①

NPS52-88-028
NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISTRIBUTED COMPUTER COMMUNICATIONS
IN SUPPORT OF
REAL-TIME VISUAL SIMULATIONS

Theodore H. Barrow

John M. Yurchak

Michael J. Zyda

September 1988

88 10 18 158

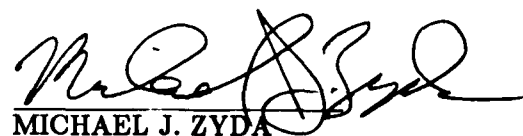## NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. C. Austin
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research conducted for the Naval Ocean Systems Center and the Naval Underwater Systems Center and funded by the Naval Postgraduate School.

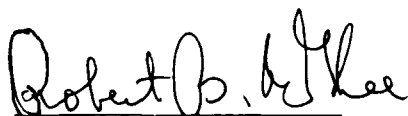Reproduction of all or part of this report is authorized.
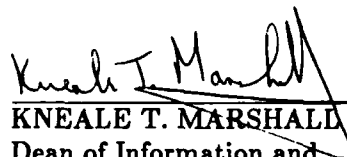
This report was prepared by:

MICHAEL J. ZYDA
Associate Professor
of Computer Science

Reviewed by:

Released by:

ROBERT B. MCGHEE
Chairman
Department of Computer Science

KNEALE T. MARSHALL
Dean of Information and
Policy Science

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE				
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-028		5 MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL *(If applicable)* 52	7a. NAME OF MONITORING ORGANIZATION 1) Naval Ocean Systems Center 2) Naval Underwater Systems Center		
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943		7b ADDRESS *(City, State, and ZIP Code)* 1) San Diego, CA 92152 2) Newport, RI 02841		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Naval Postgraduate School	8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER O&MN, Direct Funding		

8c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*

DISTRIBUTED COMPUTER COMMUNICATIONS IN SUPPORT OF REAL-TIME VISUAL SIMULATIONS (U)

12. PERSONAL AUTHOR(S)
BARROW, Theodore H., YURCHAK, John M., and ZYDA, Michael J.

13a. TYPE OF REPORT Summary	13b TIME COVERED FROM 87/10 TO 88/09	14. DATE OF REPORT *(Year, Month, Day)* 1988, Sep	15 PAGE COUNT 168

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18 SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	distributed systems, visual simulation, workstations,
			Theses. (Appl)

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Complex visual simulations can strain the capability of a single workstation. A mix of different workstations is often more economical than the use of a large processor for such simulations. Methods of communicating between such workstations are needed that allow the developer to spend effort on the simulation and not on communications. Simple protocols are developed to support both broadcast and direct-connect communications between workstations using TCP/IP on an Ethernet network. Comparisons are made between broadcast and direct connect protocols.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael J. Zyda	22b TELEPHONE *(Include Area Code)* (408)646-2305	22c OFFICE SYMBOL 52Zk

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

Distributed Computer Communications
in Support of
Real-Time Visual Simulations

Theodore H. Barrow, John M. Yurchak and Michael J. Zyda *
Naval Postgraduate School
Code 52, Dept. of Computer Science,
Monterey, California 93943-5100

ABSTRACT

Complex visual simulations can strain the capability of a single workstation. A mix of different workstations is often more economical than the use of a large processor for such simulations. Methods of communicating between such workstations are needed that allow the developer to spend effort on the simulation and not on communications. Simple protocols are developed to support both broadcast and direct-connect communications between workstations using TCP/IP on an Ethernet network. Comparisons are made between broadcast and direct connect protocols.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

* Contact author.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The Graphics and Video Laboratory of the Department of Computer Science at the Naval Postgraduate School permits the researcher to create three-dimensional visual simulations from digital terrain data [Ref. 1]. Specialized graphics hardware allows the display of such simulations in near-real time. The goal of a good part of the work in the lab is the creation of a movie-like view of movement over and on terrain, with increasingly complex movement animation models. Such projects have strained the equipment's capabilities. One method of increasing available computing power is to harness multiple heterogeneous machines together in some distributed computing organization. It requires communication between the various machines, as well as carefully matching each machine's capabilities to its assigned tasks.

A. PROBLEM

Rapid turnover of inexperienced students at the Naval Postgraduate School makes the creation of complex simulations difficult to manage. The learning curve becomes steeper as the lab's capabilities increase. One of the areas of difficulty has been inter-computer communications. So much time has been spent on designing, coding, and debugging communication software, little has been left for the original research. We set out to provide an easy-to-use, yet powerful, set of tools to aid in the development of multi-computer projects.

1. Approach

A communication protocol can be optimized for large data transfers, or small data transfers, or both. Efforts to optimize for both are both complex and difficult [Refs. 2, 3]. File transfer protocols such as FTP in the Defense Advanced Research Project Agency (DARPA) Internet domain and *uucp* in the UNIX domain can be used for

1

large data transfers. Their overhead[1] is high. This overhead cannot be tolerated in a real-time problem[2]. Our *visual simulation* efforts rely on small data transfers to communicate among machines. These small messages are typically commands and changing status indicators. Transferring the entire "world view" is only a reasonable task during initialization or reset. Hence, we designed our protocols for small messages.

2. Design Criteria

The design criteria for developed protocols were simplicity, ease of use, portability, and efficiency. Rapid turnover of inexperienced students at the Naval Postgraduate School makes simplicity of paramount importance. Inevitably, changes will be required and only a simple protocol is easily modified to take advantage of new capabilities. Much the same argument, and generally good software design practice, make ease of use only slightly less important. Almost all operating system-level aspects are hidden from the application. The number of other machines to be connected to, a use of dynamic memory allocation, and the names of the other machines are the only concerns for the application setting up a connection. The synchronization, or lack thereof, in communication between machines is a design decision.

Portability dictated our use of TCP/IP, an integral part of the Defense Data Network (DDN). Efficient use of processor power was considered more important than efficient use of the network resources. The network is shared by the entire Computer Science Department, but is not heavily loaded.

---

[1] The cost of creating a file and then spawning a process to send it is high. On the receiving end, there is the cost of creating the file and then reading it. Even a zero-cost file transfer protocol will require all this overhead.

[2] Large data transfers, in real-time systems, will not be possible until 100 MByte/Sec networks are commonly available.

B. BACKGROUND

1. Visual Simulation

a. Vision and Information Presentation

The eye has the largest bandwidth of any human sensory organ. Proper use of this capability is a challenge to all scientists. Static graphs are used in most disciplines to show the relationships between a limited number of variables. These two-dimensional representations convey information more readily to human beings than would a table of the underlying numbers. [Ref. 4: pp. 8-12]

Time, a common independent variable, is often one dimension on a graph. The other dimension is a single dependent variable. To portray additional variables in one presentation is a frequently occurring requirement. Various techniques such as multiple colored lines, multiple icons, and perspective drawing are used. With each technique, only a few additional variables are added before the graph becomes incomprehensible.

Pictures, particularly those in color, have a dense information content. Unless blind, we live in a world of pictures. Human beings can recognize many differences between two similar pictures. One presentation portrays many different variables. When a series of pictures are presented, the time variable is easily correlated to the actual time of presentation. When a series of pictures is presented rapidly, the experience approaches reality, partly explaining the success of moving pictures and television.

Animation creates visual images with an explicit time dimension, in addition to two or three spatial dimensions. Using actual time to portray the experimental time variable allows at least one more dependent variable on the display. Images can be as simple as a changing graph, or as complex as a feature-length cartoon.

3

However, animation creates its effect with the playback of prerecorded scenes [Ref. 5]. It is not suitable for providing immediate feedback to a researcher.

b.   Definition

Visual simulation is the creation, by computer, of a realistic, easily-modified, moving image from the mathematical model of a phenomenon. Realism implies high-resolution, color graphics. Movement implies adequate floating point calculation capacity to recalculate the model and its graphical representation between display refresh cycles. Easy modification implies a well-designed computer application.

Visual simulation allows a researcher to experiment easily with his subject. Ideally, we display a realistic approximation of part of the world. The experimenter then manipulates some part of this *visual simulation* and receives immediate visual feedback. The rapidly refreshed display is one key to visual realism. Such a display allows the direct manipulation of the *visual simulation*, making it easy and intuitive to use [Ref. 6]. Ease of use allows the researcher to concentrate on the research question, not the display methodology or the computer interface.

c.   Examples

Recent *visual simulation* projects of the Graphics and Video Laboratory include speed control of autonomous vehicles [Ref. 7], control of autonomous walking machines [Ref. 8], rule-based control of autonomous underwater vehicles [Ref. 9], interactive moving platforms [Ref. 10] and combat vehicle control [Ref. 11]. Each of these projects exceeded the capacity of a single workstation. The speed control and interactive moving platform projects, written entirely in C, used two Silicon Graphics, Inc. IRIS workstations, allowing multiple simultaneous views. The other projects all required a rule-based artificial intelligence component, best programmed in Lisp for ease of modification. Running the Lisp subsystem on the IRIS workstation gave an unacceptably low refresh rate and correspondingly poor realism [Ref. 12]. Placing the

4

Lisp subsystem on another machine improved the refresh rate of the IRIS workstation used for the graphics display.

2. Computer System Architecture

Computer systems can have a distributed or a non-distributed architecture. Distributed architectures have only one characteristic in common, more than one processor used to accomplish the task. Beyond this, many different approaches have been tried [Ref. 13]. Identical processors give a homogeneous architecture. Different processors give a heterogeneous architecture. Either distributed architecture may incorporate shared memory or it may not. The separate processors can be closely or loosely coupled. Communication between processors can be via shared memory, common bus, or some form of communications network. Communication via some combination of the above, such as a file server on a local area network, is also common [Ref. 3]. In the Computer Science Department at the Naval Postgraduate School, a heterogeneous mix of stand-alone workstations, file server supported workstation clusters, and minicomputers communicates via Ethernet.

Programming distributed architectures has inspired creativity. The fundamental problems with distributed programming are the communications between processes and the temporal interaction of the processes. Communicating sequential processes [Ref. 14], distributed processes [Ref. 15], and remote procedure calls [Refs. 2, 16] have all been proposed as primitives to hide message passing from the programmer. Remote procedure calls [Refs. 2, 3] and communicating sequential processes [Ref. 17] have been implemented. However, even today, none of these is generally available as a standard mechanism across varied architectures. We have created simpler (but less general) communication routines for use among heterogeneous, distributed, standalone computers.

5

Complex projects can require the resources of more than one computer. Graphics portions are best handled by the specialized hardware of a graphics workstation, such as a Silicon Graphics, Inc. IRIS. Artificial intelligence portions are best handled by a Lisp machine, such as a Symbolics[*] or a Texas Instruments Explorer[**]. Database requests can be made to a machine with appropriate database software. A general purpose computer, such as the Digital Equipment Corporation VAX[***], can be used for additional processing power, file storage, or other administrative support. Providing easy access across such a mix of heterogeneous computers is a large task [Ref. 3]. The simple mechanism described in this work gives communication access between cooperating processes running on diverse hardware. It leaves temporal design to the application developer, while providing the tools for synchronous and asynchronous interaction.

3. Communication

Communications between computers cooperating on a task can be one-to-one, many-to-one, or one-to-many. It can be synchronous or asynchronous. Any, or all, of these can be required for one *visual simulation*.

One-to-one, or direct connect, communications puts the lowest load on the network when there are few messages to be sent. A single virtual channel between the two processes is required. Each communication between any two processes comprises one message. All messages are known to be intended for the receiving process. These messages can be sent synchronously or asynchronously. Direct connect communication requires one action by the sender and one by the receiver. With more processors,

---

[*] Symbolics is a trademark of Symbolics, Incorporated.

[**] Explorer is a trademark of Texas Instruments Incorporated.

[***] VAX is a registered trademark of Digital Equipment Corporation

6

potential virtual channels grow in number geometrically. For a fully connected network, the virtual channels required can exceed capacity. The potential messages required also grow geometrically in number.

One-to-many, or broadcast, communications puts the lowest load on the sending process. A message is sent to all other processes that are connected to it. It requires one action by the sender, and two actions by each receiver (the reception and a decision on whether the message is intended for that receiver). It also places one to n messages on the network (depending on how the network and the broadcast protocols are designed). It is primarily used in an asynchronous mode, although synchronous protocols could be designed.

Many-to-one communications puts the highest load on the receiving process. It requires two actions by the receiver on every message that is sent by any connected process. It is also a primarily asynchronous method. The receiver portion of a process sees many-to-one whenever broadcast protocols are the only ones used in a *visual simulation*.

C. ORGANIZATION

The previous sections of this chapter provide background on visual simulation, distributed architectures, and communication paradigms. Chapter II describes the hardware and software environment in the Computer Science Department at the Naval Postgraduate School. The protocols developed are discussed in Chapter III. Chapter IV describes the implementation of the protocols. Chapter V covers the use of these protocols. The performance of the protocols is detailed in Chapter VI. Chapter VII concludes with a discussion of limitations, future extensions and research topics, and summarizes the research conducted. Listings of the program source code for each of the hardware systems are included as Appendices.

II. EXISTING SYSTEM

A. INTRODUCTION

The distributed architecture available in the Naval Postgraduate School Computer Science Department Graphics and Video Laboratory is Ethernet-connected workstations and minicomputers. The workstations include IRIS 2400, 3120, and 4D graphics, Symbolics 36xx* and TI Explorer Lisp, ISI AI, and Sun-3s**. The minicomputers include VAX 11/785 and an ISIV minicomputer complex providing database services. All computers, except the Symbolics and TI, use some version of UNIX*** as the primary operating system.

B. HARDWARE

1. Network

Ethernet connects all the computers in our lab. There is a backbone network and subnetworks for certain groups of computers. Currently there are two subnetworks, one for the ISIV minicomputers and one for the ISI AI workstations. Subnetworks are planned for the IRIS workstations, the Sun Workstations****, and the Symbolics and TI workstations. Figure 2.1 illustrates the network configuration.

---

* Symbolics 3600, Symbolics 3640, Symbolics 3650, and Symbolics 3675 are trademarks of Symbolics, Inc.

** Sun-3 is a trademark of Sun Microsystems, Inc.

*** UNIX is a trademark of AT&T Bell Laboratories

**** Sun Workstation is a registered trademark of Sun Microsystems, Inc.
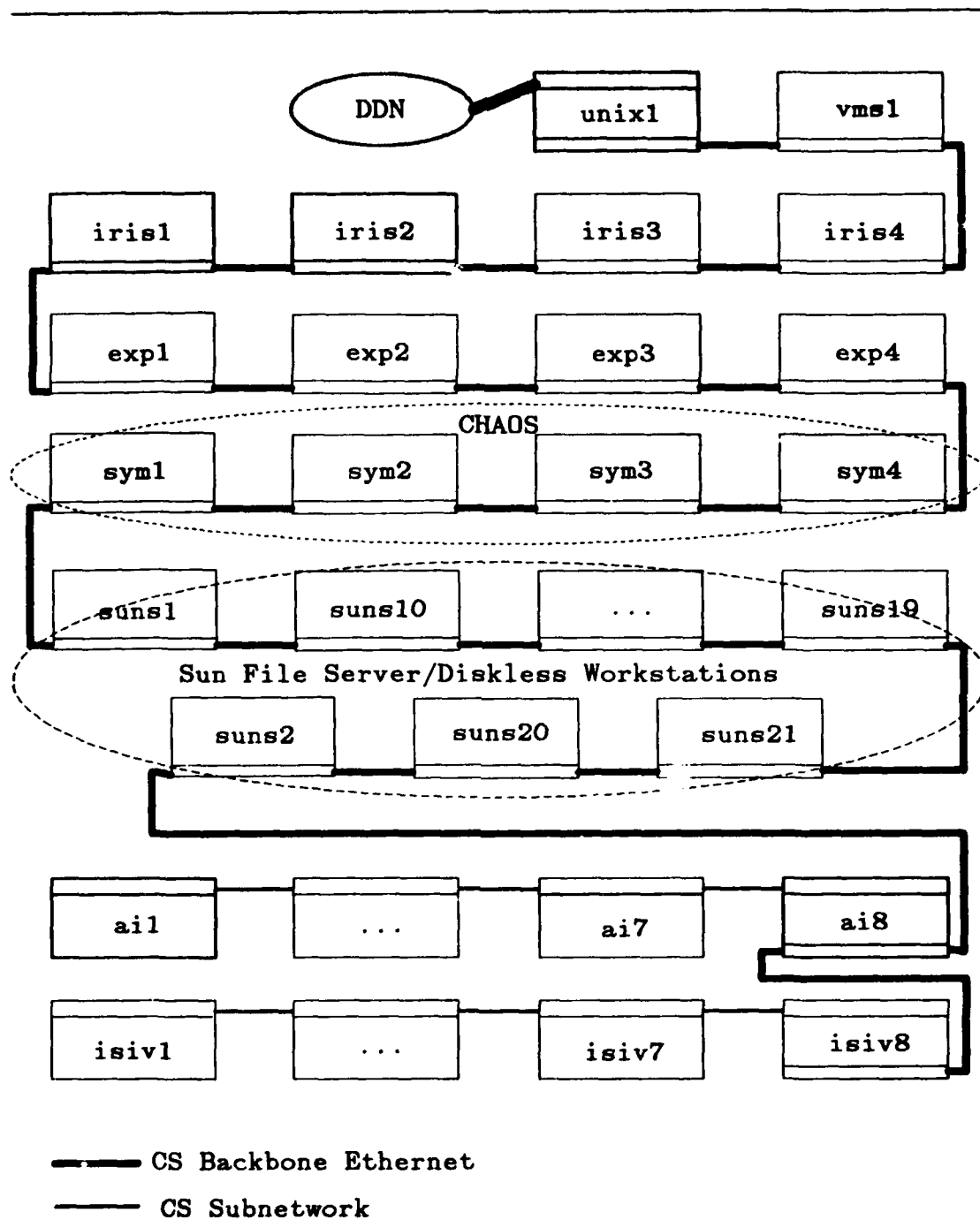
Figure 2.1   Network Configuration

9

All computers support TCP/IP protocols. The Symbolics Lisp machines also use the CHAOS protocol to provide file server services from *sym1* to the other Symbolics machines. This logical local area network (LAN) uses the Ethernet backbone for its messages. The Sun file servers also support their diskless nodes over the backbone Ethernet.

2. Workstations

    a. Silicon Graphics, Inc. IRIS

    Table 2.1 shows the IRIS workstation configurations. All are connected directly to the backbone Ethernet. The proprietary Geometry Engines in each of these workstations allows three dimensional color graphics displays to be generated and updated in real-time. The primary use of these machines is for color graphics.

    b. ISI AI

    Table 2.2 shows the ISI AI workstation configurations. Only *ai8* is connected directly to the backbone Ethernet. The other workstations are connected to it in a subnetwork. These workstations are used primarily for artificial intelligence projects. The *ai8* machine provides, as well as a gateway to the backbone Ethernet, file server support for the other workstations. Their high resolution black on white monitors, although bitmapped, have rudimentary graphics capabilities.

Table 2.1   IRIS WORKSTATION CONFIGURATIONS

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Bit Planes	Floating Point Accelerator	Screen Resolution
iris1	4D/70G	8	380MB	56	N/A	1280x1024
iris2	2400 Turbo	6	144MB	32	Y	1024x768
iris3	3120	4	144MB	32	N	1024x768
iris4	4D/70G	8	380MB	56	N/A	1280x1024

Table 2.2 ISI AI WORKSTATION CONFIGURATIONS

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Bit Planes	Screen Resolution
ai1	V8WS	4	101MB	2	1280x1024
ai2	V8WS	4	101MB	2	1280x1024
ai3	V8WS	4	101MB	2	1280x1024
ai4	V8WS	4	101MB	2	1280x1024
ai5	V8WS	4	101MB	2	1280x1024
ai6	V8WS	4	101MB	2	1280x1024
ai7	V8WS	4	101MB	2	1280x1024
ai8	V16WS	4	403MB	2	1280x1024

c. Sun-3/50

Table 2.3 shows the Sun Workstation configurations. All are connected directly to the backbone Ethernet. The black-on-white monitors of the Sun diskless workstations are primarily used for administrative tasks at this time.

d. Symbolics 36xx

Table 2.4 shows the Symbolics workstation configurations. All are connected directly to the backbone Ethernet. The Symbolics workstations are used for a

Table 2.3 SUN WORKSTATION CONFIGURATIONS

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Bit Planes	Screen Resolution
suns1	3/180S	12	490MB	2	1280x1024
sun10	3/50	4	N/A	2	1280x1024
sun11	3/50	4	N/A	2	1280x1024
sun12	3/110	4	N/A	2	1280x1024
sun13	3/110	4	N/A	2	1280x1024
sun14	3/60	4	N/A	2	1280x1024
sun15	3/60	4	N/A	2	1280x1024
sun16	3/60LC	4	N/A	10	1280x1024
sun17	3/50	4	N/A	2	1280x1024
sun18	3/50	4	N/A	2	1280x1024
sun19	3/50	4	N/A	2	1280x1024
suns2	3/180S	12	490MB	2	1280x1024
sun20	3/60LC	4	N/A	10	1280x1024
sun21	3/60LC	4	N/A	10	1280x1024

11

Table 2.4   SYMBOLICS WORKSTATION CONFIGURATIONS

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Bit Planes	Color	Screen Resolution
sym1	3675	5	1GB	24	Y	1280x1024
sym2	3640	1	180MB	1	N	1280x1024
sym3	3640	1	180MB	8	Y	1024x1024
sym4	3650	5	512MB	1	N	1280x1024

variety of research projects involving artificial intelligence. The *sym1* machine provides file server support for the other Symbolics machines using the Chaos protocol and its one GigaByte (unformatted) storage capacity. The color-capable systems are used to display static information with color providing an easier human interface.

e.   Texas Instruments Explorer

Table 2.5 shows the Explorer workstation configurations. All are connected directly to the backbone Ethernet. The TI Explorers are also used for artificial intelligence projects. They have the least graphical capabilities of any of the workstations.

3.   Digital Equipment Corporation VAX 11/785

Table 2.6 shows the two DEC* VAX 11/785 computer configurations. Both are connected directly to the backbone Ethernet. Only the *unix1* machine was included in this project. The *vms1* machine may not be available in the future, so the effort to

Table 2.5   EXPLORER WORKSTATION CONFIGURATIONS

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Bit Planes	Screen Resolution
exp1	I	4	280MB	1	1024x808
exp2	I	8	420MB	1	1024x808
exp3	I	8	420MB	1	1024x808
exp4	I	2	140MB	1	1024x808

---

* DEC is a registered trademark of Digital Equipment Corporation

Table 2.6   VAX CONFIGURATIONS

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Operating System
unix1	11/785	24	1395MB	UNIX
vms1	11/785	8	1442MB	VMS

develop appropriate code was deemed unnecessary. The *unix1* machine is *nps-cs.arpa* on MILNET and is the sole external access point to other machines connected locally via Ethernet. It supports the various dial-up lines, as well as other administrative functions.

4.   ISIV minicomputers

The computers in Table 2.7 make up the ISIV minicomputer complex. Only *isiv8* is connected to the backbone Ethernet. The other machines are connected to *isiv8* in an Ethernet subnetwork. The ISIV minicomputers provide a high performance, multi-backend distributed database. Any of the high-resolution black on white monitors can be used with any of the hosts on the subnetwork. The character displays can also be used on any of the subnetwork hosts. The graphics capabilities of these machines are limited.

Table 2.7   ISIV DATABASE MACHINE CONFIGURATION

Nickname	Model No.	Memory (MBytes)	Disk Capacity	Bit Planes	Screen Resolution
isiv1	V24S	4	602MB	N/A	80x24char
isiv2	V24WS	4	500MB	2	1280x1024
isiv3	V24WS	4	602MB	2	1280x1024
isiv4	V24WS	4	500MB	2	1280x1024
isiv5	V24S	4	602MB	N/A	80x24char
isiv6	V24S	4	602MB	N/A	80x24char
isiv7	V24WS	4	602MB	2	1280x1024
isiv8	V24WS	4	459MB	2	1280x1024
isiv9	V24S	4	602MB	N/A	80x24char

C. SOFTWARE

1. UNIX Machines

Two versions of UNIX are commonly used. The machines purporting to use System V*, also incorporate characteristics of 4.2BSD and 4.3BSD. The relevant incorporation is the Berkeley socket mechanism.

a. 4.3BSD

A "pure" 4.3BSD system (4.3 BSD UNIX #11) exists only on *unix1*. The ISIV minicomputers use 4.2 BSD UNIX Release 3.07, with a multi-backend database system installed [Refs. 18-20]. The ISI AI workstations use IS68K 4.3 BSD UNIX: 4.0D #2.

b. System V

The IRIS 4D systems use UNIX System V-based version 4D1-2.2. The IRIS 2400 and 3120 systems use UNIX System V-based version GL2-W3.6. Both have extensive 4.3BSD extensions. The Sun-3 uses an almost System V version of 4.2BSD UNIX. The currently installed release is 3.4.

2. Lisp Machines

a. Genera

The Symbolics Lisp Machines first used Genera 6.0 software. All machines are now on Genera 7.1.

b. Explorer

The TI Explorer lisp machine first used Explorer version 1.0.2 software. All machines are now on version 3.4 except *expl*, which is still on version 3.2.

_____

* UNIX System V is a trademark of AT&T Bell Laboratories

14

D. SUMMARY

The configuration described above is constantly changing. Additional machines are acquired. Older machines receive hardware upgrades. The network is reconfigured. Software releases are updated (especially 4.2BSD UNIX to 4.3BSD UNIX). The fundamental needs for distributed computation in this heterogeneous environment remain.

III. PROTOCOLS

A. INTRODUCTION

Our *visual simulation* efforts rely on small data transfers to communicate among machines. These small messages are typically commands and changing status indicators. Hence, we optimized our protocols for small messages. Overhead to optimally encode and decode packets was deemed inappropriate. The design criteria for developed protocols were simplicity, ease of use, portability, and efficiency.

B. DIRECT CONNECTION

The client/server paradigm is used for direct connection. The client requests services from the server, so establishing communications is asymmetrical. Once cor..nunications are established, however, the protocol used is completely symmetrical. [Ref. 21: p. 17]

1. High-Level Protocol

The variety of data types supported is limited (see Table 3.1). Each message contains exactly one instance of one type of data. All integer or float data is converted to an ASCII character string before it is sent. It is converted back to the proper type after

Table 3.1   DATA TYPES SUPPORTED

Type	Length (Bytes)	Elements	Code	Available
character	1	single	B	Y
		array	C	Y
integer	4	single	I	Y
		array	J	N
float	4	single	R	Y
		array	S	N

16

reception. While the conversion is unnecessary when communicating between similar architectures, it greatly simplifies the task of communicating between fundamentally different architectures. Knowledge of the other machine's architecture *is not required*. The inherent portability of this solution outweighs the processing cost.

A message is created with three fields. The *type field* is a one-character field. It contains the appropriate code from Table 3.1. The *length field* is a four-character field. It contains an ASCII string from 0001 to 9999. This string gives the length of the *data field*. The *data field* is a variable length field containing the ASCII representation of the data element. Figure 3.1 illustrates these fields.

While C programmers are continuously concerned with data types, Lisp programmers are not. The Lisp routines support arrays of characters, single integers, and single floating point numbers. Each of these is an object. Objects, not types (as implied in Table 3.1), are received and sent by lisp applications. The underlying protocol is the same, the application interface is different[3].

Position

1	2	3	4	5	6	7	...	n
T y p e		Length				Data		

Figure 3.1   Message Format

---

[3] Chapter 5 discusses applications' use.

2. Supporting Protocols

Full-duplex *stream sockets* are used to provide sequenced, reliable connection between machines. The sockets are created in the DARPA Internet[4] domain. The Internet pseudo-protocol is used [Ref. 22]. No out-of-band capability was included. We could not envision a use for it, since our protocol is inherently asynchronous. If a strictly synchronous protocol was used, out-of-band transmission might be necessary to interrupt for an urgent message. In an asynchronous protocol, however, encoding the next message gives the same effect. Processing overhead for encoding is no greater than for continuous monitoring for an out-of-band message. With only a small volume of data transfers expected, no urgent message waits very long.

Two ports, each with its own *stream socket*, are used for each channel between machines. Although full-duplex, the *stream sockets* are used in a simplex mode. The separate sockets are used because two processes cannot be bound to the same socket at the same time. Two separate UNIX processes then monitor the independent send and receive sockets. Blocking sockets are used, avoiding processing overhead for busy-waiting. While non-blocking sockets are available in 4.3BSD [Ref. 21: p. 25], they were not explicitly available in 4.2BSD [Ref. 22]. Operating systems might include 4.2BSD sockets rather than 4.3BSD versions and so the blocking socket mechanism was deemed more portable. Both TCP/IP and the C routines provide buffering.

On the TI Explorer, sockets were also blocking[5]. Direct access was made to the TCP methods provided. Lisp *streams* are used for the Symbolics lisp routines. The

---

[4] This is the underlying mechanism of the Defense Data Network (DDN) and was chosen for its wide availability and applicability to Department of Defense problems.

[5] Version 1.0 of the Explorer TCP/IP software uses blocking sockets. Version 2.0 uses non-blocking sockets. There has been no update of this system's TI Explorer lisp routines to version 2.0.

18

lisp stream mechanism isolates the code from the issues revolving around blocking versus non-blocking sockets.

C. BROADCAST

A broadcast message is sent to all machines on a local Ethernet. Those machines that are waiting for some broadcast message will probably[6] receive it. If a machine on a subnetwork is to get a broadcast message, an application must run on the gateway machine that will rebroadcast on the subnetwork any messages received on the backbone Ethernet. Machines not expecting a broadcast message must nevertheless process it and reject it as inappropriate. The extra load on all machines connected to the Ethernet restricts broadcasting to infrequent occurences until most of the machines used in simulations[7] are on a private subnetwork.

1. High-Level Protocol

We expect users of the broadcast protocol to mix its use with the use of direct connections. The same data types and messages are supported (see Table 3.1).

