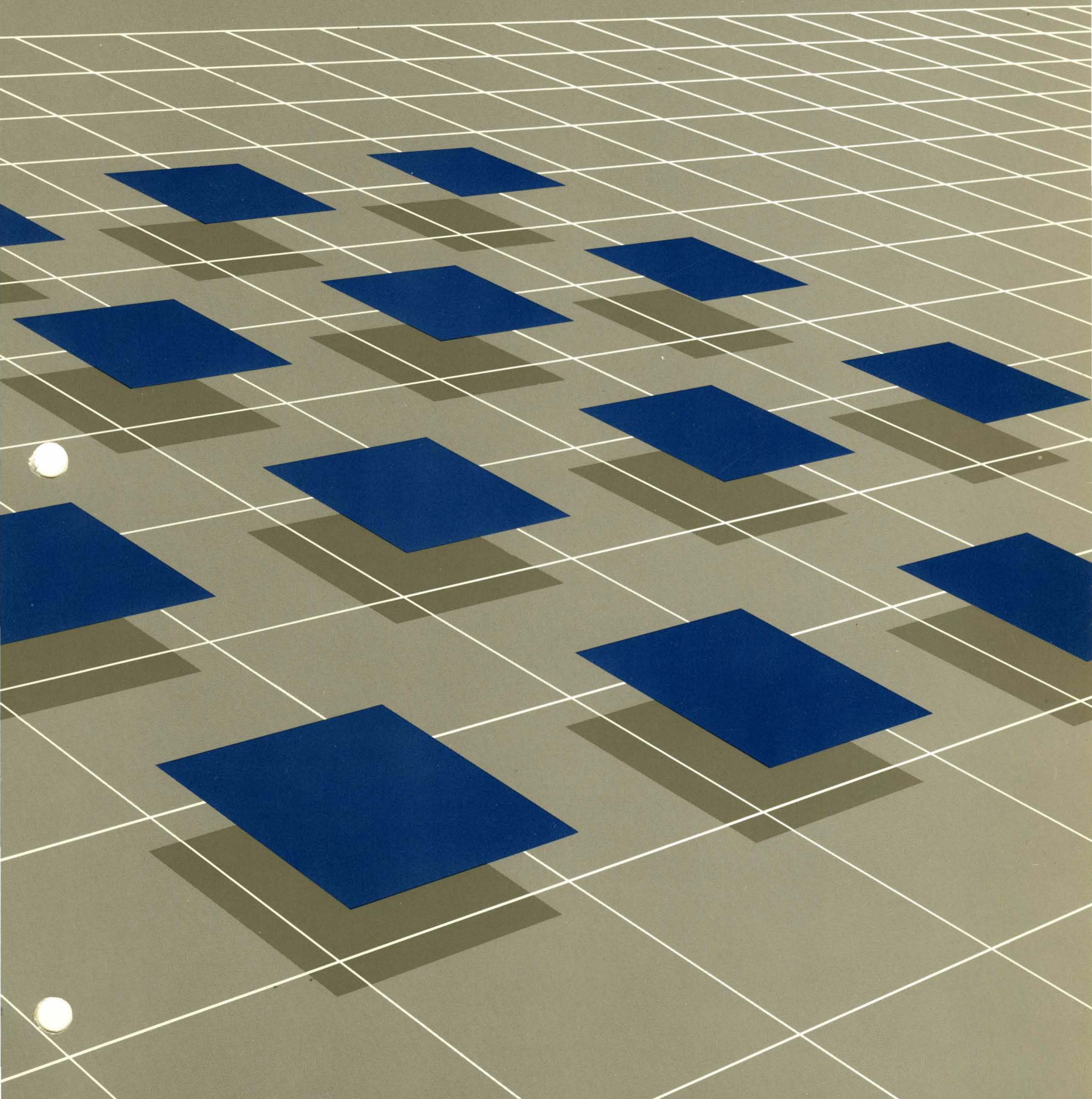
Bridge
Communications
Inc.

Ethernet System Product Line
Software Technical Reference Manual
Volume One -- Kernel and Support Software



BRIDGE COMMUNICATIONS, INC.

ETHERNET SYSTEM PRODUCT LINE

SOFTWARE TECHNICAL REFERENCE MANUAL

VOLUME ONE -- KERNEL AND SUPPORT SOFTWARE

Ø9-ØØ16-ØØ

July, 1983

Comments on this publication or its use are invited and should be directed to:

Bridge Communications, Inc.
Attn: Technical Publications
10440 Bubb Road
Cupertino, CA 95014

PUBLICATION CHANGE RECORD

This page records all revisions to this publication, as well as any Publication Change Notices (PCNs) posted against each revision. The first entry posted is always the publication's initial release. Revisions and PCNs subsequently posted are numbered sequentially and dated, and include a brief description of the changes made. The part numbers assigned to revisions and PCNs use the following format:

aa-bbbb-cc-dd

where "aa-bbbb" identifies the publication, "cc" identifies the revision, and "dd" identifies the PCN.

PCN Number	Date	Description	Affected Pages
09-0016-00	06/83	First Release	All

PREFACE

The ESPL Software Reference Manual provides the Bridge Communications customer with the information necessary to add software to a Bridge ESPL product.

The manual was prepared based on the following assumptions of reader knowledge:

1. The reader should be familiar with the information provided in the Bridge Communications Ethernet System Product Line Overview and CS/1 User's Guide.

2. The reader should be familiar with the Ethernet Specification, Version 1.Ø (see reference [4]).

3. The reader should be familiar with the Xerox Network System high-level protocols (see references [5], [6] and [7]).

4. The reader should have some familiarity with the UNIX* operating system (see reference [8]).

5. The reader should be familiar with the "C" language (see reference 9), or other high-level structured languages.

The Software Reference Manual is divided into three volumes. The information in Volume One is grouped in six major sections, whose contents are as follows:

Section 1.Ø - Introduction: Provides an overview of the Bridge Communications Ethernet System Product Line (ESPL), and describes the purpose and scope of this manual. Recommendations on how to use this manual are included.

Section 2.Ø - Software Architecture: Provides an overview of the ESPL software architecture, system resource management, and the protocol handling processes.

Section 3.Ø - Software Development Environment: Describes the tools necessary for development of software to be integrated into the ESPL products.

Section 4.Ø - MCPU Monitor: Describes the MCPU monitor and the monitor commands used for debugging, system generation and floppy utilities.

---

*UNIX is a Trademark of Bell Laboratories.

Section 5.Ø - Kernel Interface: Describes the resource management services provided by the Kernel and the access to these services available to processes running in an ESPL system.

Section 6.Ø - Floppy Disk I/O Interface: Describes the Floppy Disk interface available to processes running in an ESPL system.

Volume Two of this manual describes the packet-processing protocols used in the ESPL, and Volume Three describes the ESPL drivers and firmware.

REFERENCES

The following publications describe the Bridge Communications Ethernet System Product Line (ESPL):

[1] Ethernet System Product Line Overview, Bridge Communications, Inc.

[2] ESPL Communications Server/1 User's Guide, Bridge Communications, Inc.

[3] ESPL Software Reference Manual, Volumes Two and Three, Bridge Communications, Inc.

