the same rules in its own external interfaces. We must therefore introduce a new set of primitives.

Dealing with authentication first: this is achieved in Cambridge by UID-sets [Girling82]. Enhancements have been defined to the existing protocols to allow the inclusion of such UID-sets [Johnson82]. By employing this mechanism the MACE can ensure that it is only loaded and debugged by those remote clients who can present UID-sets with the necessary privileges.

The original reason for inventing the Ancilla was that the Type 2 ring interfaces were not sufficiently powerful enough to go to the Fileserver themselves to get the load file. This is certainly not true of the MACE which is fully capable of this (the MACE alone is arguably a more powerful machine than the Ancilla!). Therefore, in response to a suitable SSP request the MACE should go to the Fileserver and load the Host from the specified file. The reply should be generated only when this has been done. Ignoring for the moment its continued need by the Type 2s, the Ancilla has now become redundant. Its only remaining function is to convert the names of load files into Fileserver FUIDs and this could easily be subsumed into the Resource Manager, where one might consider it to genuinely belong.

If the debugging primitives were simply converted to authenticated SSP exchanges it would be necessary to interact with the AOT manager for each one. This is clearly an unacceptable overhead. The alternative that first comes to mind is to perform an initial authenticated interaction to which, if it were satisfied, the MACE could reply with a randomly generated connection identifier. The debugger could then use ordinary SSP requests,

quoting the connection identifier in each, to access the Host. This is a waste of effort when we have a protocol that can not only give us a secure connection, but has the additional properties of being error free and flow controlled: BSP. The debugger can therefore open an authenticated byte stream to the MACE through which they can exchange commands and data. The advantages of this are probably only slight for interactive debugging, where the user will only be reading small parts of the Host's memory at any one time. The mechanism, however, also allows dumps of the machine state to be taken easily. The debugger, or dump program, need only give the MACE a single command to read the entire memory of the Host. The MACE will deliver this only as fast as the debugger can take it because of the flow control in the stream. Using any other method the debugger would have to do this in many smaller transfers since it would be unable to accept all this data at ring-speed.

Chapter 7

The Implementation of a High Level Interface

This chapter is a general overview of the internal structure of the High Level Interface implemented in the MACE. It examines some of the problems involved in moving protocols to an interface processor and describes the solutions found. At the end of this chapter we give some comparative performance figures and draw some conclusions.

7.1 Modules and Tasks

The complexity of the software involved in a High Level Interface is such that it would be impossible to implement it as a single monolithic program. It was also originally intended that the code be used for other 6809 based interface processors (the GIZMO and the MACE2) with the minimum of re-writing. It must therefore be modularised both in the interests of manageability and portability. The MACE is expected to be running many independent activities concurrently, typically several byte streams, random SSP interactions and the more mundane tasks of dealing with Host requests and replies. It is therefore essential that the basic environment be multi-threaded, so multi-tasking of some form is required.

Conventionally there are two ways of dealing with a transaction in a multi-tasked system. The first is to assign a task to the job that makes the necessary module calls to serve the transaction. This task is either created for the purpose or obtained from a pool. The second way is for each task to perform a specific function and for the transaction to be passed between them by message passing. The choice here is between a procedure based or message based system; Lauer and Needham [Lauer78] have shown that these are equivalent. The first mechanism is unsatisfactory because it can lead to a profusion of tasks. In general a separate instance of each module for each task is necessary to avoid synchronisation problems, and where modules must be shared a monitor mechanism is needed. The second suffers from the expense involved in a message passing system. We would really like to mix the two approaches: eliminate message passing but keep the number of tasks and module

instances in the system under control (static if possible). This can be done by inverting the conventional structure, making modules the principals in the system and relegating the tasks to a secondary status. A transaction can now be processed by simple external procedure calls between modules. Internally a module may contain one or more tasks to handle asynchronous events, or to serialize its activities, but this need not necessarily be the case. This may be summarized by saying that while in conventional systems a task will contain several modules, in this system a module may contain several tasks.

This model bears much similarity to, and was influenced by, that expounded by Clark [Clark82a]. The principal difference is that whereas Clark retained messages for inter-task communication and synchronisation, no such feature exists here. Having said that it must be admitted that there remains a need for message-like objects. These Control Blocks are needed for several reasons. First, since the entire system in written in assembly code, and there are often too many parameters to pass in registers, they must be passed in store with a pointer in a register. Secondly, there is a need for data to persist beyond a single procedure call, for example, the address of the requesting packet must be passed through to enable the reply to be generated. Thirdly, since a module may execute in several different tasks while servicing a request, the parameters cannot be saved on the calling stack. This does not really constitute a message passing system since an individual module can handle control blocks in an application dependant manner, and while conventions exist about the format of control blocks, no general mechanism exists. Control Blocks are used, therefore, to represent a single transaction, being passed from module to module as the need arises. This also means that if the same parameters need to be passed on to another module, this same control block can be used, avoiding copying. Modules can place their own private data, including pointers to other control blocks, after the end of the defined part of the control block. Control Blocks, it should be stressed, are by no means universal, there are many modules that have a straightforward procedure call interface.

The normal model of inter-module interaction is that the calling module does not regain control until the called module has completed. In a multi-tasking environment this means that the calling task must be suspended until the call completes. In a real-time environment events may occur that require that task to perform some other duty before the call completes. External procedures should therefore be defined to return control in a finite time. For some modules the result will be available in this time, so the return is the end to the interaction. Other services, however, cannot

be completed so rapidly, for example, ring interactions may take an unknown period to complete. The computation performed in the external procedure should therefore merely initiate the service before returning. The procedure may do this itself, or it may schedule an internal task of the module to do it.

Some time later the module will have finished and will need to supply a result to the caller (this usually means returning the control block). In message passing systems this is achieved by sending the message back to its originating process. This is not what we require, the original task may no longer have any connection with the request, and in any case we are attempting to avoid this. We really want to send the result back to the originating module. This can be achieved by including among the parameters in the control block a pointer to a procedure in the calling module. To return a result the called module need merely call this procedure. This is Clark's up-call, although I favour the term Reverse Call since 'up-call' implies a hierarchial structure, which is not always the case, and the reverse call need not be made back to the calling module.

Reverse calls are a powerful mechanism and can be exploited in several ways. The most important development is to include several reverse call pointers in a control block, these can be used to represent different completion modes, or can be used by the called module to gain further information from the caller. Reverse calls are, as far as the called module is concerned, the same as its external procedures, and the same real-time constraints apply.

Tasks are used by modules in two ways. First, they are used to allow a module to sequence its processing of events and avoid synchronisation problems. In these cases the external procedures and reverse calls tend to be simple, merely asserting an event flag or incrementing a counter, and scheduling a task to run. The other purpose is to simply move the locus of control from the calling task to that of the module. This is used mainly by those modules that deal directly with interrupting devices driving modules. Reverse calls are made from the device driver into these modules in interrupt state. To avoid making any further reverse calls here, and possibly upsetting any real-time constraints on the currently executing task, these reverse calls are as simple as possible and usually just schedule a task to make any subsequent calls. This means that above this level all reverse call procedures can assume that they are executing in a task that has voluntarily called them, and has not had the reverse call thrust upon it by being unfortunate enough to be executing while an interrupt occurred. Some modules exploit this to avoid owning any tasks

themselves, doing all their work either in the external procedures or in the reverse calls, even to the extent of making further external and reverse calls.

An example will show how this works in practice. The modules mentioned will be described in more detail later, we are interested only in the communication mechanisms at present. Consider a Nameserver lookup. These are handled by the module DESC, which is given a Lookup Control Block (LuCb) by the client containing the name to be looked up plus a reverse call procedure. In its external procedure DESC allocates an SSP Control Block (SspCb) and fills in the necessary fields, including the service descriptor identifier (for the lookup function), a reverse call procedure, the name and two buffers for the reply each carried by its own Buffer Control Block (Db, for historical reasons). A pointer to the LuCb is installed in a spare field beyond the defined end of the SspCb. This is submitted to the SSPREQ module whose external procedure places the SspCb on an internal queue, increments a counter and awakens the module's internal task. The SSP external procedure now returns, as does the external procedure of DESC so the client regains control.

A short time later the SSP task will be run. It takes the SspCb off the queue and using the information found there, particularly that found in the service descriptor, fills in its two Ring Control Blocks (Cb's, again for historical reasons), one for transmission and one for reception. These are submitted to the the respective ring driving modules and the SSPREQ task waits for the reception request to complete. The actions of the ring drivers will be described later.

When the reply has been received, or the request timed out, a reverse call is made to a procedure that reawakens the SSP task. When the SSP task restarts it examines the return codes in the Cbs and sets an appropriate one in the SspCb. It then makes the reverse call in the SspCb back to the DESC module. This uses the data in the two reply buffers, which are arranged so that the lookup data goes in one and the extended or transformed name from the Nameserver goes in the other, to construct a descriptor. The Descriptor Identifier is installed in the LuCb and the reverse call in that control block is taken back to the client module. When that call returns the DESC module can tidy up, releasing the SspCb and the three Db's and itself return to the SSPREQ module, which can now proceed to the next SSP request.

This description ignores descriptor caching, so before the DESC module does anything else it searches the cache. If it finds the name in the cache it places its identifier in the LuCb and invokes the reverse call. It is therefore possible for a module to receive the reverse call from a module it is invoking even before the external call has returned.

7.2 The Modules

This section describes the various modules that make up the High Level Interface. There are about thirty separate modules comprising about 15K bytes of code and static workspace. These will not all be discussed. A schematic representation of the system is shown in figure 7.1, the arrows represent only the external calls.

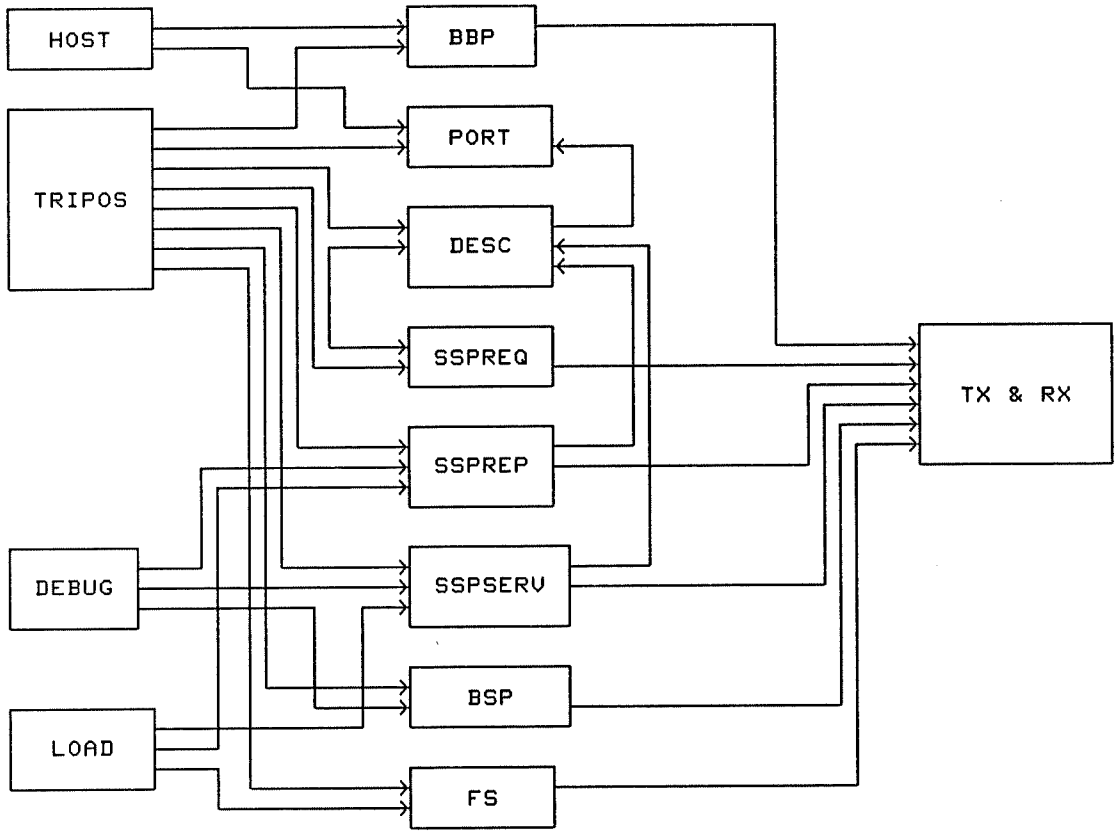


Figure 7.1 SuperMACE Modules

7.2.1 The Operating System

The operating system consists of four modules: COORD the task coordinator, STOREMAN and SMMAN the store managers, and CLOCK the clock and interrupt handler.

The coordinator supports a simple multi-tasked environment. A task consists of a Task Control Block (Tcb) and a stack, which are allocated in the same memory segment. Each Tcb contains four fields: a link word, a state byte, the value of the stack pointer when it is suspended, and the base address of the stack/Tcb memory segment. All the tasks in the system are linked in a circular chain and rescheduling consists of searching forwards along the chain for one that is runnable. Tasks may be in one of two states: suspended, represented by a zero in the state byte, and runnable, represented by any other value. The coordinator has three external procedures that affect scheduling. The Suspend entry zeroes the state byte of the current task and enters the search loop. The Release entry decrements the state byte of the given task ensuring it is non zero and returns; note that this does not cause entry to the search loop, so the current task remains in control and it may be called safely from interrupt routines. The Pause entry simply enters the search loop without suspending the task. This causes the coordinator to proceed round the task chain, running any tasks, until it returns to the pausing task which, still being runnable, is re-entered. This is used by tasks that need to busy wait, or to force other tasks to run without themselves being suspended.

Once a task is running it cannot be pre-empted and the coordinator will only be re-entered by an explicit call of Suspend or Pause. This is exploited in the coordinator by not saving any task state beyond the program counter, condition codes and stack pointer. All tasks arrange to suspend or pause only when they have little or no state to save, which makes task switching fast. Also, since interrupts have no effect on scheduling beyond marking a task to be run the coordinator may be executed with interrupts enabled. Indeed, since the task search loop is where the system will idle it is essential that this is executed with interrupts enabled.

Because of the simplicity of this coordinator tasks must take care when deciding whether to suspend. If suspending depends on some condition that is set by an interrupt routine the test and suspend must be executed with interrupts disabled to prevent race conditions. If the condition is set by another task no such precaution is necessary.

There are two levels of store management supported by the system. At the lowest level is STOREMAN, which is based on the algorithm in section 2.5 of [Knuth73]. It manages large memory segments and is typically used for obtaining task stack/Tcb segments, ring reception buffers, large control blocks and character strings. The second store manager, SMMAN, is designed to manage small memory segments (3 to 63 bytes) efficiently;

it is used for allocating control blocks. SMMAN never coalesces free store, instead any freed blocks are added to a queue of blocks of the same size; there are, therefore, sixty such queues. When asked for a block of a given size it first looks at the relevant queue for a free block. If the queue is empty it carves a fresh one out of a larger memory segment obtained from STOREMAN for this purpose. If there is insufficient room in this segment it obtains a new one. The theory behind this is that after a short time the MACE will gather a "working set" of small blocks and SMMAN will always find a block on the queue.