2. Supporting Protocols

Full-duplex *datagram sockets* are used to provide connectionless broadcast capability. The sockets are created in the DARPA Internet domain. As with our use of *stream sockets* for the direct connection protocol, we use these full-duplex *datagram sockets* in a simplex mode. We use a sending socket for one-way sending of a broadcast message to all other machines on a single network or subnetwork. We use a receiving socket for one-way receiving from a specific broadcasting machine on the network or

---

[6] Unlike the direct connect protocol, the broadcast protocol does NOT guarantee reception. Trying to provide such a guarantee requires a feedback mechanism so that the sender knows that the machines expected to receive the broadcast did so. This is difficult without resorting to a direct connection or flooding the network with messages.

[7] The IRIS machines and the Lisp machines are the ones principally used for *visual simulation*.

subnetwork. Direct connection, with its use of guaranteed reliable *stream sockets*, is used for any other communication, including return messages. [Ref. 21: pp. 32-34]

As in the direct connection protocol, independent UNIX processes are bound to the sockets. Since broadcasting is a one-way activity, a sender or receiver only spawns one[8] UNIX process.

D. SUMMARY

By building our high-level protocols on top of DARPA TCP/IP standards, we provide the highest degree of portability possible today. By using full-duplex *stream sockets* and *datagram sockets* in a simplex mode, we do not make full utilization of a socket's capabilities. However, this concern is outweighed by the increased simplicity and resultant maintainability of the code. The use of ASCII character strings for the messages is simple and makes interconnection with diverse architectures straightforward.

---

[8] If broadcasting were used exclusively for complete connectivity, each of *n* machines would spawn *n* processes. If direct connection was used exclusively for complete connectivity, each of *n* machines would spawn 2*n*-2 processes.

IV. IMPLEMENTATIONS

A. INTRODUCTION

The first connection was between the IRIS 2400-Turbo and TI Explorer. Then the Symbolics Lisp machines were included. These routines have had extensive use [Refs. 8, 9, 11]. The IRIS functions were updated for the IRIS 4D, coincidentally providing Mex support on the older IRIS machines. Broadcast capability was added for UNIX-based machines. A port to 4.3BSD UNIX (application calls unchanged) was begun.

B. SYSTEM V UNIX

All our System V UNIX-based systems include the *socket* mechanism first introduced by 4.2BSD. Sockets are a key aspect of all implementations. We expect they will become part of System V or its successors [Ref. 23]. The System V-unique *semaphore* and *shared memory* interprocess communication (IPC) capabilities are also used.

1. Silicon Graphics, Inc. IRIS 2400

a. Sockets

The *socket* was introduced in 4.2BSD as the preferred metaphor for IPC. It was easy and efficient to implement and the *select* mechanism could be used to implement remote procedure calls, if desired [Ref. 23]. System V had no comparable mechanism until version 3 was released with *streams*. The BSD sockets were included by many vendors, Silicon Graphics, Inc. included[9]. While the use of sockets could be

---

[9] The System V version available on the IRIS machines, at the start of the project, was version 2 and so streams were not considered.

replaced with streams, device drivers would have to be written. The advantage of streams is the ability to filter them between streamhead and the actual device driver. These filters, however, reside in the kernel's address space and have the kernel's permissions [Ref. 24]. In our environment, the potential performance increase is not as important as the requirement for simplicity.

The system call for socket creation is *socket*. The system calls supporting socket configuration are *setsockopt*, *bind*, *connect*, and *accept*[10] [Ref. 22]. To simplify their use, these are all repackaged into four high level routines: *connect_server* and *connect_client* for direct connection, *start_broadcast* and *broadcast_receive* for broadcast. These routines are encapsulated in netV.c. netV.c can be separately linked with any application that needs to make a server/client connection using *stream sockets* or a broadcasting connection using *datagram sockets*. Table 4.1 describes the four routines.

Using the socket number[11], a process can transmit data through the socket. In our system, sockets for inter-computer communication are created and used by the *send* and *receive* processes exclusively. The file netV.c is not linked with the application at all.

---

[10] The *accept* system call is only relevant to *stream sockets*. The *setsockopt*, *bind*, and *connect* system calls are used with both *stream sockets* and *datagram sockets*.

[11] In the direct connect protocol, the server process reads from and writes to a remote socket number. The client process reads from and writes to its local socket number. The reason for this is that a server could be connected to different clients (although not in our implementation) at different times. The client, meanwhile, is only going to connect to the one server. In the Internet domain, all necessary routing information, for either server or client, is contained in a *sockaddr_in* structure and is accessed (transparently) via the socket number.

In the broadcast protocol, both the broadcaster and receiver(s) use their local socket number because they are using connectionless *datagram sockets*. The routing information is also contained in a *sockaddr_in* structure.

Table 4.1   SOCKET SUPPORT FUNCTIONS

Function	Description	Use
connect_server	Creates socket. Binds that socket to remote client address and port. Waits to accept the remote client connection. Returns the socket number for the remote client.	int connect_server(remote_client_name, port_number) char remote_client_name[]; int port_number; remote_socket = connect_server(remote_client_name, port_number)
connect_client	Creates socket. Binds that socket to remote server address and port. Connects with remote server. Returns the local socket number.	int connect_client(remote_server_name, port_number) char remote_servei_name[]; int port_number; local_socket = connect_client(remote_server_name, port_number)
start_broadcast	Creates socket. Sets it to broadcast mode. Binds it to local address and specified local port. Returns the local socket number.	int start_broadcast(port_number) int port_number; local_socket = start_broadcast(port_number)
broadcast_receive	Creates socket. Binds it to local address and specified port. Adds broadcaster address and port. Returns the local socket number.	int broadcast_receive(broadcaster_name, broadcaster_port) char broadcaster_name[]; int broadcaster_port; local_socket = broadcast_receive(broadcaster_name, broadcaster_port)

b.   Semaphores

The semaphore mechanism was chosen as the least expensive, in both space and time, for communication between processes. Signals could have been used, but implementation would have been more complex and less reliable. Signal-based communication functions would also have been more difficult for the application programmer to use [Ref. 25: p. 10]. There are two semaphore ids maintained for each connection[12]. One is used to communicate with the *send* process; one is used to communicate with the *receive* process. The two semaphores are both used to signal their process when it is safe to proceed. A *send* process is permitted to proceed only after the

---

[12] Two semaphore ids are required for direct connect protocol connections since both a *send* and a *receive* process are spawned. Two semaphore ids are still created for broadcast protocol connections, even though only one process is spawned.

23

application has requested a write action[13] on the channel. A *receive* process is permitted to proceed only after the application has read all data from the shared memory buffer. Neither the *send* nor the *receive* process is executing more than absolutely necessary, assuring maximum availability of the local processor to the application.

The system calls supporting semaphores are *semget*, *semop*, and *semctl*. To simplify their use, they are repackaged into three high level routines: *semtran*, *P*, and *V* [Ref. 25: pp. 188-190]. These routines (and a support routine *semcall*) are encapsulated in **semaphore.c**. It can be separately linked with any application that needs semaphores. Table 4.2 describes the three routines.

c.   Shared Memory

A cost barrier to IPC in UNIX is the cost of copying data from one process to the kernel and then from the kernel to another process. Using a *shared memory segment*, as a buffer, minimizes this overhead. To further reduce overhead from system calls, only a single segment is created. An application accesses the entire segment, while a *send* or *receive* process accesses only its preassigned section. Figure 4.1 displays the layout. The *message* area of each section is used for several purposes. It is formatted as

Table 4.2   SEMAPHORE SUPPORT FUNCTIONS

Function	Description	Use
semtran	Creates a semaphore associated with a key. Returns a semaphore id.	int semtran(key) int key; sid = semtran(key);
P	Acquire semaphore	void P(sid) int sid;
V	Release semaphore	void V(sid) int sid;

---

[13] The data must also be valid in the shared memory buffer. All this is transparent to the application, which only issues a write command.

24

Receive Section						
Message				Data		
0	1	2	3	1	...	n

Send Section						
Message				Data		
0	1	2	3	1	...	n

Shared Memory Segment								
Receive			Send			Protocol		
0	...	n + 3	n + 4	...	2n + 7	2n + 8	...	2n + 19
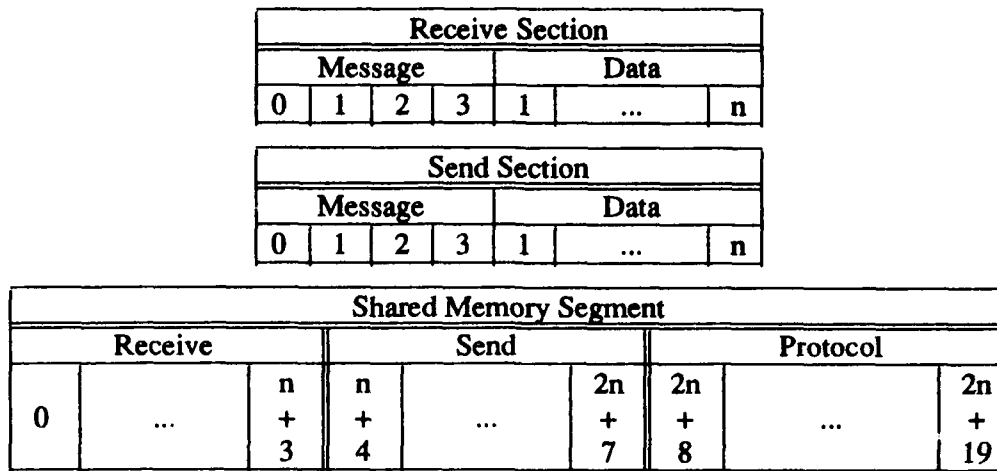
where n = LARGESTREAD from *shared.h*

Figure 4.1   Shared Memory Segment Data Assignment

a long (4-byte) integer. Table 4.3 describes the meaning of three-state values placed in this area.

Table 4.3   SHARED MEMORY MESSAGES

Value	Meaning to *send*	Meaning to *receive*	Meaning to Application
positive	Data of length given is in shared memory, ready to be sent.	Application has not finished reading data from shared memory.	*send*: Data in shared memory has not yet been sent to other machine.
			receive: Valid data of length given is in shared memory, ready to be read.
zero	Nothing ready to be sent.	Application has read data from shared memory. Message from other machine can be read, up to LARGESTREAD bytes.	*send*: Previous message has been sent. Ready to send next message.
			receive: No valid data in shared memory.
negative	Signal to terminate.	Signal to terminate.	N/A

The system calls supporting shared memory are *shmget, shmat, shmdt*, and *shmctl* [Ref. 25: pp. 192-198]. To simplify their use, they are repackaged into four high level routines: *sharedsegment, dynamicsharedsegment, detachsharedsegment*, and *deletesharedsegment*. These routines (and a support routine *attach_within_datasegment*) are encapsulated in **shareseg.c**. It can be separately linked with any application that needs shared memory. Table 4.4 describes the four routines.

The implementation of shared memory on the IRIS 2400 and IRIS 3120 was a surprise. A basic UNIX memory allocation scheme is shown in Figure 4.2. Each process has its own text, data, and stack sections. Neither the relative locations of these sections nor the direction of growth for stack and data sections is specified for UNIX. The shared memory segments are logically part of the data section [Ref. 26: p. 151].

Table 4.4 SHARED MEMORY SUPPORT FUNCTIONS

Function	Description	Use
sharedsegment	Creates (if not already in existence) a shared memory segment associated with a key. Attaches application to that shared memory segment. Returns a shared memory segment address and id. Does not permit subsequent dynamic memory allocation.	char *sharedsegment(key, nbytes, shmid) long key; long nbytes; int *shmid; segment = sharedsegment(key, nbytes, shmid)
dynamicsharedsegment	Creates (if not already in existence) a shared memory segment associated with a key. Attaches application to that shared memory segment. Returns a shared memory segment address and id. Permits subsequent dynamic memory allocation.	char *dynamicsharedsegment(nummachines, key, nbytes, shmid, freespace) int nummachines; long key; long nbytes; int *shmid; int freespace; segment = dynamicsharedsegment(nummachines, key, nbytes, shmid, freespace)
detachsharedsegment	Detach shared memory segment from application	void detachsharedsegment(segment) char *segment;
deletesharedsegment	Delete shared memory segment	void deletesharedsegment(segment, shmid) char *segment; int shmid;

26

```
              ┌──────────────────┐
              │                  │
              │      Stack       │
              │                  │
              ├──────────────────┤
              │    |        |    │
              │    growth        │
              │    ↓        ↓    │
              │                  │
              │                  │
              │                  │
              │                  │
              │    ↑        ↑    │
              │    growth        │
              │    |        |    │
              ├──────────────────┤
              │                  │
              │      Data        │
              │                  │
              ├──────────────────┤
              │                  │
              │      Code        │
              │                  │
              └──────────────────┘
```
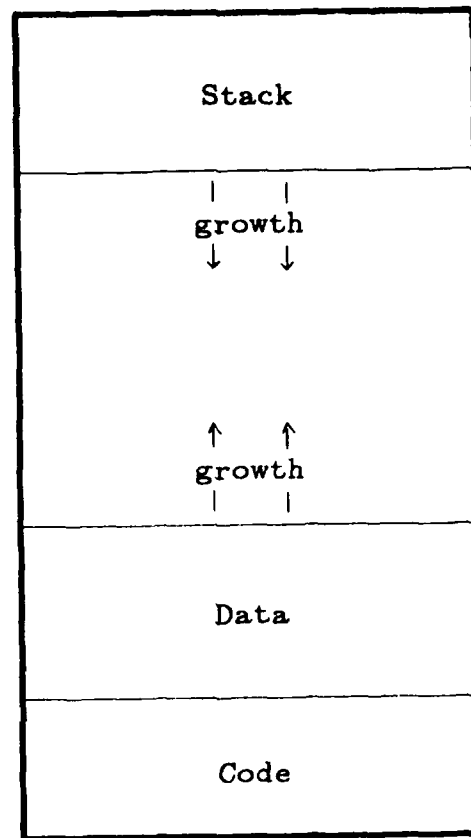
Figure 4.2   UNIX Memory Allocation

Actual implementation is left to the team porting UNIX to the machine. The Silicon

Graphics, Inc. implementation attaches a shared memory segment to the first available

valid[14] address within the data section. However, the beginning of shared memory

delimits the size of all other sections [Ref. 26: pp. 367-370]. Figure 4.3 illustrates this

---

[14] Shared memory segments must begin on a page boundary. This allows easy table-driven access by multiple processes. On the IRIS 2400 and 3120 machines, the Motorola 68000 architecture is used. The pages are 8KBytes.

27
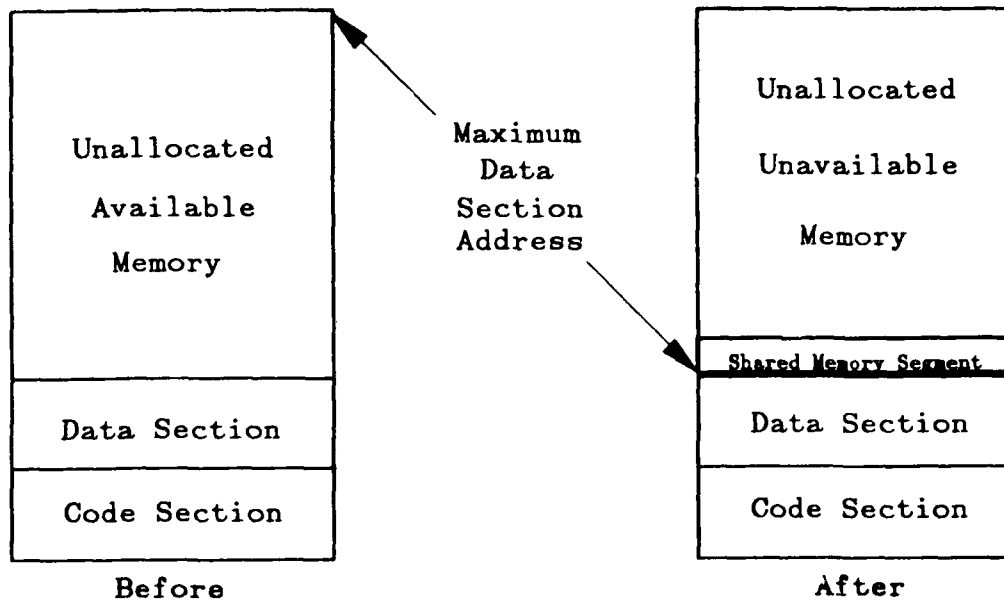
Figure 4.3   IRIS 2400 Default Shared Memory Attachment

relationship. While no dynamic memory calls[15] are made, the default arrangement works fine. But when dynamic memory allocation—linked lists and *makeobj()* calls are examples—is needed, the technique fails.

To allow dynamic memory allocation, the shared memory segment must be attached at an address beyond the greatest ever required for regular data. Dynamic allocation can then occur without reaching the shared memory segment. Attaching at an unknown address both within the data section and sufficiently beyond existing data to permit dynamic data section growth, can be done at least two ways. First, the data section can be expanded until it is as large as possible, then the shared memory segment

---

[15] Dynamic memory allocation is made with system call *brk* or alternate *sbrk*. Library functions *malloc*, *realloc*, and *calloc* use *brk* and so also do dynamic memory allocation.

28

can be attached at a valid location just inside this maximum value. While minimizing application programmer effort, this technique requires many system calls to grow the data section. It also has the fatal flaw of limiting the stack section, if the stack section and data section grow into the same unallocated memory. Second, the application can be required to prespecify the maximum amount of dynamic memory allocation it might use.

The solution adopted is adding a **freespace** parameter to the *sharedsegment* function; and renaming it the *dynamicsharedsegment* function. The *sharedsegment* function was retained for backward compatibility. The **freespace** parameter gives the caller the ability to specify the maximum additional memory required for the application. A request for this additional space is made before the shared memory segment is attached. After acquiring (and freeing) the additional space, the next available address is determined and the shared memory segment is attached to the next valid address. We have now established the shared memory segment beyond the specified growth of the application's data.

When multiple machines are connected together, there must be a separate shared memory buffer for each channel. There is no way to connect a second shared memory segment. The solution adopted is adding a **nummachines** parameter to the *dynamicsharedsegment* function. The **nummachines** parameter requires the application developer to specify, in advance, the maximum number of channels that can be created in the application. The first *dynamicsharedsegment* call establishes a shared memory segment big enough for *nummachines* maximum requested channels. Subsequent *dynamicsharedsegment* calls return the same shared memory id as the first; but return a different address within the segment. Since the application does not directly access these functions, there were no problems caused by this parameter list change.

29

The shared memory functions are isolated from the application by the *machinepath*, *dynamicmachinepath*, *dynamicmachinepaths*, and *deletemachinepath* functions[16]. For the direct connect protocol, each *machinepath*, *dynamicmachinepath*, or *dynamicmachinepaths* call spawns both a *send* and a *receive* process. For the broadcast protocol, these calls spawn only a *send* process (for the broadcaster) or a *receive* process (for the receiver). In all cases, the spawned processes issue a *sharedsegment* call to attach to the shared segment earlier created by the spawning function. A command line parameter is passed providing the offset into the shared memory segment that the spawned process is to use. Figure 4.4 illustrates a system with three machines and two channels.

d.   Buffering

(1)   Direct Connect.   When a *receive* process is quiescent, waiting for the application to read from the shared memory buffer, anything sent to it is buffered by TCP/IP. The buffering provides the reliable delivery promised by a *stream socket*. The next read command will deliver up to LARGESTREAD bytes into the receive data area of the shared memory buffer. Since the messages are variable length, there cannot be a guarantee that only one message was read[17]. Multiple messages might be in the shared memory buffer. A partial message might be in the last bytes.

The shared memory buffer management is handled by the various read functions[18] provided. Each read, requested by the application, is satisfied from the

---

[16] See Chapter 5, Sections A.1.b(1) and A.1.b(3) for more information on these functions.

[17] The idea to pad all messages to some arbitrary size was considered and rejected. Whatever size was chosen would always be too small for some character array. If the maximum Ethernet packet size was chosen, an unnecessary network dependence would be introduced. The cost of application buffer management is considered acceptable, especially since it is incurred only on reads.

[18] See Chapter 5, Section A.1.b(2) for more information on these functions
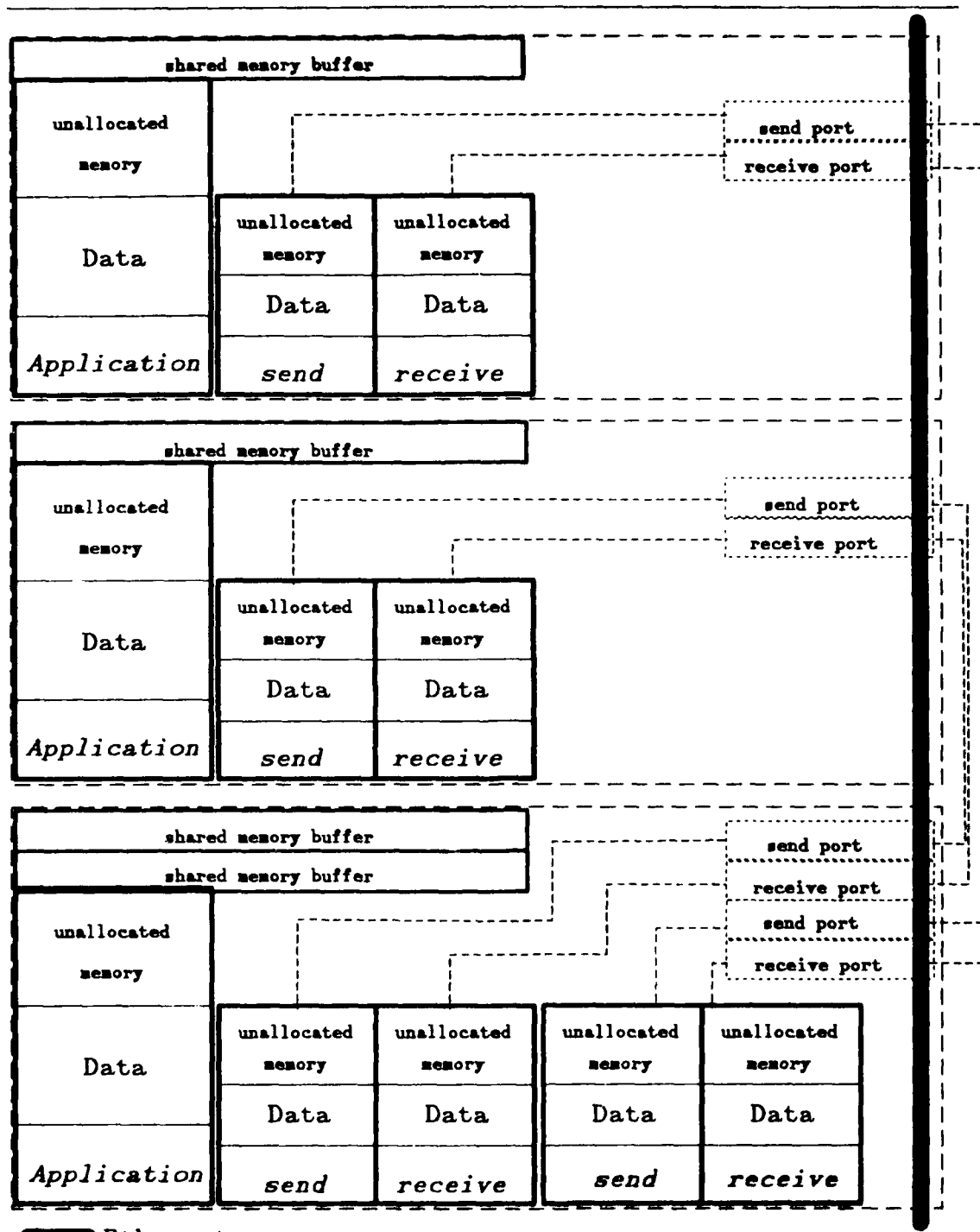
Figure 4.4 Three-Machine Interconnection

31

shared memory buffer. Remaining valid data is shifted into the low order positions of the data area. The count of valid bytes, held in the message area, is decremented. The shared memory buffer now appears as it would have, if it had only received the remaining data and not the first message at all. As long as only entire messages are received (one or more at a time), this works well. When the TCP/IP buffer has more data than the data area can take at one time, however, the *receive* process deposits LARGESTREAD bytes in the shared memory data area. It is highly unlikely that this will be on a message boundary.

A socket read overwrites all data in the data area. A partial data reception must be stored and concatenated with bytes from the next socket read to get a complete message. The *protocol area* was introduced to retain the protocol information[19] required to decipher the variable length messages. The count of already received bytes of a message is held here between socket reads. A message's protocol information is stored here, too. *Protocol information is built up until complete (covering the possibility that the break is in the protocol information itself). It is then maintained until the entire message is received and read by the application. The buffering works with data areas as small as four bytes[20].*

(2) Broadcast. The *datagram socket* used by the broadcast protocol preserves message boundaries. Each *recvfrom* call to a socket returns only one message. This message must be no longer than LARGESTREAD bytes. The shared memory buffer management routines are not needed.

---

[19] See Chapter 3, Section B.1 for a description of the protocol

[20] LARGESTREAD must be specified in multiples of four bytes. The smallest possible data area is therefore four bytes.

TCP/IP keeps unread messages on a queue. This queue may not be in sending sequence. If the queue buffer becomes full, subsequent messages are lost [Ref. 21: p. 8-8]. The sending buffer can easily be filled if many messages are broadcast in a short period of time. Each broadcast message must be processed by every host on the Ethernet. Only then can the next be sent. No access for manipulation of the TCP/IP sending buffer is provided because its size is normally specified during system generation and is not easily manipulated by an application program.

2. Silicon Graphics, Inc. IRIS 3120

There are no required changes to the IRIS 2400-Turbo code. The *Makefile* must be changed to remove the -Zf compile flag, since there is no floating point accelerator board in this machine.

3. Silicon Graphics, Inc. IRIS 4D

The IRIS 4D required programming changes only to the shared memory module, *shareseg.c*. The path name for user directories is also different. Changes were necessary to the *Makefile* because the */usr/include* directory structure changed.

The IRIS 4D is based on the MIPS RISC architecture. The UNIX implementation was done differently than that for the Motorola 68020. Shared memory segments are not attached to addresses within the data section, as illustrated in Figure 4.5. They are attached at a much higher address, yet accessing them does not result in a *segmentation violation*. This is a more robust technique that obviates any manipulation of attachment addresses. Multiple shared memory segments are easily attached, using default system calls. The *sharedsegment* call suffices, even when dynamic memory allocation is needed. To maintain backward compatibility for application code, *dynamicsharedsegment* calls *sharedsegment*, ignoring the **freespace** parameter, when compiled on an IRIS 4D, and calls *attach_within_datasegment* when compiled on an older IRIS machine.

33
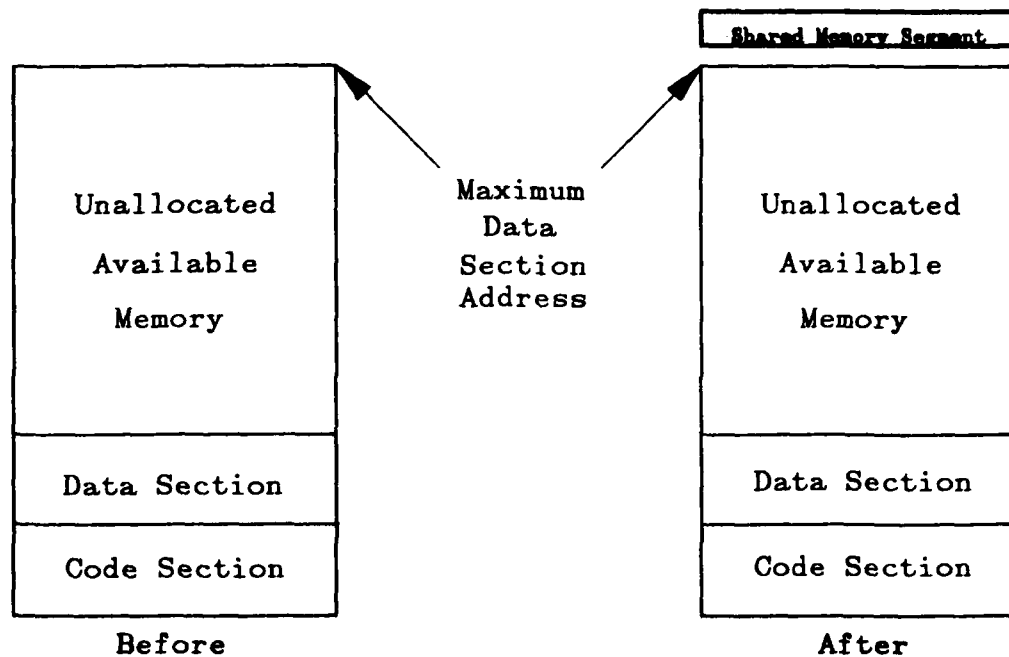
Figure 4.5 IRIS 4D Default Shared Memory Attachment

C. 4.3BSD UNIX

The **netV.c** file functions properly on a 4.3BSD machine that is connected to only one network. The *start_broadcast* function does not properly handle multiple networks. The other functions work correctly, even when the machine is connected to multiple networks.

All other functions depend upon semaphores and shared memory for communication between the spawned processes and the main application. Stream sockets[21] could be used to provide the IPC between these processes under 4.3BSD. The

---

[21] Unidirectional stream sockets are equivalent to pipes.

34

three channels[22] used will have to be multiplexed into one, but the implementation is otherwise straightforward.

D. LISP MACHINES

The communication code is a flavor to be mixed with the application [Ref. 11]. The Explorer software is syntactically equivalent to Genera 6 on the Symbolics. With a simple change in the sequence of method and flavor names, the Genera 7 code runs on the TI Explorer. The older flavor, originally developed for the Explorer, is also presented to illustrate working directly with TCP/IP instead of using a stream.

1. Texas Instruments Explorer I

This older flavor works with Release 1.0 of the Explorer TCP/IP software. It will not work with Release 2.0 as the implementation was changed from blocking to non-blocking [Ref. 27].