The following publications describe Ethernet and the Xerox Network System products:

[4] The Ethernet, A Local Area Network; Data Link Layer and Physical Layer Specifications, Version 1.0 (Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, 1980)

[5] Internet Transport Protocols, XSIS 028112 (Xerox Corporation, 1981)

[6] Courier: The Remote Procedure Call Protocol, XSIS 038112 (Xerox Corporation, 1981)

[7] D. Oppen, Y. Dalal, The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment (Xerox Corporation, 1981)

The following publications describe other related specifications:

[8] UNIX Programmer's Manual, Seventh Edition, Virtual VAX-11 Version, (University of California, Berkeley, 1981)

[9] B. Kernighan, D. Ritchie, The C Programming Language (Prentice Hall, Inc., 1978)

[10] MC68000 Microprocessor User's Manual, Second Edition MC68000UM(AD3) (Motorola Corporation, 1982)

[11] MC68000 Educational Computer Board User's Manual, Second Edition

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.Ø INTRODUCTION

This publication provides the Bridge Communications customer with the information necessary to add software to an Ethernet System Product Line product. In addition, it provides information about the existing ESPL software modules.

The Software Reference Manual is divided into three volumes. Volume One (this manual) describes the ESPL overall software architecture, the software development environment, the kernel and various support software. Volume Two describes the high-level, packet-processing protocols used in the ESPL. Volume Three describes the ESPL drivers and firmware.

This section defines the purpose, scope and audience of the publication and provides an overview of the products which comprise the Ethernet System Product Line.

1.1 Purpose and Scope

The information in this publication has been prepared to fulfill the needs of the OEM-level customer who wishes to add software to an ESPL system. In addition, this publication provides technical information about existing ESPL system software for the sophisticated user (e.g., the Network Manager).

The publication makes no attempt to present tutorial-level material aimed at the end user; please refer to the appropriate User's Guide for tutorial material.

1.2 How to Use This Manual

The ESPL products are designed to be customized by the user. Many different levels of customization are possible, but most applications will fit within a few major categories.

The following subsections present several categories of customized user applications, and indicate which interfaces are needed for each and which portions of the Software Reference Manual are applicable.

1.2.1 Adding Device Interfaces

One type of product customization consists of adding a new device interface (e.g., synchronous device support or IEEE 488 interface) to the CS/1 Connection Service. This may involve either:

- o Adding new hardware to the CS/1 in the form of a Multibus or iSBX board, and implementing firmware on the new board,
- o Replacing the ESPL firmware on the existing SIO board.

Both of these approaches require that the user either modify the existing SIO agent process running on the MCPU board or create a new agent process to provide the interface to the new device driver. This new agent would interface to the CS/1 Connection Service via the VT Program Interface, and could either replace the existing SIO agent or coexist with it.

The user planning either approach to this type of customization should read Sections 2.Ø through 5.Ø of Volume One, Section 5.Ø of Volume Two, and Section 3.Ø of Volume Three.

1.2.2 Adding Filters

A second type of customization consists of adding a filter between the Connection Service and the serial ports. For example, a filter application might be used to multiplex virtual connections over a single port, or to limit access to a port via software control. The application would require interfaces to the VT Connection Service and to the SIO module. The user planning an application of this type should read Sections 2.Ø through 5.Ø of Volume One, Section 5.Ø of Volume Two, and Section 3.Ø of Volume Three.

1.2.3 Adding XNS Protocol-Based Applications

Other customized applications may be added to the CS/1 Connection Service. For applications based on XNS protocols (e.g., file transfer or disk access) interfaces would be required to SPP or to IDP. The user planning an application of this type should read Sections 2.Ø through 5.Ø of Volume One and Sections 2.Ø and 4.Ø of Volume Two.

1.2.4 Replacing Protocols

Other protocols may be used to replace all CS/1 Connection Service protocols above XNS level Ø. This type of customization would utilize the CS/1 as a "protocol machine" to perform translation from the Ethernet to a different set of protocols. The user planning an application of this type should read Sections 2.Ø through 6.Ø of Volume One and Section 2.Ø of Volume Two.

1.2.5 Adding Non-Ethernet Network Interfaces

This type of customization represents the obverse of the type described in the previous subsection. In this type, the ESPL Data Link Service would be replaced or modified by either:

- Adding new hardware to the CS/1 to replace the existing Ethernet Controller boards, and implementing firmware on the new board(s).

- Replacing the ESPL firmware in the existing Ethernet Controller module.

Both of these approaches require that the user either modify the existing ESB agent process running on the MCPU board or create a new agent process to provide the interface to the new network driver. This new agent could either replace the Ethernet agent portion of the Data Link Service or coexist with it.

The user planning either approach to this type of customization should read Sections 2.Ø through 5.Ø of Volume One and Section 2.Ø of Volume Two.

1.3 Ethernet System Product Line

Bridge Communications' Ethernet System Product Line consists of Ethernet-based system products. Ethernet is a packet-switched Local Area Network (LAN) providing communications capability to and interconnection between various types of data processing equipment.

The Ethernet technology is described in detail in reference [4], which specifies the physical level and data link level protocols. Xerox, the original developer of the Ethernet technology, utilizes a set of published, high-level protocols in their Xerox Network System (XNS) products. These XNS protocols are described in references [5], [6] and [7].

Bridge Communications offers a full range of products compatible with both the Ethernet technology and the XNS high-level protocols. The products are designed for a maximum of performance, functionality, modularity and expandability, with a minimum of cost.

Initially, the Bridge Communications Ethernet System Product Line consists of the following XNS-compatible products:

- The Communications Server (CS/1) provides a bridge between an XNS Ethernet network and individual devices, and provides virtual connection services. Devices supported include most terminals, printers, host computers, modems, word processors, and other devices with a serial device interface.

Because XNS high-level protocols are implemented, access can also be provided to XNS workstations, file servers and print servers.

- The Gateway Server/1 (GS/1) connects an XNS Ethernet network to a host or network that has an X.25 interface, and provides virtual connection and interconnection services between devices on either network. The GS/1 can be used to extend the services of the CS/1 to include long-haul communications.

- The Gateway Server/3 (GS/3) connects two geographically distant XNS Ethernet networks by means of a medium- to high-speed communications link and provides a virtual interconnection service between devices on either network.

2.0 SOFTWARE ARCHITECTURE

The overall ESPL structure utilizes shared memory as the means of communication between modules residing on different processor boards.

Section 2.1 describes the basic ESPL functional modules and the hardware and software architecture of the ESPL products. Section 2.2 describes the software module responsible for system resource management, and Section 2.3 describes the software modules responsible for protocol processing and device handling.

2.1 Overview

Each ESPL product consists of three basic functional modules: the Central Communications Processor (CCP) module, and two external interface modules (I1 and I2).

The CCP is made up of a Main CPU (MCPU) board, a multitasking kernel (the operating system), and protocol software. The CCP provides the internal interface between the two external interface modules. The I1 external interface in any ESPL product is the Ethernet Controller (EC/1), which implements the Ethernet data link functions and buffers packets transmitted to and from the Ethernet.

The second external interface (I2) differs for each ESPL product. For the CS/1, the I2 external interface is the Serial I/O (SIO) interface which contains serial device interface software. For the GS/1, I2 is the X.25 interface, a two-board set which implements X.25 processing functions and handles packets traveling to and from an X.25 network or host.