The final operating system module is the CLOCK module. This is really only part of the operating system because the MACE routes all interrupts through the same vector. The clock interrupt detects Ring and Host interrupts and passes them on to the relevant routine. The clock itself maintains a pair of short duration timers for the Ring drivers and a 50 Hz clock for general use. Ticks of the 50 Hz clock have the effect of scheduling the clock task to be run. Modules may obtain the services of the clock by submitting a Timer Vector (Tv) containing an enable flag, a countdown and a reverse call procedure. Each time it is awakened the clock task examines the chain of Tv's and decrements the counter on each one that is enabled. If any counter goes to zero the reverse call in that vector is taken. Because the Tv can remain accessible to the originating module this mechanism makes it extremely cheap to enable or disable a timer or change its counter.

The Operating System of the MACE only supports multi-tasking and store management. This does not mean that a module must do everything else itself as modules can be supplied to handle such common activities as queue management, synchronization and event handling; it would even be possible to implement a Tripos-like message passing system, although this has not been done.

7.3 The Ring Drivers

The MACE Ring driving modules, TX and RX, present a somewhat novel interface to the ring. This interface is designed to allow protocol packages to exert some control over the reception or transmission of basic blocks. To see why this is necessary consider the reception of a BSP block. The first four bytes of the block are protocol information and the final destination of the rest is dependant on these. For example if the input command were DATA the rest of the block must be put into data buffers, unless the control, or qualifier, bit is set in the command, in

which case it must go into a different set of buffers. If the first command is RESET, CLOSE or EXPEDITED [JNT82] the rest would have to be put in yet another set of buffers. If we are to avoid any unnecessary copying of data it is essential that it be put in the correct buffers from the beginning, which can often only be done after some of the block has been received. It is wrong for the ring driver itself to interpret the format of a protocol specific block, so the reverse call mechanism is used to ask advice of the protocol handler.

To see how this works, consider a ring reception. A reception is initiated by a client submitting a Cb to the RX module. This control block contains the station and port numbers on which to listen, a timeout, plus three reverse call procedures: Fail, End and BufferRequest. It also contains a number of other fields whose use will be made clear later. This control block is added to one of sixteen chains depending on the bottom four bits of the port number.

When a basic block comes in the reception driver receives the header and port numbers and searches for a matching control block. If one is not found the reception is aborted, otherwise it checks the Immediate Data Size field. If this is non-zero it receives that many bytes into another field in the control block before calling the BufferRequest procedure. The client module now briefly regains control and using the data already received can perform any validation or make any decisions necessary. When this procedure returns it must tell the ring driver either to abort the reception or where to put the rest of the block. This last is done by returning a Buffer Control Block. The ring driver proceeds to insert the rest of the block into this buffer until the either the buffer is full or the block is finished. If the buffer fills up before the block ends RX makes a further call on the BufferRequest procedure. It continues in this manner until the block is finished. If the client module has performed validation or made decisions on the first call of BufferRequest it is unlikely to want to do this again. To avoid this it can use two BufferRequest procedures; when the first is satisfied with the block it can replace the pointer to itself in the Cb with a pointer to the second, which can be a much simpler routine. When the block has been successfully received the return code is installed in the Cb and the End procedure called. The control block is not removed from the port chain by the driver, if this is required the End procedure must do it explicitly via an external call. This is an optimisation that allows protocol handlers to leave control blocks in the chain and avoid the cost of linking and unlinking them. An Inhibit bit in the control block prevents the driver considering it when searching for a match.

If the reception fails or the request times out the Fail routine is called with an appropriate return code. Although the definition of BBP does not strictly allow clients to be told about failed receptions this must be allowed here since the BufferRequest routines may already have committed resources to the block and the client module must be given a chance to tidy up.

The interface to the transmission module is almost identical to this. The transmitter has no need to perform validity checks or decide which buffers to transmit out of; such things can be decided before starting the transmission. The same interface is retained because of its flexibility and because it allows the same data structures to be used throughout the system. Consequently data may be transmitted directly from the control block, reverse calls are made to obtain further data, and control blocks may be left with the transmitter in an inhibited state. Internally control blocks are kept on a circular chain around which the transmitter proceeds, ignoring the inhibited ones. The header of a block is retried sixteen times in the face of errors each time the Cb is encountered and if it does not get through a count in the control block is decremented and the transmitter proceeds on to the next Cb. If the count reaches zero the transmission is deemed to have failed. This strategy has the effect of spreading the retries out in time, increasing the possibility of the header getting through while preventing the MACE from wasting too much time on unsuccessful retries when transmissions to other stations may get through. Packet level retries of this sort are handled by the driver, block level retries must be handled by the client module. Retries can be accomplished in the Fail procedure simply by clearing the inhibit bit in the Cb.

The parameters of a buffer are its size, base address and type. There are two basic types of buffer: Local and Host, which indicate whether the buffer is in the MACE's memory or the 68000's. The size and address are always expressed in byte quantities, this allows buffers of both types to be manipulated by higher levels in a type independent manner. It is only in the ring drivers that they are finally differentiated. This causes a problem for the MACE when the buffers are in the Host, because not only is the Ring a 16-bit device, but the DMA hardware works in 16-bit quantities. For full generality buffers must be allowed that start on odd or even addresses and are an odd or even number of bytes long. This also implies that a buffer may run out, and the next start, in the middle of a ring packet. Combining these three gives eight possible cases to be dealt with. Fortunately this does not mean that eight separate sections of driving code must be supplied. The transfer of the bulk of the block may

be reduced to two cases, that where packet and DMA boundaries agree, and that where they are staggered. This leaves the end effects to be dealt with. These too can be reduced to a smaller number of cases, some of which are only trivially different and can share code. Unfortunately no two combinations of beginning, middle and end cases turn out to be identical. The driver must therefore determine the current combination when starting a buffer and look up in a table the correct sequence of routines to deal with it. Access to local memory is much easier and is byte oriented, so handling the cases for local type buffers reduces to a couple of tests at the beginning and end of a transfer.

7.3.1 The Protocol Handlers

There are six separate protocol handling modules in the system. At the lowest level is BBP, the Basic Block Protocol module. This presents a much simpler interface to the Ring Drivers and will handle block level retrys and multi-block transfers. It is not used by any of the other protocol handlers and merely exists to serve the Host's basic block requests.

At the next level are three modules to deal with SSP protocol. SSPREQ handles SSP requests, SSPSERV waits for SSP requests to arrive and SSPREPLY generates SSP replies. The interface to these modules is designed so that they can generate OPEN or OPENACK transactions, using the protocol type field in the descriptor to decide which. This facility is only used internally. These are all multiple instance modules, and it is customary for each client to have its own private instance.

Also at this level is the Fileserver protocol module, FS. This only implements the READ and WRITE operations, these being the only ones that do not conform to normal SSP. If the MACE were to implement the full set of Fileserver primitives it is best done via a module that makes use of SSPREQ and FS.

The final protocol handling module is BSP. This is divided into four sub-modules according to function. BSPEXT contains not only the external procedures of the module but the reverse call procedures invoked by the ring drivers and clock handler. Most of these routines perform a simple manipulation of the Byte Stream Control Block (BsCb), set an event flag and release the modules internal task. BSPROOT contains the main loop of the task, examining the event flags each time it is awakened and performing any necessary actions. Among the things implemented here are the stream RESET and CLOSE interactions and the decoding of commands in received basic blocks. The remaining two modules, BSPIN and BSPOUT, are the input

and output state machines. This implementation of BSP will be covered more fully in a later section.

7.3.2 The Descriptor Module

Descriptors are handled entirely by the module DESC, which is also responsible for Nameserver interaction. The only real sources of descriptors are the Lookup function and the SSPSERV module, which must generate reply descriptors. All descriptors are entered into a hash table indexed by their descriptor identifier. Additionally all descriptors obtained via Lookup are entered into a second hash table indexed by the name under which they were looked up. The Lookup function first tries this table to see if it already has the name and if a match is found returns the given descriptor identifier. An external procedure is supplied to create a reply descriptor for use by the SSPSERV module.

Descriptors are always referenced by their identifier, both by the Host and internally by the MACE itself. When a module, for example SSPREQ, needs to obtain the service information it calls an external procedure, DescFind, to look up the identifier in the hash table and return a pointer to the descriptor. The client can now extract the information it wants but it will not retain the pointer; if it needs this information again it will repeat the mapping. The principal reason for this is that a side effect of DescFind is to refresh the timeout on the descriptor. Descriptor timeouts are decremented once per second by a routine called from the clock handler.

A descriptor identifier is a sixteen bit value. This address space is sub-divided into a 'system' area from 0 to 1023 and a 'user' area from 1024 up. System descriptors are those that the MACE has looked up for its own purposes, and User descriptors are those that the Host has looked up for itself. At start of day there are four system descriptors available, for the Lookup, Reverse Trace and Own Name entries of the Nameserver, and for the local machine.

7.3.3 The Host Interface

This is in the hands of two modules: HOST and TRIPOS. The HOST module contains two tasks plus an interrupt routine. The interrupt routine, when called, sets a flag and releases the Request task before returning. When the Request task runs it reads the Host request vector into a fixed area and decodes the function. HOST implements the interface described in Chapter 4 so if the function code turns out to be one of

those it submits the necessary request to the BBP module. A fourth function code is now also possible, which causes the HOST module to hand control over to the TRIPOS module.

The TRIPOS module is only concerned with the first four bytes of the request, the first is the function code, and the next three are the BCPL address of the Tripos packet that caused this request. It uses this to obtain the function code from a known offset in the packet. This is looked up in a hash table from which it gets the number of argument bytes the function requires plus the address of a procedure to handle it. It reads all the required arguments into a fixed buffer and then calls the procedure. This will set up all the necessary control blocks and call the necessary modules before returning. Some time later a reverse call will be made back into the TRIPOS module to generate the reply. This procedure can free all the control blocks and construct a reply in a Reply Control Block. Replies consist of the address of the originating packet[1], plus sixty four bits of results, which will be installed in the two result words of the packet by the Host device driver. This control block is placed on an internal queue in the HOST module and the reply task awakened to deal with it.

7.3.4 Miscellaneous Modules

In addition to the major modules described so far there are several lesser, support, modules that require a mention.

Once the coordinator and store manager have set up the environment the first module entered is ROOT. This is responsible for getting the system going by calling the initialisation procedures of all the modules that require it. The simplest way of destroying any state information the MACE contains between reincarnations of the Host system is to simply restart it. All the initialisation procedures must therefore fully initialise the static state of of each module. Some things are not initialised in this way, the last port number used and the last user descriptor identifier in particular, so the new incarnation will not use the same ones as the old.

The modules LOAD and DEBUG implement the loading and debugging primitives described earlier. These constitute a second command interface driven by the ring rather than the Host, and consequently work in much the same way as TRIPOS.

1 The request handling procedure had to ensure that this was preserved through to this point. This is usually done by tacking it onto the end on a control block.

7.4 BSP and the Intelligent Interface

There are a number of problems associated with implementing BSP in an interface processor in an efficient manner. Most of these arise from the desire to avoid copying and to place the data in the Host where it will be of most use to the client: in its own buffers.

The general buffer handling scheme employed by MACE BSP has already been described. The MACE will only send a DATA command when it has at least enough data in hand to fill a buffer of the negotiated size; it will send a NODATA when the total drops below this threshold. Similarly it will only send a RDY when the total outstanding buffer size is above the negotiated block size and will send a NOTRDY when it falls below. This is essential in the input case, since there is no way of knowing what size block the other end may send so the MACE must be ready for the largest. It is not, however, necessary in the output case, and the MACE could legitimately send blocks smaller than the negotiated size if it has the data in hand. This is not done partly because it can waste bandwidth and partly because it may be useful for the client to build a single block out of smaller buffers by submitting them one at a time to the MACE. An example of this might be marshalling the parameters for a Remote Procedure Call. It is perfectly within the MACE's rights to save data in this way since if the client does want the data sent it could issue a push. These considerations result in a slightly altered state table, shown in figure 7.2.

This threshold mechanism can result in an unfortunate effect on the reception of data. The MACE will block the flow of data with a NOTRDY when the size of its buffer pool falls below the threshold. If this is part of a pre-negotiated transfer for which the Host has submitted buffers of precisely the right size then the quantity of data that the transmitter has to send will exactly fit in the buffers. It will never be sent, however, resulting in a possible deadlock, even if the transmitter has a push behind it. This is a fault of BSP, which implements flow control in block units whereas the Hosts often want to do it in byte units. To avoid this the Host could lie to the MACE about the size of the last buffer, rounding the total up to a multiple of the negotiated block size. If the transmitter is not going to send more than the Host expects this is safe, since the extra bytes in the buffer will not be over-written. But if the acknowledgement to the block before the last is lost, the transmitters retry will fill up the entire buffer, including the extra 'phantom' bytes. The only safe way of doing this is to round the buffer size up to the next multiple, or add an extra dummy buffer at the end to do the rounding.

Event	State		
	E	N	I
E[rep] DATA(n-1)/ RDY(n)	Retransmit RDY(n)/DATA(n)	Retransmit NOTRDY(n)/ NODATA(n)	Protocol Error
E[exp] DATA(n)/ RDY(n+1)	Return Buffers n +:= 1 BuffSize<BlockSize? no: Transmit RDY(n)/ DATA(n) Goto E yes: Transmit NOTRDY(n)/ NODATA(n) Goto N	Protocol Error	Return Buffers (Input only) n +:= 1 BuffSize<BlockSize? no: Transmit RDY(n)/ DATA(n) Goto E yes: Transmit NOTRDY(n)/ NODATA(n) Goto N
N[exp] NODATA(n)/ NOTRDY(n+1)	Return Buffers (Output only) Goto I	Protocol Error	No Action
Buffer Arrived	No Action	BuffSize<BlockSize? no: Transmit RDY(n)/ DATA(n) Goto E yes: Transmit NOTRDY(n)/ NODATA(n) Goto N	No Action
Timeout	Retransmit RDY(n)/DATA(n)	No Action	No Action
Idle Handshake Timer Expires	Protocol Error	Protocol Error	Transmit RDY(n)/DATA(n) Goto E

Notes:
n is the block sequence number (mod 16).
BlockSize is the number of bytes in a block of the negotiated size.
BuffSize is the number of bytes of buffering in hand.
Whenever a buffer arrives or is returned BuffSize is adjusted accordingly.

Figure 7.2 Modified BSP State Transition Table

Another problem is concerned with the possible events following the loss of a basic block. Consider two machines, A and B, running a byte stream between them. A sends a DATA command, which fills up all B's buffers causing them to be returned to the client. B sends back a NOTRDY, which gets lost and never arrives at A. The timeout in A will eventually expire and A will retransmit its last block, the DATA command. In conventional implementations B would receive this block into a standard buffer, identify it as a repetition and retransmit the NOTRDY in response. It can do this because it always keeps this fixed buffer ready , and copies data from it into client buffers as necessary. The MACE, on the other hand, transfers

data straight into the client's buffers, and if it has none of these in hand it cannot receive the block. It will therefore abort the reception of this block which will cause A to retransmit again after a timeout. This will continue until either B's client submits more buffers or A gives up and shuts the stream down. Note that if there are buffers available the repeated block will be received into them successfully, but the next proper DATA block will overwrite it with no harm done.