Messages to the flavors in the **ip** package are made together with messages to the **tcp** flavors. Network-independent addressing is not used. Table 4.5 describes the addressing schemes possible [Ref. 28: pp. 4-2—4-3]. Class C addressing is used by the Computer Science Department. Figure 4.6 shows the simple encapsulation of the addresses for *iris1*, *iris2*, and *iris3*. Extension to include other machines is easy.

Table 4.5 INTERNET ADDRESSING CLASSES

Class	No. Networks	No. Hosts
A	128	16,777,216
B	16,384	65,536
C	2,097,152	256

---

[22] These are the semaphore, the message areas of the shared memory buffer, and the data areas of the shared memory buffer. The first is unidirectional from application to spawned process. The second is bidirectional and three state (see Table 4.3).

```
(defvar *iris1-address* 3221866502)
(defvar *iris2-address* 3221866504)
(defvar *iris3-address* 3221866505)

(defvar *dest-address*  nil)                    ; the tcp-ip or internet address
                                                ; look in network configuration

(defun iris (x)
  (cond ((equal x 1) (setq *dest-address* *iris1-address*))
        ((equal x 3) (setq *dest-address* *iris3-address*))
        (t                 (setq *dest-address* *iris2-address*)) ) )
```

Figure 4.6   Encapsulation of *IRIS* Addresses

A port is acquired by using the **:get-port** method of the **tcp-handler** flavor. Here, shown in Figure 4.7, we use the global instance, ***tcp-handler***[23] to create specific instances of the Transmission Control Block (TCB) for each of the two ports. Only the client side of the server/client paradigm has been implemented. The client is created by using the **:active** *mode* argument to the **:open** method of the **tcp-port** flavor. Both the sending and receiving ports are full duplex, but are only used in a simplex mode. Figure 4.8 shows the creation of the sending port [Ref. 28: pp. 4-12—4-18].

The three fields in a message are sent and received separately. Each field is then treated as a separate object. Figure 4.9 illustrates sending a message. For all fields, the *urgent* argument is specified as **nil**. The *push* argument is specified as **nil** until the

```
(defvar *tcp-handler1* (send ip::*tcp-handler* :get-port))
(defvar *tcp-handler2* (send ip::*tcp-handler* :get-port))
```

Figure 4.7   Lisp Port Acquisition

---

[23] The double : allows the **tcp-handler** to be found, since it was not created "exportable" in the **ip** package.

36

```
(send talking-port :open
    :active                          ; tcp will begin the procedure to establish
                                     ; connection  (default  vs :passive)
    talking-port-number             ; port number of destination host
    destination                      ; machine name or address if blank and
                                     ; in :passive mode local machine waits for
                                     ; connection
    30 )                             ; set max seconds before read request times out
```

Figure 4.8   Opening a Lisp Client Connection

```
(progn
    (send talking-port :send
        typebuffer
        1
        nil
        nil )
    (if (= (length lengthbuffer) 4)
        (send talking-port :send
            lengthbuffer
            4
            nil
            nil )
        (progn
        (loopfor *loopvariable* (length lengthbuffer) 4
              (send talking-port :send "0" 1 nil nil) )
        (send talking-port :send lengthbuffer (length lengthbuffer) nil nil) ) )
    (send talking-port :send
        buffer
        buffer-length
        t
        nil ) )
```

Figure 4.9   Sending a Message

data buffer is sent, when it is specified as t. The entire message is thus sent as a unit to the other machine.

2.   Symbolics 36xx

Genera 7 syntactic conventions are followed. The principle difference with Genera 6 conventions is in the *defmethod* function. In Genera 6 (and the TI Explorer), the method name follows the flavor name. In Genera 7, the method name precedes the

37

flavor name. Figure 4.10 shows the difference. It also shows the other main difference with the earlier code, that *streams* are used. The use of streams improves portability and eliminates the need for the **:reuse-iris** method[24]. It may be slightly slower, but any difference has been unnoticeable.

Another change was to remove the dependence on hard-coded addresses. The method **:init-destination-host** was added to the **conversation-with-iris** flavor (see Figure 4.11). By using the **net:parse-host** function, the application need only know the name of another machine. As network tables are updated, no change to the application code is necessary unless a different machine is desired.

```
(defmethod (conversation-with-iris :stop-iris)
   ()
   (progn (send talking-port :close)
          (send listening-port :close) ) )
```

Genera 6

```
(defmethod (:stop-iris conversation-with-iris)
   ()
   (progn (send talking-stream :close)
          (send listening-stream :close) ) )
```

Genera 7
Figure 4.10   Genera 6 and 7 *defmethod*

```
(defmethod (:init-destination-host conversation-with-iris)
      (name-of-host)
   (setf destination-host-object (net:parse-host name-of-host)) )
```

Figure 4.11   Generic Host Addressing

---

[24] The **:reuse-iris** method is retained for backward compatibility.

E. SUMMARY

For UNIX-based machines, generic routines are developed for semaphore use, shared memory use, and socket use. The socket routines use both *stream sockets* and *datagram sockets* in a simplex mode to provide directly connected client/servers and unconnected broadcasting communications. IRIS 2400, 3120, and 4D systems are fully supported. 4.3BSD systems are supported with mid-level socket calls only.

For Lisp machines, stream-based functions are available for direct connection as clients only. These functions are available directly if using Genera 7 syntax and with minor modification if using Genera 6 syntax.

V. USE BY APPLICATIONS

A. INTRODUCTION

The application using either direct connect or broadcast protocol is not concerned with system-level implementation details. Almost all aspects of shared memory, semaphore, and socket use are hidden. The number of other machines to be connected to, the use of dynamic memory allocation, and the names of the other machines are all that concern the application in setting up a connection. The synchronization, or lack thereof, in communication between machines is a design decision, not a protocol decision.

B. DIRECT CONNECT

A UNIX-based machine can be either a server, waiting for a client to call and establish a connection, or the client. A Lisp machine is always a client.

1. UNIX-Based Machines

The functions provided for UNIX-based machines are all written in C. They must be linked into the application program using them. Figure 5.1 is an example **make** file for creation of an application program on an IRIS system.

There are two independent processes, *send* and *receive*, that are spawned to create the sockets and monitor them. They are made separately with the *makefile*[25] contained in their subdirectory.

---

[25] See Appendix A

```
CFLAGS = -Zg -lm -g -p

SHARE = /work/barrow/share3/

MAIN = carsimu.c

OBJS = First group of .o files

OBJS1 = Second group of .o files

OBJS2 = Third group of .o files

OBJS3 = $(SHARE)io_single.o \
    $(SHARE)mpath.o \
    $(SHARE)semaphore.o \
    $(SHARE)shareseg.o \
    $(SHARE)support.o

OBJS4 = Fifth group of .o files

carsimu: $(MAIN) $(OBJS) $(OBJS1) $(OBJS2) $(OBJS3) $(OBJS4)
    cc -o carsimu $(MAIN) $(OBJS) $(OBJS1) $(OBJS2) $(OBJS3) $(OBJS4) $(CFLAGS) -lbsd

$(MAIN): const.h vars.h

$(OBJS): const.h vars.h

$(OBJS1): consi.h objects.h

$(OBJS2): const.h

$(SHARE)mpath.o: $(SHARE)shared.h
    cc -c -o $(SHARE)mpath.o $(SHARE)mpath.c $(CFLAGS)

$(SHARE)support.o: $(SHARE)shared.h
    cc -c -o $(SHARE)support.o $(SHARE)support.c $(CFLAGS)

$(SHARE)semaphore.o:
    cc -c -o $(SHARE)semaphore.o $(SHARE)semaphore.c $(CFLAGS)

$(SHARE)io_single.o: $(SHARE)shared.h
    cc -c -o $(SHARE)io_single.o $(SHARE)io_single.c $(CFLAGS)

$(SHARE)shareseg.o:
    cc -c -o $(SHARE)shareseg.o $(SHARE)shareseg.c $(CFLAGS)
```

Figure 5.1   Sample Application make File

a.   Application Setup

The server process must be started first. The application can set up the

communications paths as part of initialization, or it can do so only in response to a

specific operator command. In either case, there will be two messages returned to the terminal for **each** direct connection setup. Figure 5.2 illustrates a normal, single connection, response. Since the *receive* and *send* processes that provide the messages are independent, the two lines shown may be jumbled. A variety of errors can occur at this point. Table 5.1 gives the most common error messages, their cause, and solution.

```
Server waiting to connect to name
Server waiting to connect to name
```

Figure 5.2   Normal Server Response

Table 5.1   SERVER ERROR RESPONSES

Message	Cause	Solution
Server couldn't open a local socket:	Socket in use due to previous run not terminating with *deletemachinepath*	Run **ps**. Use **kill** to terminate any *receive* or *send* processes still running
Server couldn't bind address to local socket:	Socket in use due to previous run not terminating with *deletemachinepath*	Run **ps**. Use **kill** to terminate any *receive* or *send* processes still running
shmget: Permission denied	The shared memory segment already exists, but is owned by another *uid*	Change *key* in *machinepath* call, recompile, and rerun
shmget: Invalid argument	The shared memory segment already exists, but is too small because the value of LARGESTREAD has been increased	Run *rmshare* and rerun application
shmat: Permission denied	Someone else's *send* or *receive* process is being spawned	Check that proper path is used in *shared.h*, for application's *include* of *shared.h*, and in application's *Makefile*. Correct and recompile.
	Outdated software is being used.	Ensure that all modules are the most current. If some are not, get updated modules and recompile— especially *send* and *receive*.

42

The client process must not attempt connection until after the server is properly running (the messages in Figure 5.2 have been received). The application can set up the communications paths as part of initialization, or it can do so only in response to a specific operator command. When client communications setup is part of the initialization, care must be taken to wait for a ready server before starting the client. In either case, there will be two messages returned to the terminal for each direct connection setup. Figure 5.3 illustrates a normal, single connection, response. Since the *receive* and *send* processes that provide the messages are independent, the two lines shown may be jumbled. A variety of errors can occur at this point. Table 5.2 gives the most common error messages, their cause, and solution.

    b.   Coding Practices

    (1) <u>Connection</u>.   Making a connection requires two acts. The first is to set aside space for the data required. Figure 5.4 shows this code when local declaration is used. The *Machine* structure can also be declared globally. The second is to request the connection with a *machinepath*, *dynamicmachinepath*, or *dynamicmachinepaths* call. Table 5.3 compares the three types of call, while Figure 5.5 gives a server example for *dynamicmachinepath*. A description of the parameters used is in Appendix A, Section 2.a.

    For flexibility, there is often a requirement for command line specification of the machine to be connected to. For ease of use, there is often a

---

```
Connection established with name
Connection established with name
```

Figure 5.3   Normal Client Response

Table 5.2 CLIENT ERROR RESPONSES

Message	Cause	Solution
Client couldn't open a local socket:	Socket in use due to previous run not terminating with *deletemachinepath*	Run ps. Use **kill** to terminate any *receive* or *send* processes still running
Client couldn't connect to the remote server socket:	The server has not successfully started	Terminate client, restart server, restart client when server started
	The port numbers used by client do not correspond to those of server	Correct, recompile, and rerun
shmget: Permission denied	The shared memory segment already exists, but is owned by another *uid*	Change *key* in *machinepath* call, recompile, and rerun
shmget: Invalid argument	The shared memory segment already exists, but is too small because the value of *LARGESTREAD* has been increased	Run *rmshare* and rerun application
shmat: Permission denied	Someone else's *send* or *receive* process is being spawned	Check that proper path is used in *shared.h*, for application's *include* of *shared.h*, and in application's *Makefile*. Correct and recompile.
	Outdated software is being used.	Ensure that all modules are the most current. If some are not, get updated modules and recompile—especially *send* and *receive*.

```
#include "/work/barrcv/share3/shared.h"

main(argc,argv)

/***************************************************************

        LOCAL      DECLARATIONS

***************************************************************/

    Machine cardriver;              /* structure for communications system */
```

Figure 5.4   Creation of *Machine* Structure

Table 5.3   PATH CONNECTION

Function	Purpose
machinepath	Creates a link between two machines No subsequent dynamic memory allocation allowed
dynamicmachinepath	Creates a link between two machines Subsequent dynamic memory allocation allowed
dynamicmachinepaths	Creates a link between two machines Subsequent dynamic memory allocation allowed Multiple calls provide multiple links to one or more other machines

```
main(argc,argv)

/*************************************************************

    S Y S T E M      I N I T I A L I Z A T I O N S

*************************************************************/

        /* Open up the net path to other machine (iris3 default) */
    dynamicmachinepath(2,other_machine,4,5,"server",&cardriver,2000000);
```

Figure 5.5   Server Creation

requirement for a default specification. Figure 5.6 illustrates one way to accomplish this for a client. This example does not require that the network alias be defined to the system as it uses the complete address. The user, however, only enters the alias.

(2) Program Use. The simplest high-level communication paradigm is reading from and writing to the other machine. It closely parallels handling files and terminals in C. It was chosen for these reasons.

Twelve high-level functions are available. Four provide status information, four write to other_machine, and four read from other_machine. Table 5.4 describes these functions. The parameters used by these calls are described in Appendix A, Sections 1.a and 9.a.

45

```
main(argc,argv)

int argc;        /* argument count */
char *argv[];    /* pointers to the passed in arguments */
{
/************************************************************

         D A T A        D E C L A R A T I O N

************************************************************/

   char other_machine[50];          /* name of other machine */

/************************************************************

   S Y S T E M       I N I T I A L I Z A T I O N S

************************************************************/

   /* pull out the string from the argument list */
   if(argc > 2)
   {
      printf("NAV: incorrect argument count!  use nav <alias>\n");
      exit(1);
   }
   /* pull out the name of the other string, if it exists */
   if( argc == 2 )
   {
      strcpy( other_machine, "npscs-" );
      strcat( other_machine, argv[1] );
   }
   else
      strcpy( other_machine, "npscs-iris2" );

         /* Open up the net path to other machine (iris2 default) */
   dynamicmachinepath(2,other_machine,5,4,"client",&car,2000000);
```

Figure 5.6   Command Line Direction for Connection

There is a variety of ways to use these functions. Figure 5.7 illustrates a typical scenario. This code is from the display station of a two-workstation driver simulation. The display station provides its status (that of the "world") on each pass through its graphical display loop. The control station must read that status on each pass, to update the vehicle position on its track diagram. On each pass, the display station checks to see if any commands have been received. This is an asynchronous communication, as the display station continues with or without a control station

46

command. The asynchronous reads are guarded by a *receiver_has_data* call that detects arrival of a message. Other *receiver_has_data* calls are used to "busy wait" for the next message. In practice, it has not been necessary to include any but the first "busy wait" *receiver_has_data* call. TCP/IP buffers messages when they are not immediately read. It then blocks them into the largest grouping possible and delivers them when the next read occurs. The LARGESTREAD defined constant in *shared.h* determines this maximum grouping. The first message is read by *receive*. The socket is then ignored until the application reads the data. During this time, the other messages have all been sent and buffered by TCP/IP. There is a slight delay between the time the first message is read and the block containing all the rest is read. Thus the necessity for the first "busy wait" *receiver_has_data* call. The other "busy wait" *receiver_has_data* calls are simply for robustness.

The "busy wait" *sender_is_free* call determines if something has happened to the other machine or Ethernet. The first *write* will always succeed, as it goes to a buffer. If there is a communications problem, TCP/IP will not accept it and the

Table 5.4   COMMUNICATION FUNCTIONS

Function	Action
sender_is_free	Returns TRUE if a message can be sent.
receiver_has_data	Returns TRUE if a new message has been received.
received_type	Returns a character indicating the type of the message. CHARACTER_TYPE, INTEGER_TYPE, and FLOAT_TYPE are predefined. CHARACTER_ARRAY_TYPE, INTEGER_ARRAY_TYPE, and FLOAT_ARRAY_TYPE are predefined.
number_received	Returns an integer indicating how many elements in message.
write_character write_integer write_float write_characters	Send a single value of the type to other machine.
read_character read_integer read_float read_characters	Move single value of named type from buffer to application program storage.

47

```
main(argc,argv)
      .
      .
      .

/*****************************************************************

        M A I N     S I M U L A T I O N     L O O P

*****************************************************************/

    while(vehicle.command.condition != DONE)
    {
        /*****************************************************************
           Get commands (if any) from navigator.  Commands are all sent
           or none are sent so no information is needed as to which value
           is which.
        *****************************************************************/

        if( receiver_has_data( &cardriver ) )
        {
            read_integer(&cardriver, &vehicle.command.condition);
            while( !receiver_has_data( &cardriver ) ) /*printf("1")*/;
            read_integer(&cardriver, &vehicle.command.brakepedal);
            while( !receiver_has_data( &cardriver ) ) /*printf("2")*/;
            read_integer(&cardriver, &remote_mousex);
            while( !receiver_has_data( &cardriver ) ) /*printf("3")*/;
            read_float(&cardriver, &cmdspeed);
        }
            .
            .
            .


        /*****************************************************************
           Report all status information to navigator every cycle.
        *****************************************************************/

        write_float(&cardriver, &vehicle.state_vector[1]);
        while( !sender_is_free(&cardriver) ) printf("b");
        write_float(&cardriver, &vehicle.state_vector[2]);
        write_float(&cardriver, &vehicle.state_vector[3]);
        write_float(&cardriver, &vehicle.situation.distance_traveled);
        write_integer(&cardriver, &vehicle.command.condition);
        write_integer(&cardriver, &vehicle.command.brakepedal);
        write_integer(&cardriver, &vehicle.situation.lightcolor);
            .
            .
            .

    } /* while loop */
      .
      .
      .


}   /* main */
```

Figure 5.7   Synchronous Write / Asynchronous Read

sender_is_free call will return FALSE. This often occurs when there is a delay by the client in connecting to the server (the display station here). If there is a good connection, TCP/IP will accept and buffer all input. No other "busy wait" calls are needed. The other side of the communication is shown in Figure 5.8.

(3) Disconnection. Termination, with a *deletemachinepath* call for each path opened, is mandatory. If not performed, the sockets (and shared memory segment on System V UNIX machines) will not be returned to the system. Problems[26] may then occur on the next run. Figure 5.9 is an example termination when multiple paths have been opened [Ref. 11].

2. Lisp Machines

All necessary functions are contained in a single file. This file must be loaded before use. Figure 5.10 is an example. A Lisp machine is always a client and is started second. Figure 5.11 illustrates the message returned with a successful connection. Unsuccessful connections "hang" and return nothing.

a. Connection

The address of the server and the ports it is using must be specified. Figure 5.12 shows the ports specified as part of the loaded file. When using the older TI Explorer functions, the addresses are specified in the same way (see Figure 4.5) and then the machine desired is requested by number[27] (shown in Figure 5.13). When using the stream-based functions, the addresses are not specified by the user at all. The network tables are accessed, by host name, through the **select-host** function provided (shown in Figure 5.14). Once the instance of **conversation-with-iris** flavor has been completed

---

[26] See Tables 5.1 and 5.2

[27] A throwback to connection only with different IRIS machines.

49

```
main(argc,argv)
    .
    .
    .

    while(condition != DONE)
    {
        /*******************************************************************
            Receive all status information from car every cycle.
        ******************************************************************/

        while( !receiver_has_data( &car ) ) ;
        read_float(&car, &cy);
        while( !receiver_has_data( &car ) ) ;
        read_float(&car, &cx);
        while( !receiver_has_data( &car ) ) ;
        read_float(&car, &velocity);
        while( !receiver_has_data( &car ) ) ;
        read_float(&car, &rdistance);
        while( !receiver_has_data( &car ) ) ;
        read_integer(&car, &condition);
        while( !receiver_has_data( &car ) ) ;
        read_integer(&car, &brakeposition);
        while( !receiver_has_data( &car ) ) ;
        read_integer(&car, &lightcolor);
            .
            .
            .


        /*******************************************************************
            Send commands (if any) to car.  Commands are all sent
            or none are sent so no information is needed as to which value
            is which.
        ******************************************************************/

        if(anything_has_changed)
        {
            anything_has_changed = FALSE;
            write_integer(&car, &condition);
            while( !sender_is_free( &car ) ) printf("a") ;
            write_integer(&car, &brakeposition);
            while( !sender_is_free( &car ) ) printf("b") ;
            write_integer(&car, &mousex);
            while( !sender_is_free( &car ) ) printf("c") ;
            write_float(&car, &cmdvelocity);
        }   /* if(anything_has_changed) */
            .
            .
            .

    } /* while */
    .
    .
    .
} /* main */
```

Figure 5.8   Reciprocal Synchronous Read / Asynchronous Write

```
deletemachinepath(&TI);
deletemachinepath(&SYM3);
deletemachinepath(&SYM1);
delctemachinepath(&SYM4);

exit();
}
```

Figure 5.9   Connection Termination

```
;;; this is the communication package
(load "irisflavor")
```

Figure 5.10   Loading Lisp Flavor

```
"A conversation with the iris machine has been established"
```

Figure 5.11   Lisp Connection Message

```
(defvar *iris1-port1* 1027)                    ; this is the send port
(defvar *iris1-port2* 1026)                    ; this is the receive port
```

Figure 5.12   Setting Port Numbers with *defvar*

```
;;; get the network going
(iris 1)
(setq *battle* (make-instance 'conversation-with-iris))
(if (y-or-n-p "start networking ?") (send *battle* :start-iris))
```

Figure 5.13   Specifying Server in Lisp

51

```
(select-host iris2)
```

Figure 5.14   Specifying Server by Name in Lisp

with port numbers and host addresses, the connection is established with the method
:start-iris, see Figure 5.13.

b.   Program Use

The method **:get-iris** returns with the object sent by one message. The
method (**:put-iris object**) sends the object as one message. Figure 5.15 illustrates both.
Note how methods are added to flavor **conversation-with-iris** to *simplify the*
application interface even further. [Ref. 11]

c.   Disconnection

Disconnection is accomplished with the method **:stop-iris**, shown in
Figure 5.16.

C.   BROADCAST

Only UNIX-based machines support our broadcast protocol at this time. It is a
unidirectional protocol, but nothing prevents the establishment of two unidirectional
channels in opposite directions. Using two broadcast channels to emulate a direct
connect channel, however, loads all other machines on the network by requiring every
other machine to process each message. It is also less reliable. Broadcasting is good for
sending status information to many other machines, as long as those machines can
tolerate missing reports.

1.   Similarities With Direct Connect Protocol Use

Using the broadcast protocol is similar to using the direct connect protocol.
The same functions are used in the same way. Each connection must set aside space as

52

```
;;; definitions:
;;;
;;;     object: "n"    name:          character "1" .. "5"
;;;             x      x coordinate:  real
;;;             y      y coordinate:  real
;;;             z      z coordinate:  real
;;;             spd    speed:         real      speed of vehicle -10.00 to 25.00
;;;             dir    direction:     real      compass dir in degrees from GN
;;;
;;;     in lisp  ("n" (x y z spd dir))


;;; get an object in graphics environment (defined as above)

(defmethod (conversation-with-iris :object)
           ()
  (makeobj
    (send self :get-iris)
    (send self :get-iris)
    (send self :get-iris)
    (send self :get-iris)
    (send self :get-iris)
    (send self :get-iris) ) )

;;; vision returns a list of objects in the tank's field of vision (100m radius)
;;; this is effectively an association list

(defmethod (conversation-with-iris :vision)
           (tank)
  (let ((field nil)
        (n-objects 0) )
       (progn (send self :put-iris "V")
              (send self :put-iris tank)
              (if (equal "V" (send self :get-iris))
                  (progn (setq n-objects (send self :get-iris))
                         (dotimes
                             (x n-objects field)
                             (setq field (cons (send self :object) field)) ) )
              (progn
                (print "iris did not respond to the vision command sent from ")
                (princ "tank ")
                (princ tank) ) ) ) ) )
```

Figure 5.15   Application Communication in Lisp

in Figure 5.4. The same criteria for using a specific *machinepath* call apply (see Table 5.3). The same communications functions are available as in Table 5.4. Each connection must be terminated as in Figure 5.9.

```
(if (y-or-n-p "stop iris connection ?") (send *battle* :stop-iris))
```

Figure 5.16   Termination of Communications in Lisp

2.   Differences With Direct Connect Protocol Use

a.   Application Setup

The broadcast protocol is not directly modeled as a server/client relationship. The broadcaster broadcasts to whomever is prepared to receive. The receiver must be ready and so must be started first. Since the broadcaster is more similar to the server in a server/client model, this connection order seems exactly backward. No error will result if the broadcaster starts first, messages will simply not be received. The receiver message is shown in Figure 5.17. The broadcaster message is shown in Figure 5.18. When a direct connect channel is also required between the same two machines, achieving proper startup order is easy. Establish the direct connect channel first, then the soon-to-be broadcasting process sends a message telling the receiver to start up. Once started, the receiver process sends a message permitting the broadcaster to start.

ready to receive from *broadcaster_name*

Figure 5.17   Normal Receiver Response

Waiting to broadcast

Figure 5.18   Normal Broadcaster Response

b. Coding Practices

The parameters to the *machinepath* family of functions are used differently for the broadcast protocol. All are required to be present, but some are ignored (see Table 5.5). Since a broadcast channel is unidirectional, the *receive_type* application calls are meaningless to the broadcaster (the *receiver_has_data* call always returns false). The *send_type* application calls are meaningless to the receiver (the *sender_is_free* call always returns false).

D. SUMMARY

Using the same functions, an application can either broadcast or directly connect to another machine. The same steps of setup, connection, use, and termination are common to both protocols. Care must be taken in the timing of the two (or more) machines setup. After that, an application merely reads or writes data.

Table 5.5   MACHINEPATH PARAMETERS

Parameter	Function		
	machinepath	dynamicmachinepath	dynamicmachinepaths
nummachines	N/A		Number of channels that could be created by application. This includes both DIRECT CONNECT and BROADCAST channels.
segmentnum	Arbitrary integer. Should be different than another user's application.		
			Only first call's value used.
mname	DIRECT CONNECT and BROADCAST (receiver only): Name of machine to connect to.		
	BROADCAST (broadcaster only): Required but ignored		
sendportnum	DIRECT CONNECT: Number (0-3076) of port to be used to send to other machine.		
	BROADCAST (broadcaster only): Number (0-3076) of port to be used for broadcast.		
	BROADCAST (receiver only): Required but ignored		
receiveportnum	DIRECT CONNECT: Number (0-3076) of port to be used to receive from other machine.		
	BROADCAST (broadcaster only): Required but ignored		
	BROADCAST (receiver only): Number (0-3076) of port to be used for broadcast.		
server	"server": Create DIRECT CONNECT channel as a server.		
	"client": Create DIRECT CONNECT channel as a client.		
	"broadcast": Create BROADCAST channel as a broadcaster.		
	"receive": Create BROADCAST channel as a receiver.		
instructure	Address of Machine structure created to hold channel information.		
freespace	N/A	Amount of space to be used for dynamic memory allocation.	
			Only first call's value used.

56

VI. PERFORMANCE

A. INTRODUCTION

We look at the size of packets from our protocols. We also look at the effect of real applications on the network. We try to do this for both direct connect and broadcast protocols. However, no application making good use of broadcast protocols exists. Hence, we used a direct connect test application and replaced the channel with two broadcast channels.

B. DATA COLLECTION

The LANalyzer* EX 5500 network analyzer was used to gather Ethernet statistics. Version 2.0 of the software was used. The LANalyzer 5500 is a COMPAQ PORTABLE II** with a coprocessor board installed. The coprocessor board has an Intel 80286 CPU, an Intel 82586 LAN coprocessor, and two MBytes of memory. It performs packet collection, packet filtering, and network statistics calculation. The COMPAQ PORTABLE II processor handles user software control, screen updating and disk I/O. [Ref. 29]

Samples were taken while direct connect applications were running on *iris2* and *iris3*. To compare direct connect protocol with the broadcast protocol, test programs were used[28]. Table 6.1 summarizes the information collected. These programs send a character string, an integer, and a floating point number in a rotating sequence. The messages are either sent to the machine specified on the command line or are broadcast to all machines on the local network but only received from the machine specified.

---

* LANalyzer is a registered trademark of Excelan, Inc.

** COMPAQ PORTABLE II is a tradmark of the COMPAQ Computer Corporation.

[28] See programs *prog.c*, *prog2.c*, *gprog.c*, and *gprog2.c* in Appendix D.

57

Table 6.1   DIRECT CONNECT VERSUS BROADCAST STATISTICS

Run Number	Direct Connect			Broadcast		
	Number of Packets	Ave Packet Size (bytes)	Max Test Load (%)	Number of Packets	Ave Packet Size (bytes)	Max Test Load (%)
1	1031	91	.10	9498	69	1.0
2	1047	111	.05	9860	69	1.0
3	465	96	<.05	4000	68	1.0
4	698	95	.05	2556	68	1.0
5	334	103	.10	1262	68	1.0

The *visual simulation* application measured was a modified version of the driving simulator [Ref. 7]. Table 6.2 summarizes the information collected. This data was taken during the day[29]. The application's communication code is shown in Figure 5.7 and Figure 5.8. One trip around the track took approximately five minutes. Seven messages are sent every cycle to report status. Four messages are sent in the opposite direction, as required, to control the car. One circuit was driven, on autopilot, for each test run. There were about 500 cycles per test. Approximately 3600 messages were generated per test. The number of packets sent was less than half of this. The apparent discrepancy exists for two reasons. First, each packet sent also generates an

Table 6.2   APPLICATION NETWORK USE STATISTICS

Run Number	Number of Packets	Average Packet Size (bytes)	Peak Network Load (%)	Peak Test Load (%)	Average Network Load (%)
1	3747	89	13	.10	.5
2	3297	89	11	.15	1.0
3	4152	89	15	<.05	.5
4	2848	89	17	.15	.9
5	22830	89	17	.10	.3

---

[29] At night, with less competition for network re ources, the results were similar.

acknowledgement packet in return. By acknowledging each packet, the *stream socket* guarantee of delivery and proper sequence is met. Second, after the first packet (containing the first message) is received, the remaining three or six messages are immediately sent. The receiving process has often not yet handled the first one. The remaining messages are combined into one and all are read as one block. This reduces the interchange to a typical total of four packets per cycle, two with data and two for acknowledgement. Similarly, four packets are usually generated whenever the navigator process issues a command sequence to the car.