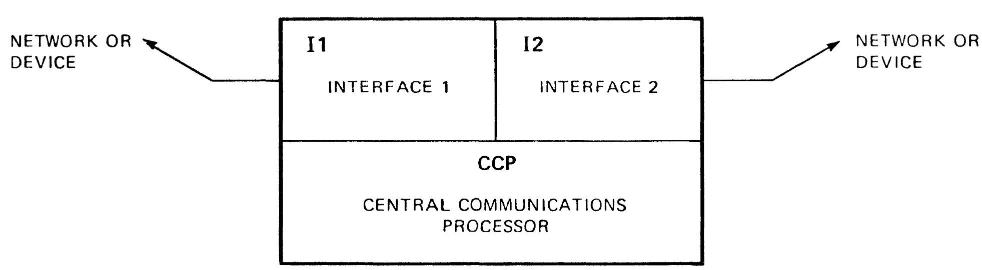


Figure 2-1 Basic Functional Modules

The ESPL products are based on multiple microprocessors for performance and flexibility. The multiprocessor architecture used is one in which independent processors communicate via shared memory and a bidirectional interrupt capability. The kernel, running on the MCPU board, controls the allocation of the shared memory on the ESB board. For each other microprocessor in the system, there is an "agent" on the MCPU board. The agent allocates memory and communicates with other kernel processes on behalf of its associated microprocessor. The ESB agent and the SIO agent (described in Volume Three, Sections 2.0 and 3.0, respectively) are examples of agent processes.

The hardware architecture of the Ethernet System products is illustrated by the CS/1 hardware block diagram shown in Figure 2-2. The boards are interconnected via an IEEE 796 Multibus backplane. Additional device or network interfaces can be added by replacing the I2 module, as is done for the GS/1 product, or by adding other Multibus or iSBX boards.

The software architecture of the ESPL is illustrated by the CS/1 software block diagram shown in Figure 2-3. Software support for additional device or network interfaces is achieved by replacing the I2 module, as is done for the GS/1 product, or by adding another module.

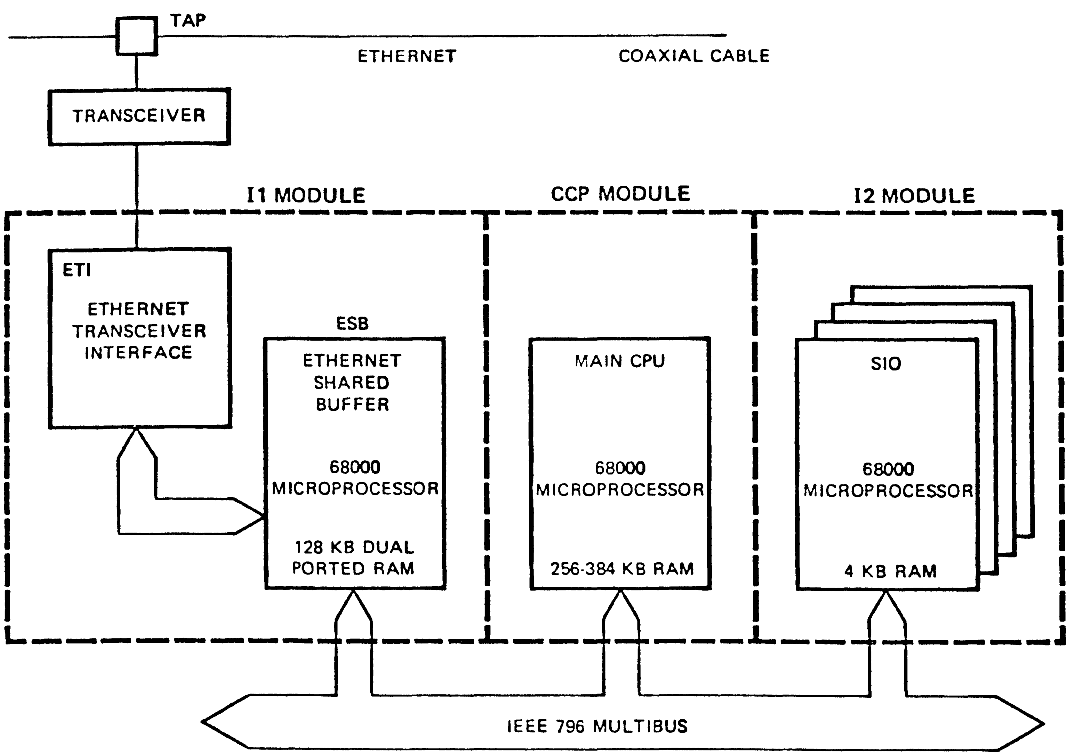


Figure 2-2 ESPL Hardware Architecture

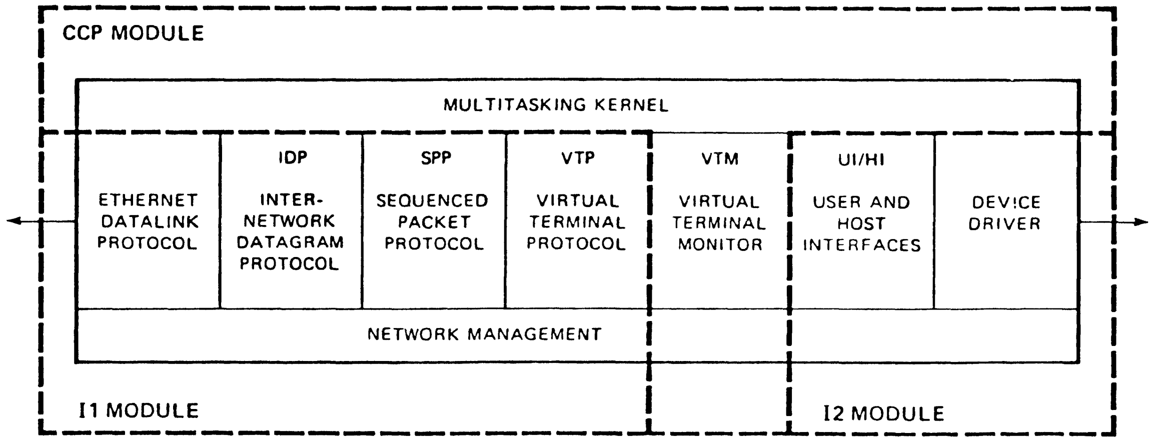


Figure 2-3 Software Architecture

The Kernel module provides a multiprocess environment for all protocol and user modules. It includes a message-based interprocess communication facility, a shared buffer manager, a storage allocator, an interrupt processing dispatcher, and time-of-day and alarm facilities. The Kernel resides on the MCPU.

The Data Link module performs the functions of the Ethernet Data Link Protocol (XNS Level Ø). These functions include transmitting and receiving frames, keeping statistics on network traffic, frame characteristics and errors, and supporting diagnostic aids including self-test diagnostics and higher level testing. Part of the module resides as firmware on the ESB board; the remainder consists of a software agent residing on the MCPU.

The Internet Datagram Protocol (IDP) module addresses, routes and delivers internet datagram packets. IDP is the XNS Level 1 protocol. IDP provides a best-effort internet delivery service, but does not guarantee reliable delivery or provide sequenced, flow-controlled transmission. The IDP module resides on the MCPU.

The Sequenced Packet Protocol (SPP) module provides reliable, sequenced, flow-controlled transmission of user packets or byte streams across the internet system. SPP is an XNS Level 2 protocol residing on the MCPU.

The Virtual Terminal module (which includes the Virtual Terminal Monitor (VTM), Virtual Terminal Protocol (VTP) and User Interface (UI) processes) provides a virtual circuit service to its clients. The VT service includes name lookup, establishment of virtual circuits, negotiation of terminal parameters, reliable exchange of data, attention signaling, and synchronized disconnection. VT implements a Virtual Terminal Protocol utilizing XNS Courier protocol functions. The User Interface (UI) provides the terminal user or the host with the capacity to control the interface to the CS/1 by specifying parameters that describe transmission and device characteristics. The terminal user specifies these parameters interactively; host interface parameters are set via program control. The VT module resides on the MCPU.

The Serial Device Driver module is an interrupt-driven driver that transfers data, attention and flow control signals to and from serial devices attached to the SIO board using an asynchronous protocol. Part of the module resides as firmware on the SIO board; the remainder consists of a software agent residing on the MCPU.