The solution to this is to define an extra buffer type to be give to the RX module in the BufferRequest procedure if it ever runs out of buffers. This buffer is a sink and tells the device driver to throw the data received away. This is quite safe because A should never send genuine data unless there is enough buffering to have caused B to send a RDY command. The only important part of the block as far as the BSP module is concerned is the first four bytes, and these have already been received into the control block when the first BufferRequest call is made.

Another area where the buffering strategy causes problems is in the treatment of control, or qualified, data. BSP has the facility to mark the data in a block with a 'control' or 'qualifier' bit, allowing it to be treated specially by the clients. If a single set of input buffers are used it is possible for ordinary and control data to be mixed in a single buffer. To prevent this a separate set of control buffers must be supplied by the client. When transmitting these buffers are simply put into the data stream and care taken not to mix them with ordinary data. Effectively this means that a control buffer has an implicit push in front of it and one behind (if the client is doing things properly these pushes will be present anyway). On the input side the control buffers must be kept in a separate queue and data diverted to them when the control bit is detected in the BufferRequest procedure. These buffers cannot contribute towards the total size of buffering available because of the possibility that only ordinary data will be sent. The MACE may find itself in a position where it has enough control buffers to accept a block but insufficient ordinary buffers and must send a NOTRDY. The data flow will be blocked, even if the other end has a block of control data to send next, which could, in theory, be accepted. The only way to avoid this is for any client that expects control data to give the MACE enough ordinary buffers so as not to block the flow.

Most BSP implementations handle RESET and CLOSE by adding extra states to one of the data flow state machines and disabling the other during the exchange. This somewhat obscures the true structure of the augmented state machine and since RESET and CLOSE are relevant to the

whole stream this is the wrong place to handle them. MACE BSP uses a higher 'Super State' which is driven directly by the events that occur in the stream. Some, like InputBufferReady translate directly into events in the relevant data flow state machine. Others, like BlockReceived or IdleTimeout, cause events in both state machines and others, like RemoteReset, have no direct equivalent. The result of this is a simple and elegant protocol handler.

7.5 Host Software

The High Level Interface required some changes to the Host's software. Several problems arose from this.

7.5.1 The Ring Device Driver and Handler Task

The original device driver for the 68000 was designed to drive the request/reply vector interface of SPECTRUM. It did this by copying certain field of the Tripos packet sent to it into the Request Vector before interrupting the MACE, and copying fields out of the reply vector into the packet on the return interrupt. The driver also put all packets sent to it onto an internal queue, and searched this queue for the packet when a reply was generated. This was primarily a debugging aid.

The driver was converted to interface to the new MACE program largely by deleting the argument copy and the packet queue. Now it simply places the address of the requesting packet in the request vector before interrupting the MACE. When the MACE interrupts back the reply vector will contain the packet address plus the two reply words. These are installed in the packet, which is usually returned to its origin. The MACE is able to specify in the reply that the packet is not to be returned to origin but is to be dropped. Here the driver sets the destination field to point back to itself and sets the link field to NotInUse. This is used solely to implement the Ring Handler compatible cancellation feature, which is defined to cause all cancelled reception request packets to be dropped in this way.

The removal of the Ring Handler Task presents a couple of problems. First, the task identifier of the ring handler is obtained by client programs from a well known location in the operating system's data structures. The TaskId is normally placed there by the handler itself when it starts. In the new system we want to place the device identifier of the device driver there instead. The driver could place it there in its

initialisation routine, but since this number is fixed when the system is linked it is simpler to initialise this location then. Because the ring handler was a well known task it was also used to locate the BSP handler task. When it started up the BSP handler would plant its TaskId in the Ring Handler's global vector. Any task that needed to communicate with it would examine this same global location. An alternative to this must now be found. There are two possible mechanisms; the first is to allocate another well known location similar to that used to find the Ring Handler. A more aesthetically acceptable mechanism is to set the BSP handler up as a pseudo-device and use the standard mechanisms for finding it.

7.5.2 The BSP Handler Task

To allow Tripos to use the MACE implementation of BSP with the minimum of alteration an interface must be supplied that will transparently replace the existing BSP Handler. The most desirable state of affairs is to remove the BSP Handler Task entirely and run the byte stream directly from the client task. Unfortunately this is not possible since there are some asynchronous events (Remote RESET and CLOSE, and returning buffers) which, owing to the limiting nature of Tripos' message passing system, could not be handled transparently. A BSP Handler Task is therefore still required.

To the client this BSP Handler present an identical interface to the original, based on Tripos streams. Indeed most of its effort is directed to this end rather than interfacing to the MACE. Internally the handler is entirely event driven by the arrival of packets from the client and from the MACE. The actions prompted by a packet are determined solely by its type, and possibly its return code; there are no timers or state machines involved.

MACE BSP can easily be used directly from a client program if it is written to do this. To this end a small library of interface routines have been written to aid in this. The most important feature of this library is that it must be allowed to take a look at all incoming packets before the client so that it can filter out any that belong to it.

7.6 Performance

This section gives the results of some performance measurements.

7.6.1 Basic Block Performance

To allow comparison of raw throughput between the High Level Interface and the previous two interfaces the same experiment has been performed of transmitting one hundred full sized basic blocks to various destinations. Some of these figures are reproduced from chapter 4.

Source	Destination			
	SINK	SPECTRUM	Type 2	SuperMACE
SPECTRUM	341	321	341	334
Type 2	367	353	356	355
SuperMACE	318	288	318	315

Figure 7.3 Comparison of Raw Throughput

These show that the High Level Interface is slower than SPECTRUM by about the same factor as that program is slower than the Type 2. This reduction is not really surprising since SPECTRUM is optimised to an extreme extent and could not maintain this rate if it had to do high level protocol processing as well.

7.6.2 SSP Performance

That connectionless protocols benefit from being moved into the MACE can be demonstrated with a Nameserver interaction. Four experiments were carried out to lookup an unknown name ("XYZ") in the Nameserver using different mechanisms. The reason for looking up an unknown name was, first, to prevent it being cached by the MACE Lookup entry and, second, because the Nameserver responds more rapidly. A fifth experiment supplied a valid name to the MACE Lookup function to demonstrate the effect of caching. The results are given in figure 7.4.

Method	Lookups per second	Milliseconds per Lookup
Tripos SSPLIB Lookup.Name Function	88	11.36
DIY using Basic Blocks	155	6.45
MACE SSP Function	185	5.40
MACE Lookup Function	159	6.29
MACE Lookup with name in cache	400	2.50

Figure 7.4 SSP Times

It can seen that the MACE functions are consistently superior to the Host. The form of the experiment is to time how long it takes to do five hundred Nameserver interactions; experiment two achieves its figure by setting up its buffers and packets before timing starts and re-using them each time whereas the others treat each lookup as a separate instance. Experiment 2 is a substantial program while the others are essentially a single statement to the client.

7.6.3 BSP Performance

The first experiment here involves driving BSP via the standard stream interface using the BCPL wrch and rdch routines. This is exactly how all clients of BSP use it and the results of this experiment show what practical improvement can be expected. Three experiments were conducted in each of the four possible arrangements of BSP Handler Task (BSPH) and MACE BSP. Experiment 1 was to transmit 100K bytes in full sized blocks, containing 2044 bytes of data each, resulting in the transmission of 51 blocks. Experiment 2 was to transmit 100K bytes in 64 byte blocks which each contained 124 bytes and resulted in the transmission of 826 data blocks. Experiment 3 was to transmit 1652 bytes in three packet blocks. This last resulted in the same number of blocks as in experiment 2 but with the minimum number of bytes of data in each. The purpose of this experiment was to discover how quickly the protocol implementations can deal with protocol events while not obscuring the figures with copy or transmission times. Remember that for each data block sent an acknowledgement must be transmitted in the opposite direction. The times taken did not include the time to open or close the stream. The results are shown in figure 7.5 converted to Kbits/second in experiments 1 and 2, and to block exchanges per second in experiment 3.

	Source	Destination	Experiment		
			1	2	3
A	BSPH	BSPH	35	21	53
B	BSPH	MACE	45	31	86
C	MACE	BSPH	37	22	66
D	MACE	MACE	77	66	205

Figure 7.5 BSP Throughput
(1 and 2 in KBits/second, 3 in block exchanges/second)

The results in column one indicate that the MACE can perform more than twice as fast as the BSP Handler so long as the other end of the stream is equally as fast.

Experiment 2 deals in the same quantity of data as experiment one, and therefore suffers from the same overheads of data copying in the client. The difference is in the number of data blocks sent; a factor of sixteen greater. As expected the transfer rate is lower since the stream must work harder, the point of interest is the difference between the figures for this experiment and experiment 1. The difference in line **A** is 40% while that in line **D** is only 15%. This shows that the per-block overheads in the MACE are substantially lower. The figures for experiment 3 show that they are at least a factor of four better.

Lines **B** and **C** give us a valuable clue about where this difference is critical. The figures in line **C** are close to those in line **A**, and line **B** tells us that the BSP handler is capable of a better transmission rate than in line **A**. This tells us that the limiting factor in lines **A** and **C** is the destination protocol package. This limitation is the speed with which the destination can process an incoming DATA block and get the next RDY back to the source.

The above experiments were carried out using the stream interface, which involves a byte-by-byte copy of all the data using procedures at both source and destination. If we eliminate this and drive the BSP Handlers directly at the Tripos packet level we can discover the raw throughput of the stream. We can do this by simply returning buffers to the stream as fast as possible, not even bothering to fill or empty them. Figure 7.6 give figures for an experiment similar to 1 above.

Source	Destination	
	BSPH	MACE
BSPH	50	81
MACE	52	318

Figure 7.6 Raw BSP Throughput in KBits/second

The interesting figure here is that for MACE to MACE transfer which is the raw throughput of the MACE given in figure 7.3. The overheads of protocols processing have become totally insignificant compared with the time taken to transmit the data[1]. These figures also indicate where the gains were made in the first set of experiments. Assuming the copy and transmission times were constant the improvements were made solely in reducing the amount of time spent in the protocol package. This demonstrates just how much CPU time the BSP Handler actually consumes.

MACE BSP can also be driven directly from the client program by exchanging blocks with the MACE device driver. Doing this, however, produces no better figures than the 318K bits/second produced by going through the dummy BSP Handler. This figure was produced by breaking the rules of Tripos and could never be used in practice. Driving the MACE directly does allow this throughput to be achieved and has the added advantage that the client can submit any number of buffers of whatever size he wants.

7.6.4 Effects on the Host

There are two measurements that can be made on Tripos to determine the effect of the High Level Interface: the change in system size, and the amount of work it has to do for any particular task.

The version of Tripos that uses the High Level Interface (MACE-Tripos) is about 9 Kbytes smaller than the normal Tripos system. Approximately 5K bytes of this derive from the reduction in code size: reductions in the device driver and Bsp Handler Task, and the elimination of the Ring Handler code. The the other 4 Kbytes is the result of the removal of the Ring Handler Task stack, global vector and other data structures. This

1 This is more an indictment of the MACE hardware than evidence of efficient software.

reduction is not really important in a 512 Kbyte machine whose memory is not fully utilised anyway.

The difference in workload on the Host is somewhat more difficult to measure. The best way of measuring this is to record the length of time each task in the system executes for the duration of a particular activity. Unfortunately the only measure of time available is a 50Hz clock, and most tasks execute at any one time for much less than this. The best we can do is to run a task at maximum priority that wakes up every 20ms and records those tasks it has interrupted. By doing this over a period of time a reasonably accurate picture of the system's behaviour can be built up, with due regard for Heisenberg's Principle. The results of running such a program are given in figure 7.7. Experiment 1 was to type the contents of a 16 Kbyte file on the terminal and is a good way of exercising the whole system. Experiment 2 was to transmit 200 Kbytes of data down a byte stream to a sink running in a MACE system and using a program that drives the BSP Handler directly. Experiment 3 was to transmit 200 full sized basic blocks to SINK.

Task	Experiment					
	1		2		3	
	Normal Tripos	MACE Tripos	Normal Tripos	MACE Tripos	Normal Tripos	MACE Tripos
Idle Task	56.49	68.46	59.33	92.40	91.88	97.93
Task 1	19.86	18.14	1.19	2.62	2.43	1.76
Task 3	9.23	11.10	0.04	0.41	0.20	0.07
Task 4	0.65	0.80	0.07	0.55	0.06	0.22
Task 5	4.80	--	1.62	--	5.07	--
Task 6	8.93	1.47	37.72	4.00	0.33	--
Elapsed Time	40.76s	40.62s	46.86s	10.40s	10.66s	10.34s

Tasks:
1 = Command Task
3 = Console Task
4 = File Handler Task
5 = Ring Handler Task (Not present in MACE system)
6 = BSP Handler Task

Figure 7.7 Comparison of Per-Task CPU Utilization Percentages

Apart from the complete elimination of the Ring Handler Task the main difference is in the time the BSP Handler runs. It is also interesting to note that the extra CPU cycles freed by the High Level Interface are almost all absorbed by the Idle Task.

Another measurement that can be made is of the number of task switches that occur during an activity. These can only be measured by making a small change to the Kernel to increment a counter in the Tcb of a task

each time it is entered. This is only one instruction, so the perturbation of the system is minimal. The measure is of actual task switches and not packet receptions or transmissions, which may not cause rescheduling. Invocations of devices are not counted, because they are called as subroutines of the invoking task, but the results of a device interrupt are counted if it causes a task switch. Figure 7.8 shows the results for the same three experiments.

Task Switches	Experiment					
	1		2		3	
	Normal Tripos	MACE Tripos	Normal Tripos	MACE Tripos	Normal Tripos	MACE Tripos
Idle Task	728	659	286	234	246	219
Task 1	2290	1629	469	452	446	236
Task 3	1734	1575	108	28	34	26
Task 4	370	247	61	34	42	26
Task 5	3335	--	1011	--	526	--
Task 6	4023	1666	1190	433	103	35
Total	12480	5776	3125	1181	1397	542
Elapsed Time	40.66s	40.34s	39.87s	10.15s	10.72s	10.24s

Figure 7.8 Comparison of Task Switches between Normal and MACE Systems

The totals for each experiment show that there is a reduction of between fifty and sixty percent in the total number of task switches. Most of this difference is due to the removal of the ring handler, although the BSP handler task also exhibits a significant reduction in the number of times it is entered.

7.6.5 Inter-Machine Communication

A potential bottleneck is the overheads associated with getting commands from the Host to the MACE and replies back. A simple test, using a MACE function that does nothing but generate a reply, indicates that about 800 commands can be exchanged in a second. This compares with a rate of 2000 packet exchanges per second between two Tripos tasks. In none of these experiments were commands exchanged at anything near this rate, so it did not constitute a limiting factor.