An evaluation of a five-workstation application [Ref. 11] was also made. This application used three Symbolics (*sym1*, *sym3*, and *sym4*), *exp1*, and *iris2* to perform its tasks. Statistics were similar to the other application, but the Symbolics **irisflavor.lisp**[30] exhibited some problem behavior. It sent three packets for every message. The first packet contained the *type field* only. The second packet contained both the *type field* and the *length field*. The third contained the entire message. If a second message immediately followed the first, three more packets were sent, each adding one field to the previous packet. Only one acknowledgement was received, as all packets in a group had the same identification number.

C. DISCUSSION

Attempting to use broadcast protocol with the simple test programs failed. One problem encountered was overflow of the sending buffer within the TCP/IP layers. The rapidity of attempted transmission was the cause. Higher network loading exacerbated the problem. When the test application was slowed down with *printf* calls (and the output redirected into a file) the buffer could keep up with sending requests. Using

---

[30] See Appendix C

59

broadcast protocol within a graphics display loop should pose no problems unless numerous data elements are transmitted at one time.

Without acknowledgement packets, broadcasting put fewer packets on the network than did the direct connect protocol. When overall load was haevy, some were lost. This poses a serious problem for *visual simulation* applications. Without an elaborate application-level protocol, the receiving process will never know what was intended to be sent. Since only one data object is transmitted at a time, labeling the data objects is difficult. All that is available is to alternately send different types and, after checking the type received, make a determination of the likely intent of the sending process. If a block of data, containing different types, could be sent as a single message, the decoding problem would become one of simply sequence checking. Missing status packets can be safely ignored in many situations. At most, a simple averaging algorithm can smooth any discontinuities caused by a missing packet. Timestamping, with a virtual timestamp, of each packet would eliminate the averaging requirement.

The Symbolics *stream* version is much less efficient, in terms of network utilization, than is the Explorer's. It still functions correctly, with no noticeable delay. As the amount of data to transmit increases, the Symbolics flavor will eventually have noticeable performance degradation.

The interconection of five machines loads the network only slightly more than does that of two. The limitation will be from the process swap overhead, not the network.

D. SUMMARY

The direct connect protocol sends fewer packets than messages. Half of the packets sent are acknowledgements. These acknowledgements provide the reliability of the direct connect protocol. The broadcast protocol sends one packet for each message. These packets tend to be smaller than those for the direct connect protocol. Until a

mechanism exists to bundle several messages into one broadcast packet, the broadcast protocol is of small value.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. LIMITATIONS

There are two primary limitations. First, the Lisp and C functions differ at the user level. This was done to allow each to be used readily by programmers "thinking" in their respective language. We have found this to be confusing to students who are inexperienced in both languages. Second, there is no simple means to transmit a block of data or an entire file. Each data element, unless it is part of an array of characters, must be sent separately. This was done to "hit a middle ground" between a complex facility—*printf* function—and low-level system calls. As long as only the direct connect protocol existed, this was only an annoyance. As discussed in Chapter 6, this is a critically limiting factor for the broadcast protocol.

The port to BSD UNIX systems without shared memory and semaphores was not completed. The socket handling aspects are portable, but the shared memory aspects are interwoven throughout the system. The difficult part of the porting will be designing the message-passing protocol for the pipe between the application and the *send* and *receive* processes, as discussed in Chapter 4. Other specific limitations include:

- no broadcast capability for Lisp machines
- no server capability for Lisp machines
- limited communication error handling—no signals are sent from the *send* or *receive* processes to the application process if they encounter problems
- limited read/write error handling—a read or write of the wrong type will be attempted and usually produce garbage
- no out-of-band capability
- Symbolics **iris-flavor.lisp** creates three packets per message

B. FUTURE RESEARCH AREAS

Implementation of the missing structure data type is one key area in which more work could be done. The most straight-forward solution to this would be to add messages to the send section of the shared memory array without signalling the *send* process to send it until the entire block was ready. Such a solution eliminates any need to change the receiving functions at the cost of either an additional sending function or an additional parameter to the existing send functions. The additional send function would be a *push* function and the existing send functions would be modified to never signal the *send* process to send. That would be left to the new *push* function. Adding a parameter to each send function would allow any send function to *push*. While in some respects simpler, changes to any application sending a block of data would have to carefully monitor which send function actually is *pushing*.

Creation of a Lisp flavor that mimics the UNIX functions would prove useful to C programmers who find a need for Lisp modules in their *visual simulation*. Adding server and broadcast capabilities would increase the applicability of the protocols to future *visual simulation* projects. Functions to break complex Lisp objects into simple ones and then combine these into a single message are necessary for the broadcast protocol. The Symbolics version should be corrected to send a packet only at message boundaries.

C. SUMMARY AND CONCLUSION

The routines described herein have already proved useful to researchers at the Naval Postgraduate School. With Ethernet loading never exceeding one percent, these routines are efficient enough to use without concern. With the additions mentioned above, the goal of an easy-to-use yet powerful system will be reached.

APPENDIX A - IRIS MODULE DESCRIPTIONS

1. io_single.c

a. Calling Protocols

This module contains functions that are intended for the application's use and functions that are used exclusively by them. The parameters for externally accessible functions are described below.

i. *number_received*

```
number_received( instructure )

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment
            */
```

ii. *read_character*

```
read_character(instructure,character_out)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment */
    char *character_out;   /* pointer to output character */
```

iii. *read_characters*

```
read_characters(instructure,outarray,arraysize)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment */
    char outarray[];       /* output character buffer */
    int arraysize;         /* the number of characters to be returned */
```

iv. *read_float*

```
read_float(instructure,float_out)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment */
    float *float_out;      /* pointer to output float */
```

v. *read_integer*

```
read_integer(instructure,integer_out)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment */
    int *integer_out;      /* pointer to output integer */
```

io_single.c

      vi. *received_type*

```
char received_type( instructure )

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment
            */
```

      vii. *write_character*

```
write_character(instructure,character_in)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment
            int instructure.sendsem    the semaphore to the sender */
    char *character_in;    /* pointer to input character */
```

      viii. *write_characters*

```
write_characters(instructure,inarray,arraysize)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment
            int instructure.receivesem    the semaphore to the receiver. */
    char *inarray;    /* input character buffer */
    long arraysize;        /* the number of characters input */
```

      ix. *write_float*

```
write_float(instructure,float_in)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment
            int instructure.sendsem    the semaphore to the sender */
    float *float_in;    /* pointer to input float */
```

      x. *write_integer*

```
write_integer(instructure,integer_in)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment
            int instructure.sendsem    the semaphore to the sender */
    int *integer_in;    /* pointer to input integer */
```

b. Code and Description

```
/******************************************************************
*                                                                *
*   TITLE   : Inter-Computer Communication Package               *
*                                                                *
*   MODULE : io_single.c                                         *
*                                                                *
*   VERSION: 3.0                                                 *
*                                                                *
*   DATE    : 15 December 1987                                   *
*                                                                *
*   AUTHOR : Theodore H. Barrow                                  *
*                                                                *
******************************************************************
*                                                                *
*   HISTORY:                                                     *
*                                                                *
*       VERSION: 1.0                                             *
*                                                                *
*       DATE    : 27 May 1987                                    *
*                                                                *
*       AUTHOR : Theodore H. Barrow                              *
*                                                                *
*       DESC.   : Originally part of support.c.  Contains the documented read *
*                 and write calls for use by the application programmer.      *
*                                                                *
*       VERSION: 2.0                                             *
*                                                                *
*       DATE    : 21 October 1987                                *
*                                                                *
*       AUTHOR : Theodore H. Barrow                              *
*                                                                *
*       DESC.   : Modified read routines to use a global array to manage the  *
*                 possibility of a partial message receipt.      *
*                                                                *
*       VERSION: 3.0                                             *
*                                                                *
*       DATE    : 15 December 1987                               *
*                                                                *
*       AUTHOR : Theodore H. Barrow                              *
*                                                                *
*       DESC.   : Modified read routines to use part of a buffer set instead of *
*                 the global array to manage the reception of a partial message. *
******************************************************************
*                                                                *
*                    RECORD OF CHANGES                           *
*                                                                *
*Version*  Date  *  Author                  *           * Affected  *Reqd*
*       *        *  Change Description       *           * Modules   *Vers*
******************************************************************
*       *        *          *               *           *       *   *
*       *        *          *               *           *       *   *
******************************************************************/
```

io_single.c

```c
#include "shared.h"
#include "gl.h"

/* The following routine copies a character into the shared segment.
   It puts the type CHARACTER_TYPE in the first byte and the
      length 0001 into the next four bytes.
   It then puts the total size at the top of the shared segment.
   It then sends a wakeup to the sender program.
   It uses an input structure since called by main program
*/

write_character(instructure,character_in)

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment

            int instructure.sendsem     the semaphore to the sender */

char *character_in;   /* pointer to input character */

{
   int msgsize  = 5 + CHARACTER_SIZE;    /* size of message */

   char *senderstart = instructure->segment + SENDEROFFSET;

   /* the + 9 is to skip over the first 4 bytes for the size
      of the shared memory data and the 5 bytes of header information */
   char *datastart = senderstart + 9;

   long *sentlength = (long *)instructure->segment + WSENDEROFFSET;

   /* insert the type code */
   *(senderstart + 4) = CHARACTER_TYPE;

   /* insert the length IN BYTES of the input data */
   sprintf((senderstart + 5), "%04d", CHARACTER_SIZE);

   /* move the data bytes */
   memcpy(datastart, character_in, CHARACTER_SIZE);

   /* copy out the size of the data from the shared segment top */
   *sentlength = msgsize;

   /* at this point, we send a wakeup to the sender program,
      indicating that he can reuse the shared segment.
   */
   V(instructure->sendsem);

}   /* write_character */
```

io_single.c

```c
/* The following routine converts an integer to a string and copies it
        into the shared segment.
   It puts the type INTEGER_TYPE in the first byte and the string length
        (in bytes) as an integer (in string format) into the next four bytes.
   It then puts the total size at the top of the shared segment.
   It then sends a wakeup to the sender program.
   It uses an input structure since called by main program

*/

write_integer(instructure,integer_in)

Machine *instructure;   /* includes

             char *instructure.segment    a pointer to the shared segment

             int instructure.sendsem      the semaphore to the sender */

int *integer_in;    /* pointer to input integer */

{
    char integer_string[20];   /* string for integer conversion */

    int length;                /* length of integer string */

    int msgsize;               /* size of message */

    char *senderstart = instructure->segment + SENDEROFFSET;

    /* the + 9 is to skip over the first 4 bytes for the size
        of the shared memory data and the 5 bytes of header information */
    char *datastart = senderstart + 9;

    long *sentlength = (long *)instructure->segment + WSENDEROFFSET;

    /* convert integer to string */
    sprintf( integer_string,"%d", *integer_in );

    /* find length of integer string and thus message */
    length = strlen( integer_string );
    msgsize  = 5 + length;

    /* insert the type code */
    *(senderstart + 4) = INTEGER_TYPE;

    /* insert the length IN BYTES of the input data */
    sprintf((senderstart + 5), "%04d", length);

    /* move the data bytes */
    memcpy(datastart, integer_string, length);

    /* copy out the size of the data from the shared segment top */
    *sentlength = msgsize;

    /* at this point, we send a wakeup to the sender program,
        indicating that he can reuse the shared segment.
    */
    V(instructure->sendsem);

}   /* write_integer */
```

68

io_single.c

```
/* The following routine converts a float to a string and copies it
       into the shared segment.
    It puts the type FLOAT_TYPE in the first byte and the length
       (in bytes) as an integer (in string format) into the next four bytes.
    It then puts the total size at the top of the shared segment.
    It then sends a wakeup to the sender program.
    It uses an input structure since called by main program
*/

write_float(instructure,float_in)

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment

            int instructure.sendsem      the semaphore to the sender */

float *float_in;   /* pointer to input float */

{
    char float_string[30];   /* string for float conversion */

    int length;                 /* length of float string */

    int msgsize;                /* size of message */

    char *senderstart = instructure->segment + SENDEROFFSET;

    /* the + 9 is to skip over the first 4 bytes for the size
       of the shared memory data and the 5 bytes of header information */
    char *datastart = senderstart + 9;

    long *sentlength = (long *)instructure->segment + WSENDEROFFSET;

    /* convert float to string */
    sprintf( float_string, "%f", *float_in );

    /* find length of float string and thus message */
    length = strlen( float_string );
    msgsize  = 5 + length;

    /* insert the type code */
    *(senderstart + 4) = FLOAT_TYPE;

    /* insert the length IN BYTES of the input data */
    sprintf((senderstart + 5), "%04d", length);

    /* move the data bytes */
    memcpy(datastart, float_string, length);

    /* copy out the size of the data from the shared segment top */
    *sentlength = msgsize;

    /* at this point, we send a wakeup to the sender program,
       indicating that he can reuse the shared segment.
    */
    V(instructure->sendsem);

}   /* write_float */
```

io_single.c

```
/* This routine returns the type of data received. */

char received_type( instructure )

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment
            */
{
    return( *(instructure->segment + RECEIVEROFFSET + 4) );
}
```

io_single.c

```c
/* This routine returns the number of data items received. */

number_received( instructure )

Machine *instructure;   /* includes

            char *instructure.segment   a pointer to the shared segment */
{
    int temp_int;

    char *protocolhold   = instructure->segment + PROTOCOLHOLDOFFSET;

    long *partreceived   = (long *)protocolhold;

    long *receivedlength = (long *)instructure->segment + WRECEIVEROFFSET;

    char *receiverstart  = instructure->segment + RECEIVEROFFSET;

    /* check if only part of protocol information received */

    if( *receivedlength < 5)
    {
        /* move data received (as well as length field) to holding area */
        memcpy( protocolhold, receiverstart, *receivedlength + 4 );

        /* get next message(s) */
        free_receiver(instructure->segment);
        V(instructure->receivesem);
        while( receiver_is_free(instructure->segment) ) /* wait */ ;

        /* copy rest of protocol data into holding area */
        memcpy( (protocolhold + *partreceived + 4), (receiverstart + 4),
                (5 - *partreceived) );
    }
    else
    {
        /* copy protocol data into holding area */
        memcpy( protocolhold, receiverstart, 9);

        /* initialize *partreceived so it can be used later */
        *partreceived = 0;
    }

    /* determine the length of the received integer string and thus message */
    sscanf( protocolhold + 5, "%d", &temp_int );

    switch( *(protocolhold + 4) )
    {
        case CHARACTER_TYPE:
            return( 1 );
            break;
        case INTEGER_TYPE:
            return( 1 );
            break;
        case FLOAT_TYPE:
            return( 1 );
            break;
        case CHARACTER_ARRAY_TYPE:
            return( temp_int/CHARACTER_SIZE );
            break;
        case INTEGER_ARRAY_TYPE:
            return( temp_int/INTEGER_SIZE );
            break;
        case FLOAT_ARRAY_TYPE:
            return( temp_int/FLOAT_SIZE );
    }
}   /* number_received */
```

io_single.c

```
/* The following routine returns a character from the shared segment.
   It frees the receiver side of the shared segment if it is empty.
   It then sends a wakeup to the receiver program.
   It uses an input structure since called by main program.
*/


read_character(instructure,character_out)

Machine *instructure;   /* includes

            char *instructure.segment   a pointer to the shared segment */

char *character_out;   /* pointer to output character */


{
    /* temporary storage for move of received data or for protocol information
       when partial receipt */
    char temp[LARGESTREAD];

    char *protocolhold   = instructure->segment + PROTOCOLHOLDOFFSET;

    /* first four bytes of holding area as integer */
    long *partreceived   = (long *)protocolhold;

    int msgsize          = 5 + CHARACTER_SIZE;    /* size of message */

    char *receiverstart  = instructure->segment + RECEIVEROFFSET;

    /* the + 9 is to skip over the first 4 bytes for the size
       of the shared memory data and the 5 bytes of header information */
    char *datastart      = receiverstart + 9;

    long *receivedlength = (long *)instructure->segment + WRECEIVEROFFSET;

    /* check if first part of protocol information is missing */
    if( *partreceived == 0 )
    {
        /* check if only part of protocol information received */
        if( *receivedlength <= 5)
        {
            /* move data received (as well as length field) to holding area */
            memcpy( protocolhold, receiverstart, *receivedlength + 4 );

            /* get next message(s) */
            free_receiver(instructure->segment);
            V(instructure >receivesem);
            while( receiver_is_free(instructure->segment) ) /* wait */ ;
        }
    }

    /* reset msgsize and datastart to correspond to partial receipt */
    msgsize    -= *partreceived;
    datastart  -= *partreceived;

    /* move the bytes */
    memcpy(character_out, datastart, CHARACTER_SIZE);

    /* make buffer ready for next read */
    reset_buffer( receivedlength, msgsize, instructure, datastart,
                  CHARACTER_SIZE, partreceived, receiverstart );

}   /* read character */
```

io_single.c

```c
/* The following routine converts a string in the shared segment
      into the returned integer.
   It frees the receiver side of the shared segment if it is empty.
   It then sends a wakeup to the receiver program.
   It uses an input structure since called by main program.
*/


read_integer(instructure,integer_out)

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment */

int *integer_out;    /* pointer to output integer */
{
   char integer_string[LARGESTREAD];   /* string storage for received data */

   char *protocolhold   = instructure->segment + PROTOCOLHOLDOFFSET;

   /* first four bytes of holding area as integer */
   long *partreceived   = (long *)protocolhold;

   int length;                          /* length of integer string read */

   long segmentlength;                  /* length of data of partial massage */

   int msgsize;                         /* size of message */

   char *receiverstart = instructure->segment + RECEIVEROFFSET;

   /* the + 9 is to skip over the first 4 bytes for the size
      of the shared memory data and the 5 bytes of header information */
   char *datastart = receiverstart + 9;

   long *receivedlength = (long *)instructure->segment + WRECEIVEROFFSET;

   /* determine proper protocol info and reset variables if necessary */
   get_protocol( protocolhold, partreceived, receivedlength, receiverstart,
                 instructure, &length, &msgsize, &datastart );

   /* check if only part of data has been received */

   if( *receivedlength < msgsize )
   {
      get_data( &segmentlength, receivedlength, partreceived,
                integer_string, &datastart, &msgsize,
                receiverstart, instructure, &length);

      /* convert to string */
      integer_string[segmentlength + msgsize] = '\0';

   else
   {
      /* move the integer string bytes */
      memcpy(integer_string, datastart, length);

      /* convert to string */
      integer_string[length] = '\0';
   }

   /* convert the received string to an integer */
   sscanf( integer_string, "%d", integer_out );

   /* make buffer ready for next read */
   reset_buffer( receivedlength, msgsize, instructure, datastart, length,
                 partreceived, receiverstart );

}   /* read integer */
```

io_single.c

```c
/* The following routine converts a string in the shared segment
       into the user supplied float.
    It frees the receiver side of the shared segment if empty.
    It then sends a wakeup to the receiver program.
    It uses an input structure since called by main program.
*/

read_float(instructure,float_out)

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment */

float *float_out;    /* pointer to output float */
{
    char float_string[LARGESTREAD];   /* string storage for received data */

    char *protocolhold    = instructure->segment + PROTOCOLHOLDOFFSET;

    /* first four bytes of holding area as integer */
    long *partreceived    = (long *)protocolhold;

    int length;                         /* length of float string read */

    long segmentlength;                 /* length of data of partial massage */

    int msgsize;                        /* size of message */

    char *receiverstart = instructure->segment + RECEIVEROFFSET;

    /* the + 9 is to skip over the first 4 bytes for the size
       of the shared memory data and the 5 bytes of header information */
    char *datastart = receiverstart + 9;

    long *receivedlength = (long *)instructure->segment + WRECEIVEROFFSET;


    /* determine proper protocol info and reset variables if necessary */
    get_protocol( protocolhold, partreceived, receivedlength, receiverstart,
                  instructure, &length, &msgsize, &datastart );

    /* check if only part of data has been received */
    if( *receivedlength < msgsize )
    {
        get_data( &segmentlength, receivedlength, partreceived,
                  float_string, &datastart, &msgsize,
                  receiverstart, instructure, &length);

        /* convert to string */
        float_string[segmentlength + msgsize] = '\0';
    }
    else
    {
        /* move the float string bytes */
        memcpy(float_string, datastart, length);

        /* convert to string */
        float_string[length] = '\0';
    }

    /* convert the received string to an float */
    sscanf( float_string, "%f", float_out );

    /* make buffer ready for next read */
    reset_buffer( receivedlength, msgsize, instructure, datastart, length,
                  partreceived, receiverstart );

}   /* read_float */
```

74

io_single.c

```
/* The following routine copies characters from an array
       into the shared segment.
   It puts the type CHARACTER_ARRAY_TYPE in the first byte and the
       array length (in bytes) as an integer into the next four bytes.
   It then puts the total size at the top of the shared segment.
   It then sends a wakeup to the sender program.
   It uses an input structure since called by main program

*/

write_characters(instructure,inarray,arraysize)

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment

            int instructure.receivesem     the semaphore to the receiver. */

char *inarray;    /* input character buffer */

long arraysize;       /* the number of characters input */

{
    int datasize = arraysize * CHARACTER_SIZE; /* size of data field */

    int msgsize  = 5 + datasize;                   /* size of message */

    char *senderstart = instructure->segment + SENDEROFFSET;

    /* the + 9 is to skip over the first 4 bytes for the size
        of the shared memory data and the 5 bytes of header information */
    char *datastart = senderstart + 9;

    long *sentlength = (long *)instructure->segment + WSENDEROFFSET;

    /* insert the type code */
    *(senderstart + 4) = CHARACTER_ARRAY_TYPE;

    /* insert the length IN BYTES of the input data */
    sprintf((senderstart + 5), "%04d", (int)datasize);

    /* move the data bytes */
    memcpy((datastart), inarray, datasize);

    /* copy out the size of the data from the shared segment top */
    *sentlength = 5 + datasize;

    /* at this point, we send a wakeup to the sender program,
        indicating that he can reuse the shared segment.
    */
    V(instructure->sendsem);

}   /* write_characters */
```

75

io_single.c

```
/* The following routine copies bytes from the shared segment
     into the user supplied array.
   It frees the receiver side of the shared segment if it is empty.
   It then sends a wakeup to the receiver program.
   It uses an input structure since called by main program.

*/

read_characters(instructure,outarray,arraysize)

Machine *instructure;   /* includes

           char *instructure.segment   a pointer to the shared segment */

char outarray[];    /* output character buffer */

int arraysize;      /* the number of characters to be returned */

{
    char *protocolhold   = instructure->segment + PROTOCOLHOLDOFFSET;

    /* first four bytes of holding area as integer */
    long *partreceived   = (long *)protocolhold;

    int length;                            /* length of character string read */

    long segmentlength;                    /* length of data of partial massage */

    int datasize = arraysize * CHARACTER_SIZE; /* size of requested data field */

    int requestsize;                           /* size of message */

    int msgsize  = 5 + datasize;               /* size of requested message */

    char *receiverstart = instructure->segment + RECEIVEROFFSET;

    /* the + 9 is to skip over the first 4 bytes for the size
       of the shared memory data and the 5 bytes of header information */
    char *datastart = receiverstart + 9;

    long *receivedlength = (long *)instructure->segment + WRECEIVEROFFSET;


    /* determine proper protocol info and reset variables if necessary */
    get_protocol( protocolhold, partreceived, receivedlength, receiverstart,
                  instructure, &length, &msgsize, &datastart );

    /* check if all of data (or more) was requested */

    if( length <= arraysize )
    {
        /* check if only part of data has been received */
        if( *receivedlength < msgsize )
        {
            get_data( &segmentlength, receivedlength, partreceived,
                      outarray, &datastart, &msgsize,
                      receiverstart, instructure, &datasize );
        }
        else
        {
            /* move the character bytes */
            memcpy(outarray, datastart, length);
        }

        /* make buffer ready for next read */
        reset_buffer( receivedlength, msgsize, instructure, datastart, datasize,
                      partreceived, receiverstart );
    }
```

76

```
    else
    {
        /* move the bytes */
        memcpy(outarray, datastart, datasize);

        /* make buffer ready for next read */
        reset_buffer( receivedlength, msgsize, instructure, datastart, datasize,
                      partreceived, receiverstart );
    }
}   /* read_characters */
```

io_single.c

```
/* These are various support routines used by several of the preceding
   functions.
*/

reset_buffer(receivedlength, msgsize, instructure, datastart, datasize,
             partreceived, receiverstart)

long *receivedlength;   /* first four bytes of receive part of shared seg */

int msgsize;            /* size of message read */

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment;

            int instructure.receivesem    the semaphore to the receiver. */

char *datastart;        /* address data starts in receive part of shared seg */

int datasize;           /* length of data part of message */

long *partreceived;     /* length of message received in previous block */

char *receiverstart;    /* address receive part of shared seg starts */

{
   char temp[LARGESTREAD];   /* temporary storage for move of received data */

   /* free the receiver segment if this is only message received */
   if(*receivedlength == msgsize)
   {
      free_receiver(instructure->segment);

      /* at this point, we should send a wakeup to the receiver program,
         indicating that he can reuse the shared segment.
      */
      V(instructure->receivesem);

   }
   else    /* shift data forward in shared memory segment */
   {
      *receivedlength -= msgsize;

      memcpy(temp, (datastart + datasize), (LARGESTREAD - msgsize));

      memcpy((receiverstart + 4), temp, (LARGESTREAD - msgsize));
   }

   /* reset *partreceived for next read */
   *partreceived = 0;

}   /* reset_buffer */
```

io_single.c

```
get_protocol( protocolhold, partreceived, receivedlength, receiverstart,
              instructure, length, msgsize, datastart )

char *protocolhold;      /* protocol holding area */

long *partreceived;      /* length of message received in previous block */

long *receivedlength;    /* first four bytes of receive part of sh*red seg */

char *receiverstart;     /* address receive part of shared seg starts */

Machine *instructure;    /* includes

           char *instructure.segment    a pointer to the shared segment

           int instructure.receivesem    the semaphore to the receiver. */

int *length;             /* length of data field in message */

int *msgsize;            /* length of message */

char **datastart;        /* address data starts in receive part of shared seg */

{
    /* check if first part of protocol information is missing */
    if( *partreceived == 0 )
    {
        /* check if only part of protocol information received */
        if( *receivedlength <= 5)
        {
            /* move data received (as well as length field) to holding area */
            memcpy( protocolhold, receiverstart, *receivedlength + 4 );

            /* get next message(s) */
            free_receiver(instructure->segment);
            V(instructure->receivesem);
            while( receiver_is_free(instructure->segment) ) /* wait */ ;

            /* copy rest of protocol data into holding area */
            memcpy( (protocolhold + *partreceived + 4), (receiverstart + 4),
                    (5 - *partreceived) );
        }
        else
        {
            /* copy protocol data into holding area */
            memcpy( protocolhold, receiverstart, 9);

            /* initialize *partreceived so it can be used later */
            *partreceived = 0;
        }
    }

    /* determine the length of the received data string and thus message */
    sscanf( protocolhold + 5, "%d", length );
    *msgsize  = 5 + *length - *partreceived;

    /* reset datastart to compensate for possible  partial receipt */
    *datastart -= *partreceived;

}   /* get_protocol */
```

io_single.c

```c
get_data( segmentlength, receivedlength, partreceived, string_array,
          datastart, msgsize, receiverstart, instructure, datasize )

long *segmentlength;    /* length of partial data */

long *receivedlength;   /* first four bytes of receive part of shared seg */

long *partreceived;     /* length of message received in previous block */

char string_array[];    /* storage for incoming characters */

char **datastart;       /* address data starts in receive part of shared seg */

int *msgsize;           /* length of message */

char *receiverstart;    /* address receive part of shared seg starts */

Machine *instructure;   /* includes

            char *instructure.segment    a pointer to the shared segment

            int instructure.receivesem   the semaphore to the receiver. */

int *datasize;          /* length of data field in message */
{
   /* determine length of data that has been received */
   *segmentlength = *receivedlength - 5 + *partreceived;

   /* copy the first segment of data to holding array */
   memcpy( string_array, *datastart, *segmentlength );

   /* reset msgsize and datastart to correspond to partial receipt */
   *msgsize    -= *segmentlength + 5 - *partreceived;
   *datastart   = receiverstart + 4;

   /* get next message(s) */
   free_receiver(instructure->segment);
   V(instructure->receivesem);
   while( receiver_is_free(instructure->segment) ) /* wait */ ;

   /* cycle through as many messages as it takes */
   while( *receivedlength < *msgsize )
   {
      /* copy the next segment of data to holding array */
      memcpy( &string_array[*segmentlength], *datastart, *receivedlength );

      /* reset msgsize and segmentlength to correspond to partial receipt */
      *msgsize       -= *receivedlength;
      *segmentlength -= *receivedlength;

      /* get next message(s) */
      free_receiver(instructure->segment);
      V(instructure->receivesem);
      while( receiver_is_free(instructure->segment) ) /* wait */ ;
   }

   /* copy the last segment of data to holding array */
   memcpy( &string_array[*segmentlength], *datastart, *msgsize );

   /* reset datasize to properly reflect last segment size */
   *datasize = *msgsize;

}   /* get data */
```

2. mpath.c

a. Calling Protocols

All functions in this module are meant to be accessible by the application.