The Network Management module provides a variety of functions, including performance monitoring, network control and configuration management. The XNS Error Protocol and Echo Protocol are implemented within the Network Management module. This module resides on the MCPU.

In addition to these major modules, the CS/1 includes the following miscellaneous software:

- Floppy Driver
- PROM Monitor/Debugger
- Boot Loader
- Power On Diagnostics

2.2 System Resource Handling

The kernel is the heart of the Bridge operating system. It provides centralized access to system resources in a transparent manner, so that the user of these resources need not be concerned with their underlying form. In a system optimized to perform specialized tasks efficiently, system resources and the methods of accessing them are in the domain of the kernel.

System resources include mailboxes, storage, buffers and clock control structures. The CPU is controlled by a round-robin, prioritized scheduler, which chooses a process from one of eight prioritized ready lists. Other processes are accessed by means of an inter-process communication system, which features multiple mailboxes with selective receives, and the ability to peek into mailboxes. The kernel also supports semaphores for synchronizing access to shared data structures.

The CPU manipulates two kinds of memory: memory private to the CPU, and memory that is shared between the main CPU and other processors in the system. There are two means of accessing memory: via a storage allocator, which allocates blocks of memory from private or shared memory by returning to the requesting process a pointer to the physical block; and via a buffer management system built on top of it, which instead returns the identifier of a buffer descriptor, providing a logical view of memory.

External devices are manipulated by agents that execute in the main CPU space. An agent may be a process, or may run on behalf of a process. The basic structure of these agents includes a natural division between synchronous functions (request interface) and asynchronous functions (interrupt servicing). The kernel provides a centralized interrupt dispatch routine, so that interrupt identifiers can be used instead of process IDs in messages originating from interrupt routines, and the kernel always knows the current nested interrupt level.

The clock facility provides an alarm function, so processes can put themselves to "sleep" for a determined period of time. The facility also includes the ability to set and read the system clock.

The kernel functions and the means of accessing them are described in detail in Section 5.0.

2.3 Protocol and Device Handling

The CS/1 protocol software (briefly described in Section 2.1) is organized as a set of processes that run on top of the kernel. Figure 2-4 shows how these processes interact.

The Data Link and SIO Drivers each consist of firmware (on the ESB and SIO boards, respectively) as well as agent code on the MCPU board.

Of the remaining processes, some exist as a single incarnation and others as multiple incarnations. The single incarnations include IDP; multiple incarnations include the SPP and VT modules. SPP consists of both a single "parent SPP" incarnation, responsible for dynamically creating and deleting "child SPP" incarnations, plus an individual "child SPP" incarnation for each current session. VT consists of a "parent VT" incarnation, responsible for dynamically creating and deleting "child VT" incarnations, plus an individual "child VT" incarnation for each active port.

The protocol processes are described in Volume Two of this manual, and device handling processes are described in Volume Three.

```
                  +-------------+
                  |     SIO     |          SIO Board
                  |  Firmware   |
                  +------+------+
                         |
  -----------------------|----------------------
                         |
                  +-------------+
                  |     SIO     |
                  |    Agent    |
                  +-+------+---+
                 /  |      |    \          MCPU Board
                /   |      |     \
               /    |      |      \
              /     |      |       \
+----------+    +--+--+ +--+---+     +-----+
|  Parent  |---| VT  | | VT  | ...| VT  |
|  VT      |   |     | |     |    |     |     (1)
+----+---+-+   +--+--+ +--+--+    +-+-+-+
     |   |    /   |       |         |  \
     |   +---/-+  |       |         |   \
     |      /  |  |       |         |    \
+----+-----+   +--+--+ +--+--+  +--+--+ +-----+
|  Parent  +---+ SPP | | SPP |...| SPP | | SPP |  (2)
|  SPP     |   |     | |     |   |     | |     |
+----------+   +--+--+ +--+--+  +-----+ +-----+
           \      |       |     /       /
            \     |       |    /       /
             \    |       |   /       /
              +--+--------+--+       /
              |     IDP      +------+
              |              |
              +------+-------+
                     |
              +------+-------+
              |     ESB      |
              |    Agent     |
              +-----+--------+
                    |
  ------------------|-----------------------
                    |
              +-----+------+
              |    ESB     |          ESB Board
              |  Firmware  |
              +------------+

  Notes:  (1)  One VT process per port
          (2)  One SPP process per session
```

Figure 2-4 Process Interaction

3.0 SOFTWARE DEVELOPMENT ENVIRONMENT

This section describes the requirements of the environment within which software is developed for integration into an ESPL product, as well as the development support optionally provided with the ESPL product.

3.1 Requirements of the Development Environment

For the current release of ESPL product software, all software development must be performed on a host running UNIX Version 7.0 (or Berkeley Version 4.1bsd) or later.

Bridge Communications does not provide the development host, but does optionally provide a package of tools and utilities to facilitate the software development and integration process. This package, which can be ordered as the Bridge Development Environment (BDE) package, is described in the Section 3.2. In addition, the customer who purchases object or source software receives other development utilities as part of the software distribution; these are described in Section 3.3.

The development strategy involves linking customer-added code with the required ESPL release software (which is purchased separately in either object or source form) using the BDE tools and utilities. The customer can optionally use other tools, so long as the resulting compiler/assembler output can be linked with the Bridge software modules.

3.2 Bridge Development Environment Package

This section briefly describes the tools and utilities included in the Bridge Development Environment (BDE) package, and provides some general information on the transportability of the package among various UNIX hosts.

The BDE package is shipped on a 9-track, 1600 bpi magnetic tape. On-line documentation for each program is available (via the UNIX "man" command) as part of the package. The BDE tools and utilities programs are as follows:

- Cc68 is the portable UNIX "C" compiler, modified for the 68000. This flexible program is used to translate between various types of files, including "C" source files, assembly language files and relocatable binary files. Arguments to cc68 may specify options as well as filenames, and the amount of processing performed by cc68 may be decreased or increased by the action of the options. In general, cc68 translates each "C" source file or assembly language file into a relocatable binary file using ccom68 and/or as68 (described below); then cc68 link-edits all binary files into a single binary output file.

- Ccom68 is the translator component of cc68, and is used to translate "C" source files to assembly language files.
- O68 is the assembly language optimizer component of cc68.
- As68 is the 68000 assembler component of cc68, and is used to generate relocatable binary files.
- Ld68 is the link-editor component of cc68, which combines several binary files into one, resolving external references and searching libraries.

In addition to these major utilities, the BDE package includes several miscellaneous utilities, which operate on binary files to print extended statistics (pr68), print symbol tables (nm68), print relocation commands (rl68), print segment sizes (size68); or which translate binary files into a form readable by the 68000 by reversing byte order in some fields and padding or repacking other fields (rev68).

The steps necessary to install the BDE package on the UNIX host, and the usage of the listed utility programs are described in Section 3.3.

The "C" compiler (cc68) should run successfully on any of a wide variety of machines. Bridge Communications utilizes a VAX 11/750 running Berkeley VAX UNIX Version 4.1. The compiler should also run with little or no modification on any implementation of Version 7 UNIX, and with more extensive modification on most implementations of System 3 UNIX.

None of the utility programs require more than 64K bytes of code or 64K bytes of data. However, the total amount of memory needed by the compiler itself is more than 64K bytes, so it will not run on the smaller address space machines which do not support separate code and data spaces (e.g., the PDP-11/34).

In the standard BDE package, the programs cc68, as68, ld68, rev68 and dl68 are installed in the directory /usr/bin, and the program ccom68 is installed in the directory /usr/sun/lib/ccom68. The library (libc.a) and the run-time startup routines (crt0.b) are installed in /usr/sun/lib. The user should place all standard include files in the directory /usr/sun/include.