7.7 Discussion and Conclusions

The figures in the previous section show that, using the same hardware, implementing a higher level of interface in the MACE improves system performance. Since the raw throughput of the High Level MACE is less than that of the Type2 and SPECTRUM this increment must be derived from other factors.

Is this improvement due to the increases in MACE functionality, or is it caused by some other, more mundane, feature? We have moved the protocol implementations from a high level language to hand crafted assembly code and would be surprised if this did not result in an improvement. Is the second processor really necessary, and would we not get the same improvement by implementing the protocols in machine code in the Host? In answer to this we must bear in mind that the 6809 is a slower processor than the 68000, has a less powerful instruction set, and inter-machine communication is a factor of three slower than inter-task communication. These would tend to put the MACE code at the same level as BCPL in the 68000, so any comparison in other areas should be approximately equivalent.

A Processing Server running Tripos is not the best machine on which to base our observations. It is a single user system and while a little asynchronous activity does exist it is largely a single program one too. Therefore when a ring interaction is performed there is little work the machine can do while it is waiting for it to complete. The result of this is that a processing server tends to spend most of its time idle, even when it is in theory working hard. Increasing this by a few percent is no great achievement. More genuine gains would be observed in a multi-user timeshared machine that has other users to service while waiting for a ring interaction to complete Here the system overheads in the Host are also much greater and reducing these can only be an advantage. In the Cambridge environment multiple client service machines like the Fileserver or the Filing Machine would also gain in performance.

In addition to reducing the amount of work the Host does in order to operate a protocol, there is the additional advantage of reducing the length of time spent in protocol related processing. This is where the customised environment in the NIP is of advantage. The use of a separate processor also allows this processing to be carried out in parallel, removing it from the critical path. The improvements seen in the performance of Tripos lie almost exclusively in this category.

It is interesting to note that we find in the MACE BSP implementation an example of the general buffer chaining mechanism we were striving for in the Type 2. There it was somewhat unsuccessful whereas here it is not only successful but the most natural way to do things. What is the difference? Largely it is a matter of implementation chaining being too complex for the limited abilities of the Type 2. The major infelicity, however, lay in what to do with errors. If, for example, an error occurred while processing a buffer the Host had marked to be dropped the Type 2 was presented with a dilemma: should it return the buffer with the error report, or drop it as ordered, making the Host do a lot of work to find out what happened. The MACE avoids this problem by always returning buffers. BSP is also different from Basic Block protocol in that there are no real errors to be reported back to the client, buffers contain return codes that are for information only.

A workable basic block chaining scheme, based on a mixture of the Type 2 mechanism and the MACE BSP mechanism could be devised. In a High Level Interface that outlaws Basic Block Protocol this is not what we want. It would be more useful to be able to submit buffer chains to functions like SSP and Fileserver reads and writes. Internally the MACE already uses buffer chains for all these purposes, so the change is only in the interface. However only a few special applications will need such a facility, we should not force simple applications to use is too. This would only allow static buffer chains, which could not be added to once submitted, because the MACE would slave the entire chain in its own memory.

The special chaining mechanism required by the Fileserver, allowing it to add new buffers while the transfer is in progress and to cycle the same buffer several times between ring and disc, is a different matter. This is best achieved by maintaining the chain in the MACE only, and cycling buffers through the inter-machine communication mechanism.

Even this may not be necessary, we have seen that the MACE BSP protocol implementation can deliver data as fast as the Host can itself using raw basic blocks. It would be possible to perform all file accesses using a byte stream interface. A distributed filing system being developed at Cambridge is going to do exactly this. The advantages of using BSP in this way are similar to those described in chapter 6 under debugging. The protocol handles errors and retries without the Host's intervention, and the flow control allows data to be read off, or written to, disc as fast as it will allow[2] .

2 Here the disc unit is slower than the ring, but this smoothing factor will work equally well in reverse.

SuperMACE gains its speed primarily by never copying data, which is itself achieved by switching buffers in mid-block. It is worth emphasising that this 'trick' is only made possible by the unique features of the Cambridge Ring. The use of busy backoff as a primitive flow-control mechanism gives the MACE the necessary time in which to find and setup the next buffer. Such a thing would be almost totally impossible with a network that shared the medium with a much larger grain size, or did not return any information to the transmitter. To the author's knowledge the Cambridge Ring is the only LAN that has the required properties.

The internal structure of the MACE, using the Reverse Call mechanism to effect inter-module communication has proved to be worthwhile. It started as a simple mechanism to allow protocol packages to control the reception and transmission of their blocks and make validity checks on the fly. Extending it throughout the system was an experiment, which in the author's opinion, has succeeded in that the resulting program is both more efficient and simpler than if message passing or worker processes had been used exclusively. A contributing factor to this is that the entire system is written in assembly code, which gives the author complete freedom over program and data structures. The mechanism can, however, be implemented in any language that allows modules and procedure variables, for example Modula-2 [Wirth80].

The MACE system has avoided any of the problems of inter-task synchronisation by employing a non-preemptive coordinator. For genuine real-time response it must be possible to preempt low priority activities if favour of higher ones. This kind of preemption goes on to a small extent in the MACE whenever an interrupt occurs; synchronisation problems are solved here by turning interrupts off for a short period whenever shared data is accessed. The requirements for synchonisation are relatively small; a simple implementation of semaphores, for example, would meet the needs adequately and efficiently in a preemptive system.

The entire system only works because all the programs have been written with the time constraints in mind. There is no easy way to enforce these and remain efficient. Attempting to extend this mechanism to a general purpose operating system, where reverse calls into untrustworthy user programs would have to be made, may not be totally successful. On the whole reverse calls are really no more than an interesting design method for self contained, real-time, systems, we should not try to push it too far.

Chapter 8

Further Aspects of a High-Level Interface

This chapter covers those aspects of High Level Interfaces not covered in chapters six and seven. These features have not been implemented because of lack of time, hardware or because they are inappropriate in the current environment. Of necessity this chapter is somewhat disjoint.

8.1 Cambridge Specific Features

There are several aspects that are specific to the Cambridge environment or the Ring alone. The loading and debugging requirements of a processing server have already been discussed. Here I wish to cover the MACEs role in resource management and authentication, and examine how the BSP implementation might be enhanced to a network independent Transport Service.

8.1.1 Resource Management

Processing servers are allocated by the Resource Manager (RM). To enable RM to retrieve and reallocate machines when they crash a "dead-man's handle" is worked by the processing server. If this is not worked for some timeout period the machine is reclaimed. The handle consists of an SSP exchange every thirty seconds or so; this is clearly a candidate for removal to the MACE.

Merely arranging to send an SSP to RM every 30 seconds is of little use. The Host can easily crash without affecting the MACE, which will continue working the handle regardless, keeping the machine allocated. To avoid this the MACE could maintain a timer which is reset whenever a command is received from the Host. If the timer expires the MACE can assume the Host has crashed. Unfortunately this does not always work, the Host may be involved in a CPU intensive computation, and may not use the Ring for some time. The MACE would not be able to detect the difference between this and a genuine crash.

The solution is for the Host to maintain a dead-man's handle to the MACE. Most of the time the operation of this will be implicit in the exchange of requests, only when the Host has not used the Ring for some time will an idle exchange be necessary. To prevent the Host having to maintain a timer of its own, the initiative for the exchange should come from the MACE. This may be achieved by the Host giving the MACE a packet to use for this in the same manner as Remote Resets and Closes are implemented in BSP. The Host merely need to bounce this packet back to the MACE whenever it is sent. Note that this bouncing should not be performed in the interrupt routine, it is likely that this would survive a higher level crash of the machine. The packet should instead be delivered to a reasonably high level in the system to ensure that it is still alive.

At first sight this does not appear to gain us anything. We have simply replaced one dead-man's handle by another one. If we consider, however, what the MACE should do in the case of a Host crash we find that this is not so. The initial reaction, to tell RM to release the machine, is not correct, since the user may wish to debug it and find out why it has crashed. The MACE should therefore keep the handle going while the debugging is being done. It should not, however, keep the handle going if there is no intention to debug; the following mechanism accomplishes this.

When the MACE believes the Host has crashed it reports this fact, either by sending a message to the system log or by communicating with a well known debug service. At this point it stops working the dead-man's handle to RM. If the user is not going to debug the machine RM will eventually reallocate it; this is no worse than when the machine works the handle itself. If the machine is to be debugged a byte stream for this purpose will be connected into the MACE. When this happens the MACE can resume the dead-man's handle for the duration of the debugging session. When the stream is closed the MACE can hand the machine back to RM, or continue if the user has managed to fix up the problem.

This mechanism allows the user the RM timeout period in which to get a debug service and connect in to the MACE. It also means that any other time-dependent connections the machine has, particularly the terminal byte stream, will not be lost, and if the machine is allowed to continue it will be in the same state.

8.1.2 Authentication

The Cambridge Authentication system is described in chapter 2. To support this the Host normally maintains a "Fridge", the purpose of which is to refresh the timeouts on any UID-sets placed in it. The Fridge is clearly a candidate for removal to the MACE for the same reasons given in the previous section.

Authenticated versions of both the the OPEN and SSP protocols have been defined, [Johnson82]. It should be simple to enhance the relevant modules to handle these as well. The Host can reference UID-sets using UID-set identifiers and should quote one of these whenever making an authenticated call. Likewise any UID-sets in received requests can be detected and removed, a UID-set being returned in their place.

The automatic validation of received requests, while possible, is probably not a good idea. This is because the Host may not want the validation to be carried out immediately, or may want to try the UID-set out against several authentities. It makes more sense, therefore, to supply an explicit validation function that takes two UID-sets as arguments. The first UID-set provides the PUID/TUID pair to be authenticated and the second the authentity under which this test should be made; these may, of course, be the same UID-set.

The MACE should also supply the Identify function of the AOT, which takes and entire UID-set and checks that it is represented as a single entry in the AOT. The AOT has three other functions: Refresh, GetTUID and Enhance. Refresh is implicitly provided by the Fridge, and need not be supplied to the client explicitly. The other two are used by authentication authorities. While it is unlikely that authentication servers will be implemented in processing servers, it is possible that a static service (the Filing Machine for example) will be an authority. The protocols used between an authority and its client are not at present well defined, so the MACE would not be able to extract or insert UID-sets. These protocols are best left, then, to being implemented explicitly by the server itself using the SSP entry. The service will, of course, be able to use the Fridge for storage.

Finally functions must be available to allow clients to examine and install UID-sets in the Fridge. Since we are not concerned here with protecting the mechanism from the user, it has its own in-built protection, this is acceptable. Attempts to install invalid UID-sets in the Fridge will be detected when the first refresh is attempted, so they can be evicted.

8.1.3 Transport Service Compatability

In chapter 5 it was mentioned that the protocol level implemented in the NIP should not go beyond ISO OSI Transport Service. At present the MACE BSP implementation does not go even this far. Of interest here are the "Orange Book" [JNT82] V-service and N-service definitions.

MACE BSP is already close to V-service (which is BSP under another name) lacking only the ability to include user data in the RESET and CLOSE exchanges, and the EXPEDITED message type. The latter can be implemented as a couple of extra super-states, and both changes require the addition of an extra buffer type (with associated internal queues). This would also require enhancements to the Host interface, partly to include the Expedited functions, and partly to bring the existing functions in line with the standard. The OPEN exchange must also be altered because V-service expresses the negotiation sizes in bytes while BSP uses packets.

The N-service enhances V-service to the full Transport Service defined in [BT80]. It does this by imposing a message structure on the the service. Raw data is still sent in the same way but various control messages are defined to occupy the data fields of the OPEN, RESET, EXPEDITED and CLOSE messages. These may also extend to following data blocks with the qualifier bit set. The ADDRESS N-service message is not directly associated with any V-service message but occupys a block with the qualifier bit set.

A major obstacle to implementing N-service efficiently is that it requires all parameters to be divided up into fragments of less than 64 byte each for transmission. While it would certainly be possible to fragment or reconstruct parameters on the fly by clever use of reverse calls and Buffer Control Blocks, it would require the BSP module to be made N-service specific. Fortunately this is not necessary as some of the N-service parameters (Called Address, Calling Address, Recall Address etc.) are of interest to V-service in order to establish the call. Consequently these must be read into the MACE. If all the parameters are read into the MACE the fragmentation or reconstruction can be carried out in store. Control messages are likely to be rare, so the overheads associated with processing them in this way will not be too restrictive, the bulk of data can still be transferred at full speed. This arrangement also means that N-service can be implemented as a simple layer on top of V-service.

8.2 The Hardware of a High Level Interface

With the experience of having written ring driving software for two separate sets of hardware it is possible to make some suggestions for the hardware of an improved NIP. Some of the following features were designed and implemented by J. J. Gibbons in the GIZMO [Gibbons81a] which itself influenced the design of the MACE2. Both these machines were 6809 based and it was originally intended that the program described in previous chapters would be transported to at least one of them. This has not been done.

The most important feature to be added is vector DMA hardware so that Ring to memory and memory to Ring transfers can be carried out without the NIP's intervention. This also means that automatic checksumming is necessary. Vector DMA is simple on reception, a memory cycle is performed on demand each time a ring packet arrives. Transmission is more complicated since it is possible for a previous packet to need retrying. The speed will come from the pre-fetching of data into a buffer to await transmission. Data will only be transferred from here to the ring when the previous packet has been sent successfully. To avoid involving the NIP in the process the retrys must be done in hardware.

The hardware to do this pre-fetching and retrying is known as the Forward Transmit Buffer (FTB) [Gibbons81b]. In addition to automating the transmission process it allows any machine to which it is fitted to gain a little more point-to-point bandwidth. A program driven interface is only able to use a ring packet at maximum once every ring revolution plus two packets. In theory it is possible to reduce this to a revolution plus one packet, but no program is fast enough to test the response bits and give the next packet to the station in the few microseconds necessary to achieve this. With the next packet already buffered, and hardware to test the responses this is possible. As an example, consider a ring with 80 bits, this will have two packets and a gap of four bits, a program driven interface can achieve about 1 Mbit/s transmission rate, an FTB can increase this to 1.35 Mbit/s.