These functions set up and tear down the communications path between two machines.

i. *deletemachinepath*

```
deletemachinepath(instructure)

    Machine *instructure;   /* structure to hold segment and semaphore info:
            char *instructure.segment -- returned ptr to the shared segment.
            int instructure.shmid -- returned system generated shared mem id
            int instructure.sendsem -- the returned send semaphore.
                                    We base it on the send portnumber.
            int instructure.receivesem -- the returned receive semaphore.
                                    We base it on the receive portnumber.
            */
```

ii. *machinepath*

```
machinepath(segmentnum,mname,sendportnum,receiveportnum,server,instructure)

    long segmentnum;    /* the key to use for the created shared segment */
    char mname[];       /* machinename character string */
    long sendportnum,receiveportnum;   /* send and receive port numbers */
    char server[];      /* this character string is either "client" or "server".
                        It indicates whether the sender/receiver should open
                        up as either a client or server. The first guy open
                        must be the server.
                        */
    Machine *instructure;   /* structure to hold segment and semaphore info:
            char *instructure.segment -- returned ptr to the shared segment.
            int instructure.shmid -- returned system generated shared mem id
            int instructure.sendsem -- the returned send semaphore.
                                    We base it on the send portnumber.
            int instructure.receivesem -- the returned receive semaphore.
            */
```

iii. *dynamicmachinepath*

```
dynamicmachinepath(segmentnum,mname,sendportnum,receiveportnum,server,
        instructure,freespace)

    long segmentnum;    /* the key to use for the created shared segment */
    char mname[];       /* machinename character string */
    long sendportnum,receiveportnum;   /* send and receive port numbers */
    char server[];      /* this character string is either "client" or "server".
                        It indicates whether the sender/receiver should open
                        up as either a client or server. The first guy open
                        must be the server.
                        */
    Machine *instructure;   /* structure to hold segment and semaphore info:
            char *instructure.segment -- returned ptr to the shared segment.
            int instructure.shmid -- returned system generated shared mem id
            int instructure.sendsem -- the returned send semaphore.
                                    We base it on the send portnumber.
            int instructure.receivesem -- the returned receive semaphore.
                                    We base it on the receive portnumber.
            */
    int freespace;      /* amount of freespace desired for dynamic memory allocation
                        after this routine has been called. */
```

81

iv. *dynamicmachinepaths*

```
dynamicmachinepaths(nummachines,segmentnum,mname,sendportnum,receiveportnum,
           server,instructure,freespace)

   int nummachines;    /* the maximum number of other machines to be attached */
   long segmentnum;    /* the key to use for the created shared segment */
   char mname[];       /* machinename character string */
   long sendportnum,receiveportnum;   /* send and receive port numbers */
   char server[];      /* this character string is either "client" or "server".
                          It indicates whether the sender/receiver should open
                          up as either a client or server. The first guy open
                          must be the server.
                    */
   Machine *instructure;   /* structure to hold segment and semaphore info:
               char *instructure.segment -- returned ptr to the shared segment.
               int instructure.shmid -- returned system generated shared mem id
               int instructure.sendsem -- the returned send semaphore.
                                        We base it on the send portnumber.
               int instructure.receivesem -- the returned receive semaphore.
                                        We base it on the receive portnumber.
                    */
   int freespace;      /* amount of freespace desired for dynamic memory allocation
             after this routine has been called. */
```

b.   <u>Code and Description</u>

```
/***********************************************************************
*                                                                     *
*  TITLE   : Inter-Computer Communication Package                     *
*                                                                     *
*  MODULE  : mpath.c                                                  *
*                                                                     *
*  VERSION : 5.0                                                      *
*                                                                     *
*  DATE    : 31 May 1988                                              *
*                                                                     *
*  AUTHOR  : Theodore H. Barrow                                       *
*                                                                     *
***********************************************************************
*                                                                     *
*  HISTORY:                                                           *
*                                                                     *
*     VERSION: 1.0                                                    *
*                                                                     *
*     DATE    : 6 February 1987                                       *
*                                                                     *
*     AUTHOR  : Michael J. Zyda                                       *
*                                                                     *
*     DESC.   : Contains routines machinepath and deletemachinepath for *
*               link creation/removal at a high level of abstraction. *
*                                                                     *
*     VERSION: 2.0                                                    *
*                                                                     *
*     DATE    : 27 May 1987                                           *
*                                                                     *
*     AUTHOR  : Theodore H. Barrow                                    *
*                                                                     *
*     DESC.   : Converted to use a structure for ease of use.         *
*                                                                     *
*     VERSION: 3.0                                                    *
*                                                                     *
*     DATE    : 21 October 1987                                       *
*                                                                     *
*     AUTHOR  : Theodore H  Barrow                                    *
*                                                                     *
*     DESC.   : Added function dynamicmachinepath to allow dynamic memory *
```

mpath.c

```
*              allocation after communications link established.           *
*                                                                          *
*  VERSION: 4.0                                                            *
*                                                                          *
*  DATE    : 15 December 1987                                              *
*                                                                          *
*  AUTHOR  : Theodore H. Barrow                                            *
*                                                                          *
*  DESC.   : Added function dynamicmachinepaths to allow use with multiple *
*            links.  Modified all creation routines to place sequence      *
*            numbers at end of command line for send and receive processes.*
*                                                                          *
*  VERSION: 5.0                                                            *
*                                                                          *
*  DATE    : 31 May 1988                                                   *
*                                                                          *
*  AUTHOR  : Theodore H. Barrow                                            *
*                                                                          *
*  DESC.   : Added broadcast and receive capability - one process spawned  *
****************************************************************************
*                                                                          *
*                     RECORD OF CHANGES                                     *
*                                                                          *
*Version*  Date  *  Author                    *          * Affected  *Reqd*
*        *       *    Change Description       *          * Modules   *Vers*
****************************************************************************
*        *       *        *                    *          *      *    *
*        *       *        *                    *          *      *    *
****************************************************************************/
```

83

mpath.c

```
#include "shared.h"   /* my special defines */
#include <gl.h>

deletemachinepath(instructure)

Machine *instructure;   /* structure to hold segment and semaphore info:

            char *instructure.segment -- returned ptr to the shared segment.

            int instructure.shmid -- returned system generated shared mem id

            int instructure.sendsem -- the returned send semaphore.
                                    We base it on the send portnumber.

            int instructure.receivesem -- the returned receive semaphore.
                                    We base it on the receive portnumber.
                */
{

    /* kill the receiver process... */
    kill_receiver(instructure->segment,instructure->receivesem);

    /* kill the sender process... */
    kill_sender(instructure->segment,instructure->sendsem);

    /* detach and delete the shared segment... */
    deletesharedsegment(instructure->segment,instructure->shmid);
}
```

mpath.c

```
/*
   For direct connection, both send and receive processes are spawned.
   For broadcast, either send or receive process is spawned.
   The machinepath routine performs the following:

   (1) creates a shared segment.
   (2) creates a send and/or receive semaphore based on the send and receive
       port numbers.
   (3) free_sender(segment) and/or free_receiver(segment)
   (4) spawns off the send and/or receive processes.
       system("send sharedseg# machinename port# server/client/broadcast 0&");
       system("receive sharedseg# machinename port# server/client/receive 0&");
   (5) the send and receive semaphores, the pointer to the shared segment,
       and the id of the shared segment are placed in a structure of type
       Machine that is declared in the calling program.
*/
machinepath(segmentnum,mname,sendportnum,receiveportnum,server,instructure)


long segmentnum;   /* the key to use for the created shared segment */

char mname[];      /* machinename character string */

long sendportnum,receiveportnum;   /* send and receive port numbers */

char server[]      /* this character string is either "client", "server",
                      "broadcast", or "receive". If direct connection wanted,
                      it indicates whether the sender/receiver should open
                      up as either a client or server. The first guy open
                      must be the server.  If broadcast wanted, it indicates
                      whether to open up as broadcaster or receiver.
                   */

Machine *instructure;   /* structure to hold segment and semaphore info:

                char *instructure.segment -- returned ptr to the shared segment.

                int instructure.shmid -- returned system generated shared mem id

                int instructure.sendsem -- the returned send semaphore.
                                               We base it on the send portnumber.

                int instructure.receivesem -- the returned receive semaphore.
                   */
{
   char *sharedsegment();       /* shared segment creation function */

   int semtran();               /* semaphore creating routine. */

   char temp[200], temp2[200]; /* temp character arrays */


   /* create the shared segment */
   instructure->segment = sharedsegment(segmentnum,MAXSHAREDSIZE,&instructure->shmid);

   /* create the send semaphore. (unused if receiving broadcast messages) */
   instructure->sendsem = semtran(sendportnum);

   /* create the receive semaphore (unused if broadcasting messages) */
   instructure->receivesem = semtran(receiveportnum);

   /* free the sender and receiver parts of the shared segment */
   init_shared_buffer(instructure->segment);

   /* spawn off the sender process */

   if( strcmp( server, "receive" ) != 0 )
   {
```

85

```
    /* add the start of the line, i.e. the program to run */
    strcpy(temp,SENDLOCATION);
    strcat(temp," ");

    /* add the number of the sharedsegment in text */
    sprintf(temp2,"%d",instructure->shmid);
    strcat(temp,temp2);
    strcat(temp," ");

    /* add on the machine name */
    strcat(temp,mname);
    strcat(temp," ");

    /* add the port number */
    sprintf(temp2,"%d",sendportnum);
    strcat(temp,temp2);
    strcat(temp," ");

    /* indicate whether a server, a client, or a broadcaster */
    strcat(temp,server);
    strcat(temp," 0");

    /* spawn off into the background */
    strcat(temp,"&");

    /* spawn off the sender */
    if( system(temp) == -1 )
        perror("SEND system call failed");
}
else
{
    /* kill sender (which really doesn't exist anyway) so that the
       sender_is_free() call will always return FALSE.
       A similar thing does not have to be done for receiver_has_data()
       in a broadcasting path since it will always return FALSE anyway */
    kill_sender( instructure->segment, instructure->sendsem );
}


/* spawn off the receiver process */

if( strcmp( server, "broadcast" ) != 0 )
{
    /* add the start of the line, i.e. the program to run */
    strcpy(temp,RECEIVELOCATION);
    strcat(temp," ");

    /* add the number of the sharedsegment in text */
    sprintf(temp2,"%d",instructure->shmid);
    strcat(temp,temp2);
    strcat(temp," ");

    /* add on the machine name */
    strcat(temp,mname);
    strcat(temp," ");

    /* add the port number */
    sprintf(temp2,"%d",receiveportnum);
    strcat(temp,temp2);
    strcat(temp," ");

    /* indicate whether a server, a client, or a broadcast receiver */
    strcat(temp,server);
    strcat(temp," 0");

    /* spawn off into the background */
    strcat(temp,"&");
```

```
        /* spawn off the receiver */
        if( system(temp) == -1 )
            perror("RECEIVE system call failed");
    }
}
```

mpath.c

```
/*
   For direct connection, both send and receive processes are spawned.
   For broadcast, either send or receive process is spawned.
   The dynamicmachinepath routine performs the following:

   (1) creates a shared segment and attaches it to the main program virtual
       space after an allocation of free memory space.
   (2) creates a send and/or receive semaphore based on the send and receive
       port numbers.
   (3) free_sender(segment) and/or free_receiver(segment)
   (4) spawns off the send and/or receive processes.
       system("send sharedseg# machinename port# server/client/broadcast 0&");
       system("receive sharedseg# machinename port# server/client/receive 0&");
   (5) the send and receive semaphores, the pointer to the shared segment,
       and the id of the shared segment are placed in a structure of type
       Machine that is declared in the calling program.
*/


dynamicmachinepath(segmentnum,mname,sendportnum,receiveportnum,server,
                   instructure,freespace)


long segmentnum;    /* the key to use for the created shared segment */

char mname[];       /* machinename character string */

long sendportnum,receiveportnum;    /* send and receive port numbers */

char server[];      /* this character string is either "client", "server",
                       "broadcast", or "receive". If direct connection wanted,
                       it indicates whether the sender/receiver should open
                       up as either a client or server. The first guy open
                       must be the server. If broadcast wanted, it indicates
                       whether to open up as broadcaster or receiver.
                    */

Machine *instructure;   /* structure to hold segment and semaphore info:

                char *instructure.segment -- returned ptr to the shared segment.

                int instructure.shmid -- returned system generated shared mem id

                int instructure.sendsem -- the returned send semaphore.
                                           We base it on the send portnumber.

                int instructure.receivesem -- the returned receive semaphore.
                                              We base it on the receive portnumber.
                */

int freespace;      /* amount of freespace desired for dynamic memory allocation
                       after this routine has been called. */
{
   char *dynamicsharedsegment();    /* shared segment creation function */

   int semtran();                   /* semaphore creating routine. */

   char temp[200], temp2[200],      /* temp character arrays */


   /* create the shared segment */
   instructure->segment = dynamicsharedsegment(1,segmentnum,MAXSHAREDSIZE,
                                               &instructure->shmid,freespace);

   /* create the send semaphore. (unused if receiving broadcast messages) */
   instructure->sendsem = semtran(sendportnum);

   /* create the receive semaphore (unused if broadcasting messages) */
   instructure->receivesem = semtran(receiveportnum);
```

88

mpath.c

```c
/* free the sender and receiver parts of the shared segment */
init_shared_buffer(instructure->segment);

/* spawn off the sender process */

if( strcmp( server, "receive" ) != 0 )
{
    /* add the start of the line, i.e. the program to run */
    strcpy(temp,SENDLOCATION);
    strcat(temp," ");

    /* add the number of the sharedsegment in text */
    sprintf(temp2,"%d",instructure->shmid);
    strcat(temp,temp2);
    strcat(temp," ");

    /* add on the machine name */
    strcat(temp,mname);
    strcat(temp," ");

    /* add the port number */
    sprintf(temp2,"%d",sendportnum);
    strcat(temp,temp2);
    strcat(temp," ");

    /* indicate whether a server, a client, or a broadcaster */
    strcat(temp,server);
    strcat(temp," 0&");

    /* spawn off the sender into the background */
    if( system(temp) == -1 )
        perror("SEND system call failed");
}
else
{
    /* kill sender (which really doesn't exist anyway) so that the
       sender_is_free() call will always return FALSE.
       A similar thing does not have to be done for receiver_has_data()
       in a broadcasting path since it will always return FALSE anyway */
    kill_sender( instructure->segment, instructure->sendsem );
}


/* spawn off the receiver process */

if( strcmp( server, "broadcast" ) != 0 )
{
    /* add the start of the line, i.e. the program to run */
    strcpy(temp,RECEIVELOCATION);
    strcat(temp," ");

    /* add the number of the sharedsegment in text */
    sprintf(temp2,"%d",instructure->shmid);
    strcat(temp,temp2);
    strcat(temp," ");

    /* add on the machine name */
    strcat(temp,mname);
    strcat(temp," ");

    /* add the port number */
    sprintf(temp2,"%d",receiveportnum);
    strcat(temp,temp2);
    strcat(temp," ");

    /* indicate whether a server, a client, or a broadcast receiver */
    strcat(temp,server);
    strcat(temp," 0&");
```

89

mpath.c

```
        /* spawn off the receiver into the background */
        if( system(temp) == -1 )
            perror("RECEIVE system call failed");
    }
}
```

mpath.c

```
/*
   For direct connection, both send and receive processes are spawned.
   For broadcast, either send or receive process is spawned.
   The dynamicmachinepaths routine performs the following:

   (1) creates a shared segment large enough for multiple attachments
       and attaches it to the main program virtual space after an allocation
       of free memory space.
   (2) creates a send and/or receive semaphore based on the send and receive
       port numbers.
   (3) free_sender(segment) and/or free_receiver(segment)
   (4) spawns off the send and/or receive processes.
       system("send sharedseg# machinename port# server/client/broadcast 0&");
       system("receive sharedseg# machinename port# server/client/receive 0&");
   (5) the send and receive semaphores, the pointer to the shared segment,
       and the id of the shared segment are placed in a structure of type
       Machine that is declared in the calling program.
*/


dynamicmachinepaths(nummachines,segmentnum,mname,sendportnum,receiveportnum,
                    server,instructure,freespace)

int nummachines;    /* the maximum number of other machines to be attached */

long segmentnum;    /* the key to use for the created shared segment */

char mname[];       /* machinename character string */

long sendportnum,receiveportnum;    /* send and receive port numbers */

char server[];      /* this character string is either "client", "server",
                       "broadcast", or "receive".  If direct connection wanted,
                       it indicates whether the sender/receiver should open
                       up as either a client or server.  The first guy open
                       must be the server.  If broadcast wanted, it indicates
                       whether to open up as broadcaster or receiver.
                    */

Machine *instructure;    /* structure to hold segment and semaphore info:

                char *instructure.segment -- returned ptr to the shared segment.

                int instructure.shmid -- returned system generated shared mem id

                int instructure.sendsem -- the returned send semaphore.
                                            We base it on the send portnumber.

                int instructure.receivesem -- the returned receive semaphore.
                                               We base it on the receive portnumber.
                    */

int freespace;      /* amount of freespace desired for dynamic memory allocation
                       after this routine has been called. */
{
   char *dynamicsharedsegment();    /* shared segment creation function */

   int semtran();                   /* semaphore creating routine. */

   char temp[200], temp2[200];      /* temp character arrays */

   static Boolean firsttime = TRUE; /* flag to detect multiple requests */

   static int sequencenum  = 0;     /* sequence number for receive/send */

   static int totmachines;          /* max attachments permitted */

   /* check for first time called and establish max possible attachments */
   if( firsttime )
```

91

```
{
    totmachines = nummachines;
    firsttime   = FALSE;
}
else
    ++sequencenum;

/* check for violation of maximum attachments */
if( sequencenum >= totmachines )
{
    perror("mpath: Too many attachments attempted");
    exit( -1 );
}

/* create the shared segment */
instructure->segment = dynamicsharedsegment(nummachines,segmentnum,
                                            MAXSHAREDSIZE,
                                            &instructure->shmid,freespace);

/* create the send semaphore. (unused if receiving broadcast messages) */
instructure->sendsem = semtran(sendportnum);

/* create the receive semaphore (unused if broadcasting messages) */
instructure->receivesem = semtran(receiveportnum);

/* free the sender and receiver parts of the shared segment */
init_shared_buffer(instructure->segment);

/* spawn off the sender process */

if( strcmp( server, "receive" ) != 0 )
{
    /* add the start of the line, i.e. the program to run */
    strcpy(temp,SENDLOCATION);
    strcat(temp," ");

    /* add the number of the sharedsegment in text */
    sprintf(temp2,"%d",instructure->shmid);
    strcat(temp,temp2);
    strcat(temp," ");

    /* add on the machine name */
    strcat(temp,mname);
    strcat(temp," ");

    /* add the port number */
    sprintf(temp2,"%d",sendportnum);
    strcat(temp,temp2);
    strcat(temp," ");

    /* indicate whether a server, a client, or a broadcaster */
    strcat(temp,server);
    strcat(temp," ");

    /* add the machine sequence number */
    sprintf(temp2,"%d",sequencenum);
    strcat(temp,temp2);

    /* spawn off into the background */
    strcat(temp,"&");

    /* spawn off the sender */
    if( system(temp) == -1 )
        perror("SEND system call failed");
}
else
{
    /* kill sender (which really doesn't exist anyway) so that the
```

```
            sender_is_free() call will always return FALSE.
            A similar thing does not have to be done for receiver_has_data()
            in a broadcasting path since it will always return FALSE anyway */
        kill_sender( instructure->segment, instructure->sendsem );
    }


    /* spawn off the receiver process */

    if( strcmp( server, "broadcast" ) != 0 )
    {
        /* add the start of the line, i.e. the program to run */
        strcpy(temp,RECEIVELOCATION);
        strcat(temp," ");

        /* add the number of the sharedsegment in text */
        sprintf(temp2,"%d",instructure->shmid);
        strcat(temp,temp2);
        strcat(temp," ");

        /* add on the machine name */
        strcat(temp,mname);
        strcat(temp," ");

        /* add the port number */
        sprintf(temp2,"%d",receiveportnum);
        strcat(temp,temp2);
        strcat(temp," ");

        /* indicate whether a server, a client, or a broadcast receiver */
        strcat(temp,server);
        strcat(temp," ");

        /* add the machine sequence number */
        sprintf(temp2,"%d",sequencenum);
        strcat(temp,temp2);

        /* spawn off into the background */
        strcat(temp,"&");

        /* spawn off the receiver */
        if( system(temp) == -1 )
            perror("RECEIVE system call failed");
    }
}
```

93

3. **netV.c**

   a. <u>Calling Protocols</u>

   This module contains the low-level socket-managing calls. No functions in this module are intended for application programs. This module is only linked into the *send* and *receive* processes.

   b. <u>Code and Description</u>

```
/*********************************************************************
*                                                                   *
*  TITLE   : Inter-Computer Communication Package                   *
*                                                                   *
*  MODULE  : netV.c                                                 *
*                                                                   *
*  VERSION: 5.0                                                     *
*                                                                   *
*  DATE    : 31 May 1988                                            *
*                                                                   *
*  AUTHOR  : Theodore H. Barrow                                     *
*                                                                   *
*********************************************************************
*                                                                   *
*  HISTORY:                                                         *
*                                                                   *
*      VERSION: 1.0                                                 *
*                                                                   *
*      DATE    : 19 November 1986                                   *
*                                                                   *
*      AUTHOR  : Michael J. Zyda                                    *
*                                                                   *
*      DESC.   : Contains routines connect_server and connect_client to allow *
*                two machines with Unix System V to communicate via sockets.  *
*                                                                   *
*      VERSION: 2.0                                                 *
*                                                                   *
*      DATE    : 29 April 1987                                      *
*                                                                   *
*      AUTHOR  : Michael J. Zyda                                    *
*                                                                   *
*      DESC.   : Converted to work with 4.2BSD sockets.            *
*                                                                   *
*      VERSION: 3.0                                                 *
*                                                                   *
*      DATE    : 27 May 1987                                        *
*                                                                   *
*      AUTHOR  : Theodore H. Barrow                                 *
*                                                                   *
*      DESC.   : Eliminated excess variables, some unused and some unnecessary.*
*                                                                   *
*      VERSION: 4.0                                                 *
*                                                                   *
*      DATE    : 21 August 1987                                     *
*                                                                   *
*      AUTHOR  : Theodore H. Barrow                                 *
*                                                                   *
*      DESC.   : Improved reliability of socket connection and disconnection. *
*                                                                   *
*      VERSION: 5.0                                                 *
*                                                                   *
*      DATE    : 31 May 1988                                        *
```

94

netV.c

```
*                                                                        *
*       AUTHOR : Theodore H. Barrow                                      *
*                                                                        *
*       DESC.   : Added start_broadcast() and broadcast_receive() to provide *
*                 datagram sockets for broadcast use.  These sockets use the *
*                 default Internet broadcast addressing.                 *
*************************************************************************
*                                                                        *
*                       RECORD OF CHANGES                                *
*                                                                        *
*Version*  Date  *  Author                    *           * Affected  *Reqd*
*         *       *    Change Description       *           * Modules   *Vers*
*************************************************************************
*   4.1  * 4Jan88 * T. H. Barrow              *           * send.c    *4.0 *
*        *        *  Changed include library pathnames for IRIS 4D.* receive.c *4.0 *
*************************************************************************
*        *        *          *                *           *           *    *
*        *        *          *                *           *           *    *
***************************************************************************/
```

95

netV.c

```
/*
This segment, when linked into a program on a computer with a UNIX 4.2 BSD
operating system, will allow the program to communicate with programs
executing on other computer systems over an Internet network.
*/

#define TRUE 1

/* include files for UNIX 4.2 BSD.  These are all called from the bsd
   subdirectory in /usr/include.  The file sys/types.h also exists and is
   included when bsd/sys/types.h is used.  This was done for ease of change
   if and when Silicon Graphics changes the include library structure. */
#include <sys/types.h>
#include <sys/socket.h>
#include <bsd/netinet/in.h>
#include <bsd/netdb.h>

/*****************************************************************

The connect_server(remote_client_name, port_number) function performs
the actions required to connect a server system to a remote client system

*****************************************************************/


int connect_server(remote_client_name, port_number)

char remote_client_name[];  /* name of the remote client system */

int  port_number;           /* port number to the remote client system */

{
    char *ptr_client_name;      /* pointer to the remote client system's name */

    int local_server_socket;    /* local socket number */

    int socket();               /* function that opens a socket */

    int accept();               /* function that accepts a connection from
                                   a remote client socket */

    int remote_client_socket = -1;  /* socket number of remote client system */

    /* protocol and address data structure for socket */
    static struct sockaddr_in address  = { AF_INET };

    long remote_client_address;      /* address of the remote client system */

    short remote_client_port;        /* port number of the remote client system */

    int address_size;            /* size of address of remote client system */


    /* create socket structure from input parameters */

    /* get a pointer to the remote client system's name */
    ptr_client_name = remote_client_name;

    /* convert the remote client system name to its address.
       Note that gethostbyname() requires a pointer to a pointer */
    remote_client_address = (long)gethostbyname(&ptr_client_name);

    /* set the remote client port number above the system reserved ports
       by adding the remote client port number to the number of reserved ports */
    remote_client_port = IPPORT_RESERVED + port_number;

    /* remote client system address family (Internet in this case) */
    address.sin_family = AF_INET ;
```

96

```
/* place the remote client port number into the address data structure
   in network byte order */
address.sin_port = htons(remote_client_port);

/* place the remote client system's address in the address data structure */
address.sin_addr.s_addr = remote_client_address;

/* find number of bytes in the remote client address */
address_size = sizeof(remote_client_address);

/* attempt to open a local socket */
local_server_socket = socket(AF_INET, SOCK_STREAM, 0);

if(local_server_socket < 0)
    perror("Server couldn't open a local socket:");
else
{
    if(bind(local_server_socket, (caddr_t)&address, sizeof(address)) < 0)
        perror("Server couldn't bind address to local socket:");

    /* set the maximum number of remote client systems to be connected to */
    listen(local_server_socket, SOMAXCONN);

    printf("Server waiting to connect to %s\n", remote_client_name);

    /* attempt to accept a connection */
    remote_client_socket = accept(local_server_socket, &address,
                                  &address_size);

    if(remote_client_socket < 0)
    {
        /* an error occurred in the server attempting to
           accept a connection from remote client system */
        perror("Server couldn't accept connection from remote client system:");

        shutdown(local_server_socket, 2);
        close(local_server_socket);
    }

    /* else the server accepted a connection from the remote client system */

}

/* return the socket number of the remote client system */
return(remote_client_socket);

} /* connect_server */
```

netV.c

```
/******************************************************************

The connect_client(remote_server_name, port_number) function performs
all the actions required to connect a client system to a remote server
system

******************************************************************/


int connect_client(remote_server_name, port_number)

char remote_server_name[];    /* name of the remote server system */

int  port_number;             /* port number to the remote server system */

{
    int local_client_socket;     /* local socket number */

    int socket();                /* function that opens a socket */

    /* function that connects local socket to remote server socket */
    int connect();

    int remote_server_socket;    /* socket number on remote server system  */

    /* the protocol and address data structure specified for the socket */
    static struct sockaddr_in address = { AF_INET };

    struct hostent *remote_server_address; /* address of remote server system */

    short remote_server_port;             /* port number of remote system */


    /* create socket structure from input parameters */


    /* convert the remote server system name to its address.
       Note that gethostbyname() requires a pointer only in this case */
    remote_server_address = gethostbyname(remote_server_name);

    /* clear out the address structure */
    bzero((char *)&address, sizeof(address));

    /* copy the remote server address structure into the address structure */
    bcopy(remote_server_address->h_addr,
          (char *)&address.sin_addr,
          remote_server_address->h_length);

    /* set remote server port number above the system reserved ports by adding
       the user's remote server port number to the number of reserved ports */
    remote_server_port = IPPORT_RESERVED + port_number;

    /*  remote server system address family(Internet in this case) */
    address.sin_family = AF_INET;

    /* place the remote server port number into the address structure
       in network byte order */
    address.sin_port = htons(remote_server_port);

    /* attempt to obtain a local socket */
    local_client_socket = socket(AF_INET, SOCK_STREAM, 0);

    if(local_client_socket < 0)
        perror("Client couldn't open a local socket:");
    else
    {
        /* place Internet address family type in address structure */
        address.sin_family = AF_INET;
```