Most absolute pathnames are specified in cc68.c, and can be changed if necessary to suit the needs of the customer. The pathnames of the library and the startup routine crtØ.b are defined in ld68. The library name must not be changed, since there is some code in ld68 which takes advantage of the number of characters in the library pathname.

If the customer's development machine orders bytes differently than the VAX, the customer may need to make some changes to as68, ld68, rev68 and ds68 (these are the programs that deal with object code). If changes to any of the utility programs are required, the program(s) must be recompiled. This should present no problems when recompiling under Version 7 or Berkeley Version 4.1 UNIX. However, undefined references may be reported when recompiling under some System 3 implementations. These are typically due to references to Version 7 routines that do not exist in System 3.

3.3 Software Development

This section lists the major steps involved in the software development process; the following subsections describe each step in detail.

1. The first major step in the software development process is the installation of the magnetic tape containing the BDE package on the customer's UNIX host. It includes creating the appropriate directories and loading utility programs and UNIX manual sections into them. This portion of the process is described in Section 3.3.1.

2. The second major step is the installation of the magnetic tape containing the ESPL binary and/or source software files on the customer's UNIX host. This portion is described in Section 3.3.2.

3. The third major step in the development process is the optional creation of software to be added to or substituted for existing ESPL software. The information necessary for this portion of the process (e.g., information about interfaces to the Bridge kernel or to protocol processes) is contained in Volumes Two and Three of this manual. In addition, the customer must include information about the new software in the appropriate initialization table and ensure that sufficient memory is available for the new software. These steps are described in Section 3.3.3.

4. The fourth major step in the development process is compiling, assembling, link/loading and formatting source files on the UNIX host into a single binary file to be downloaded to the target ESPL unit. This is accomplished using Make files, described in Section 3.3.4. The formatting utility is described in more detail in Section 3.3.5.

5. The fifth major step is the download process itself. Refer to Section 3.3.6 for instructions. Section 3.3.7 describes the procedure used to download firmware only.

6. The sixth major step, debugging the software downloaded to the ESPL unit, is described in Section 3.3.8.

7. Once software has been downloaded and debugged, it is ready to be saved onto the floppy diskette. Section 3.3.9 contains instructions on creating a new floppy diskette, as well as information on the diskette directory structure.

3.3.1 BDE Distribution Tape Installation

This section describes the procedure used to install the BDE distribution tape on the customer's UNIX host. The BDE distribution tape is a 1600 bpi, high-density UNIX "tar" tape. The installation steps are as follows:

1. Log in to the root directory.

2. Create a directory called "/usr/sun":

   ```
   mkdir /usr/sun
   ```

3. Specify the directory "/usr/sun" as the current directory:

   ```
   cd /usr/sun
   ```

4. Mount the BDE distribution tape on the tape drive; ensure the drive is configured for 1600 bpi, high-density tape.

5. Read the tape into the directory "/usr/sun":

   ```
   tar xf /dev/rmt0 .
   ```

   Note that in this example, "/dev/rmt0" is the mnemonic for the tape device; the mnemonic may differ depending on host and peripheral device configuration.

6. A summary of these instructions may now be obtained from the file called "READ_ME" in the directory "/usr/sun".

7. Copy all files from the directory "/usr/sun/bin" into the directory "/usr/bin":

   ```
   cd /usr/bin
   cp /usr/sun/bin/* .
   ```

8. Copy all files from the directory "/usr/sun/man" into the directory "/usr/man/manl":

   ```
   cd /usr/man/manl
   cp /usr/sun/man/* .
   ```

9. If the BDE is installed in a different directory than /usr/sun, the files ./src/cmd/cc68.c, ./src/cmd/ld68/ld68.c and ./src/cmd/Makefile must be edited to modify the absolute pathname dependencies, then recompiled and relinked.

3.3.2 Software Distribution Tape Installation

This section describes the directory structure of the ESPL software distribution tape, the procedure used to create the appropriate directory structure on the target UNIX host and the procedure for installing the tape on the host.

Binary code and/or source code must be purchased separately from the BDE. The software files are distributed on a 1600 bpi, high-density UNIX "tar" tape. There are organizational differences between files containing floppy-loaded software (typically run on the ESPL unit's MCPU) and files containing firmware.

Software files are structured in a tree organization, as follows:

- There is one master root node directory, named "xxxrlse", where "xxx" is an abbreviation code designating the product (e.g., "cslrlse").

- The master root node directory contains one subdirectory per module, as well as a bin subdirectory and an integ.test subdirectory.

- The module subdirectories contain the source, object list and individual make files in an "src" subdirectory, and the header files in an "h" subdirectory.

- The bin subdirectory contains various utility programs and shell scripts, including a global makefile (supermake) and associated make rules (csl_make_rules). Supermake is used to recompile and load all modules in the cslrlse directory. On-line documentation is available via the UNIX "man" command for each of the utilities except make rules.

- The integ.test subdirectory contains the system initialization table (the file "csl.c") and the makefile used to load (but not recompile) all modules in the cslrlse directory.

This tree structure is illustrated in Figure 3-1.

```
cslrlse
        ./bin
                supermake
                csl_make_rules
                .
                .
                .
        ./integ.test
                csl.c
                makefile
                .
                .
                .
        ./kernel
                ./src
                ./h
        .
        .
        .
```

Figure 3-1 Distribution Directory Organization

Firmware files are distributed in UNIX "archive" files. A description of archive files is provided in reference [8], under "ar(1)".

Firmware consists of Level 1 Diagnostics, an optional monitor, and one or more sets of optional driver code (e.g., Ethernet driver or SIO driver), and may be restored as follows:

- o Level 1 Diagnostics (and monitor code, if included) are stored in a single archive named "xxMON", where "xx" is an abbreviation code designating the PROM set. Diagnostic and monitor code are independent of driver code, and may be restored in a subdirectory anywhere in the cslrlse tree structure.

- o Driver code (if included) is stored in an "h" archive and an "src" archive, named respectively "xxH" and "xxSRC". Driver code is archived from "xxxrlse/yyy" (where "xxx" designates the product, and "yyy" designates the driver module; e.g. "cslrlse/sio") and should be restored into this software structure.

To install the software distribution tape, set the current directory to the directory where the subtree is to be appended, mount the distribution tape on the tape drive, and then use the "tar" command to read the tape into the current directory. For example:

```
cd /usr/cslrlse
tar xf /dev/rmtØ .
```

In this example, "/usr/cslrlse" is the directory into which the tape is to be read and "/dev/rmtØ" is mnemonic for the magnetic tape device, which may differ depending on the customer's installation.

3.3.3 Adding OEM Files or Modules

Adding files or modules should be done in a manner consistent with the file organization described in Section 3.3.2.

When adding a file to an existing module, the following steps are required:

1. Place the file in the existing "src" subdirectory of the module.

2. Update the module's makefile dependencies (refer to Section 3.3.4 for descriptions and examples of makefiles).

When adding a new module, the following steps are required:

1. Create a new module directory (including "src" and "h" subdirectories).

2. Create a makefile for the new module (refer to the existing individual makefiles for examples).

3. Optionally, update the "supermake" file to reflect the addition of a new individual makefile. This step is required only if the customer intends to use "supermake" to compile an entire system, rather than use "make" to compile a single module.