The MACE has to go to considerable trouble to concatenate buffers that are oddly aligned with respect to memory words and ring packets. The size of DMA transfers and ring packets should be independent, so on transmission data should be moved a byte at a time from the memory to the ring, transmissions and DMA transfers being performed on demand as the buffers fill and empty. This can have the unfortunate effect of losing the 'next packet' advantage if the ring and DMA sizes disagree, or they get out

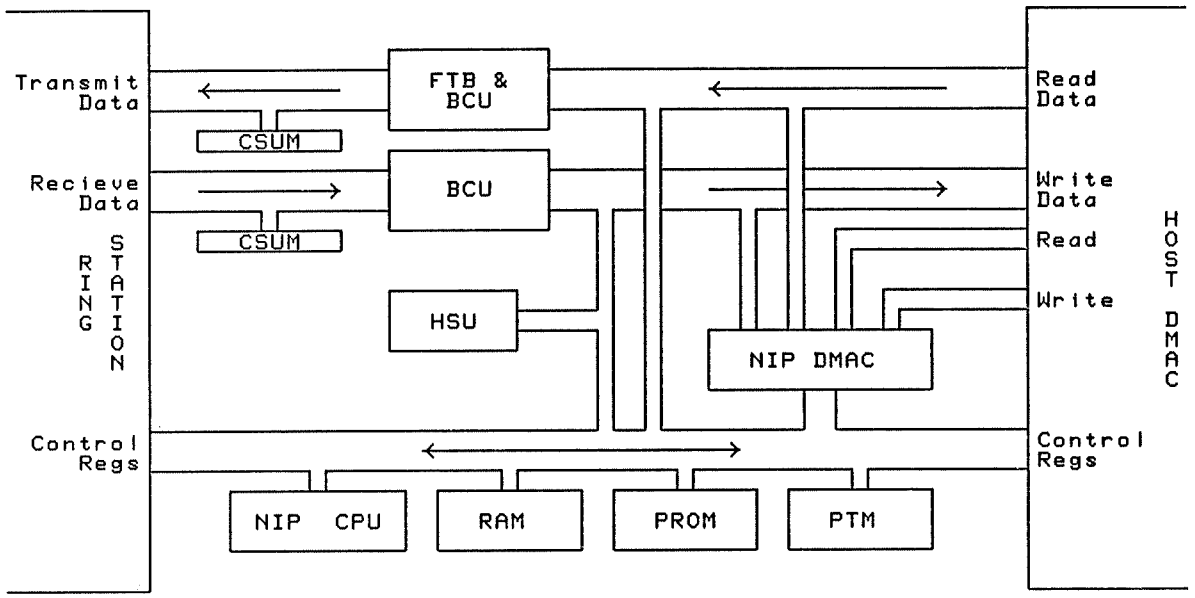
of step. A possible alternative is to replace the buffer by a byte wide FIFO with enough room for several packets and cause a DMA transfer whenever there is room for it.

The GIZMO was fitted with a Byte Swap Unit [Gibbons81c] which not only allowed buffer concatenation, but allowed the bytes of a packet to be swapped over. This was to cater for machines whose byte ordering in a word was not the same as that on the ring; this could lead to confusion when transferring integer values from one machine to another. It should be noted that while the byte swap unit does what is required for 16-bit values, it does not solve the problem of 32- or 64-bit ones.

So far most of the hardware support has been for transmission, can anything be done for reception? Nothing equivalent to the FTB is necessary, but a buffer concatenation unit would be. There are also a couple of useful things that could be done in hardware. First, the search for valid header packets can be automated. The hardware is given a bit pattern and a mask, it receives packets until one is received that matches the bit pattern after masking. The source address of the packet is automatically transferred to the select register and only then is the NIP interrupted. The second optimisation is for when an unwanted block is to be rejected by setting the select register to zero for a short period. A piece of hardware could, when enabled, set the SAR to zero at the same time set a counter (or a mono-stable circuit). When the count reaches zero the SAR is reset to select anybody and the header search hardware enabled. Neither of these will increase data transfer speeds, but they do absolve the NIP of some work. This is particularly true for deselection, the length of time this needs to be done is so short that it is easier to do it in a busy wait loop rather than set up a clock timer.

Turning away from the ring for the moment, a potential bottleneck in the system is the Host/NIP interface. The 68000/MACE packet exchange rate of 800 commands a second is adequate when large buffers are being used, but it is of no use when we want to transfer small quantities of data at speed, for example when driving a VDU in single character mode. Part of the problem is the need to do two DMA transfers. One of these could be eliminated if a pair of parallel I/O ports were provided between the two machines as in the Type 2. An even better solution would be a small quantity of shared memory, which would allow simple commands to be transferred in one go.

So far vector DMA hardware has only been considered for the Host memory to ring path. Such hardware would also be useful for transferring data from the NIPs own memory to the Ring or vice versa. This would enable the NIP to use the FTB and checksum hardware, and to maintain the same data throughput. Vector DMA cannot be used as a straight replacement for the simpler single packet interface of the MACE and Type 2 since there is still a need to transmit and receive single packets to deal with header and port packets. Vector DMA between the NIP memory and Host memory would also help with the Host Interface, this would require DMA hardware at both ends. A schematic of a possible configuration is shown in figure 8.1.



Legend:
BCU: Buffer Concatenation Unit
CSUM: Automatic Checksum Unit
DMAC: Direct Memory Access Controller
FTB: Forward Transmit Buffer
HSU: Header Search Unit (and Deselect Unit)
PTM: Programmable Timer Module

Figure 8.1 Possible High Level Interface Hardware

The MACE2 is designed to cater for large timeshared computers, and to this end it also has a large buffer store between it and the Host. This is so the Host does not have to keep many buffers locked down in store for the MACE to receive into. Instead all data is received into the buffer store and is transferred from there into the Host when a buffer is available. If the MACE2 processor has easy access to this memory it could use transfers to and from this to replace DMA Transfers using its own memory. The buffer store does have the disadvantage, however, that it introduces an extra copy operation in the pipeline.

The cost of such an interface should not be unduly high. Most of the components are small- and medium-scale TTL, the expensive parts being the DMA controllers, the NIP's memory and the processor itself. These are likely to be overshadowed by the cost of the software development, both in the NIP and in the Host machine.

Future LANs will operate at transmission speed an order of magnitude greater than the Cambridge ring (e.g. the Fast Ring [Banerjee83]). The design of a NIP for such a network may need to be significantly different to cope with the greater real-time requirements. The Cambridge Fast Ring already contains hardware to do the jobs of the FTB and BCU, and so would be a more suitable device from which to start.

8.3 Security and Encryption

If the data to be shipped between machines on a network is of a sensitive or secret nature the sender wants to ensure that it is only read by the intended recipient and is protected from unauthorised access. Since there is no way of preventing physical access to the network it must be assumed that anybody can listen to any communication going on and, more seriously, interpose a computer of their own in any communication path. The only secure means of transferring data under these conditions is to encrypt it.

It has been suggested that encryption be performed in the network interface and material encrypted on transmission and decrypted on reception. This has been rejected [Needham78] because one needs to be able to multiply encrypt an item, or attempt decryption with several different keys. This objection is based on the assumption of an un-intelligent or low level network interface. If we have a High Level Interface, into which we move all encryption duties the objection is removed. Indeed the only reason multiple encryption, and decryption attempts, are necessary is to implement key-passing protocols, in which the Host is now no longer involved.

It is undeniably useful to be able to encrypt or decrypt data on-the-fly as it is received or transmitted. When the secure connection has been set up this is exactly what we want. Positioning encryption units in the main data paths will achieve this (in figure 8.1 they would probably be combined with the Buffer Concatenation Units). This assumes that suitably fast hardware encryption units are available. DES [DES75] chips are available that can encrypt a 64-bit block in 5 microseconds using a 56-bit key.

These can be configured with external hardware to operate as byte-by-byte stream encryptors. An interesting feature of this configuration is that for both encryption and decryption the chip works in encryption mode only.

Placing the encryptors in the main data pipelines gives the NIP the same problems that caused the original objection by Needham and Schroeder. There are two ways of allowing the NIP multiple encryption and decryption. The most elegant solution is to provide it with a third encryptor, possibly cross-connecting two of its own DMA channels to allow asynchronous operation. If encryption units are at a premium (which they arguably should be), arrangements could be made to pass data through one of the pipeline encryptors when there was no transfer going on.

The NIP is can now implement any key passing protocols necessary. To show how this would be achieved consider how a secure byte-stream might be established between two clients, **A** and **B**. For the purposes of illustration we use a modified form of the protocol and notation from section 4 of [Needham78].

To the Hosts the interactions used would be identical to those to establish a non-secure connection. The decision that this is a secure communication need not lie with the Hosts and may be a property of the service description. The first interaction therefore is:

$$A \rightarrow NIP_A : B, \text{ Parameters} \qquad (1)$$

The parameter field should not contain any sensitive information, merely some initial parameters to allow **B** to decide whether to accept the connection. If **A** has communicated with **B** before it may have cached a connection key already, which would be stored in the description along with the messages to be sent initially to **B**: $[CK , A]^{KB}$. If there were no cached key the NIP would have to obtain a new one from the Authentication Server with the following interaction:

$$NIP_A \rightarrow AS : A , B , I_{A1} \qquad (2)$$

$$AS \rightarrow NIP_A : [I_{A1} , B , CK , [CK , A]^{KB}]^{KA} \qquad (3)$$

I_{A1} is a transaction identifier, which is generated by the NIP for this transaction only, KA is **A**'s private key and KB is **B**'s private key. Note that since the NIP knows that this message is going to be encrypted with KA it can decrypt this as it is received. CK and $[CK , A]^{KB}$ can now be

cached for later use. The NIP now transmits the following message:

$$NIP_A \rightarrow NIP_B : [\ CK\ ,\ A\ ]^{KB}\ ,\ [\ I_{A2}\ ]^{CK}\ ,\ [\ Parameters\ ]^{CK} \qquad (4)$$

As far as A is concerned this can be encrypted on-the-fly, the first part does not need to be encrypted, and the latter two parts are encrypted once with CK as they pass out of the machine. This is also a candidate for on-the-fly decryption provided the message can be identified as secure connection attempt from its header. It is known that the first part of the message will be encrypted with KB so this can be set in the pipeline during reception. Once this part has been received CK can be removed from the newly decrypted data and set in the pipeline decryptor to receive the rest of the message.

NIP_B now performs the following exchange with B:

$$NIP_B \rightarrow B : A\ ,\ Parameters \qquad (5)$$

$$B \rightarrow NIP_B : A\ ,\ Results \qquad (6)$$

And returns the following message to NIP_A:

$$NIP_B \rightarrow NIP_A : [\ I_{A2}\ ,\ I_B\ ]^{CK}\ ,\ [\ Results\ ]^{CK} \qquad (7)$$

Again this can be encrypted/decrypted on-the-fly. NIP_A in turn does:

$$NIP_A \rightarrow A : Results \qquad (8)$$

The results indicate whether B will accept the connection. If the connection is rejected the exchange stops here. If this happens NIP_A cannot retain CK in its cache since it is now known by another service. If this was genuinely B there is no problem, but an eavesdropper could have used the interaction so far to obtain CK with the intention of listening in on a subsequent conversation between A and B. If B accepts the connection the NIP_A responds with:

$$NIP_A \rightarrow NIP_B : [\ I_B\ ]^{CK} \qquad (9)$$

The connection is now established, any subsequent communication may be carried out with CK set in the pipeline encryptors during transfers. It is interesting to note that in this protocol we have used only on-the-fly encryption. This has been possible because at all stages we have either known beforehand which key to use, or have been able to extract it from an

earlier part of the same message. Placing the encryption unit in the Ring pipelines has turned out to be no great disadvantage, and if we restrict ourselves to authentication protocols that can exploit on-the-fly encryption we can dispense with memory-memory encryption entirely. This also shows us that at the lowest level keys should be associated with data buffers, and not with entire blocks, allowing different parts of the block to be encrypted with different keys. It should be easy to arrange that when the NIP switches buffers it also switches the current key. This is why some of the messages above (4 and 7) show two parts of the message encrypted with the same key, these are not coalesced because they originate from different buffers.

The usual method of checking the validity of an encrypted block is to include in it a checksum of the data it contains. Like basic block checksums this can be calculated by hardware and transmitted or validated on-the-fly. This is necessary if we are going to remove it from the data-stream, and catch invalid communication attempts early.

The NIP is only concerned with using encryption to establish interactive connections across the network. There are many other applications of encryption: secure mail, digital signatures, cryptographic sealing [Gifford81], which are outside its domain. These are more suitably implemented in the Host processor, although there is no reason for the NIP not supplying an encryption service to the Host for these purposes.

This example uses secret key encryption; public key encryption does not lend itself so easily to similar treatment. This is partly because at present public key algorithms are unsuitable for hardware implementations, and software implementations are slow. Also, once the secure connection has been established all data transferred has to be encrypted first with the secret key of the sender and then with the public key of the receiver.

8.4 Customised NIPs

The MACE program described in chapters six and seven was designed for general purpose use by a processing server. There is also the possibility of producing NIP programs that are tailored to the needs of special purpose Host machines. These would typically be static servers such as the Fileserver or the Filing Machine.

The Filing Machine is a typical example. It has no need of BSP but makes heavy use of SSP and SSP based protocols. The advantages of implementing the Filing Machine protocol in the MACE have already been

mentioned. The service end of this protocol is only necessary in the Filing Machine itself. FM is also the only machine that can make full use of the Fileserver protocols. A MACE program could be put together which implement these protocols exclusively.

Similarly the Fileserver only needs the service end of the Fileserver protocol. Here we could probably go further and include validation and function decoding, and keep the tag field in the request in the reply descriptor to prevent it being passed around the Fileserver itself.

8.5 The Use of Stable Storage in a NIP

It is generally considered useful that machines that engage in network-wide transactions should have a quantity of stable, crash-proof, memory at its disposal. This may be used to, for example, store the commitment state of an atomic transaction. This is often supplied by disc storage and conventional multiple copy techniques. Where the expense of a disc is too much, or is inappropriate, other techniques must be used.

The requirements of stable-storage are that it be immune to both software and power failure. Resistance to power failure can be achieved either by using a memory medium that preserves its state without power (e.g. Bubble memory) or giving it a standby power supply using batteries. Immunity to software error can be achieved by preventing it being accidentally written. The technique given in [Needham83] allows the stable memory to be read in the same way as any other part of the address space, but only written if the previous content of the location is inserted in a special register first. The paper describes this in terms of a microcoded machine, but the same technique can be applied in hardware by a purpose built memory board. There are a couple of disadvantages to this. First, it is Host specific, a new micro-program or memory board must be implemented for each different machine type. Secondly if many locations contain the same value, and this value is in the special register, a rogue program may be able to write nonsense over much of the stable memory with little effort.

An alternative is to install the stable memory in the NIP. A Host specific interface must be implemented for this in any case, once this has been done the stable storage will become available at no extra expense and in exactly the same way on all machines. The formality of communications with the NIP provides the protection necessary against software failures, and since we can put some trust in the NIP software the stable storage

need not be protected against reading and writing in an arbitrary manner. Indeed if we are not concerned about protection against power failures the stable storage need be no more than a reserved area of the NIPs normal RAM. At the other end of the scale we can give the entire NIP a reserve power supply, so if the Host loses power it can report the fact. It can even take measures to dump the contents of the stable memory to a Fileserver before the batteries run out.

Finally it should be noted that stable storage is only useful to certain types of machine. These are machines that, when they are restarted, will be performing the same job, for example static servers and personal machines. Stable storage of this kind is of little use to a processing server since it is uncertain whether a crashed program will be reloaded into the same physical machine. A limited amount of such storage would be useful to the NIP itself, however, to keep the port number sequence, descriptor cycle etc. Here only a few bytes are necessary, and can be supplied easily.

Chapter 9

Protocols and Closed Networks

This chapter discusses two issues associated with Intelligent Interfaces. The first is the influence a NIP may have on the design of network protocols, and the second is the possible advantages to be gained by the universal use of NIPs in a network.