98

netV.c

```
/* attempt to connect local client socket to remote server socket */
remote_server_socket = connect(local_client_socket, (caddr_t)&address,
                               sizeof(address));

if(remote_server_socket < 0)
{
    /* error occurred in attempting to connect to remote server socket */
    perror("Client couldn't connect to the remote server socket:");

    shutdown(local_client_socket, 2);
    close(local_client_socket);

    /* set local_client_socket so that negative value is
       always returned when an error occurs
    */
    local_client_socket = remote_server_socket;
}
else
    /* successfully connected to the remote server system */
    printf("Connection established with %s.\n",remote_server_name);

}

/* return the socket number of the local client system */
return(local_client_socket);

} /* connect_client */
```

netV.c

```c
/***************************************************************
The start_broadcast(port_number) function performs
the actions required to initiate a datagram broadcast socket.

***************************************************************/


int start_broadcast(port_number)

int port_number;              /* port number for the remote receiver system */

{
   int broadcast_socket;     /* local socket number */

   int socket();             /* function that opens a socket */

   int setsockopt();         /* function that sets a socket to allow broadcast */

   int on = TRUE;            /* to set broadcast toggle on for socket */

   /* protocol and address data structure for socket */
   static struct sockaddr_in address  = { AF_INET };

   short broadcast_port;        /* port number broadcast heard from */


   /* create local socket structure from input parameters */

   /* set the broadcast port number above the system reserved ports
      by adding the broadcast port number to the number of reserved ports */
   broadcast_port = IPPORT_RESERVED + port_number;

   /* system address family (Internet in this case) */
   address.sin_family = AF_INET ;

   /* place the port number into the address data structure
      in network byte order */
   address.sin_port = htons(broadcast_port);

   /* place the local address in the address data structure
      in network byte order */
   address.sin_addr.s_addr = htonl(INADDR_ANY);

   /* attempt to open a local socket */
   broadcast_socket = socket(AF_INET,SOCK_DGRAM,0);

   if(broadcast_socket < 0)
      perror("Broadcaster couldn't open a local socket:");
   else
   {
      /* set the broadcast_socket for broadcasting */
      if(setsockopt( broadcast_socket, SOL_SOCKET, SO_BROADCAST,
                  &on, sizeof(on) ) < 0)
         perror("Broadcaster couldn't set socket to broadcast:");

      else if(bind( broadcast_socket, (struct sockaddr *)&address,
                  sizeof(address) ) < 0)
         perror("Broadcaster couldn't bind to local socket:");
      else
      {
         printf("Waiting to broadcast\n");
      }
   }
   /* return the socket number */
   return(broadcast_socket);

} /* start_broadcast */
```

netV.c

```
/*********************************************************************

The broadcast_receive(broadcaster_name,port_number) function performs
all the actions required to set up a broadcast receiving socket

*********************************************************************/


int broadcast_receive(broadcaster_name,port_number)

char broadcaster_name[];    /* name of the broadcaster system */

int  port_number;           /* port number for the broadcaster */

{
    int local_socket;       /* local socket number */

    int socket();           /* function that opens a socket */

    int broadcaster_socket; /* socket number on broadcaster system  */

    /* the protocol and address data structure specified for the socket */
    static struct sockaddr_in address = { AF_INET };

    struct hostent *broadcaster_address; /* address of broadcaster system */

    short broadcaster_port;             /* port number of remote system */


    /* create socket structure from input parameters */


    /* convert the broadcaster system name to its address.
       Note that gethostbyname() requires a pointer only in this case */
    broadcaster_address = gethostbyname(broadcaster_name);

    /* clear out the address structure */
    bzero((char *)&address, sizeof(address));

    /* copy the broadcaster address structure into the address structure */
    bcopy(broadcaster_address->h_addr,
          (char *)&address.sin_addr,
          broadcaster_address->h_length);

    /* set broadcaster port number above the system reserved ports by adding
       the user's broadcaster port number to the number of reserved ports */
    broadcaster_port = IPPORT_RESERVED + port_number;

    /*  broadcaster system address family(Internet in this case) */
    address.sin_family = AF_INET;

    /* place the broadcaster port number into the address structure
       in network byte order */
    address.sin_port = htons(broadcaster_port);


    /* attempt to obtain a local socket */
    local_socket = socket(AF_INET, SOCK_DGRAM, 0);

    if(local_socket < 0)
    {
        perror("Receiver couldn't open a local socket:");
    }
    else
    {
        /* attempt to connect local socket to broadcaster socket */
        broadcaster_socket = connect(local_socket, (struct sockaddr *)&address,
                                     sizeof(address));
```

```
        if(broadcaster_socket < 0)
        {
            /* error occurred in attempting to insert broadcaster information */
            perror("Receiver couldn't find broadcaster:");

            shutdown(local_socket, 2);
            close(local_socket);

            /* set local_socket so that negative value is
               always returned when an error occurs
            */
            local_socket = broadcaster_socket;
        }
        else
            /* successfully listening to the broadcaster system */
            printf("ready to receive from %s.\n",broadcaster_name);
    }

    /* return the socket number of the local system */
    return(local_socket);

} /* broadcast_receive */
```

4. receive.c

a. Calling Protocols

This program monitors a socket, like a daemon. It is spawned transparently to the user and receives its initialization data through the command line.

b. Code and Description

```
/********************************************************************
*                                                                  *
*  TITLE  : Inter-Computer Communication Package                   *
*                                                                  *
*  MODULE : receive.c                                              *
*                                                                  *
*  VERSION: 3.0                                                    *
*                                                                  *
*  DATE   : 31 May 1988                                            *
*                                                                  *
*  AUTHOR : Theodore H. Barrow                                     *
*                                                                  *
********************************************************************
*                                                                  *
*  HISTORY:                                                        *
*                                                                  *
*     VERSION: 1.0                                                 *
*                                                                  *
*     DATE   : 6 February 1987                                     *
*                                                                  *
*     AUTHOR : Michael J. Zyda                                     *
*                                                                  *
*     DESC.  : Background process to receive messages over link.   *
*                                                                  *
*     VERSION: 2.0                                                 *
*                                                                  *
*     DATE   : 15 December 1987                                    *
*                                                                  *
*     AUTHOR : Theodore H. Barrow                                  *
*                                                                  *
*     DESC.  : Added capability to get sequence number from command line *
*              and use it to get offset into shared memory segment. *
*                                                                  *
*     VERSION: 3.0                                                 *
*                                                                  *
*     DATE   : 31 May 1988                                         *
*                                                                  *
*     AUTHOR : Theodore H. Barrow                                  *
*                                                                  *
*     DESC.  : Added broadcast receive capability                  *
********************************************************************
*                                                                  *
*                    RECORD OF CHANGES                             *
*                                                                  *
*Version* Date  * Author              *          * Affected *Reqd* 
*        *      *    Change Description              * Modules  *Vers*
********************************************************************
*        *      *      *                    *        *     *    *
*        *      *      *                    *        *     *    *
********************************************************************/
```

receive.c

```c
#include "shared.h"
#include "gl.h"

main(argc,argv)

int argc;    /* argument count */

char *argv[];    /* pointers to the passed in arguments */

{
    /* we need to declare character variables for everything passed in */

    char shmidstr[10];    /* shared segment string holding the integer key*/

    int shmid;               /* integer pulled out of the string */

    char *segment;           /* character pointer to the shared segment */

    int receivesem;          /* receive semaphore */

    char *sharedsegment();/* create shared segment function */

    char mname[100];         /* machine name */

    char portstr[10];        /* port number string */

    long portnum;            /* port number pulled from the string */

    char server[10];         /* server string */

    char seqnostr[10];       /* sequence # string holding integer sequence # */

    long sequencenum = 0;  /* integer pulled out of the string (default 0) */

    int socket;              /* the opened socket descriptor */

    int connect_server();

    int connect_client();

    int broadcast_receive();

    int receiver_is_free();

    int receiver_should_die();

    int semtran();          /* semaphore creation routine. */

    /* pull out the strings from the argument list */
    if(argc < 5)
    {
        printf("RECEIVE: incorrect argument count!\n");
        exit(1);
    }

    /* pull out the shared memory string */
    strcpy(shmidstr,argv[1]);
    sscanf(shmidstr,"%d",&shmid);

    /* pull out the machinename string */
    strcpy(mname,argv[2]);

    /* pull out the port number string */
    strcpy(portstr,argv[3]);
    sscanf(portstr,"%d",&portnum);
```

104

receive.c

```c
/* create the receive semaphore */
receivesem = semtran(portnum);

/* pull out the client/server string */
strcpy(server,argv[4]);

/* pull out the sequence number string */
if( argc > 4 )
{
    strcpy(seqnostr,argv[5]);
    sscanf(seqnostr,"%d",&sequencenum);
}

/* attach to the shared memory segment */
if((int)(segment = (char *)shmat(shmid, 0, 0666)) < 0)
{
    perror("RECEIVE:shmat");
    exit(0);
}

/* create the shared segment address to use */
segment += sequencenum * MAXSHAREDSIZE;

/* open the socket connection to the named machine */
if(strcmp(server,"server") == 0)
{
    /* we should open as the server */
    socket = connect_server(mname,portnum);
}
else if(strcmp(server,"receive") == 0)
{
    /* we should open as the broadcast receiver*/
    socket = broadcast_receive(mname,portnum);
}
else
{
    /* we should open as a client */
    socket = connect_client(mname,portnum);
}

/* check to make sure socket was opened, exit if not */
if(socket < 0)
{
    printf("RECEIVE: socket connection NOT made!\n");
    exit(1);
}


/* the infinite loop... */

if(strcmp(server,"receive") == 0)
while(TRUE)
{
    /* should the receiver die??? */
    if(receiver_should_die(segment,receivesem))
    {
        /* exit after detaching shared segment and cleaning up socket */
        detachsharedsegment(segment);
        shutdown(socket, 0);
        close(socket);
        exit(0);
    }

    /* if the receiver part of the segment is free, read onto it */
    if(receiver_is_free(segment))
    {
        /* check socket and read into segment if proper message */
        if(broadcast_into_segment(socket,segment,mname,portnum) > 0)
```

receive.c

```
        {
            /* at this point, sleep until we receive a signal from the
                graphics program that the receiver segment is free, i.e.
                the data has been read out */
            P(receivesem);
        }
    }

}   /* end while true for broadcasting*/
else
    while(TRUE)
    {
        /* should the receiver die??? */
        if(receiver_should_die(segment,receivesem))
        {
            /* exit after detaching shared segment and cleaning up socket */
            detachsharedsegment(segment);
            shutdown(socket, 0);
            close(socket);
            exit(0);
        }

        /* if the receiver part of the segment is free, read onto it */
        if(receiver_is_free(segment))
        {
            /* read socket into segment */
            read_socket_into_segment(socket,segment);
        }

        /* at this point, sleep until we receive a signal from the
            graphics program that the receiver segment is free, i.e.
            the data has been read out */
        P(receivesem);

    }   /* end while true for direct connections*/

}
```

5. semaphore.c

a. Calling Protocols

This module repackages the low-level semaphore calls into a *P* and a *V* semaphore operation. No functions in this module are intended for application programs.

b. Code and Description

```
/*********************************************************************
*                                                                   *
*  TITLE   : Inter-Computer Communication Package                   *
*                                                                   *
*  MODULE  : send.c                                                 *
*                                                                   *
*  VERSION: 1.0                                                     *
*                                                                   *
*  DATE    : 11 February 1987                                       *
*                                                                   *
*  AUTHOR : Michael J. Zyda                                         *
*                                                                   *
*********************************************************************
*                                                                   *
*  HISTORY:                                                         *
*                                                                   *
*     VERSION: 1.0                                                  *
*                                                                   *
*     DATE    : 11 February 1987                                    *
*                                                                   *
*     AUTHOR : Michael J. Zyda                                      *
*                                                                   *
*     DESC.  : Implements P and V semaphore operations for Unix system V. *
*              Based on an example from Advanced Unix Programming.  *
*********************************************************************
*                                                                   *
*                   RECORD OF CHANGES                               *
*                                                                   *
*Version*  Date  *  Author                 *          * Affected   *Reqd*
*       *       *   Change Description      *          * Modules    *Vers*
*********************************************************************
*         *        *                 *          *           *   *
*         *        *                 *          *           *   *
*********************************************************************/
```

semaphore.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semtran(key) /* translate semaphore key to ID  */
int key;
{
    int sid;

    if ((sid = semget((key_t)key,1,0666| IPC_CREAT)) == -1)
        {
          perror("semget");
        }

    return(sid);
}



static void semcall(sid,op) /* call semop  */
int sid;
int op;
{

    struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = op;
    sb.sem_flg = 0;

    if(semop(sid,&sb,1) == -1)
        {
            perror("semop");
        }

}



void P(sid)  /* acquire semaphore  */
int sid;
{
    semcall(sid, -1);
}



void V(sid)  /* release semaphore */
int sid;
{
    semcall(sid, 1);
}
```

6. send.c

a. Calling Protocols

This program monitors a socket, like a daemon. It is spawned transparently to the user and receives its initialization data through the command line.

b. Code and Description

```
/****************************************************************
 *                                                              *
 *  TITLE   : Inter-Computer Communication Package              *
 *                                                              *
 *  MODULE  : send.c                                            *
 *                                                              *
 *  VERSION : 3.0                                               *
 *                                                              *
 *  DATE    : 31 May 1988                                       *
 *                                                              *
 *  AUTHOR  : Theodore H. Barrow                                *
 *                                                              *
 ****************************************************************
 *                                                              *
 *  HISTORY:                                                    *
 *                                                              *
 *      VERSION : 1.0                                           *
 *                                                              *
 *      DATE    : 6 February 1987                               *
 *                                                              *
 *      AUTHOR  : Michael J. Zyda                               *
 *                                                              *
 *      DESC.   : Background process to send messages over link.*
 *                                                              *
 *      VERSION : 2.0                                           *
 *                                                              *
 *      DATE    : 15 December 1987                              *
 *                                                              *
 *      AUTHOR  : Theodore H. Barrow                            *
 *                                                              *
 *      DESC.   : Added capability to get sequence number from command line *
 *                and use it to get offset into shared memory segment.       *
 *                                                              *
 *      VERSION : 3.0                                           *
 *                                                              *
 *      DATE    : 31 May 1988                                   *
 *                                                              *
 *      AUTHOR  : Theodore H. Barrow                            *
 *                                                              *
 *      DESC.   : Added broadcast capability                    *
 ****************************************************************
 *                                                              *
 *              RECORD OF CHANGES                               *
 *                                                              *
 *Version*  Date  *  Author            *          * Affected *Reqd*
 *       *        *  Change Description *          * Modules  *Vers*
 ****************************************************************
 *       *        *  *                  *          *    *    *
 *       *        *  *                  *          *    *    *
 ****************************************************************/
```

109

send.c

```c
#include "shared.h"
#include "gl.h"

main(argc,argv)

int argc;    /* argument count */

char *argv[];    /* pointers to the passed in arguments */

{
    /* we need to declare character variables for everything passed in */

    char shmidstr[10];      /* shared segment string holding the integer shmid */

    int shmid;              /* integer pulled out of the string */

    char *segment;          /* character pointer to the shared segment */

    int sendsem;            /* send semaphore */

    char *sharedsegment();/* create shared segment function */

    char mname[100];        /* machine name */

    char portstr[10];       /* port number string */

    long portnum;           /* port number pulled from the string */

    char server[10];        /* server string */

    char seqnostr[10];      /* sequence # string holding integer sequence # */

    long sequencenum = 0; /* integer pulled out of the string (default 0) */

    int socket;             /* the opened socket descriptor */

    int connect_server();

    int connect_client();

    int start_broadcast();

    int sender_has_data();

    int sender_should_die();

    int semtran();          /* semaphore creation routine. */


    /* pull out the strings from the argument list */
    if(argc < 5)
    {
        printf("SEND: incorrect argument count!\n");
        exit(1);
    }

    /* pull out the shared memory string */
    strcpy(shmidstr,argv[1]);
    sscanf(shmidstr,"%d",&shmid);

    /* pull out the machinename string */
    strcpy(mname,argv[2]);

    /* pull out the port number string */
    strcpy(portstr,argv[3]);
    sscanf(portstr,"%d",&portnum);

    /* create the send semaphore */
```

```c
sendsem = semtran(portnum);

/* pull out the client/server string */
strcpy(server,argv[4]);

/* pull out the sequence number string */
if( argc > 4 )
{
   strcpy(seqnostr,argv[5]);
   sscanf(seqnostr,"%d",&sequencenum);
}

/* attach to the shared memory segment */
if((int)(segment = (char *)shmat(shmid, 0, 0666)) < 0)
{
   perror("SEND:shmat");
   exit(0);
}

/* create the shared segment */
segment += sequencenum * MAXSHAREDSIZE;

/* open the socket connection to the named machine */
if(strcmp(server,"server") == 0)
{
   /* we should open as the server */
   socket = connect_server(mname,portnum);
}
else if( strcmp( server, "broadcast" ) == 0 )
{
   /* we should open as a broadcaster */
   socket = start_broadcast( portnum );
}
else
{
   /* we should open as a client */
   socket = connect_client(mname,portnum);
}


/* check to make sure socket was opened, exit if not */
if(socket < 0)
{
   printf("SEND: socket connection NOT made!\n");
   exit(1);
}


/* the infinite loop... */

if( strcmp( server, "broadcast" ) == 0 )
   while(TRUE)
   {
       /* should the sender die??? */
       if(sender_should_die(segment,sendsem))
       {
          /* exit after detaching segment and cleaning up socket */
          detachsharedsegment(segment);
          shutdown(socket, 1);
          close(socket);
          exit(0);
       }

       /* if there is data in the shared memory segment, ... */
       if(sender_has_data(segment))
       {
          /* write the data in the shared segment onto the socket */
          send_socket_from_segment(socket,portnum,segment);
```

111

send.c

```
        }

        /* at this point, sleep until we receive a signal from the graphics
           program. The signal will indicate that the graphics program
           has put more data into the shared segment.
        */
        P(sendsem);

    }    /* end while true for broadcasting*/
else
    while(TRUE)
    {
        /* should the sender die??? */
        if(sender_should_die(segment,sendsem))
        {
            /* exit after detaching segment and cleaning up socket */
            detachsharedsegment(segment);
            shutdown(socket, 1);
            close(socket);
            exit(0);
        }

        /* if there is data in the shared memory segment, ... */
        if(sender_has_data(segment))
        {
            /* write the data in the shared segment onto the socket */
            write_socket_from_segment(socket,segment);
        }

        /* at this point, sleep until we receive a signal from the graphics
           program. The signal will indicate that the graphics program
           has put more data into the shared segment.
        */
        P(sendsem);

    }    /* end while true for direct connection*/
}
```

112

7. **shared.h**

    a. <u>Calling Protocols</u>

    This module has all the predefined constants and type definitions. It must be included in the application.

shared.h

b. Code and Description

```
/******************************************************************
 *                                                                *
 * TITLE   : Inter-Computer Communication Package                 *
 *                                                                *
 * MODULE  : shared.h                                             *
 *                                                                *
 * VERSION : 4.0                                                  *
 *                                                                *
 * DATE    : 15 December 1987                                     *
 *                                                                *
 * AUTHOR  : Theodore H. Barrow                                   *
 *                                                                *
 ******************************************************************
 *                                                                *
 * HISTORY:                                                       *
 *                                                                *
 *     VERSION: 1.0                                               *
 *                                                                *
 *     DATE   : 6 February 1987                                   *
 *                                                                *
 *     AUTHOR : Michael J. Zyda                                   *
 *                                                                *
 *     DESC.  : Contains all defines and special constants for shared *
 *              memory socket system.                             *
 *                                                                *
 *     VERSION: 2.0                                               *
 *                                                                *
 *     DATE   : 27 May 1987                                       *
 *                                                                *
 *     AUTHOR : Theodore H. Barrow                                *
 *                                                                *
 *     DESC.  : Added a typedef of structure for use by various routines. *
 *              Added message types for high level read/write protocol. *
 *                                                                *
 *     VERSION: 3.0                                               *
 *                                                                *
 *     DATE   : 21 October 1987                                   *
 *                                                                *
 *     AUTHOR : Theodore H. Barrow                                *
 *                                                                *
 *     DESC.  : Changed dependencies of buffer calculation constants so that *
 *              only one need change.  Added additional message types. *
 *                                                                *
 *     VERSION: 4.0                                               *
 *                                                                *
 *     DATE   : 15 December 1987                                  *
 *                                                                *
 *     AUTHOR : Theodore H. Barrow                                *
 *                                                                *
 *     DESC.  : Added field to buffer set so that each link would have its *
 *              own area to handle partial receipt of messages.   *
 ******************************************************************
 *                                                                *
 *                    RECORD OF CHANGES                           *
 *                                                                *
 *Version*  Date  *  Author                   *     * Affected *Reqd*
 *        *       *  Change Description        *     * Modules  *Vers*
 ******************************************************************
 *  4.1   * 4Jan88 * T. H. Barrow             *     *          *    *
 *        *        * Changed pathname to include /usr for IRIS1 *          *    *
 ******************************************************************/
```

shared.h

```
/*
   the following 3 defines are the changeable parameters

           LARGESTREAD MUST be divisible by 4
*/

#define SENDLOCATION "/usr/work/barrow/share3/send" /* the name of the program
                                                  to run for the sender */

#define RECEIVELOCATION "/usr/work/barrow/share3/receive" /* the name of program
                                                  to run for the receiver */

#define LARGESTREAD 252        /* the largest read (i.e. buffer size) */


/* The following defines are constants or are derived from LARGESTREAD */

#define SENDEROFFSET (LARGESTREAD + 4)    /* the sender data starts here */

#define WSENDEROFFSET (SENDEROFFSET / 4) /* long word offset for sender data */

#define RECEIVEROFFSET 0        /* the receiver data starts at byte 0 */

#define WRECEIVEROFFSET 0       /* the receiver data starts at long word 0 */

#define PROTOCOLHOLDOFFSET (SENDEROFFSET * 2)    /* holding area starts after
                          sender area */
#define MAXSHAREDSIZE (PROTOCOLHOLDOFFSET + 12) /* the number of bytes in the
                                                 shared segment */

#define CHARACTER_TYPE          'B'  /* code for characters */
#define INTEGER_TYPE            'I'  /* code for integers */
#define FLOAT_TYPE             'R'  /* code for floats */
#define CHARACTER_ARRAY_TYPE   'C'  /* code for character arrays */
#define INTEGER_ARRAY_TYPE     'J'  /* code for integer arrays */
#define FLOAT_ARRAY_TYPE       'S'  /* code for float arrays */

#define CHARACTER_SIZE  1           /* character size in bytes */
#define INTEGER_SIZE    sizeof(1)   /* integer size in bytes */
#define FLOAT_SIZE      sizeof(1.0) /* float size in bytes */

/* the following is the structure type definition needed for each machine
   you want to communicate to...
*/

    typedef struct {
                    char *segment;   /* ptr to shared memory segment */

                    int shmid;       /* system generated shared mem. id */

                    int sendsem;     /* semaphore used to wakeup the sender
                                        process.
                                     */

                    int receivesem;  /* semaphore used to wakeup the
                                        receiver process...
                                     */
                    } Machine ;
```

115

8. shareseg.c

a. Calling Protocols

This module contains the low-level shared-memory calls. No functions in this module are intended for application programs.

b. Code and Description

```
/*******************************************************************
 *                                                                 *
 *  TITLE   : Inter-Computer Communication Package                 *
 *                                                                 *
 *  MODULE  : shareseg.c                                           *
 *                                                                 *
 *  VERSION : 3.1                                                  *
 *                                                                 *
 *  DATE    : 24 February 1988                                     *
 *                                                                 *
 *  AUTHOR  : Theodore H. Barrow                                   *
 *                                                                 *
 *******************************************************************
 *                                                                 *
 *  HISTORY:                                                       *
 *                                                                 *
 *     VERSION: 1.0                                                *
 *                                                                 *
 *     DATE   : 6 February 1987                                    *
 *                                                                 *
 *     AUTHOR : Michael J. Zyda                                    *
 *                                                                 *
 *     DESC.  : Contains routines to manage shared memory segment. Creation *
 *              attachment, detachment and deletion are all covered. *
 *                                                                 *
 *     VERSION: 2.0                                                *
 *                                                                 *
 *     DATE   : 21 October 1987                                    *
 *                                                                 *
 *     AUTHOR : Theodore H. Barrow                                 *
 *                                                                 *
 *     DESC.  : Added function dynamicsharedsegment to allow dynamic memory *
 *              allocation after communications link established.   *
 *                                                                 *
 *     VERSION: 3.0                                                *
 *                                                                 *
 *     DATE   : 15 December 1987                                   *
 *                                                                 *
 *     AUTHOR : Theodore H. Barrow                                 *
 *                                                                 *
 *     DESC.  : Modified function dynamicsharedsegment for use with multiple *
 *              links.  First call does shared segment creation.  Subsequent *
 *              calls return address for the next buffer set.       *
 *******************************************************************
 *                                                                 *
 *                     RECORD OF CHANGES                           *
 *                                                                 *
 *Version*  Date  *  Author                  *         * Affected *Reqd* 
 *       *        *   Change Description      *         * Modules  *Vers* 
 *******************************************************************
 *  3.1  * 24Feb88* T. H. Barrow             *         * none     *    * 
 *       *        * Added compatibility for IRIS 4D.   *         *    * 
 *******************************************************************/
```

116

shareseg.c

```c
#include <sys/sysmacros.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <gl.h>

/* The following defines will have to be modified for different machines
   but one of the underlying shared memory attachment mechanisms should
   work for any system V implementation. */
#define IRIS4D    1
#define IRIS3000 2
#ifdef FLAT
#define MACHINE    IRIS4D
#else
#define MACHINE    IRIS3000
#endif

char *sharedsegment(key,nbytes,shmid)

long key;     /* the key to use for the segment */

long nbytes; /* the number of bytes in the segment */

int *shmid;  /* returned shared memory id name */

{
    char *buf;                    /* temp char pointer */

    struct shmid_ds junkbuf;    /* 1 don't care what's in this buffer */

    /* allocate a shared memory segment */
    if( (*shmid = shmget( key, nbytes, 0666 | IPC_CREAT )) < 0 )
    {
        perror("shmget");
        exit(0);
    }

    /* attach to the shared memory segment */
    if((int)(buf = (char *)shmat(*shmid, 0, 0666)) < 0)
    {
        perror("shmat");

        /* Since there was an attachment error, delete the segment */
        if( shmctl( shmid, IPC_RMID, &junkbuf ) == -1 )
            perror( "shmctl" );
        exit(0);
    }

    /* return the pointer to the shared segment */
    return(buf);

}
```

117

shareseg.c

```c
char *attach_within_datasegment( key, size, shmid, freespace )

long key;          /* the key to use for the segment */

long size;         /* the number of bytes in the segment */

int *shmid;        /* returned shared memory id name */

int freespace;     /* amount freespace desired for dynamic allocation */

{
   char *enddata, *buf;       /* temporary address pointers */

   struct shmid_ds junkbuf;   /* I don't care what's in this buffer */

   char *sbrk(), *malloc();

   /* allocate a shared memory segment */
   if( (*shmid = shmget(key, size, 0666 | IPC_CREAT)) < 0 )
   {
      perror("shmget");
      exit(0);
   }

   /* Ensure at least as much unallocated space as freespace indicates.
      Normally the top of the data region is incremented more than the
      minimum required to meet the malloc() request.  Using malloc()
      and free() ensures that this mechanism is available for subsequent
      dynamic memory allocations.  Direct use of sbrk() system call
      causes the malloc() mechanism to fail on subsequent allocation
      requests.  freespace is cast to unsigned to meet malloc() spec. */
   free( malloc( (unsigned)freespace ));

   /* find the top of data region */
   enddata = sbrk(0);

   /* round up to the next page boundary for attachment of shared
      memory segment */
   buf = (char *)((int)enddata - ((int)enddata % SHMLBA) + SHMLBA);

   /* reset top of data region to be above shared segment */
   if( brk( buf + size ) < 0 )
   {
      perror("brk");

      /* Since there was an error, delete the segment */
      if( shmctl( shmid, IPC_RMID, &junkbuf ) == -1 )
         perror( "shmctl" );
      exit(-1);
   }

   /* attach to the shared memory segment at the calculated address */
   if( (int)shmat(*shmid, buf, 0666) < 0 )
   {
      perror("shmat");

      /* Since there was an attachment error, delete the segment */
      if( shmctl( shmid, IPC_RMID, &junkbuf ) == -1 )
         perror( "shmctl" );
      exit(0);
   }

   return( buf );

}  /* attach_within_datasegment() */
```

118

shareseg.c

```c
char *dynamicsharedsegment(nummachines, key, nbytes, shmid, freespace)

int nummachines;  /* maximum number of machines to be initiated */

long key;         /* the key to use for the segment */

long nbytes;      /* the number of bytes in the segment */

int *shmid;       /* returned shared memory id name */

int freespace;    /* amount freespace desired for dynamic allocation */

{
   static Boolean firsttime = TRUE; /* allows for multiple calls */

   static char *startshared; /* start of shared memory space */

   static int *holdshmid;    /* holds shmid for subsequent calls */

   if( firsttime )
   {
      switch( MACHINE )
      {
         case IRIS4D:
            startshared = sharedsegment( key, nummachines*nbytes, shmid );
            break;
         case IRIS3000:
            startshared = (char *)attach_within_datasegment( key,
                                      nummachines*nbytes, shmid, freespace );
            break;
         default:
            perror( "shareseg: Unknown machine" );

      }  /* switch( MACHINE ) */

      holdshmid = shmid;
      firsttime = FALSE;

   }
   else
   {
      /* start next buffer immediately above last.  Return the same shmid
         for all buffers.  Assumes all buffers are same size (true if all
         from same shared.h definition. */
      startshared += nbytes;

      *shmid = *holdshmid;
   }
   /* return pointer to the proper buffer in the shared segment */
   return( startshared );

}
```

119

shareseg.c

```
detachsharedsegment(segment)

char *segment;    /* segment to detach from */

{
   int returnvalue;

   if( (int)segment % SHMLBA != 0 )
      return( 1 );
   else
   {
      if( returnvalue = shmdt(segment) < 0 )
         perror("shmdt");
      return( returnvalue );
   }
}




deletesharedsegment(segment,shmid)

char *segment;    /* character pointer to the shared segment */

int shmid;        /* shared memory id... */

{
   int returnvalue;

   struct shmid_ds junkbuf;    /* I don't care what's in this buffer */

   /* detach from the shared segment and set returnvalue */
   if( returnvalue = detachsharedsegment(segment) == 0 )

      /* remove the shared segment from the system and reset returnvalue */
      if( returnvalue = shmctl(shmid, IPC_RMID, &junkbuf) < 0 )
         perror("shmctl");

   return(returnvalue);

}
```

9. **support.c**

    a. <u>Calling Protocols</u>

    This module contains functions that are intended for the application's use and functions that are used exclusively by other routines. The parameters for externally accessible functions are described below.

    i. *receiver_has_data*

```
int receiver_has_data(instructure)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment */
```

    ii. *sender_is-free*

```
int sender_is_free(instructure)

    Machine *instructure;  /* includes
            char *instructure.segment    a pointer to the shared segment */
```

b. Code and Description