4. Edit the appropriate initialization table to include the new module. For most modules, this is the file cslrlse/integ.test/csl.c. The table contains an entry for each system process that the parent process "init" must create; an entry for the new module should be added at the end of the table. Agent processes, however, are not started up by "init"; the ESB agent is started by IDP, and the SIO agent is started by the Parent Virtual Terminal process. Refer to Section 5.2.1 for further information about the csl.c initialization table. The ESB and SIO agents are

described in Volume Three of this manual, and IDP and Parent Virtual Terminal are described in Volume Two.

5. Use the "size" utility to find the amount of memory required by the new module.

6. Edit the file cslrlse/kernel/src/keram.c to free enough memory for the new module. In the standard release of the CS/1, the existing code uses almost all of the available memory, and some of the kernel's buffers must be deallocated to make room for the new code. To do this, the following steps are required:

   o Locate the structure "privhdrs" in the file keram.c. This structure contains a list of how many buffers of what sizes are normally allocated as private header space.

   o Edit the structure to reduce the number of buffers which normally exist in quantities of 48. These may be cut back to a minimum of 32 each if necessary.

   o If this does not free enough space, reduce the number of other size buffers until adequate free memory is available for the new software module.

7. Execute the makefile residing in cslrlse/kernel/src to remake the kernel module.

8. Set the current directory to /cslrlse/integ.test. Edit the makefile in this directory to include an entry for the new module.

9. Execute the makefile edited in step 7, specifying an argument of "csl".

3.3.4 Makefiles and Utilities

The utilities which are provided as part of a software distribution tape (in the directory /cslrlse/bin) are used to simplify and standardize the process of compiling, assembling, linking and formatting code, and to download code to the ESPL system. Brief descriptions of the most important utilities are presented in this section; complete on-line documentation for most of the utilities is available via the UNIX "man" command.

- Makefiles are present in the subdirectories for each individual module. An individual module's makefile contains the commands necessary for "make" to appropriately generate compiled and assembled output files, by defining source files, objects, key files, etc. The makefile also includes an entry which references the file containing makefile rules. Since every makefile is different, this manual makes no attempt to describe each one in detail; instead, a simple example is provided in Figure 3-2.

  Note that the makefiles included in the software distribution tape require that the development host have the UNIX System 3 level make utility, which supports "include" statements.

- Makefile rules are contained in a single rules file (cslrlse/bin/csl_make_rules), and are referenced in each individual Makefile, thus assuring consistency between module subdirectories. The rules file allows a variety of requests to be passed as makefile arguments, enabling the user to remake all objects, make all expanded assembly listing files, remove all objects, make a file listing in each module or archive all source, header and other keyfiles into an archive file.

- Supermake (the file cslrlse/bin/supermake) is the global makefile, used when "making" an entire CS/1. The utility consists of a cshell script which sets the current directory to each module's subdirectory, then executes a make with the arguments present on the supermake command line. Like the rules file, supermake allows a variety of requests to be passed as arguments. If the customer adds or replaces a module in an ESPL system, the Supermake file must be updated to reflect the change.

- Srecs is the formatter utility, which translates binary files into S-record format prior to downloading to the ESPL unit. This utility is available in two forms, described in detail in Section 3.3.5.

o Oad is the download utility, used as a slave process by the monitor command "LOAD" to transmit S-record format files from the UNIX host to the ESPL unit via an RS-232-C serial download line which is connected to the ESPL unit's download port (refer to Section 4.1 for detailed information).

```
# ${BDE}/spp/src/Makefile
# Define commands and command parameter strings
ARCHV=  VAXSPLIB
CFILES= spp.c spctl.c spidp.c spuser.c sppkt.c \
        sptrace.c spVersion.c
VOBJS=  spp.o spctl.o spidp.o spuser.o sppkt.o \
        sptrace.o spVersion.o
68OBJS= spp.b spctl.b spidp.b spuser.b sppkt.b \
        sptrace.b spVersion.b
LSTS=   spp.lst spctl.lst spidp.lst spuser.lst \
        sppkt.lst sptrace.lst spVersion.lst
HFILES= ../h/spincludes.h ../h/spint.h \
        ../h/spp.h ../h/spuser.h
KEYFILES=

$(ARCHV):       $(ARCHV)(spp.o) $(ARCHV)(spctl.o)
        $(ARCHV)(spidp.o) $(ARCHV)(spuser.o) \
        $(ARCHV)(sppkt.o) $(ARCHV)(sptrace.o) \
        $(ARCHV)(spVersion.o)

include ../../bin/csl_make_rules
```

Figure 3-2 Sample Makefile for SPP Module

3.3.5 S-Record Formatter

This section describes the "srecs" utility, which translates binary files into S-record format files for transmission between systems.

S-record formatting consists of a two-level encoding method which transforms each 8-bit byte of data into two printable ASCII characters. An S-record is a sequential ASCII record starting with an "S" character (hexadecimal 53) and ending with the carriage return and linefeed characters. This formatting scheme was introduced by Motorola for use with its development system, and is now widely used in the industry.

There are three categories of S-records: header records, data records and termination records. The Motorola S-record format specification defines eight separate types of S-records, as follows:

- S0 Header record for a block of data records.
- S1 Record containing data and a 16-bit destination address.
- S2 Record containing data and a 24-bit destination address.
- S3 Record containing data and a 32-bit destination address.
- S5 Termination record containing a count of records in the previous S1 block (alternate of S9).
- S7 Termination record for a block of S3 records.
- S8 Termination record for a block of S2 records.
- S9 Termination record for a block of S1 records.

For further information on S-record format, see reference [11], Appendix A.

Two forms of the "srecs" utility are available:

1. The "srecs" program creates only record types S1 and S9, and is used only when the file being produced consists of firmware to be downloaded from the UNIX host directly to a PROM programmer. This version of the program uses a standard record length of 32 bytes.

2. The "bigsrecs" program creates only record types S2 and S8, and is used when the file being produced consists of software to be downloaded via monitor command to an ESPL unit. This version of the program uses a longer record length (currently 96 bytes), and thus reduces overhead in the download process by allowing transmission of larger packets.

Both forms of the utility use the same syntax; a complete description may be obtained on the UNIX host via the "man srecs" command. Arguments passed include the following:

```
-T <destination address>
inputfile
outputfile
```

The output file may be specified in either of two ways:

```
-o outputfile     (anywhere in the command line)
> outputfile      (at the end of the command line)
```

Thus, the following two examples will produce the same result:

```
bigsrecs -T Ø -o kernel.hex kernel.out
bigsrecs -T Ø kernel.out > kernel.hex
```

3.3.6 Downloading Software

This section lists the steps used to download software (in S-record format files) from the host to the ESPL unit. The instructions assume this unit is a CS/1; the procedure for a GS/1 is identical. The instructions also assume that a physical connection exists between the host download port and the CS/1 download port, and that a console terminal is connected to the CS/1 console port.

1. On the CS/1 console terminal, enter transparent mode by typing the command "i t".

2. Login to UNIX and change to the directory containing the file to be downloaded. Note that a search path must also exist to the directory containing the "oad" program; this path should be defined as an alternate path in the user's ".login" or ".profile" file.

3. Return to the CS/1 monitor by typing the transparent mode escape sequence ("<CTRL-caret>" followed by the letter "c").

4. Enter the Load command, as follows:

```
load <filename>
```

While the download is in progress, lines of periods will print on the console screen. When the download is complete, the console terminal bell will ring and the MCPU monitor prompt (>) will reappear. The downloaded software is now in CS/1 memory, and is ready to be debugged, saved to diskette and executed.

3.3.7 Downloading Firmware

This section describes the steps used to download SIO firmware to shared memory in order to debug the firmware using the SIO PROM debugger. This procedure requires special OEM PROMs for the SIO board (MONSA, included as part of the OEM SIO Kit). THe following steps are required:

1. The MONSA prom configures the port Ø of the SIO board as a console port, and port 2 of the SIO board as a download port. Connect the console terminal to port Ø, and connect the host to port 2.