9.1 Protocols and the Intelligent Interface

So far we have only considered implementing those protocols that have already been in use. Sometimes the protocol is designed in such a way that it is difficult to implement it in a NIP, BSP is an example of this. We can also consider implementing protocols in the NIP that would be too costly or impossible in the Host. In this section I wish to consider two protocols that are designed for an implementation in the NIP.

9.1.1 Remote Procedure Call

Interactions of a request/reply type are handled in the Cambridge environment by Single Shot Protocol which is a limited form of Remote Procedure Call. This only implements at-least-once semantics because the failure of the protocol is detected by the expiry of a timeout, and the only recovery is to retry the request. Any service that is accessed by SSP must therefore be defined in such a way that any operation is idempotent. This is often not possible, for example: if an OpenFile request to the Fileserver fails and the retry produces a reply of 'File already open' the client does not know whether this is because the reply to his previous otherwise successful request was lost, or because some other client has it open. At-least-once semantics are at odds with the 'natural' view of a procedure call, which guarantees that the operation has been performed exactly once if the call returns. It would ease the application programmers burden, and allow modules to be made local or remote in a transparent manner, if an RPC protocol were available that guaranteed exactly-once semantics. For a full discussion of the issues involved in Remote Procedure Call see [Nelson81].

Imposing this model on the interaction requires that the service can differentiate between original requests and retries. This is achieved in some systems, for example Courier [Xerox81b], by employing a stream protocol as the basic data carrier. This guarantees that the requests and replies are delivered exactly once without error by handling all flow control, error recovery and retries itself. In consequence the RPC protocol layer is merely a message structure imposed on top of the stream. If the client is going to make repeated calls on the service over a short period of time this is adequate. If the client only wants to make one call, however, it would be needlessly expensive to create a stream for the purpose.

An alternative approach is to enhance the simple message exchange protocol towards reliability. One means of doing this is to include a Unique Identifier in the request. The server records the UID of each request it receives and if any repeats arrive they can be discarded. By using a system-wide sequence number Shrivastava and Panzieri [Shrivastava81] only need to keep the last sequence number received.

This mechanism deals adequately with a lost request, and prevents the service performing an operation more than once. It does not help the client to recover from a lost reply. This can be achieved if the server caches every reply it sends; then if it gets a repeat request it simply returns the cached reply. In theory the server must cache every reply it has ever sent since it does not know how long the clients timeout is. In practice this can be reduced to just the current set of Remote Calls if the client returns an acknowledgement to the reply. Since it should do this immediately the timeout used by the server to detect the failure of this exchange can be small; if it does fail the server must retransmit the reply message. If the acknowledgement was lost the client will not have an outstanding call with that UID attached. By deriving the UIDs from a monotonic sequence the client can decide whether the UID was used recently and either generate a suitable acknowledgement or return an error message to the server.

Detection of a lost request still relies on the expiry of a long timeout. This is unsatisfactory for two reasons. First, the client must wait the entire timeout period before retrying; in theory it should be able to retransmit the request at once if it knew it had been lost. Second, the service is forced to meet this deadline; this often results in a timeout value much larger than necessary, making the client wait even longer when failures occur. This is exacerbated if the service must itself make remote procedure calls to other services, since their timeout periods must be

added on to the time for the service. We can overcome this in exactly the same way as we overcame the loss of replies: by returning an acknowledgement to the request. Again the timeout on this exchange can be short, and retries rapid. If the request was lost the retry will start the operation as required. If the acknowledgement was lost the service will already have the UID in its cache and can generate the acknowledgement while throwing the repeated request away. This scheme has the advantage that there is no need for an overall timeout on the call, which may take as long as necessary.

So far we have dealt with lost packets only; for the protocol to be complete it must also be able to deal with machine crashes. The server will detect that the client has crashed when it attempts to reply: the client will not have that UID as an outstanding request. To avoid confusing this with the case of a lost reply acknowledgement the client must keep the first UID it used in the current incarnation in stable storage. Any reply whose UID lies between this value and the current one is the result of a reply retry; any other value indicates that the reply was from an orphaned call of an earlier incarnation. The server can use the result of this response to decide whether to commit any changes it has made as a result of the call, or to undo them.

A crash of the server is not so easy to detect with the protocol as it stands, but if we introduce an idle-handshake between client and server every thirty seconds or so, both ends can discover early whether the other has crashes. Since most calls will be of short duration it is unlikely that the idle handshake will be used frequently.

Since this protocol is implemented entirely in the NIP the Host machines see an interface identical with that of SSP, consisting of just three functions: RpcRequest, RpcService and RpcReply. The client should see no difference in performance between this protocol and SSP since the request acknowledgement is delivered in parallel with the execution of the service, and the reply acknowledgement is delivered after the reply has been passed on to it. The server will see a slight degradation in the time taken for the RpcReply function since the NIP 'caches' replies by simply not returning the reply buffers to the Host until the acknowledgement has arrived. If the service expects to generate a reply almost instantly a slight optimisation can be made in which the servers NIP witholds sending the request acknowledgement, making the reply serve the purpose instead.

The full advantages of this protocol are only available on a machine with a NIP that will do all the protocol processing in parallel. The protocol is only four messages, however, and is much simpler than BSP which is the only other means of guaranteeing safe delivery, so it should be easy to implement on single processors and simple machines for compatability.

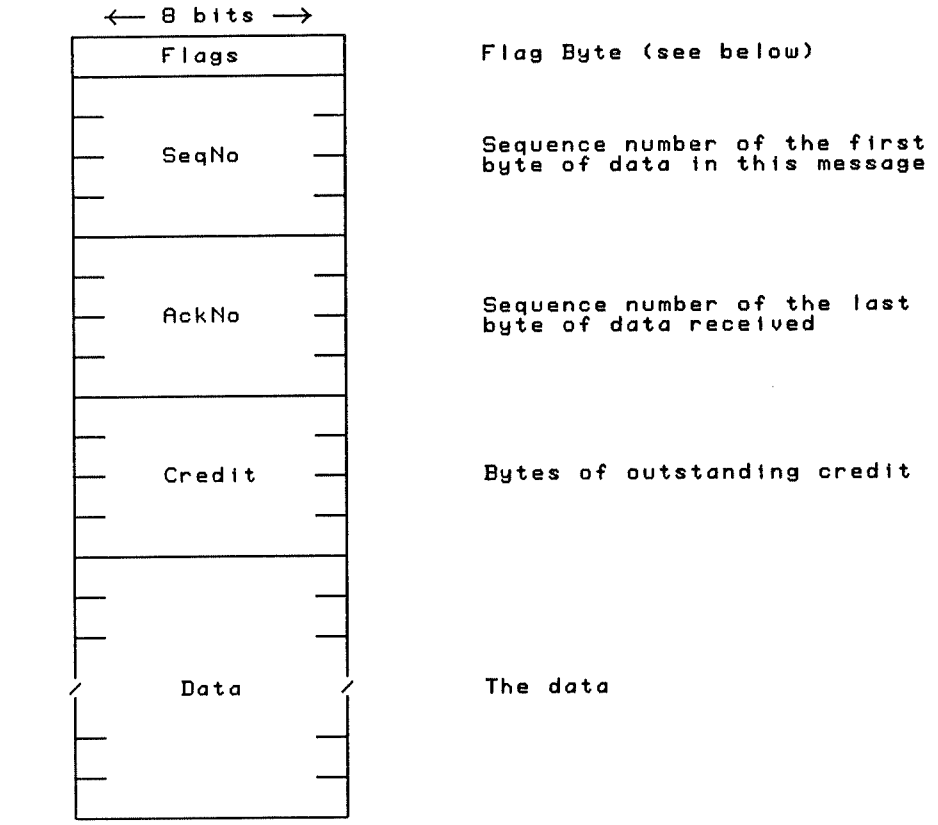
This protocol should not be confused with the fully type-checked, highly structured RPC mechanism of Courier, or described in [Nelson81]. A closer analogy would be that this protocol is to that form of RPC what a jump subroutine instruction is to normal procedure call. It is a primitive, unstructured mechanism that may be used as it is for speed, but may be built upon to achieve something more elaborate.

9.1.2 A Stream Protocol

Some of the problems associated with the implementation of BSP in an intelligent interface were described in chapter 7. These are largely a result of its original derivation from wide-area protocols such as X.25. Two important infelicities can be identified. First, the restriction to fixed sized blocks is unnecessary in a NIP based protocol, and leads to the threshold mechanism employed in the MACE. The Host should be able to submit data buffers of any size that is convenient and expect it to be delivered, and not be trapped in the stream by an unfortunate choice of buffer sizes. Secondly, every block of data sent must be acknowledged before the next is sent. This is done in the interests of simplicity and makes the protocol safe at the expense of bandwidth (it should be mentioned that this was a conscious design decision). On a LAN where the error rate is expected to be low this practice is largely redundant.

An alternative stream protocol to BSP can be devised with the following properties: flow control is at a byte rather than block level, and data may be transmitted on the assumption that it will arrive safely, special action is only taken when it does not. An example of such a protocol is now given. The intention in giving this protocol is to show how the deficiencies in BSP might be overcome and is therefore presented for the purposes of illustration only.

The stream is a full duplex bi-directional channel, although the following description considers just one direction only. The format of a stream message is shown in figure 9.1.
The stream starts up with SeqNo, AckNo and Credit set to zero. Whenever the receiver is given a buffer it sends a message to the transmitter with the Credit field set to the new size of the buffer pool. The receiver may



Flag Byte Layout

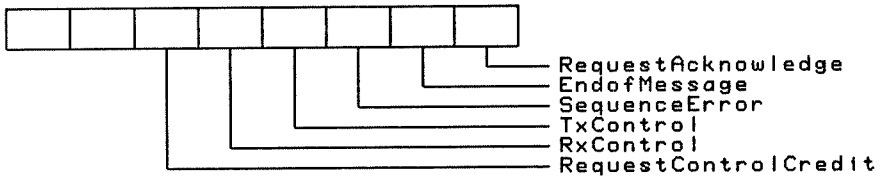


Figure 8.2 Layout of Stream Protocol Message

send a message to the transmitter at any time. When the transmitter receives a buffer of data from its Host it examines the outstanding credit from the receiver and if this is non-zero transmits as much of the buffer as it will allow, incrementing the sequence numbers in the blocks accordingly. If the Host wants to know that the data has been delivered the transmitter sets the RequestAcknowledge bit in the flags field. If a message from the receiver with the AckNo value equal to or greater than the sequence number of the last byte sent is not received within a short period of time the last message is retransmitted. Normally this bit will not be set, and acknowledgements will flow back to the transmitter with any data travelling in the opposite direction. The Host can specify an EndOfMessage bit, which is equivalent to the Push bit of BSP, and which causes the RequestAcknowledge bit to be set too.

When the receiver gets a message it checks that SeqNo is equal to the sequence number of the last byte it received plus one. If it is less then this is a retransmission, and indicates that an acknowledgement was lost, so the receiver transmits another acknowledgement. If it is equal then this is the next message in sequence and it increments AckNo by the size of the data in the message and decrements the Credit; if the RequestAcknowledge bit is set it transmits an acknowledgement. If the SeqNo in the message is greater than expected then the previous message has been lost, and the receiver sends an acknowledgement with the SequenceError bit set. When it receives such a message the transmitter restarts transmission of all data after the last byte acknowledged. Since the error message itself may have advanced AckNo this will normally result in the retransmission of just the message that was lost and the one that provoked the error.

Buffers are returned to the Host at both ends whenever AckNo advances past their last byte. When the transmitter has no more data to send or the credit is zero, the stream will be idle and no messages will normally flow. To detect a crash of a machine during this period an idle exchange can be produced if the transmitter sends a message containing no data but with the RequestAcknowledge bit set. This will force the receiver to respond; if it does not (and it should do so immediately) then a crash of the receiver is indicated. This also allows new credit to flow from the receiver to the transmitter when there is no data to send.

To enable the Hosts to send control information without defining escape sequences in the main stream the transmitter should be able to mark, or qualify, certain data buffers as Control data. The receiver should put any messages whose data is so marked into a separate set of buffers. As in BSP these control buffers cannot contribute towards the credit, but neither can they have their own flow control since they should retain their position in the stream. There is nothing to stop them having their own credit, however. Whenever the receiver gets a new control data buffer it sends a message to the transmitter with the RxControl bit set, SeqNo and AckNo can be the same as usual, but the Credit is the amount of control buffering available, not the amount of ordinary buffering. When the transmitter comes across a control data buffer in its input stream it first ensures that all previous data has been acknowledged by sending a message with the RequestAcknowledge and RequestControlCredit bits set. This causes the receiver to return the Control Credit as described above, and guards against the loss of credit in earlier messages. Data can now be transferred in exactly the same way as before. When the transmitter runs out if control data it ensures that is has all been acknowledged, and

returns to transmitting ordinary data. The actions required of the transmitter in switching from ordinary to control data and back are identical with the exception of the setting or clearing of certain bits. The only real difference between ordinary and qualified data is the set of buffers they are delivered into at the receiver.

The advantage of this protocol over BSP is its flexibility; the client can tailor its behaviour to the application. The frequency of acknowledgements is an example of this; by setting the RequestAcknowledge bit on every message we have a lock-step protocol similar to BSP. On the other hand, by only setting it at the end of a long transfer there need only be a single positive acknowledgement, although negative, sequence error, acknowledgements will occur at any time. In this mode the protocol can be used for high speed bulk data transfer with little overhead.

It should be possible to implement this protocol simply. The transmitter, for example, needs to keep just a single queue of buffers. The head of the queue points to the oldest byte yet to be acknowledged, while a pointer into the queue points to the next byte to be sent. Whenever a message from the receiver arrives the head of the queue is advanced and any buffers returned. The head of the transmitter's queue always represents, therefore, the point to which it must return to retransmit after a sequence error. The sequence numbers themselves should be large, 32 bits, as should the credit field. This means that the credit should never grow to be more than 2^{31} to avoid problems when comparing sequence numbers. The use of such large values makes the stream overheads in each message larger than that of BSP, by about a factor of three. This is balanced, however, by not having to return an explicit acknowledgement to each data message. As the errors in the network increase this protocol will do more work than BSP since it will tend to retransmit more data each time a message is lost. The protocol is therefore only suitable for low error networks.

The message format is devised to allow the receiving NIP to decode it on-the-fly. In particular it can do the sequence number comparison and route the data to the correct buffers as the message is arriving. As for BSP the NIP must maintain two timers: an idle timer which is reset each time a message is received, and should be long; and a transmission timer which is reset whenever a message is transmitted, and whose value can be small. The action to be taken when the transmission timer expires depends on which end of the protocol this is, and what the last block sent was. Normally the expiry of the timer results in no action. If the transmitter sees a timeout after sending a message with the RequestAcknowledge bit

set it should repeat it; similarly if the receiver sees a timeout after sending a sequence error acknowledgement it should retransmit.