```
/*******************************************************************
 *                                                                 *
 *  TITLE   : Inter-Computer Communication Package                 *
 *                                                                 *
 *  MODULE  : support.c                                            *
 *                                                                 *
 *  VERSION: 4.0                                                   *
 *                                                                 *
 *  DATE    : 31 May 1988                                          *
 *                                                                 *
 *  AUTHOR  : Theodore H. Barrow                                   *
 *                                                                 *
 *******************************************************************
 *                                                                 *
 *  HISTORY:                                                       *
 *                                                                 *
 *          VERSION: 1.0                                           *
 *                                                                 *
 *          DATE    : 6 February 1987                              *
 *                                                                 *
 *          AUTHOR  : Michael J. Zyda                              *
 *                                                                 *
 *          DESC.   : Contains support routines for shared memory communications *
 *                    system.                                      *
 *                                                                 *
 *          VERSION: 2.0                                           *
 *                                                                 *
 *          DATE    : 27 May 1987                                  *
 *                                                                 *
 *          AUTHOR  : Theodore H. Barrow                           *
 *                                                                 *
 *          DESC.   : Converted functions called by the application program to use *
 *                    a structure for ease of use.                 *
 *                                                                 *
 *          VERSION: 3.0                                           *
 *                                                                 *
 *          DATE    : 21 October 1987                              *
 *                                                                 *
 *          AUTHOR  : Theodore H. Barrow                           *
 *                                                                 *
 *          DESC.   : Removed functions for reading from and writing to the shared *
 *                    memory segment by the application program.   *
 *                                                                 *
 *          VERSION: 4.0                                           *
 *                                                                 *
 *          DATE    : 31 May 1988                                  *
 *                                                                 *
 *          AUTHOR  : Theodore H. Barrow                           *
 *                                                                 *
 *          DESC.   : Added functions broadcast_into_segment and   *
 *                    send_socket_from_segment for broadcasting over datagram socket*
 *******************************************************************
 *                                                                 *
 *                       RECORD OF CHANGES                         *
 *                                                                 *
 *Version*  Date  *  Author                    *          * Affected  *Reqd*
 *          *        Change Description                    * Modules   *Vers*
 *******************************************************************
 *         *         *                          *          *           *    *
 *         *         *                          *          *           *    *
 *******************************************************************/
```

122

support.c

```c
#include "shared.h"
#include <gl.h>
#include <bsd/sys/types.h>
#include <sys/socket.h>
#include <bsd/netinet/in.h>
#include <bsd/netdb.h>

/* the following routine sets up buffer area */

init_shared_buffer(segment)

char *segment;   /* pointer to the shared segment */

{
   free_sender( segment );
   free_receiver( segment );
   *(segment + PROTOCOLHOLDOFFSET + 9) = '\0';

}


/* the following routine writes zeroes at the top of the
   shared segment indicating that the segment data is no longer
   valid.
*/

free_sender(segment)

char *segment;   /* pointer to the shared segment */

{
   /* the following line zeroes the first four bytes of the sender part
      of the shared memory segment. 'segment' is a character pointer.
      I coerce it into a long integer pointer and then write a zero.
   */
   *((long *)segment + WSENDEROFFSET) = 0;

}


/* this following routine writes zeroes at the top of the
   shared segment indicating that the segment data is no longer
   valid.
*/

free_receiver(segment)

char *segment;   /* pointer to the shared segment */

{
   /* the following line zeroes the first four bytes of the receiver part
      of the shared memory segment. 'segment' is a character pointer.
      I coerce it into a long integer pointer and then write a zero.
   */
   *((long *)segment + WRECEIVEROFFSET) = 0;

}
```

support.c

```c
/* the following routine tests the first 4 bytes of the receiver
   segment to see if they are non-zero.
   it uses an input structure since called by main program
*/

int receiver_has_data(instructure)

Machine *instructure;  /* includes

     char *instructure.segment    a pointer to the shared segment */

{

   if(*((long *)instructure->segment + WRECEIVEROFFSET) > 0)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}




/* the following routine tests the first 4 bytes of the sender
   segment to see if they are non-zero.
*/

int sender_has_data(segment)

char *segment;    /* pointer to the shared segment */

{

   if(*((long *)segment + WSENDEROFFSET) > 0)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}
```

support.c

```c
/* the following routine tests the first 4 bytes of the receiver
   segment to see if they are less than zero.
*/

int receiver_should_die(segment)

char *segment;   /* pointer to the shared segment */

{

   if(*((long *)segment + WRECEIVEROFFSET) < 0)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}



/* the following routine tests the first 4 bytes of the sender
   segment to see if they are less than zero.
*/

int sender_should_die(segment)

char *segment;   /* pointer to the shared segment */

{

   if(*((long *)segment + WSENDEROFFSET) < 0)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}
```

125

support.c

```c
/* the following routine tests the first 4 bytes of the receiver
   segment to see if they are non-zero.
*/

int receiver_is_free(segment)

char *segment;   /* pointer to the shared segment */

{
   if(*((long *)segment + WRECEIVEROFFSET) == 0)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}




/* the following routine tests the first 4 bytes of the sender
   segment to see if they are non-zero.
   it uses an input structure since called by main program
*/

int sender_is_free(instructure)

Machine *instructure;   /* includes

      char *instructure.segment   a pointer to the shared segment */

{
   if(*((long *)instructure->segment + WSENDEROFFSET) == 0)
   {
      return(TRUE);
   }
   else
   {
      return(FALSE);
   }
}
```

126

support.c

```c
/* the following routine reads on the input socket into the receiver segment.*/

read_socket_into_segment(socket,segment)

int socket;   /* a socket descriptor */

char *segment; /* a ptr to the shared segment */

{

    long nbytes;   /* the number of bytes read in */

    char temp[LARGESTREAD];


    /* read the data into a temporary array to avoid segment protection
        violation since the socket does not share with the shared memory
        segment.
    */
    nbytes = read(socket,temp,LARGESTREAD);

    if(nbytes <= 0)
    {
        /* the following routine calls are commented out for the following
            reason:

            nbytes <= 0 means that the socket has been broken.

            This routine is called by the receiver process so the only
            intelligent thing to do is to terminate the receiver process,
            i.e. call exit...

            perror("read");
            printf("READ_SOCKET_INTO_SEGMENT: number of bytes read = %d\n",nbytes);
        */
        shutdown( socket, 2 );
        close( socket );
        exit(1);
    }

    /* copy the data into the shared segment */
    memcpy((segment + RECEIVEROFFSET + 4),temp,nbytes);

    /* set the number of bytes in the shared segment */
    *((long *)segment + WRECEIVEROFFSET) = nbytes;

}
```

support.c

```c
/* the following routine writes the data from the sender side
   of the shared segment to the socket */


write_socket_from_segment(socket,segment)

int socket;    /* socket descriptor */

char *segment;    /* pointer to the shared segment */

{

    long nbytes;    /* the number of bytes to write */

    char temp[LARGESTREAD];

    /* copy the data into a temporary array to avoid segment protection
       violation since the socket does not share with the shared memory
       segment.
    */
    memcpy(temp,((char *)segment + SENDEROFFSET + 4),
                    *((long *)segment + WSENDEROFFSET));

    /* write the data to the socket */
    nbytes = write(socket,temp, *((long *)segment + WSENDEROFFSET));

    if(nbytes <= 0 || nbytes != *((long *)segment + WSENDEROFFSET))
    {
        /*
        This error indicates the socket is broken. Just exit the
        sender process.

        perror("write");
        printf("WRITE_SOCKET_FROM_SEGMENT: number of bytes written = %d\n",nbytes);
        printf("Number of bytes in shared segment = %d\n",*((long *)segment + WSENDEROFFSET));
        */
        shutdown( socket, 2 );
        close( socket );
        exit(1);
    }

    /* free the sender segment */
    free_sender(segment);

}
```

support.c

```c
/* The following routine receives on the input datagram socket.
   If the message matches the mname and portnum it is copied into the
   receiver area of the shared memory segment.
   0 is returned if the message does not match mname and portnum,
   the number of bytes read is returned if it does match. */

int broadcast_into_segment(socket,segment,mname,portnum)

int socket;     /* a socket descriptor */

char *segment;  /* a ptr to the shared segment */

char mname[];   /* machine name of broadcaster */

long portnum;   /* port number of broadcaster */

{

    long nbytes;                    /* the number of bytes read in */

    char temp[LARGESTREAD];

    int flags = 0;                  /* flags = 0 indicates none set */

    struct sockaddr_in who; /* Internet structure for message sender address */

    int wholen;                     /* length of received address struct who */

    struct hostent *broadcaster;    /* pointer to structure with info on
                                        broadcaster */

    static long broadcast_address; /* address of broadcaster */

    static short broadcast_port;    /* port of broadcaster */

    static Boolean firsttime = TRUE;


    /* read the data into a temporary array to avoid segment protection
       violation since the socket does not share with the shared memory
       segment. This also allows checking for match with desired broadcaster.
    */
    nbytes = recvfrom( socket, temp, LARGESTREAD, flags,
                        (struct sockaddr *)&who, &wholen );

    if(nbytes <= 0)
    {
       perror("recvfrom:");
    }
    else
    {
       if( firsttime )
       {
          /* determine desired broadcaster address and port */

          broadcast_port = htons((short)portnum);

          broadcaster    = (struct hostent *)gethostbyname( mname );

          bcopy( broadcaster->h_addr, (char *)&broadcast_address,
                 broadcaster->h_length );
       }


       if( (broadcast_address == who.sin_addr.s_addr) &&
           (broadcast_port    == who.sin_port) )
       {
```

129

support.c

```
        /* copy the data into the shared segment */
        memcpy((segment + RECEIVEROFFSET + 4),temp,nbytes);

        /* set the number of bytes in the shared segment */
        *((long *)segment + WRECEIVEROFFSET) = nbytes;

    }
    else
    {
        nbytes = 0;
        /* Set nbytes to 0 so return of function indicates no match */
    }
}

return( nbytes );
}
```

support.c

```c
/* the following routine sends the data from the sender side
   of the shared segment to the socket for broadcast */


send_socket_from_segment(socket,portnum,segment)

int socket;              /* socket descriptor */

long portnum;     /* port number of broadcaster */

char *segment;    /* pointer to the shared segment */

{

    long nbytes;    /* the number of bytes to write */

    char temp[LARGESTREAD];

    short broadcaster_port;

    static Boolean firsttime = TRUE;

    static struct sockaddr_in network = { AF_INET }; /* structure for broadcast
                                            address */

    if( firsttime )
    {
        broadcaster_port = IPPORT_RESERVED + portnum;
        /* Set up broadcasting address structure */
        network.sin_family       = AF_INET;
        network.sin_addr.s_addr  = htonl(INADDR_BROADCAST);
        network.sin_port         = htons(broadcaster_port);
        firsttime                = FALSE;
    }

    /* copy the data into a temporary array to avoid segment protection
       violation since the socket does not share with the shared memory
       segment.
    */
    memcpy(temp,((char *)segment + SENDEROFFSET + 4),
                    *((long *)segment + WSENDEROFFSET));

    /* broadcast the data through the socket */
    nbytes = sendto( socket, temp, *((long *)segment + WSENDEROFFSET), 0,
                        (struct sockaddr *)&network, sizeof(network) );

    if(nbytes <= 0 || nbytes != *((long *)segment + WSENDEROFFSET))
    {
        /*
        This error indicates the socket is broken. Just exit the
        sender process.
        */

        perror("write");
        printf("WRITE_SOCKET_FROM_SEGMENT: number of bytes written = %d\n",nbytes);
        printf("Number of bytes in shared segment = %d\n",*((long *)segment + WSENDEROFFSET));
        shutdown( socket, 2 );
        close( socket );
        exit(1);
    }

    /* free the sender segment */
    free_sender(segment);

}
```

131

support.c

```c
/* the following routine deletes the sender by writing
   a negative byte count into the shared segment
   and then waking up the sender.
*/

kill_sender(segment,sendsem)

char *segment;    /* ptr to the segment */

int sendsem;            /* semaphore to the sender */

{

    /* write a negative number into the byte count field. */
    *((long *)segment + WSENDEROFFSET) = -1;

    /* at this point, we should send a wakeup to the sender program.
       the sender will read the bad byte count and exit.
    */
    V(sendsem);

}




/* the following routine deletes the receiver by writing
   a negative byte count into the shared segment
   and then waking up the receiver.
*/

kill_receiver(segment,receivesem)

char *segment;    /* ptr to the segment */

int receivesem;           /* semaphore to the receiver */

{

    /* we do not wait until the receiver segment is free here
       as the process that calls this routine should already
       have read the last piece of data.
    */

    /* write a negative number into the byte count field. */
    *((long *)segment + WRECEIVEROFFSET) = -1;

    /* at this point, we should send a wakeup to the receiver program.
       the receiver will read the bad byte count and exit.
    */
    V(receivesem);

}
```

APPENDIX B - TI EXPLORER MODULE DESCRIPTIONS

All functions, methods, and flavor are contained in file **irisflavor.lisp**.

1. Calling Protocols

The module contains functions, methods, and a flavor that are intended for the application's use. It also contains a macro and functions that are used internally. The parameters for externally accessible functions and methods are described below.

    a.   *iris*

```
(defun iris (x)        ;where x is number of iris machine desired
```

    b.   *start-iris*

```
(defmethod (conversation-with-iris :start-iris)
           ()
```

    c.   *get-iris*

```
(defmethod (conversation-with-iris :get-iris)
           ()
```

    d.   *put-iris*

```
(defmethod (conversation-with-iris :put-iris)
           (object)
  (let* ((buffer (cond
        ((equal (type-of object) 'bignum) (convert-number-to-string object))
          ((equal (type-of object) 'fixnum) (convert-number-to-string object))
          ((equal (type-of object) 'float) (convert-number-to-string object))
          ((equal (type-of object) 'string) object)
          (t "error") ))
```

    e.   *stop-iris*

```
(defmethod (conversation-with-iris :stop-iris)
           ()
```

    f.   *reuse-iris*

```
(defmethod (conversation-with-iris :reuse-iris)
           ()
```

2. Code and Description

```lisp
(defmacro loopfor (var init test exp1 &optional exp2 exp3 exp4 exp5)
  '(prog ()
      (setq ,var ,init)
      tag
      ,exp1
      ,exp2
      ,exp3
      ,exp4
      ,exp5
      (setq ,var (1+ ,var))
      (if (= ,var ,test) (return t) (go tag)) ) )

(defun convert-number-to-string (n)
  (princ-to-string n) )

(defun convert-string-to-integer (str &optional (radix 10))
  (do ((j 0 (+ j 1))
       (n 0 (+ (* n radix) (digit-char-p (char str j) radix))) )
      ((= j (length str)) n) ) )

(defun find-period-index (str)
  (catch 'exit
    (dotimes (x (length str) nil)
      (if (equal (char str x) (char "." 0))
        (throw 'exit x) ) ) ) )

(defun get-leftside-of-real (str &optional (radix 10))
  (do ((j 0 (1+ j))
       (n 0 (+ (* n radix) (digit-char-p (char str j) radix))) )
      ((or (null (digit-char-p (char str j) radix)) (= j (length str)))  n) ) )

(defun get-rightside-of-real (str &optional (radix 10))
  (do ((index (1+ (find-period-index str)) (1+ index))
       (factor 0.10 (* factor 0.10))
       (n 0.0 (+ n (* factor (digit-char-p (char str index) radix)))) )
      ((= index (length str)) n) ) )

(defun convert-string-to-real (str &optional (radix 10))
  (+ (float (get-leftside-of-real str radix)) (get-rightside-of-real str radix)) )


(defvar *tcp-handler1* (send ip::*tcp-handler* :get-port))
(defvar *tcp-handler2* (send ip::*tcp-handler* :get-port))

(defvar *iris1-port1* 1027)                     ; this is the send port
(defvar *iris1-port2* 1026)                     ; this is the receive port

(defvar *iris1-address* 3221866502)
(defvar *iris2-address* 3221866504)
(defvar *iris3-address* 3221866505)

(defvar *dest-address*  nil)                    ; the tcp-ip or internet address
                                                ; look in network configuration

(defun iris (x)
  (cond ((equal x 1) (setq *dest-address* *iris1-address*))
        ((equal x 3) (setq *dest-address* *iris3-address*))
        (t           (setq *dest-address* *iris2-address*)) ) )


(defflavor conversation-with-iris ((talking-port-number    *iris1-port1*)
                                   (listening-port-number  *iris1-port2*)
                                   (talking-port            *tcp-handler1*)
                                   (listening-port          *tcp-handler2*)
                            (destination          *dest-address*) )
```

134

```
                        ()
                                :gettable-instance-variables
                  :settable-instance-variables
                  :initable-instance-variables )

(defmethod (conversation-with-iris :start-iris)
          ()
  (progn
    (send talking-port :open
         :active                       ; tcp will begin the procedure to establish
                                       ; connection  (default  vs :passive)
         talking-port-number           ; port number of destination host
         destination                   ; machine name or address if blank and
                                       ; in :passive mode local machine waits for
                                       ; connection
         30 )                          ; set max seconds before read request times out
    (send listening-port :open
         :active                                          ; :passive
         listening-port-number
         destination
         30 )
    '"A conversation with the iris machine has been established" ) )

(defmethod (conversation-with-iris :reuse-iris)
          ()
  (setq *tcp-handler1* (send ip::*tcp-handler* :get-port)
     *tcp-handler2* (send ip::*tcp-handler* :get-port)
     talking-port   *tcp-handler1*
     listening-port *tcp-handler2* ) )

(defmethod (conversation-with-iris :get-iris)
          ()
  (let* ((typebuffer    " ")
        (lengthbuffer "    ")
         (buffer        " ")
        (buffer-length 1) )
        (progn
            (send listening-port :receive
               typebuffer
               buffer-length
               30
               :wait )
          (send listening-port :receive
               lengthbuffer
               4
               30
               :wait )
        (setq buffer-length (convert-string-to-integer lengthbuffer))
        (setq buffer (make-string buffer-length :initial-element (character 32)))
        (send listening-port :receive
               buffer
               buffer-length
               30
               :wait )
        (cond ((equal typebuffer "I") (convert-string-to-integer buffer))
              ((equal typebuffer "R") (convert-string-to-real   buffer))
              ((equal typebuffer "C") buffer)
              (t nil) ) ) ) )

(defmethod (conversation-with-iris :put-iris)
          (object)
  (let* ((buffer (cond
          ((equal (type-of object) 'bignum) (convert-number-to-string object))
              ((equal (type-of object) 'fixnum) (convert-number-to-string object))
          ((equal (type-of object) 'float) (convert-number-to-string object))
          ((equal (type-of object) 'string) object)
        (t "error") ))
      (buffer-length  (length buffer))
```

135

```
(typebuffer        (cond ((equal (type-of object) 'bignum) "I")
                        ((equal (type-of object) 'fixnum) "I")
                        ((equal (type-of object) 'float) "R")
                        ((equal (type-of object) 'string) "C")
                        (t "C") ))
(lengthbuffer     (convert-number-to-string buffer-length))
(*loopvariable* 0) )
    (progn
      (send talking-port :send
        typebuffer
        1
        nil
        nil )
    (if (= (length lengthbuffer) 4)
        (send talking-port :send
            lengthbuffer
            4
            nil
            nil )
        (progn
        (loopfor *loopvariable* (length lengthbuffer) 4
            (send talking-port :send "0" 1 nil nil) )
        (send talking-port :send lengthbuffer (length lengthbuffer) nil nil) ) )
    (send talking-port :send
        buffer
        buffer-length
        t
        nil ) ) ) )

(defmethod (conversation-with-iris :stop-iris)
        ()
  (progn (send talking-port :close) (send listening-port :close)) )
```

136

APPENDIX C - SYMBOLICS MODULE DESCRIPTIONS

All functions, methods, and flavor are contained in file **irisflavor.lisp**.

1. Calling Protocols

The module contains functions, methods, and a flavor that are intended for the application's use. It also contains a macro and functions that are used internally. The parameters for externally accessible functions and methods are described below.

    a. *select-host*

```
(defun select-host (host-name)
```

    b. *start-iris*

```
(defmethod (:start-iris conversation-with-iris)
           ()
```

    c. *get-iris*

```
(defmethod (:get-iris conversation-with-iris)
           ()
```

    d. *put-iris*

```
(defmethod (:put-iris conversation-with-iris)
           (object)
  (let* ((buffer (cond
        ((equal (type-of object) 'bignum) (convert-number-to-string object))
          ((equal (type-of object) 'fixnum) (convert-number-to-string object))
        ((equal (type-of object) 'single-float) (convert-number-to-string object))
        ((equal (type-of object) 'string) object)
        (t "error") ))
```

    e. *stop-iris*

```
(defmethod (:stop-iris conversation-with-iris)
           ()
```

    f. *reuse-iris*

```
(defmethod (:reuse-iris conversation-with-iris)
           ()
```

137

Symbolics irisflavor.lisp

2. Code and Description

```lisp
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER -*-

; handy macro to have in the send message farthur down

(defmacro loopfor (var init test exp1 &optional exp2 exp3 exp4 exp5)
  '(prog ()
     (setq ,var ,init)
     tag
       ,exp1
       ,exp2
       ,exp3
       ,exp4
       ,exp5
       (setq ,var (1+ ,var))
       (if (= ,var ,test) (return t) (go tag)) ) )


(defun convert-number-to-string (n)
  (princ-to-string n) )

(defun convert-string-to-integer (str &optional (radix 10))
  (do ((j 0 (+ j 1))
       (n 0 (+ (* n radix) (digit-char-p (char str j) radix))) )
      ((= j (length str)) n) ) )

(defun find-period-index (str)
  (catch 'exit
    (dotimes (x (length str) nil)
      (if (equal (char str x) (char "." 0))
          (throw 'exit x) ) ) ) )

(defun get-leftside-of-real (str &optional (radix 10))
  (do  ((j 0 (1+ j))
        (n 0 (+ (* n radix) (digit-char-p (char str j) radix))) )
       ((or (null (digit-char-p (char str j) radix)) (= j (length str)))  n) ) )

(defun get-rightside-of-real (str &optional (radix 10))
  (do ((index (1+ (find-period-index str)) (1+ index))
       (factor 0.10 (* factor 0.10))
       (n 0.0 (+ n (* factor (digit-char-p (char str index) radix)))) )
      ((= index (length str)) n) ) )

(defun convert-string-to-real (str &optional (radix 10))
  (+ (float (get-leftside-of-real str radix)) (get-rightside-of-real str radix)) )


(defvar *iris-port1* 1027)            ; this is the send port
(defvar *iris-port2* 1026)            ; this is the receive port
(defvar *local-talk-port* 1500)       ; this is the local send port
(defvar *local-listen-port* 1501)     ; this is the local receive port


(defflavor conversation-with-iris ((talking-port-number      *iris-port1*)
                     (listening-port-number    *iris-port2*)
                     (local-talk-port-number   *local-talk-port*)
                     (local-listen-port-number *local-listen-port*)
                     (talking-stream)
                     (listening-stream)
                     (destination-host-object) )
                          ()
                  :initable-instance-variables )

(defmethod (:init-destination-host conversation-with-iris)
       (name-of-host)
  (setf destination-host-object (net:parse-host name-of-host)) )
```

138

```lisp
(defmethod (:start-iris conversation-with-iris)
      ()
  (setf talking-stream
     (tcp:open-tcp-stream destination-host-object
                    talking-port-number
                    local-talk-port-number ) )
  (setf listening-stream
     (tcp:open-tcp-stream destination-host-object
                    listening-port-number
                    local-listen-port-number ) )
  "A conversation with the iris machine has been established" )

(defmethod (:reuse-iris conversation-with-iris)
      ()
  )

(defun read-string (stream num-chars)
  (let ((out-string ""))
    (dotimes (i num-chars)
      (setf out-string (string-append out-string (read-char stream))) )
    out-string ) )

(defmethod (:get-iris conversation-with-iris)
      ()
  (let* ((typebuffer    " ")
      (lengthbuffer "    ")
        (buffer        " ")
    (buffer-length 1) )
    (progn
      (setf typebuffer
         (read-string listening-stream 1) )
      (setf lengthbuffer
         (read-string listening-stream 4) )
      (setf buffer-length
         (convert-string-to-integer lengthbuffer) )
      (setf buffer
         (read-string listening-stream buffer-length) )

      (cond ((equal typebuffer "I") (convert-string-to-integer buffer))
         ((equal typebuffer "R") (convert-string-to-real    buffer))
         ((equal typebuffer "C") buffer)
         (t nil) ) ) ) )

(defvar *step-var* 0)

(defun my-write-string(string stream)
  (let* ((num-chars (length string)))
    (dotimes (i num-chars)
      (write-char (aref string i) stream) ) ) )

(defmethod (:put-iris conversation-with-iris)
      (object)
  (let* ((buffer (cond
         ((equal (type-of object) 'bignum) (convert-number-to-string object))
            ((equal (type-of object) 'fixnum) (convert-number-to-string object))
            ((equal (type-of object) 'single-float) (convert-number-to-string object))
            ((equal (type-of object) 'string) object)
         (t "error") ))

      (buffer-length  (length buffer))

      (typebuffer      (cond ((equal (type-of object) 'bignum) "I")
                   ((equal (type-of object) 'fixnum) "I")
                   ((equal (type-of object) 'single-float) "R")
                   ((equal (type-of object) 'string) "C")
                   (t "C") ))

      (lengthbuffer    (convert-number-to-string buffer-length)) )
```

139

```
       (progn
         (my-write-string typebuffer talking-stream)
         (send talking-stream :force-output)
      (if (= (length lengthbuffer) 4)
         (write-string lengthbuffer talking-stream)
         (progn
             (loopfor *step-var* (length lengthbuffer) 4
               (write-string "0" talking-stream) )

             (my-write-string lengthbuffer talking-stream) ) )
      (send talking-stream :force-output)
      (my-write-string buffer talking-stream)
      (send talking-stream :force-output) ) ) )

(defmethod (:stop-iris conversation-with-iris)
      ()
  (progn (send talking-stream :close)
      (send listening-stream :close) ) )


(defun select-host (host-name)
  (send talk :init-destination-host host-name) )
```

APPENDIX D - TEST AND UTILITY PROGRAMS

1. gprog.c

a. Calling Protocols

This is a test program for the direct connect protocol. By command line argument, another machine to receive direct connect messages from can be specified. The default is to receive messages from *iris2*. It must be run in conjunction with **gprog2.c** to function properly, as the port assignments are hardcoded. Since it is the *server* program, it must be started before **gprog2.c**.

b. Code and Description

```
/* this is file gprog.c

    It is a sample top level program for the asynchronous reading
    and writing of sockets via shared memory and two other processes.

    This program spawns off the required processes.

    This program uses structure type Machine declared in file shared.h.

    This is the SERVER side program and runs first!!!.

*/

#include "shared.h"
#include "gl.h"
#include "device.h"


main(argc,argv)

int argc;   /* argument count */
char *argv[];   /* pointers to the passed in arguments */

{
    Machine remotemachine;          /* structure for remote machine */

    char other_machine[50];         /* name of other machine */

    char mybuffer[LARGESTREAD];     /* received data */

    char outgoing[LARGESTREAD];     /* outgoing message's buffer */

    int mybuffer1[LARGESTREAD/INTEGER_SIZE]; /* received integer data */

    int outgoing1[LARGESTREAD/INTEGER_SIZE]; /* outgoing integer message's buffer */

    float mybuffer2[LARGESTREAD/FLOAT_SIZE]; /* received float data */

    float outgoing2[LARGESTREAD/FLOAT_SIZE]; /* outgoing float message buffer */

    long noutgoing;                 /* size of the outgoing message */
```

141

gprog.c

```c
char temp[10];                      /* temp array used to make outgoing message */

long count = 0;                     /* message counter */

char received_type();

char type_received;

int elements_received;

long i;                             /* temp loop variable */

long j = 0;                         /* variable to control message sending */


/* pull out the string from the argument list */
if(argc > 2)
{
    printf("GPROG: incorrect argument count!  use gprog <alias>\n");
    exit(1);
}
/* pull out the name of the other string, if it exists */
if( argc == 2 )
{
    strcpy( other_machine, "npscs-" );
    strcat( other_machine, argv[1] );
}
else
    strcpy( other_machine, "npscs-iris1" );


/* create a path to a particular machine (iris1 default) */
/* the first argument is the key for the shared memory segment.
   the second argument is the name of the machine to connect to.
   the third argument is the sending port number for the socket to use.
   the fourth argument is the receiving port number for the socket to use.
   the fifth argument indicates whether the processes should
       act as a server or a client.
   the sixth argument is the returned pointer to the structure
       remotemachine.
       it includes the pointer to the shared memory segment,
       the system generated shared memory id, the sendsem id,
       and the returned receivesem id.
   the seventh argument is the amount of freespace desired for dynamic
       memory allocation during execution of the program.
*/
dynamicmachinepath(1,other_machine,1,2,"server",&remotemachine,2000000);


/* the loop for polling the shared segment */
while(TRUE)
{
    /* make an outgoing message */
    strcpy(outgoing,"GPROG ORIGINATED MESSAGE: ");

    count = count - 1;

    outgoing1[0] = count;

    noutgoing = strlen(outgoing);

    outgoing2[0] = count;


    /* is there data in the shared segment? */
    if(receiver_has_data(&remotemachine))
    {
        type_received = received_type(&remotemachine);
```

gprog.c

```c
        printf("The message received by GPROG is of type %c \n",
                type_received);

        switch (type_received)
        {
           case CHARACTER_ARRAY_TYPE:
              elements_received = number_received(&remotemachine);

              printf("The message received by GPROG is %d elements long!\n",
                      elements_received);

              read_characters(&remotemachine, mybuffer, elements_received);
              break;

           case INTEGER_TYPE:
              read_integer(&remotemachine,mybuffer1);
              break;

           case FLOAT_TYPE:
              read_float(&remotemachine,mybuffer2);
              break;
        }
        /* at this point in the program, process the received data...*/
        printf("GPROG has received the following data:\n");

        switch (type_received)
        {
           case CHARACTER_ARRAY_TYPE:
              for(i=0; i < elements_received; i+=1)
              {
                 printf("%c",mybuffer[i]);
              }
              break;

           case INTEGER_TYPE:
              printf("%d",mybuffer1[0]);
              break;

           case FLOAT_TYPE:
              printf("%f",mybuffer2[0]);
              break;
        }
        printf("\n");
}

/* at this point, we would look at our system and see if we needed
   to send data. Instead, I will check if the sender is free.
   If the sender is free, I will send one of three messages */
if(sender_is_free(&remotemachine))
{
    if((j % 3) == 0)
       write_characters(&remotemachine,outgoing,noutgoing);

    /* wait until message sent before attempting to send another */
    while( !sender_is_free(&remotemachine) ) /* do nothing */ ;

    if((j % 3) == 1)
       write_integer(&remotemachine,outgoing1);

    /* wait until message sent before attempting to send another */
    while( !sender_is_free(&remotemachine) ) /* do nothing */ ;

    if((j % 3) == 2)
       write_float(&remotemachine,outgoing2);

    ++j;
}
else
```

```
          /* assume socket connection broken */
      printf("Sender wasn't free!  Terminating...\n");
          break;

   }    /* endif while TRUE */

   /* get rid of the path to the other machine...*/
   deletemachinepath(&remotemachine);

}
```

2. gprog2.c

a. Calling Protocols

This is a test program for the direct connect protocol. By command line argument, another machine to receive direct connect messages from can be specified. The default is to receive messages from *iris1*. It must be run in conjunction with **gprog.c** to function properly, as the port assignments are hardcoded. Since it is the *client* program, it be started after **gprog.c** is ready for it.

b. Code and Description

```
/* this is file gprog2.c

    It is a sample top level graphics program for the asynchronous reading
    and writing of sockets via shared memory and two other processes.