2. Perform steps 1 through 4 of Section 3.3.6.

3. Be sure to use shared memory address space (this may be the ESB board or a memory card). If the shared memory is on the ESB, the address range is from 2ØØØØØ to 4ØØØØØ. Note that code will execute slower out of shared memory.

3.3.8 Debugging

The MCPU monitor provides a complete set of interactive commands for debugging software. Monitor commands permit the user not only to examine and alter memory, but also to set multiple breakpoints (up to eight), to disassemble instructions, and to trace instructions via a single-step operation. Refer to Section 4.1 for a comprehensive list of the MCPU monitor commands.

3.3.9 Creating a New Diskette

After software has been downloaded and debugged, it may be saved on the diskette via monitor commands.

This section describes the diskette directory system and lists the steps required to create a new floppy diskette.

Diskette Directory Structure

The floppy disk drive uses double-sided, double-density diskettes, with a storage capacity of 327K bytes (formatted). Each diskette is divided into 639 blocks of 512 bytes each; the blocks are interleaved from side to side. Side Ø, track Ø, sectors 1 through 8 are used first; then side 1, track Ø, sectors 1 through 8 are used; then side Ø, track 1, sectors 1 through 8, and so on. All monitor commands refer to diskette locations by hexadecimal block number. Since the disk controller translates the block number to the appropriate side, track and sector numbers, it is not necessary for the user to calculate these. Note, however, that in the standard CS/1 disk subsystem, one block equals one sector.

The blocks on the diskette are allocated as indicated in Table 3-1.

Block Ø of each diskette is reserved for the directory system, and may not be used for storage of code. The directory system contains up to 32 structures, numbered from Ø to 1F (hexadecimal). Each structure contains information about one file on the diskette, divided into fields as indicated in Table 3-2.

The following steps are necessary to read the directory of a diskette:

1. Use the monitor command "Read" to transfer block Ø into memory (refer to Section 4.1.23).

2. Use the monitor command "Display Memory" to display the information (refer to Section 4.1.1Ø).

Table 3-1 Diskette Block Allocation

Hexadecimal Block Number(s)	Allocation
Ø	Directory system
1 through 181	ESPL software
182 through 18F	Unused
19Ø through 1EF	MCPU monitor overlay routines (e.g., copy_disk, disassembler, sysgen)
1FØ through 27F	Clearinghouse tables, UI configuration tables, and statistics data structures

Table 3-2 Diskette Directory System Fields

No.	Type (Size)	Meaning
1	Character (1 byte)	File presence; Ø = file not present, 1 = file is present
2	Character (1 byte)	Executability; Ø = file not executable, 1 = file is executable
3	Short (2 bytes)	Block no. of first block of file
4	Short (2 bytes)	Block no. of last block of file
5	Long (4 bytes)	Length of file, in bytes
6	Long (4 bytes)	Execution starting address, in hexadecimal
7	Short (2 bytes)	Padding, to make structure 16 bytes long

Creating a New Diskette

The following steps are necessary to create a new ESPL system diskette:

1. Use the monitor command "Format" to format an unused diskette (refer to Section 4.1.13 for a complete description of the command). For example:

```
> fo
```

2. Remove the newly formatted diskette from the floppy unit, and place the master system diskette in the unit.

3. Use the monitor command "Copy", with the "partial" option enabled, to copy block Ø (the directory block) from the master system diskette to the newly formatted diskette, as follows:

```
> co -p
First block ? Ø
Last block ? Ø
```

Refer to Section 4.1.6 for a complete description of the "Copy" command. The copy_disk routine will prompt for the first and last block numbers of the copy, and will indicate when to change diskettes. For this operation, both first and last block numbers should be Ø.

4. Repeat the "Copy" command, with the "partial" option enabled, to copy the monitor overlay routines, configuration tables, clearinghouse tables and statistics data structures from the master system diskette to the new diskette. Since all these are stored in contiguous blocks, a single copy operation will suffice, as follows:

```
> co -p
First block ? 19Ø
Last block ? 27F
```

5. The actions in step 3 created a directory data structure on the new diskette; next it is necessary to erase any information from the directory data structure. This is done with the "Read", "Change Word" and "Write" commands.

   a. Enter the "Read" command to read block zero of the new diskette into memory location 3ØØØ for a length of 2ØØ. For example:

   ```
   > r Ø 3ØØØ 2ØØ
   ```

   b. Enter the "Change Word" command to change the value stored in location 3ØØØ. The monitor will display the current value of location 3ØØØ and prompt for a new value; enter a zero, followed by a carriage return. The monitor will then display the current value of the next location (3002) and prompt for a new value for that location; enter the "Quit" command to exit the "Change Word" operation. For example:

```
> cw 3000
3000: 0101 ? 0
3002: 01F0 ? Q
```

c. Enter the "Write" command to write the new value from memory location 3000 back to block 0 on the new diskette for a length of 200, as follows:

```
> w 0 3000 200
```

6. Use the monitor command "Put" to save the recently down-loaded software from memory to the new diskette, as follows:

```
> p 30000 0 1
```

The newly created diskette is now ready to be used to boot the ESPL system.

Note that the save operation in step 6 is accomplished via the "Put" command rather than the "Write" command; this is done for several reasons:

- o The "Put" command automatically updates the appropriate directory structure on the diskette; "Write" does not update the directory.
- o The "Put" command prevents accidental overwriting; if the starting block number or file identifier specified by the user already has data written in it, "Put" generates an error message, aborts the operation and returns to the monitor.

Creating a Backup Diskette

To create a backup copy of a master diskette, use the monitor command "Copy" (refer to Section 4.1.6) with no options enabled. The monitor will assume that a disk format operation is to be performed first, so the "Format" command is unnecessary. The copy routine issues prompts when the master disk should be removed and the backup disk placed in the unit.

4.0 MCPU MONITOR

The MCPU monitor provides interactive access to utilities for obtaining hardware diagnostics, debugging an ESPL system, or performing system generation and floppy disk operations.

Hardware diagnostics are described fully in the ESPL Hardware Technical Reference Manual. Section 4.1 of this manual provides descriptions of the MCPU monitor commands, which include a full set of debugging aids. Section 4.2 describes the monitor error and exception messages that may be displayed on the console terminal. Section 4.3 describes program access to the monitor exception trap vectors. Section 4.4 describes the Sysgen program, and Section 4.5 summarizes the floppy disk operations available through interactive monitor commands.

4.1 Monitor Commands

This section describes the MCPU monitor commands.

4.1.1 Breakpoint Command

Syntax: BR

Description: This command is used to set a new breakpoint. The monitor displays the current breakpoint address and prompts for a new breakpoint address. The address specified should be on an instruction boundary.

4.1.2 Boot Command

Syntax: BT <file>

Description: This command is used to boot the specified file from the diskette. Execution begins at location 3000 (hexadecimal). File identifiers are hexadecimal values in the range 0 through 1F.

4.1.3 Change Address Register Command

Syntax: CA <n>

Description: This command is used to display (and optionally change) the contents of the specified address register. If <n> is omitted, the monitor assumes a default value of 0. The monitor displays all 32 bits of the register, then prompts for a new value. To change the current value, enter a new hexadecimal value. If the value specified is fewer than 32 bits, the upper bits will be set to 0. To display the next address register, enter a carriage return. To return to the monitor, enter the Quit command (q).