So far we have considered only an established stream, how should it be opened and closed? BSP uses an Open exchange that is similar to SSP and is used to negotiate the message sizes and communicate port numbers. This stream only need to exchange port numbers, and this can be performed in a simple interaction. Rather than define a full RPC-like open protocol specific to the stream protocol the initial connection and user parameter exchange should be made via the standard RPC protocol. A special exchange to close the stream is not necessary either, since all that is required is that both ends agree, which can be achieved by consent at a higher level. The NIPs must, of course, be able to handle unilateral action, but this should be treated at all levels as a serious error.

9.2 Closing the Network

Having designed our protocols with a NIP in mind the next step is to consider what would happen if we placed all machines on a network behind an intelligent interface. Such a configuration can be called a Closed Network because it is not now open to direct use (or abuse) by the Host.

The first advantage of this is in the area of protection. We have already seen how descriptors contribute a degree of protection to a single machine with a NIP. If all machines are equipped with NIPs then this can be extended throughout the network. The mechanism can be expanded to bring the otherwise divergent concepts of descriptor, stream identifier and UID-set identifier under the same umbrella. Such items are 'tickets of permission' to communicate with a paricular service, use a protocol instance, or be identified as a particular user. Conventionally these are known as capabilities [Fabry74], and are protected against forgery either by being encrypted, chosen from a sparse set, or by being hidden from the user by the operating system or machine hardware. In this case the capabilities are protected by being stored in the NIP's memory, which is inaccessible to the Host. Definition of the protocols to include these Network Capabilities in messages can greatly simplify the passing of addresses and other information that is of interest to the NIPs.

Another advantage of the Closed Network derives from the trust one NIP can have in another. Any messages the NIP receives over the network are expected to be correct, since they can only have originated at another NIP. Similarly the NIP knows what will happen with any messages it sends.

This means that protocols may be designed to be lightweight, and can make assumptions about the behaviour of the other end because there is no question of it being a bad, faulty or malicious implementation (this does not mean, however, that we can ignore the problems of lost messages or machines). This can also make security and authentication a little easier: a NIP can pass things like UID-sets or encryption keys to a remote NIP in the safe knowledge that the Host will not get hold of it. This feature of trustworthiness relies on the assumption that untrustworthy nodes cannot be added to the network. This is valid in networks where the transmission medium must be cut to introduce new nodes, as in the Cambridge Ring, but is not in system like Ethernet where a station can be tapped onto the ether at any point.

A Closed Network will be somewhat more closely coupled than the current Cambridge Distributed System owing to the unifying nature of the NIPs. It would not, however, be so closely coupled as a multiprocessor system, or a "Distributed Operating System" like Amoeba [Tannenbaum81], SODS/OS [Sincoskie80], Accent [Rashid81], or Roscoe [Solomon79]. This is because the user remains free to run whatever program or operating system he desires in the Host, without having to have a fixed protected kernel or nucleus. A consequence of this is that the NIPs work at a machine-to-machine level and not, as the above do, at a process-to-process level. On a network where most of the machines are expected to be personal computers this is not really significant, although support for multiple processes can be included in the NIP.

9.2.1 Protocols in the Closed Network

The distributed operating systems mentioned above have a somewhat restricted view of communications. They all limit themselves to just one communication protocol: either simple uni-directional message passing or a Transport Service level stream (Amoeba). This limitation is justified by the (valid) observation that any other protocols may be built as extra layers in the Host, or by engineering special cases (Amoeba allows a "Secure Datagram" to be sent by enabling the stream to be opened, used and closed in just one message). At the other end of the spectrum lies the Cambridge Distributed System which relies on convention to prevent the proliferation of ad-hoc protocols.

This second approach is unacceptable because it tends to imply access to the network at a low level, with the corresponding compromise of protection. The first approach is also unacceptable because it places the client in the position of having to do a lot of work, or accept gross

inefficiencies, if his pattern of use does not match that assumed by the system implementors. For example, while it is possible to perform remote procedure calls down a byte stream, or implement a stream protocol using remote procedure calls, both suffer from a mismatch of application and protocol.

We must therefore adopt a medium level approach and supply a small set of protocols whose use can be controlled by the NIP, but which serve the client's needs fully and efficiently. This is similar to the problems faced by a language designer in providing sufficient facilities to allow any program to be expressed, but few enough to keep the compiler down to a controllable size. If we consider the mechanisms supplied by most operating systems to access objects under their control we find that two are outstanding. Operating systems represent objects either as streams of records or bytes (files, terminals, printers etc.)[1], or as procedures (O/S primitives, library modules etc.). The most frequently used protocols in local area networks also happen to be byte streams and remote procedure calls. This is not really surprising, distributed systems tend to take their model of the desired operations from conventional operating systems.

The most suitable protocols to supply, then, are RPC and byte stream protocols. Those described in the previous section would be suitable. Provided the interfaces are defined flexibly enough they should cater for most user need. There do remain, however, some applications for which these protocols are inappropriate or too costly. An example of this is the protocol that is used between the Fileserver and its Garbage Collector to report new preservations; which uses repeated, unacknowledged transmission to ensure data delivery. These applications can be catered for by a simple datagram protocol. This is not the same as giving the Host access to the basic transport protocol, and can be as fully protected as any other protocol.

9.2.2 Network Capabilities

Network Capabilities can be used to represent many things, but all have several common attributes that must be recognised by the NIP. The first of these is its type; the NIP will recognise several types of capability and only allow certain operations on each type. The second is the identity of the creator, this will usually be the global address of the NIP that created the capability.

1 That streams are accessed with procedure calls should not disguise the underlying abstraction.

The third common attribute is the access field, which is simply a field of bits some of which are defined by the capability's type and others of which may be defined by the client. When stored in a NIP there should be two copies of this field: the Local and the Remote access fields. Whenever the capability is transmitted to another machine it is the Remote access field that is sent. This allows the client to refine the access in a capability without making a copy, and without losing his own rights.

The last common attribute is a group number which serves two related purposes. First, it enables the NIP to filter out any messages that are directed to a previous incarnation of the Host system since the group to which they are directed will not be valid. Secondly, by allowing the Host to define several groups it is possible to allow different processes in one Host to appear on the network as separate entities and to handle process crashes by the same mechanism as machine crashes. To ensure that groups are not re-used immediately they should be obtained from a monotonic sequence and the oldest and newest groups in existence recorded in stable storage. This is exactly the same property required of the UID in an RPC call, which can now be constructed from the group number of the caller plus a sequence number.

In addition to the above fields there is the type-dependent information that the capability encapsulates. In theory the capabilities can be made of variable size, but for practical reasons of store management it is expedient to make them a fixed size with eight or sixteen bytes of type dependant information. This should be sufficient for most purposes.

9.2.2.1 Protocol Capabilities

Among the types of capability the NIP will recognise the most important are those that represent the three protocols. A protocol capability effectively has two forms. In the creator's NIP it is in local form, and may not be passed out to any other machine. Attempting to transmit it elsewhere causes it to be converted into the remote form. An instance of a protocol is therefore represented by a single local capability and many remote capabilities.

These different forms are easily implemented by setting up the local and remote access fields of the newly created capability accordingly. For datagram capabilities only the holder of a local capability is able to receive messages, and possessors of the remote capability are only able to transmit to it. Similarly the holders of a remote RPC capability are only able to make calls to the holder of the local capability, which is the only machine that can receive requests.

Also associated with RPC are reply type capabilities, which represent the return link to the caller. The state associated with a current call can be associated in the client with the reply capability, freeing the local RPC capability to accept further calls. A useful facility results if we allow reply capabilities to be passed on: the reply to a request need not come from the original service. This would be particularly useful for distributed services, the request is made to the closest or most convenient instance which passes the request on to another part that executes the request and replies. It would also help with the implementation of dynamic services, which are created only in response to a request to use them.

Stream capabilities are slightly different in that they allow data transfer in both directions. When a stream is initially created it is in passive state, and occupys no more memory than it takes to store the capability. When the holder of a remote stream capability wishes to open a stream it simply activates it, at which point the NIP allocates any data structures necessary to represent the stream. It also sends an initial message to the local end to inform it of the address to be used for stream messages. If the local capability has also been activated the stream will be created, otherwise the NIP waits until it is, or reports a failure (this choice can lie with the client). While the stream is active the remote capability is fixed in place and cannot be transferred elsewhere. The stream is closed by deleting the capability.

9.2.2.2 User Capabilities

The protection provided by the capability mechanism may be extended for the use of clients and servers, and is safe so long as the user created capabilities cannot be confused with 'system' capabilities. It is expected that the active objects in the system will be represented by one or other of the protocol capabilities. For example, an open file may be represented by an open stream between the client and the Fileserver. The purpose of user capabilities is to name passive objects, and to represent access rights to them. Such capabilities should have a lifetime longer than a single incarnation of the service that created them. A Fileserver, for example, will give out capabilities that represent handles onto files; it will want to treat such capabilities as valid even if they were handed out by a previous incarnation of itself.

To segregate user capabilities into separate domains they may only be manipulated if the client can present a Type capability to the NIP. A Type capability defines the value of the type field in the user capability and confers upon its holder certain rights regarding these. Depending on the access field the posessor of a Type capability is able to examine the

contents, and create new capabilities of that type. In the absence of a Type capability the contents of a user capability is protected against both reading and writing.

While it would be possible to define a general mechanism to allow the dynamic creation of capability types this is not necessary. The number of capability types needed in the system is small and fixed, and these types can be allocated statically by a higher authority than exists in the network. In this respect they have similar properties to names in the Nameserver, or Authentities in the present authentication system.

9.2.3 The Transmission of Capabilities

The mechanism for transferring capabilities between machines must be defined to preserve their protected status; it is not possible to mix them with ordinary data. We could define a special capability transfer protocol for this purpose, but this does not match the higher level requirements. When making a remote procedure call the client will want to submit both capabilities and ordinary arguments to the service, and expect both in return. It would be unweildy and expensive to do this with two different mechanisms, so the standard protocols must be modified to allow capability transfer. This is best achieved by splitting the message into two parts, one for capabilities and one for normal data allowing the NIP to extract the capabilities on-the-fly. While this mechanism is acceptable for datagrams and RPC, it introduces the added problem of capability flow control in the stream protocol. However, a stream will largely be used for bulk data transfer, in which capabilities are not expected to appear. Any capability transfer needed can be carried out either in the initial open exchange, or with RPC's or datagrams in parallel with the stream.

The distinction of local and remote capabilities while at first sight restricting is exactly identical to the mechanism in use at present; the remote capability simply encapsulates the address of the local end. Remote capabilities will under normal circumstances be lodged in a Nameserver for general access. The exception to this is expected to be stream capabilities. Any service that is accessed via a stream will present an RPC interface initially, if it is satisfied with the request it will then pass a remote stream capability back in the reply. Streams will, therefore, only be used once.

The restrictions are imposed for purely practical reasons: if the service (local) end of an RPC protocol instance were moved it would either be necessary to inform the holder of every remote capability for that

service, or record its new home in the original NIP so that any request that arrived for it could be redirected. The first requires that the creating NIP be informed every time a remote capability is moved or copied, and generates a lot of traffic when a local capability is moved. The second can result in a request being redirected many times, and may even generate loops.

9.2.4 The Nameserver and Initial Capabilities

When a machine starts up it must have some capabilities to allow it to communicate with other machines; and if it is to have access to a user capability type it must obtain the necessary Type capability. These can either be obtained from the Nameserver, or be already present when the Host starts. The first method allows dynamic binding of names to addresses, while the second allows the rights and privileges of the client to be established in a safe way. It is in fact useful to have both these mechanisms. Initial capabilities may be installed in two ways, depending on the form of the Host. A machine, such as a processing server, which is loaded remotely from the network, will receive its initial capabilities in the loading sequence. Other machines, such as a Fileserver, which bootstraps from a different source cannot obtain their capabilities in this way. The capabilities must already be present in the NIP. Specifically, the capabilities can be stored in the NIP's stable storage and be read out into the usual capability store when the machine restarts. The capabilities must be installed in the NIP when the service is initially created, after this there is no need to change them.

The Nameserver required by this system is somewhat more complex than the simple static Nameserver in use on the ring at present. It need not, however, be a fully dynamic Nameserver, allowing the addition and deletion of names. Instead it must allow the bindings between names and capabilities to be altered as services move or come into existence. To prevent the binding being altered by anyone, the Nameserver must support a user capability type which allows a name/capability binding to be altered. A new capability will only be entered against a name if the requestor can produce a capability which controls that binding. In general only the service to which the name applies will need to do this, and will receive the capability in its initial set. By defining the Nameserver as an RPC service and ensuring that each machine is passed a capability for this in its initial set, any dependency upon the design of the Nameserver can be removed from the NIP.

9.2.5 Invalid Capabilities

The crash of a machine, its NIP, or an inter-network bridge will render some or all of the capabilities it has passed out to other machines invalid. If any of these machines attempts to use one of these capabilities it will either be unable to contact the owner, or have its communication rejected because it is addressed to an earlier incarnation. The response of SuperMACE to a failure of this kind was to attempt to look up the name again in the Nameserver, if this was appropriate. Since the Closed Network NIP does not have any knowledge of the Nameserver this is not now possible. A generalisation of this mechanism would be to record with each capability the source from which it was obtained, and generate an appeal to that address if the capability becomes invalid. However, it is not possible for the NIP to know what the capability represents to the Host, or what other information or capabilities the Host had to supply in order to obtain it. The invalidity of this capability may be a symptom of a larger system failure which can only be corrected by alternative or drastic action at higher levels. This is very similar to the problems discussed in section 5.2.2 regarding on-the-fly protocol decoding; there is no way to convey all the relevant information to the lower levels which allows them to make the correct decision in all circumstances. The only useful action the NIP can take, therefore, is to respond to the Host with an 'Invalid Capability' return code.

9.2.6 Authentication and Security

The use of protected capabilities greatly simplifies authentication. The client of a service can be sure that if the capability he possesses for a service is valid, then it will connect him with a genuine instance of that service. Similarly the service can assume that only bona fide clients will possess a capability for its entry point. The class of access allowed to the client is modified by the access bits in the capability which are also transmitted in the datagram/request message.

This is only valid if an eavesdropper cannot introduce bogus messages into the network. If he can then it is necessary to protect all communications with encryption. At the least the capability segments of messages must be encrypted, and if the clients are also exchanging sensitive data, the data segments too. Since the capability mechanism handles authentication, we are only concerned here with excluding the eavesdropper. Encrypted connections need only be established on a machine-to-machine or group-to-group basis, using the protocol given in section 8.3. Such connections need only be established between machines

when they communicate for the first time, and can be timed out after a period of non-use. This means that the Authentication Server and the entire encryption mechanism is below the level of the clients, which need not be aware of its existence.

9.2.7 Systems Aspects

The above description has been given assuming a network-wide system similar to the Cambridge Distributed System. A feature of this that does not transport well into the closed network is the use of small, single task, computers to implement most of the management functions of the system. It is not economically feasible to place such a small machine behind a NIP, which is of comparable size. While it would be possible to implement such servers in stand-alone NIPs this is a potential source of protection breaches. The alternative is to collect all these management functions together into one larger machine. While this goes against the spirit of distributed computing, it does have several advantages. First, since this machine will be of the same type as the processing servers, it reduces the number of different hardware and software configurations that needs to be supported. Secondly, the servers can share code for common activities, for example saving state and, of course, protocol handling. Thirdly, it allows reliability through redundancy to be included in the system since it will be easier to add a second 'shadow' machine ready to take over from the first if it fails.