    This program spawns off the required processes.

    This program uses structure type Machine declared in file shared.h.

    This is the CLIENT side program and runs second!!!.

*/

#include "shared.h"
#define TRUE 1


main(argc,argv)

int argc;        /* argument count */
char *argv[];    /* pointers to the passed in arguments */

{
    Machine remotemachine;           /* structure for remote machine */

    char other_machine[50];          /* name of other machine */

    char mybuffer[LARGESTREAD];      /* received data */

    char outgoing[LARGESTREAD];      /* outgoing message's buffer */

    int mybuffer1[LARGESTREAD/INTEGER_SIZE]; /* received integer data */

    int outgoing1[LARGESTREAD/INTEGER_SIZE]; /* outgoing integer message's buffer */

    float mybuffer2[LARGESTREAD/FLOAT_SIZE]; /* received float data */

    float outgoing2[LARGESTREAD/FLOAT_SIZE]; /* outgoing float message buffer */

    long noutgoing;                  /* size of the outgoing message */

    char temp[10];                   /* temp array used to make outgoing message */

    long count = 0;                  /* message counter */

    char received_type();
```

145

gprog2.c

```c
char type_received;

int elements_received;

long i;                    /* temp loop variable */

long j = 0;                      /* variable to control message sending */


/* pull out the string from the argument list */
if(argc > 2)
{
   printf("GPROG2: incorrect argument count!  use gprog2 <alias>\n");
   exit(1);
}
/* pull out the name of the other string, if it exists */
if( argc == 2 )
{
   strcpy( other_machine, "npscs-" );
   strcat( other_machine, argv[1] );
}
else
   strcpy( other_machine, "npscs-iris2" );


/* create a path to a particular machine (iris2 default) */
/* the first argument is the key for the shared memory segment.
   the second argument is the name of the machine to connect to.
   the third argument is the sending port number for the socket to use.
   the fourth argument is the receiving port number for the socket to use.
   the fifth argument indicates whether the processes should
   act as a server or a client.
   the sixth argument is the returned pointer to the structure
      remotemachine.
      it includes the pointer to the shared memory segment,
      the system generated shared memory id, the sendsem id,
      and the returned receivesem id.
*/
machinepath(1,other_machine,2,1,"client",&remotemachine);

/* the display loop and loop for polling the shared segment */
while(TRUE)
{
   /* make an outgoing message */
   strcpy(outgoing,"GPROG2 ORIGINATED MESSAGE: ");

   count = count + 1;

   outgoing1[0] = count;

   noutgoing = strlen(outgoing);

   outgoing2[0] = count;


   /* is there data in the shared segment? */
   if(receiver_has_data(&remotemachine))
   {
   type_received = received_type(&remotemachine);

      printf("The message received by GPROG2 is of type %c \n",
       type_received);

   switch (type_received)
   {
      case CHARACTER_ARRAY_TYPE:
         elements_received = number_received(&remotemachine);
```

146

```
                printf("The message received by GPROG2 is %d elements long!\n",
                 elements_received);

                read_characters(&remotemachine,mybuffer,
                     elements_received);
            break;

        case INTEGER_TYPE:
            read_integer(&remotemachine,mybuffer1);
            break;

        case FLOAT_TYPE:
            read_float(&remotemachine,mybuffer2);
            break;
    }
        /* at this point in the program, process the received data...*/
        printf("GPROG2 has received the following data:\n");

    switch (type_received)
    {
        case CHARACTER_ARRAY_TYPE:
                for(i=0; i < elements_received; i+=1)
                {
                    printf("%c",mybuffer[i]);
                }
            break;

        case INTEGER_TYPE:
                printf("%d",mybuffer1[0]);
            break;

        case FLOAT_TYPE:
                printf("%f",mybuffer2[0]);
            break;
    }
        printf("\n");
    }


    /* at this point, we would look at our system and see if we needed
        to send data. Instead, I will check if the sender is free.
        If the sender is free, I will send one of three messages */
    if(sender_is_free(&remotemachine))
    {
    if((j % 3) == 0)
        write_characters(&remotemachine,outgoing,noutgoing);

    /* wait until message sent before attempting to send another */
    while( !sender_is_free(&remotemachine) ) /* do nothing */ printf("2");

        if((j % 3) == 1)
            write_integer(&remotemachine,outgoing1);

    /* wait until message sent before attempting to send another */
    while( !sender_is_free(&remotemachine) ) /* do nothing */ printf("3");

        if((j % 3) == 2)
            write_float(&remotemachine,outgoing2);

    ++j;
    }
    else
    {
        /* assume socket connection broken */
    printf("Sender wasn't free!  Terminating...\n");
        break;
    }
```

```
    /* at this point, you can do the rest of the display loop */

}    /* endif while TRUE */

/* get rid of the path to the other machine...*/
deletemachinepath(&remotemachine);

}
```

3. **prog.c**

    a.   <u>Calling Protocols</u>

        This is a test program for the broadcast protocol. By command line argument, another machine to receive broadcast messages from can be specified. The default is to receive messages from *iris2*. It must be run in conjunction with **prog2.c** to function properly, as the port assignments are hardcoded.

    b.   <u>Code and Description</u>

```
/* this is file prog.c

   It is a sample top level program for the asynchronous reading
   and writing of sockets via shared memory and two other processes.

   This program spawns off the required processes.

   This program uses structure type Machine declared in file shared.h.

*/

#include "shared.h"
#define TRUE 1


main(argc,argv)

int argc;        /* argument count */
char *argv[];    /* pointers to the passed in arguments */

{
    Machine remotemachine1;          /* first structure for remote machine */

    Machine remotemachine2;          /* second structure for remote machine */

    char other_machine[50];          /* name of other machine */

    char mybuffer[LARGESTREAD];      /* received data */

    char outgoing[LARGESTREAD];      /* outgoing message's buffer */

    int mybuffer1[LARGESTREAD/INTEGER_SIZE]; /* received integer data */

    int outgoing1[LARGESTREAD/INTEGER_SIZE]; /* outgoing integer message's buffer */

    float mybuffer2[LARGESTREAD/FLOAT_SIZE]; /* received float data */

    float outgoing2[LARGESTREAD/FLOAT_SIZE]; /* outgoing float message buffer */

    long noutgoing;                  /* size of the outgoing message */

    char temp[10];                   /* temp array used to make outgoing message */

    long count = 0;                  /* message counter */

    char received_type();

    char type_received;
```

prog.c

```c
int elements_received;

long i;                    /* temp loop variable */

long j = 0;                       /* variable to control message sending */


/* pull out the string from the argument list */
if(argc > 2)
{
    printf("PROG: incorrect argument count!  use prog <alias>\n");
    exit(1);
}
/* pull out the name of the other string, if it exists */
if( argc == 2 )
{
    strcpy( other_machine, argv[1] );
}
else
    strcpy( other_machine, "npscs-iris2" );


/* create a pair of paths to a particular machine (iris2 default) */
/* the first argument is the maximum number of channels to be created.
   the second argument is the key for the shared memory segment.
   the third argument is the name of the machine to connect to.
   the fourth argument is the sending port number for the socket to use.
   the fifth argument is the receiving port number for the socket to use.
   the sixth argument indicates whether the processes should
   act as a receiver or a broadcaster.
   the seventh argument is the returned pointer to the structure
      remotemachine1 or remotemachine2.
      it includes the pointer to the shared memory segment,
      the system generated shared memory id, the sendsem id,
      and the returned receivesem id.
*/
dynamicmachinepaths(2,1,other_machine,2,1,"receive",&remotemachine1);

sleep(5);   /* to let both sides set up receiving channels first */

dynamicmachinepaths(2,1,other_machine,4,3,"broadcast",&remotemachine2);


/* the loop for polling the shared segment limited to avoid send buffer
   overflow */
while(TRUE)
{
    /* make an outgoing message */
    strcpy(outgoing,"PROG ORIGINATED MESSAGE: ");

    count = count + 1;

    outgoing1[0] = count;

    noutgoing = strlen(outgoing);

    outgoing2[0] = count;


    /* is there data in the shared segment? */
    if(receiver_has_data(&remotemachine1))
    {
    type_received = received_type(&remotemachine1);

        printf("The message received by PROG is of type %c \n",
        type_received);

    switch (type_received)
```

```
{
    case CHARACTER_ARRAY_TYPE:
        elements_received = number_received(&remotemachine1);

            printf("The message received by PROG is %d elements long!\n",
             elements_received);

            read_characters(&remotemachine1,mybuffer,
                elements_received);
        break;

    case INTEGER_TYPE:
        read_integer(&remotemachine1,mybuffer1);
        break;

    case FLOAT_TYPE:
        read_float(&remotemachine1,mybuffer2);
        break;
}
    /* at this point in the program, process the received data...*/
    printf("PROG has received the following data:\n");

switch (type_received)
{
    case CHARACTER_ARRAY_TYPE:
            for(i=0; i < elements_received; i+=1)
            {
                printf("%c",mybuffer[i]);
            }
        break;

    case INTEGER_TYPE:
            printf("%d",mybuffer1[0]);
        break;

    case FLOAT_TYPE:
            printf("%f",mybuffer2[0]);
        break;
}
    printf("\n");
}

 /* at this point, we would look at our system and see if we needed
    to send data. Instead, I will check if the sender is free.
    If the sender is free, I will send one of three messages */
 if(sender_is_free(&remotemachine2))
 {
 if((j % 3) == 0)
    write_characters(&remotemachine2,outgoing,noutgoing);

/* wait until message sent before attempting to send another */
while( !sender_is_free(&remotemachine2) ) /* do nothing printf("2")*/ ;

    if((j % 3) == 1)
        write_integer(&remotemachine2,outgoing1);

/* wait until message sent before attempting to send another */
while( !sender_is_free(&remotemachine2) ) /* do nothing printf("3")*/ ;

    if((j % 3) == 2)
        write_float(&remotemachine2,outgoing2);

/* wait until message sent before continuing */
while( !sender_is_free(&remotemachine2) ) /* do nothing printf("4")*/ ;

++j;
 }
 else
```

```
    {
        /* assume socket connection broken */
    printf("Sender wasn't free!\n");
        break;
    }


    /* at this point, you can do the rest of the display loop */

}   /* endif while TRUE */

/* get rid of the path to the other machine...*/
deletemachinepath(&remotemachine1);
deletemachinepath(&remotemachine2);

}
```

4. prog2.c

a. Calling Protocols

This is a test program for the broadcast protocol. By command line argument, another machine to receive broadcast messages from can be specified. The default is to receive messages from *iris1*. It must be run in conjunction with **prog.c** to function properly, as the port assignments are hardcoded.

b. Code and Description

```
/* this is file prog2.c

   It is a sample top level program for the asynchronous reading
   and writing of sockets via shared memory and two other processes.

   This program spawns off the required processes.

   This program uses structure type Machine declared in file shared.h.

*/

#include "shared.h"
#define TRUE 1


main(argc,argv)

int argc;          /* argument count */
char *argv[];      /* pointers to the passed in arguments */

{
    Machine remotemachine1;          /* first structure for remote machine */

    Machine remotemachine2;          /* second structure for remote machine */

    char other_machine[50];          /* name of other machine */

    char mybuffer[LARGESTREAD];      /* received data */

    char outgoing[LARGESTREAD];      /* outgoing message's buffer */

    int mybuffer1[LARGESTREAD/INTEGER_SIZE]  /* received integer data */

    int outgoing1[LARGESTREAD/INTEGER_SIZE]; /* outgoing integer message's buffer */

    float mybuffer2[LARGESTREAD/FLOAT_SIZE]; /* received float data */

    float outgoing2[LARGESTREAD/FLOAT_SIZE]; /* outgoing float message buffer */

    long noutgoing;                  /* size of the outgoing message */

    char temp[10];                   /* temp array used to make outgoing message */

    long count = 0;                  /* message counter */

    char received_type();

    char type_received;
```

153

prog2.c

```c
int elements_received;

long i;                    /* temp loop variable */

long j = 0;                        /* variable to control message sending */


/* pull out the string from the argument list */
if(argc > 2)
{
   printf("PROG2: incorrect argument count!  use gprog2 <alias>\n");
   exit(1);
}
/* pull out the name of the other string, if it exists */
if( argc == 2 )
{
   strcpy( other_machine, argv[1] );
}
else
   strcpy( other_machine, "npscs-iris2" );


/* create a path to a particular machine (iris2 default) */
/* the first argument is the maximum number of channels to be created.
   the second argument is the key for the shared memory segment.
   the third argument is the name of the machine to connect to.
   the fourth argument is the sending port number for the socket to use.
   the fifth argument is the receiving port number for the socket to use.
   the sixth argument indicates whether the processes should
   act as a server or a client.
   the seventh argument is the returned pointer to the structure
      remotemachine1 or remotemachine2.
      it includes the pointer to the shared memory segment,
      the system generated shared memory id, the sendsem id,
      and the returned receivesem id.
*/
dynamicmachinepaths(2,1,other_machine,3,4,"receive",&remotemachine2);

sleep(5);  /* to let both ends of the process get set up */

dynamicmachinepaths(2,1,other_machine,1,2,"broadcast",&remotemachine1);


/* the display loop and loop for polling the shared segment */
while(TRUE)
{
   /* make an outgoing message */
   strcpy(outgoing,"PROG2 ORIGINATED MESSAGE: ");

   count = count + 1;

   outgoing1[0] = count;

   noutgoing = strlen(outgoing);

   outgoing2[0] = count;


   /* is there data in the shared segment? */
   if(receiver_has_data(&remotemachine2))
   {
   type_received = received_type(&remotemachine2);

      printf("The message received by PROG2 is of type %c \n",
      type_received);

   switch (type_received)
   {
```

154

prog2.c

```c
    case CHARACTER_ARRAY_TYPE:
        elements_received = number_received(&remotemachine2);

            printf("The message received by PROG2 is %d elements long!\n",
             elements_received);

            read_characters(&remotemachine2,mybuffer,
                 elements_received);
        break;

    case INTEGER_TYPE:
        read_integer(&remotemachine2,mybuffer1);
        break;

    case FLOAT_TYPE:
        read_float(&remotemachine2,mybuffer2);
        break;
}
    /* at this point in the program, process the received data...*/
    printf("PROG2 has received the following data:\n");

switch (type_received)
{
    case CHARACTER_ARRAY_TYPE:
        for(i=0; i < elements_received; i+=1)
        {
            printf("%c",mybuffer[i]);
        }
        break;

    case INTEGER_TYPE:
        printf("%d",mybuffer1[0]);
        break;

    case FLOAT_TYPE:
        printf("%f",mybuffer2[0]);
        break;
}
    printf("\n");
}

/* at this point, we would look at our system and see if we needed
   to send data. Instead, I will check if the sender is free.
   If the sender is free, I will send one of three messages */
if(sender_is_free(&remotemachine1))
{
if((j % 3) == 0)
    write_characters(&remotemachine1,outgoing,noutgoing);

/* wait until message sent before attempting to send another */
while( !sender_is_free(&remotemachine1) ) /* do nothing printf("2")*/ ;

    if((j % 3) == 1)
        write_integer(&remotemachine1,outgoing1);

/* wait until message sent before attempting to send another */
while( !sender_is_free(&remotemachine1) ) /* do nothing printf("3")*/ ;

    if((j % 3) == 2)
        write_float(&remotemachine1,outgoing2);

/* wait until message sent before continuing */
while( !sender_is_free(&remotemachine1) ) /* do nothing printf("4")*/ ;

++j;
}
else
{
```

prog2.c

```
        /* assume socket connection broken */
    printf("Sender wasn't free!  Terminating...\n");
        break;
    }


}    /* endif while TRUE */

/* get rid of the path to the other machine...*/
deletemachinepath(&remotemachine2);
deletemachinepath(&remotemachine1);

}
```

5.　rmshare.c

a.　Calling Protocols

This is a stand-alone utility. It will remove all shared memory segments owned by the user. By command line argument, selective segments can be removed.

b.　Code and Description

```
/***********************************************************************
*                                                                     *
*  TITLE   : Inter-Computer Communication Package                     *
*                                                                     *
*  MODULE : rmshare.c                                                 *
*                                                                     *
*  VERSION: 1.0                                                       *
*                                                                     *
*  DATE    : 25 February 1988                                         *
*                                                                     *
*  AUTHOR : Theodore H. Barrow                                        *
*                                                                     *
***********************************************************************
*                                                                     *
*  HISTORY:                                                           *
*                                                                     *
*     VERSION: 1.0                                                    *
*                                                                     *
*     DATE    : 25 February 1988                                      *
*                                                                     *
*     AUTHOR : Theodore H. Barrow                                     *
*                                                                     *
*     DESC.   : Removes shared memory segments identified on command line. *
*                                                                     *
***********************************************************************
*                                                                     *
*                   RECORD OF CHANGES                                 *
*                                                                     *
*Version*  Date  *  Author                    *         Affected  *Reqd*
*        *       *   Change Description        *         Modules   *Vers*
***********************************************************************
*        *       *          *                *          *         *    *
*        *       *          *                *          *         *    *
***********************************************************************/
```

157

rmshare

```c
#include <errno.h>
#include <sys/sysmacros.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <gl.h>

/* The following defines will have to be modified for different machines
   but one of the underlying shared memory attachment mechanisms should
   work for any system V implementation. */
#define IRIS4D    1
#define IRIS3000  2
#ifdef FLAT
#define MACHINE   IRIS4D
#else
#define MACHINE   IRIS3000
#endif

extern int errno;

main( argc, argv )
int argc;        /* argument count */
char *argv[];    /* pointers to the passed in arguments */

{
    int first = 1;
    int last  = 1000;
    key_t i;
    int shmid;
    key_t key;
    static struct shmid_ds buffer;


    /* set the number of shared memory keys to remove */
    if(argc > 1)
    {
       for( i=first; i<argc; i++ )
        {
        key = atoi( argv[i] );
           if( (shmid = shmget( key, 0, 0)) == -1 )
            {
            if( errno != ENOENT )
            {
               write_error( shmid, key, errno );
            }
            }
            else
            {
               if( shmctl( shmid, IPC_RMID, &buffer ) == -1 )
                {
               write_error( shmid, key, errno );
                }
            else
               write_done( shmid, key );
            }  /* if( (shmid = shmget( i, 0, 0 )) == -1 ) */
       }  /* for */
    }
    else
    {
       for( i=first; i<last; i++ )
        {
           if( (shmid = shmget( i, 0, 0)) == -1 )
            {
            if( errno != ENOENT )
            {
               write_error( shmid, i, errno );
            }
```

```
          }
        else
        {
          if( shmctl( shmid, IPC_RMID, &buffer ) == -1 )
          {
            write_error( shmid, i, errno );
          }
        else
          write_done( shmid, i );
        }   /* if( (shmid = shmget( i, 0, 0 )) == -1 ) */
      }   /* for */
    }


    printf( "\nCompleted.\n" );

}   /* main() */


write_error( shmid, key, error )
int  shmid;
key_t key;
int error;

{
    printf( "\nShared Memory ID %d (key %d) caused error %d.",
              shmid, key, error );

}   /* write_error() */


write_done( shmid, key )
int  shmid;
key_t key;

{
    printf( "\nShared Memory ID %d (key %d) removed.", shmid, key );

}   /* write_done() */
```

6. testshare.c

a. Calling Protocols

This is a stand-alone utility. It will print current parameters for all active shared memory segments. By command line argument, selective segments can be printed.

b. Code and Description

```
/***********************************************************************
*                                                                     *
*  TITLE   : Inter-Computer Communication Package                     *
*                                                                     *
*  MODULE : testshare.c                                               *
*                                                                     *
*  VERSION: 1.0                                                       *
*                                                                     *
*  DATE    : 25 February 1988                                         *
*                                                                     *
*  AUTHOR : Theodore H. Barrow                                        *
*                                                                     *
*:********************************************************************:*
*                                                                     *
*  HISTORY:                                                           *
*                                                                     *
*     VERSION: 1.0                                                    *
*                                                                     *
*     DATE    : 25 February 1988                                      *
*                                                                     *
*     AUTHOR : Theodore H. Barrow                                     *
*                                                                     *
*     DESC.   : Determines which shmid values are used and what their *
*               parameters are.                                       *
*                                                                     *
*********************************************************************:*
*                                                                     *
*                    RECORD OF CHANGES                                *
*                                                                     *
*Version* Date  * Author              *             * Affected  *Reqd*
*        *       * Change Description  *             * Modules   *Vers*
*********************************************************************:*
*        *       *        *           *             *           *    *
*        *       *        *           *             *           *    *
*********************************************************************/
```

testshare.c

```c
#include <errno.h>
#include <sys/sysmacros.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <gl.h>

/* The following defines will have to be modified for different machines
   but one of the underlying shared memory attachment mechanisms should
   work for any system V implementation. */
#define IRIS4D     1
#define IRIS3000 2
#ifdef FLAT
#define MACHINE   IRIS4D
#else
#define MACHINE   IRIS3000
#endif

extern int errno;

main( )
{
    int first = 1;
    int last  = 1000;
    int i;
    int shmid;

    for( i=first; i<last; i++ )
    {
      if( (shmid = shmget( i, 0, 0)) == -1 )
      {
      if( errno != ENOENT )
      {
         write_error( shmid, i, errno );
      }
      }
      else
      {
      if( write_struct( shmid ) == -1 )
         write_error( shmid, i, errno );
      }   /* if( (shmid = shmget( i, 0, 0 )) == -1 ) */
    }   /* for */

    printf( "\nCompleted.\n" );

}   /* main() */


write_error( shmid, key, error )
int shmid;
key_t key;
int error;

{
    printf( "\nShared Memory ID %d (key %d) caused error %d.",
              shmid, key, error );

}  /* write_error() */


struct shmid_ds *get_struct( shmid )
int shmid;

{
    static struct shmid_ds buffer;

    if( shmctl( shmid, IPC_STAT, &buffer ) == -1 )
```

161

```
    {
        return( (struct shmid_ds *)-1 );
    }
    else
        return( &buffer );

}   /* get_struct() */


write_struct( shmid )
int shmid;

{
    struct shmid_ds *buf;

    if( (int)(buf = get_struct( shmid )) == -1 )
        return( (int)buf );

    printf( "\nShared Memory ID %d has the following structure:", shmid );
    printf( "\n    shm_perm has the following structure:" );
    printf( "\n        cuid is %d.", buf->shm_perm.cuid );
    printf( "\n        cgid is %d.", buf->shm_perm.cgid );
    printf( "\n        uid is %d.", buf->shm_perm.uid );
    printf( "\n        gid is %d.", buf->shm_perm.gid );
    printf( "\n        mode is %o.", buf->shm_perm.mode );
    printf( "\n        seq is %d.", buf->shm_perm.seq );
    printf( "\n        key is %d.", buf->shm_perm.key );
    printf( "\n    shm_segsz is %d or %x.", buf->shm_segsz, buf->shm_segsz );
    printf( "\n    shm_reg is a structure incompletely defined in region.h!" );
    printf( "\n    shm_lpid is %d.", buf->shm_lpid );
    printf( "\n    shm_cpid is %d.", buf->shm_cpid );
    printf( "\n    shm_nattch is %d.", buf->shm_nattch );
    printf( "\n    shm_cnattch is %d.", buf->shm_cnattch );
    printf( "\n    shm_atime is %d.", buf->shm_atime );
    printf( "\n    shm_dtime is %d.", buf->shm_dtime );
    printf( "\n    shm_ctime is %d.", buf->shm_segsz );

    return( 0 );

}   /* write_struct() */
```

LIST OF REFERENCES

1. Zyda, Michael J., and others, "Flight Simulators for Under $100,000," *IEEE Computer Graphics & Applications*, v. 8, no. 1, pp. 19-27, January 1988 .

2. Birrell, Andrew D. and Nelson, Bruce Jay, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, v. 2, no. 1, pp. 39-59, February 1984 .

3. Cheriton, David R., "The V Distributed System," *Communications of the ACM*, v. 31, no. 3, pp. 314-333, March 1988 .

4. Hearn, Donald and Baker, M. Pauline, *Computer Graphics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986 .

5. Magnenat-Thalmann, Nadia and Thalmann, Daniel, *Computer Animation: Theory and Practice*, Computer Science Workbench, ed. by Tosiyasu L. Kunii, Springer-Verlag, New York, 1985 .

6. Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, pp. 179-223, Addison-Wesley Publishing Company, Menlo Park, California, 1987 .

7. Dolezal, Michael J., *A Simulation Study of a Speed Control System for Autonomous On-Road Operation of Automotive Vehicles*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1987 .

8. Goodpasture, Richard Paul, *A Computer Simulation Study of an Expert System for Walking Machine Motion Planning*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987 .

9. MacPherson, David L., *A Computer Simulation Study of Rule-Based Control of an Autonomous Underwater Vehicle*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1988 .

10. Oliver, Michael R. and Stahl, David J., *Interactive, Networked, Moving Platform Simulators*, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1987 .

11. McConkle, Corinne and Nelson, Andrew H., *A Prototype Simulation System for Combat Vehicle Coordination and Motion Visualization*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1988 .

12. Nelson, Andrew H., McGhee, Robert B., and Zyda, Michael J., *Investigation into the Use of Kyoto Common Lisp For Real-Time Computer Animation*, to be published, Naval Postgraduate School, Monterey, California .

13. Newell, D. P. Siewiorek, C. G. Bell, and A., *Computer Structures: Principles and Examples*, pp. 306-485, McGraw-Hill Book Company, San Francisco, 1982 .

14. Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, v. 21, no. 8, pp. 666-677, August 1978 .

15. Hansen, Per Brinch, "Disributed Processes: A Concurrent Programming Concept," *Communications of the ACM*, v. 21, no. 11, pp. 934-941, November 1978 .

16. Lin, Kwei-Jay and Gannon, John D., "Atomic Remote Procedure Call," *IEEE Transactions on Software Engineering*, v. 11, no. 10, pp. 1126-1135, October 1985 .

17. Pountain, Dick, *A Tutorial Introduction to Occam Programming*, INMOS Limited, March 12, 1986 .

18. OSU-CISRC-TR-82-1, *The Implementation of a Multi-Backend Database System (MDBS): Part I - Software Engineering Strategies and Efforts Towards a Prototype MDBS*, by Kerr, D. S., and others , The Ohio State University, Columbus, Ohio, January 1982 .

19. NPS-52-82-008, *The Implementation of a Multi-Backend Database System (MDBS): Part II - The First Prototype MDBS and the Software Engineering Experience*, by He, X., and others , Naval Postgraduate School, Monterey, California, July 1982 .

20. NPS-52-83-003, *The Implementation of a Multi-Backend Database System (MDBS): Part III - The Message-Oriented Version with Concurrency Control and Secondary-Memory-Based Directory Management*, by Boyne, Richard D., and others , Naval Postgraduate School, Monterey, California, March 1983 .

21. Leffler, Samuel J., and others, "An Advanced 4.3BSD Interprocess Communication Tutorial," in *UNIX Programmer's Supplementary Documents Volume 1*, PS1:8, Usenix Association, 1986 .

22. Leffler, Samuel J., Fabry, Robert S., and Joy, William N., "A 4.2BSD Interprocess Communication Primer," in *Unix Programmer's Manual*, Draft of August 23, 1986 .

23. Tuthill, Bill, "IPC Facilities in 4.2BSD," *Unix Review*, v. 3, no. 4, pp. 82-87, April 1985 .

24. AT&T, *UNIX System V*, Streams Programmer Guide, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987 .

25. Rochkind, Marc J., *Advanced UNIX Programming*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985 .

26. Bach, Maurice J., *The Design of the Unix Operating System*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986 .

27. Texas Instruments Inc., *Explorer TCP/IP User's Guide*, 2537150-0001 Revision A, pp. C-1-C-7, Austin, Texas, June 1987 .

28. Texas Instruments Inc., *Explorer TCP/IP User's Guide*, 2537150-0001, Austin, Texas, March 1986 .

29. LANalyzer EX 500 Series Network Analyzer, *Reference Manual*, Publication No. 4200068-00 (Rev. B), Excelan, Inc., December 21, 1987 .

Distribution List for Dr. Michael J. Zyda

Defense Technical Information Center,
Cameron Station,
Alexandria, VA 22314   2 copies

Library, Code 0142
Naval Postgraduate School,
Monterey, CA 93943   2 copies

Center for Naval Analyses,
4401 Ford Avenue
Alexandria, VA 22302-0268   1 copy

Director of Research Administration,
Code 012,
Naval Postgraduate School,
Monterey, CA 93943   1 copy

Dr. Michael J. Zyda
Naval Postgraduate School,
Code 52, Dept. of Computer Science
Monterey, California 93943-5100   200 copies

Mr. Bill West,
HQ, USACDEC,
Attention: ATEC-D,
Fort Ord, California 93941   1 copy

John Maynard,
Naval Ocean Systems Center,
Code 402,
San Diego, California 92152   1 copy

Duane Gomez,
Naval Ocean Systems Center,
Code 433,
San Diego, California 92152   1 copy

James R. Louder,
Naval Underwater Systems Center,
Combat Control Systems Department,
Building 1171/1,
Newport, Rhode Island 02841   1 copy