The operating system installed in the Host is chosen by the user. It may be a standard operating system like Tripos or UNIX, or it may be a special system designed to fully exploit the closed network environment. Such an operating system would undoubtedly have to be capability based, perhaps using the NIP's user capabilities to represent its own internal ones in the absence of memory protection.

Chapter 10

Summary and Conclusions

This dissertation has been concerned with the design and application of Intelligent Network Interfaces.

The first device described was the Type 2 which implements Basic Block Protocol only. An attempt to supply a more flexible interface to this protocol was made in the provision of buffer chaining facilities. While these have not proved to be particularly successful the improvement that the Type 2 contributes to overall performance is clear. The principal conclusion to be drawn from the Type 2 is that significant performance gains are to be had in moving protocol processing out of the Host. In this respect the Type 2 serves to lay the foundations for the rest of the work described here.

The MACE differs from the Type 2 in that it contains a more conventional processor; what it loses in raw speed it makes up for by being more powerful. The first program written for this device, SPECTRUM, was neither machine independent nor sophisticated, modelling its facilities on those required by the Tripos operating system. This has had the effect of reducing the Host software considerably compared with the Type 2, although the difference in specification is small.

Both these interface suffer from the somewhat simple design of the hardware, which requires the NIP to participate in each packet transferred. This is caused partly by the lack of checksum hardware, which the NIP is required calculate itself on-the-fly, adding a delay before the next packet can be processed. This delay is not so great for the Type 2 owing to the faster processor, but is paid for with the instruction set.

The bulk of this dissertation describes the High Level Interface. The advantages of such an approach are threefold. First, the protocol implementation itself moves to a more benign environment, with a consequent increase in performance and decrease in complexity, since it can cooperate more closely with other levels. Secondly, the Host will lose much necessarily resident, and CPU-hungry, software, releasing more resources

for other purposes. These both contribute to a better system performance being perceived by the user. The third advantage is that the High Level Interface is able to conceal the true nature of the underlying network and protocols. This makes the Host software independent of these, and allows them to be altered transparently.

The High Level Interface implemented in the MACE demonstrates superior performance to SPECTRUM in both the major high level protocols implemented (SSP and BSP). This is true even though it suffers from the same hardware infelicities as the earlier program, and cannot take advantage of the same software optimisations. The improvements observed in Tripos are almost entirely due to the increased efficiency of the protocol implementation, since the freed CPU cycles in the Host are all absorbed by the idle task. This leads one to expect that a time-shared Host would benefit in both areas.

With the expected performance of computers of all types increasing continuously, any means to achieve this is welcome. The Network Interface Processor is one way of improving the performance of machines connected to a Local Area Network by offloading the network related processing. As the cost of hardware decreases, and the complexity of components increases, this approach becomes more attractive for smaller machines. It is therefore reasonable to assume that the NIP will become more common, particularly for the larger machines. This leads on to consider the possible benefits and advantages of placing all the Host computers on a network behind NIPs.

Several areas for further research present themselves. First, the hardware design of a NIP needs to be elaborated; the features presented in section 8.2 have not, at the time of writing, been fully realised. There is scope here for some exercises in VLSI design, for example: a DMA controller that may be remotely controlled. The problems of NIP design for high speed networks must also be addressed. Second, the design of protocols that exploit the features of a NIP may be developed. Two such protocols were given in chapter 9: the first used the NIP to provide a more reliable service (RPC) at roughly the same cost as a simpler one (SSP), the second attempted to rectify some of the problems faced in implementing BSP in a NIP. The third area of research is the full specification and development of a closed network. The construction of a network and system based on these principles would be a major undertaking.

References

Banerjee83:

Aspects of Fast Local Area Networks; R.Banerjee; Ph.D. Thesis, University of Cambridge, 1983.

Biba79:

FordNet: A Front-End Approach to Local Computer Networks; Proc. MITRE/NBS Conference on Local Area Computer Networks, Boston, May 1979.

BT80:

A Network Independant Transport Service; prepared by Study Group 3 of the British Telecom User Forum; February 1980.

Carpenter78:

A Microprocessor Based Local Network Node; J.Carpenter, J.Sokol Jr., R.Rosenthal; 17th IEEE Computer Society International Conference, Fall 1978, pp 104-109.

Clark82a:

An Alternative Protocol Implementation; D.D.Clark; Systems Research Group Document, University of Cambridge Computer Laboratory, 10th May 1982.

Clark82b:

Considerations of Message Based System Design; D.D.Clark; Systems Research Group Document, University of Cambridge Computer Laboratory, 10th May 1982.

Collinson82:

The Cambridge Ring and UNIX; R.P.A.Collinson; Software - Practice and Experience 12(6) pp 583-594, June 1982.

Dellar80:

The Distribution of Operating System Functions; C.N.R.Dellar; Ph.D. Thesis, University of Cambridge, 1980.

Dion81:

Reliable Storage in a Local Network; J.Dion; Ph.D. Thesis, University of Cambridge, Feb 1981.

DES75:

Proposed Federal Information Processing Data Encryption Standard; US National Bureau of Standards; August 1, 1975.

Fabry74:

Capability Based Addressing; R.S.Fabry; Communications of the ACM, 17(7), July 1974, pp 403-412.

Farber75:

A Ring Network; D.J.Farber; Datamation, 21(2), pp 44-46, Feb 1975.

Fuller78:

Multi-microprocesors: An Overview and Working Example; S.Fuller, J.K.Ousterhout, L.Raskin, P.Rubinfeld, P.Sindhu, R.Swan; Proc. IEEE 66(2), pp 216-228, Feb 1978.

Garnett80:

An Asynchronous Garbage Collector for the Cambridge File Server; N.H.Garnett, R.M.Needham; Operating Systems Review 14(4), pp 36-40, Oct 1980.

Gibbons80a:

The Design of Interfaces for the Cambridge Ring; J.J.Gibbons; Ph.D. Thesis, University of Cambridge, Sept 1980.

Gibbons80b:

BSPLIB - An Implementation of BSP for RSX11M; J.J.Gibbons; Ring Documentation, University of Cambridge Computer Laboratory, 5th Dec 1980.

Gibbons81a:

Design of a new High Performance Ring Interface; J.J.Gibbons; SRG Note, 18th May 1981.

Gibbons81b:

A Forward Transmit Buffer for the GIZMO; J.J.Gibbons; GIZMO Design Note, 17th August 1981.

Gibbons81c:

Efficient Manipulation of Byte Buffers on a Word Machine; J.J.Gibbons; GIZMO Design Note, 5th June 1981.

Gifford81:

Cryptographic Sealing for Information Secrecy and Authentication; D.K.Gifford; Preprints for 8th SOSP, pp 30-43.

Girling82:

Object Representation on a Heterogeneous Network; C.G.Girling; Operating Systems Review, 16(4), pp 49-59, Oct 1982.

Heart70:

The Interface Message Processor for the ARPA Computer Network; F.E.Heart, R.E.Khan, S.M.Ornstein, W.R.Crowther, D.C.Walden; Proc. AFIPS SJCC, 1970, pp 551-567, reprinted in Siewiorek, Bell & Newell, Computer Structures: Principles and Examples, McGraw-Hill, 1982.

Hopkins81:

Recent Developments on the MITRENET; G.T.Hopkins; Local Networks & Distributed Office Systems, pp97-105, May 1981.

ISO79:

Reference Model of Open Systems Interconnection; International Standards Organisation; ISO/TC97/SC16 No. 309, August 1979.

JNT82:

Cambridge Ring 82 Protocol Specification; prepared by the Joint Network Team of the Computer Board and Research Councils; November 1982.

Johnson80:

Byte Stream Protocol Specification; M.A.Johnson; Systems Research Group Documentation, University of Cambridge Computer Laboratory, April 1980.

Johnson82:

Byte Stream OPEN Blocks; M.A.Johnson; Systems Research Group Documentation, University of Cambridge Computer Laboratory, 25th

Oct 1982.

Jones79:
StarOS, a Multiprocessor Operating System for the Support of Task Forces; A.K.Jones, R.J.Chansler Jr., I.Durham, K.Schwans, S.R.Vegdahl; Proc. 7th SOSP, pp 117-127.

Knight82:
Portable Systems Software for Personal Computers on a Network; B.J.Knight; Ph.D. Thesis, University of Cambridge, April 1982.

Knuth73:
The Art of Computer Programming, Vol I: Fundamental Algorithms; D.E.Knuth; Addison-Wesley, 1973.

Lauer78:
On the Duality of Operating Systems Structures; H.C.Lauer, R.M.Needham; in Proceedings of the 2nd International Symposium on Operating Systems, IRIA, October 1978, reprinted in Operating Systems Review 13(2), April 1979, pp 3-19.

Lee78:
Interface Processor for High Speed Recirculating Data Network; C.C.Lee, A.V.Pohm; 17th IEEE Computer Society International Conference, Fall 1978, pp 194-199.

Leslie81:
Organization of Voice Communication on the Cambridge Ring; I.M.Leslie, R.Banerjee, S.J.Love; Local Networks & Distributed Office Systems, pp 465-490, May 1981.

Leslie83:
Extending the Local Area Network; I.M.Leslie; Ph.D. Thesis, University of Cambridge, Februrary 1983.

Mark81:
Protocol Model for WELNET; J.W.Mark; Online Conference on Local Area Networks and Office Systems, pp 303-317, March 1982.

Metcalfe76:
Ethernet: Distributed Packet Switching for Local Area Networks; R.M.Metcalfe, D.R.Boggs; CACM, 19(7), pp 77-65, July 1976.

Mockapetris77:

On the Design of Local Network Interfaces; D.V.Mockapetris, M.R.Lyle, D.J.Farber; Information Processing '77, pp 427-430, IFIP, Toronto Aug 1977.

Needham78:

Using Encryption for Authentication in Large Computer Networks; R.M.Needham, M.D.Schroeder. CACM, 21(12), pp 993-999, Dec 1978.

Needham82:

The Cambridge Distributed Computing System; R.M.Needham, A.J.Herbert; Addison-Wesley, 1982.

Needham83:

How to Connect Stable Memory to a Computer; R.M.Needham, A.J.Herbert, J.G.Mitchell; Operating Systems Review, 17(1), p 16, Jan 1983.

Nelson78:

Computer Cells - a Network Architecture for Data Flow Computing; D.L.Nelson, R.L.Gordon; 17th IEEE Computer Society International Conference, Fall 1978.

Nelson81:

Remote Procedure Call; B.J.Nelson; Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

Ody79:

A Single Shot Protocol; N.J.Ody; Systems Research Group Document, University of Cambridge Computer Laboratory, April 1979.

Ody81a:

The "Standard" PROM Program of the Type 1 System; N.J.Ody; Systems Research Group Documentation, University of Cambridge Computer Laboratory, 12th January 1981.

Ody81b:

Monitoring Ring Traffic using a "promiscuous" station; N.J.Ody; SRG Note, 9th June 1981.

Ody81c:

Z80 Loading and Debugging Protocols; N.J.Ody; SRG Documentation, 30th Nov 1981.

Ousterhout79:

Medusa: An Experiment in Distributed Operating System Structure; J.K.Ousterhout, D.A.Scelza, P.S.Sindhu; Preprints for 7th SOSP, pp 395-404.

Rashid81:

Accent: A Communications Oriented Network Operating System Kernel; R.F.Rashid, G.G.Robertson; Proceedings 8th SOSP, pp 64-75.

Richards79a:

TRIPOS - A Portable Operating System for Minicomputers; M.Richards, A.R.Aylward, P.Bond, R.D.Evans and B.J.Knight; Software - Practice and Experience, June 1979.

Richards79b:

BCPL: The Language and its Compiler; M.Richards, C.Whitby-Strevens; Cambridge University Press, 1979.

Richardson83:

Filing System Services for Distributed Computer Systems; M.F.Richardson; Ph.D. Thesis, University of Cambridge, 1983.

Rubinstein81:

Terminal Support on the Cambridge Ring; M.J.Rubinstein, C.J.Kennington, G.J.Knight; Local Networks & Distributed Office Systems, pp 475-490, May 1981.

Shrivastava81:

The Design of a Reliable Remote Procedure Call Mechanism; S.K.Shrivastava, F.Panzieri; University of Newcastle, Technical Report 171.

Sincoskie80:

SODS/OS A Distributed Operating System for the IBM Series I; W.D.Sincoskie, D.J.Farber; Operating Systems Review 14(3), pp 46-54, July 1980.

Solomon79:

The Roscoe Distributed Operating System; M.H.Solomon, R.A.Finkel; 7th SOSP, pp 108-114, Dec 1979.

Sommer81:

A Real-Time Protocol for a Sub-Local Network; R.Sommer; Local Networks & Distributed Office Systems, pp263-275, May 1981.

Stack81:

LAN Protocol Residency Alternatives for IBM Mainframe Open System Interconnection; T.Stack; Local Networks & Distributed Office Systems, pp435-450, May 1981.

Tannenbaum81:

An Overview of the Amoeba Distributed Operating System; A.S.Tannenbaum, S.J.Mullender; Operating Systems Review 15(3), pp 51-64, July 1981.

Thacker79:

Alto: A Personal Computer; C.P.Thacker, E.M.McCreight, B.W.Lampson, R.F.Sproull, D.R.Boggs; Siewiorek, Bell & Newell, Computer Structures: Principles and Examples, 2nd edition, McGraw Hill, 1982.

Thornton70:

Design of a Computer: The Control Data 6600; J.E.Thornton; Scott, Foresman and Co.

Tymes71:

TYMNET - A Terminal Oriented Communication Network; L.R.Tymes; AFIPS SJCC Proceedings Vol. 38, Spring 1971.

Walker78:

Basic Ring Transport Protocol; R.D.H.Walker; Systems Research Group Document, University of Cambridge Computer Laboratory, Oct 1978.

West78:

CNET - A Cheap Network for Distributed Computing; A.R.West, A.Davison; Technical Report 120, Computer Systems Laboratory, Queen Mary College, University of London, March 1978.

Wilkes79a:

The Cambridge Digital Communications Ring; M.V.Wilkes, D.J.Wheeler; Proc. Local Area Communications Networks Symposium, Mitre and NBS, Boston, May 1979.

Wilkes79b:

The Cambridge CAP Computer and Its Operating System; M.V.Wilkes, R.M.Needham; North Holland, 1979.

Wirth80:

Modula-2; N.Wirth; ETH Zurich, 1980.

Wulf74:

HYDRA: The Kernel of a Multiprocessor Operating System; W.Wulf, E.Cohen, W.Corwin, A.Jones, R.Levin, C.Pierson, F.Pollack; CACM, 17(6), pp 357-345, June 74.

Xerox81a:

Internet Transport Protocols; Xerox System Integration Standard 028112; Xerox Corporation, 1981.

Xerox81b:

Courier: The Remote Procedure Call Protocol; Xerox System Integration Standard 028112; Xerox Corporation, 1981.