4.1.4 Change Byte Command

Syntax: CB <address>

Description: This command is used to display (and optionally change) the contents of the byte at the specified address. The monitor displays the current value, then prompts for a new value. To change the value, enter a new hexadecimal value; to display the next location, enter a carriage return. To return to the monitor, enter the Quit command (q).

4.1.5 Change Data Register Command

Syntax: CD <n>

Description: This command is used to display (and optionally change) the contents of the specified data register. If <n> is omitted, the monitor assumes a default value of Ø. The monitor displays all 32 bits of the register, then prompts for a new value. If the new value specified is less than 32 bits, the upper bits will be set to Ø. To display the next data register, enter a carriage return. To return to the monitor, enter the Quit command (q).

4.1.6 Copy Diskette Command

Syntax: CO -<option> <#copies>

Description: This command is used to copy diskettes. Three options are available. The "v" option performs the copy without verification. The "f" option performs the copy without first formatting the target diskette. The "p" option performs a partial copy (and assumes the "f" option); the monitor prompts for the desired hexadecimal block numbers, then copies only the specified portion of the diskette. Note that if one or more options are specified, the first option must be preceded by a single hyphen (-). The "#copies" parameter is used to specify the decimal number of copies desired.

The monitor prompts the user to insert the source diskette, then the target diskette(s). To indicate that the appropriate diskette is in place, enter a carriage return. To abort the program, press the <BREAK> key.

The CO command may only be run immediately after a system reset; it will not execute properly if normal system code has been running since the most recent reset. In addition, because the command runs as an overlay routine, a diskette containing the copy_disk routine must be in place when the command is entered, or the routine must have already been run once immediately prior to the current run.

4.1.7 Change Process Command

Syntax: CP

Description: This command is used to display (and optionally change) the contents of the User Stack (US), Status Register (SR) and Program Counter (PC). The monitor displays the current contents of each, and prompts for a new value. To change the current value, enter a new hexadecimal value; to continue to the next display, enter a carriage return. To return to the monitor, enter the Quit command (q).

4.1.8 Change Word Command

Syntax: CW <address>

Description: This command is used to display (and optionally change) the contents of the word beginning at the specified address. Odd addresses are rounded down to the next lower even address. The monitor displays the current value, then prompts for a new value. To change the current value, enter a new hexadecimal value; to display the contents of the next location, enter a carriage return. To return to the monitor, enter the Quit command (q).

4.1.9 Disassemble Command

Syntax: DI <address>

Description: This command is used to disassemble memory at the specified address into Motorola assembler code. To continue on to disassemble the next location, enter a carriage return. To return to the monitor, enter the Quit command (q).

The DI command runs as an overlay routine, so a system diskette must be in place when the command is entered.

4.1.10 Display Memory Command

Syntax: DM <address> <length>

Description: This command is used to display a block of bytes beginning at the specified address. The bytes are displayed first in hexadecimal and then in ASCII. Non-ASCII characters are replaced by periods in the display. The "length" parameter specifies (in hexadecimal) how much data will be displayed; if omitted, the monitor assumes a default length of ten (hex) bytes. If the user specifies a length less than ten (or not divisible by ten), the monitor automatically rounds upward to the next ten-byte increment in determining how much data to display.

4.1.11 Display Registers Command

Syntax: DR

Description: This command is used to display all of the processor's internal registers in a short, tabular form.

4.1.12 Fill Byte Command

Syntax: FB <address> <length> <data>

Description: This command is used to insert the specified data, starting at the specified address, for the specified number of bytes.

4.1.13 Format Command

Syntax: FO

Description: This command is used to format both sides of the diskette currently in the disk unit.

** CAUTION **

Before entering the FO command, be sure that the appropriate diskette is in the disk unit. The FO command immediately formats whichever diskette is present, thus erasing all information written on the diskette.

4.1.14 Fill Word Command

Syntax: FW <address> <length> <data>

Description: This command is used to insert the specified data, starting at the specified address, for the specified number of bytes.

4.1.15 Sysgen Command

Syntax: GN

Description: This command is used to execute the Sysgen program. The program provides a simple, menu-driven means of displaying, altering and saving system generation parameter values. The Sysgen procedure for the CS/1 is described in more detail in Section 4.4. Individual Sysgen parameters for the ESPL utilities and for each ESPL service are described in the section(s) of this manual devoted to the appropriate service or utility.

The Sysgen program may only be run immediately after a system reset; it will not execute properly if ESPL product software has been running since the most recent reset. In addition, because the command runs as an overlay routine, a system diskette containing the routine must be in place when the command is entered, or the routine must have already been run once immediately prior to the current run.

4.1.16 Go Command

Syntax: GO <address>

Description: This command is used to start execution of system code at the specified address. If the "address" parameter is omitted, execution begins at the address stored in the Program Counter (PC) register.

4.1.17 Set UART Mode Command

Syntax: I<mode>

Description: This command sets the UART mode. The modes that may be specified are "A", "B" and "T". The "A" mode (the default) indicates that the device connected to the console port is communicating normally with the monitor.

The "B" mode indicates that the device connected to the download port is communicating with the monitor, and the device connected to the console port is disabled. "B" mode should only be used in special circumstances (e.g., when a modem connected to the download port is used to enable remote monitor functions usually performed locally via the console port). In order to change from "B" mode to "A" mode, either the device connected to the download port must transmit an "IA" command, or the device connected to the console port must transmit a <BREAK> signal.

The "T" mode (transparent mode) indicates that the device connected to the console port is communicating directly with the device connected to the download port. In order to change from "T" mode to "A" mode, the user at the device connected to the console port must enter a <CTRL-caret> followed by the letter "c".

** NOTE **

Transparent mode will not work unless the baud rate of the console port is equal to or greater than the baud rate of the download port. Baud rates of the console and download ports are set via MCPU jumpers (refer to the CS/1 User's Guide).

4.1.18 Soft Reset Command

Syntax: K

Description: This command is used to reset the MCPU monitor's stack and internal variables, and is useful after exceptions or other unusual situations (e.g., to reset stack and variables after a series of <BREAK> signals have saved the current context on the stack). Note: this command does not reset the entire machine.

4.1.19 Load Command

Syntax: L<vax-cmd>

Description: This command is most commonly used to download (via the serial download line) an S-record format, hexadecimal file previously generated on the VAX. The parameter "vax-cmd" has the format "OAD <filename>". For example:

    LOAD prog.hex

Note that the "L" portion of the command is interpreted and stripped off by the MCPU monitor, and the remainder (OAD prog.hex) is sent to the VAX. The monitor then waits for data to be transmitted to the ESPL system's download port. OAD is the VAX download utility, which must be present on the VAX. The command causes the file "prog.hex" to be transmitted record by record.

4.1.20 Move Byte Command

Syntax: MB <from> <to> <length>

Description: This command is used to copy a block of memory byte-by-byte from one address to another for a specified number of bytes.

4.1.21 Move Word Command

Syntax: MW <from> <to> <length>

Description: This command is used to copy a block of memory word-by-word from one address to another for a specified number of bytes.

4.1.22 Put Command

Syntax: P <length> <file> <block>

Description: This command is used to save a memory image on the diskette, to be used subsequently by the boot command. The "length" parameter specifies the length (in bytes) of the image to be saved. The "file" parameter specifies the file identifier, which must be a hexadecimal number in the range 0 through 1F. The "block" parameter specifies the starting block of the saved image; block 0 is reserved for directory information. The Put command will not permit existing information to be overwritten; if the specified file or block already contains data, an error message will appear.

Note that the Put command automatically updates disk directory information, while the Write command does not do so.

4.1.23 Read Command

Syntax: R <block> <address> <length>

Description: This command is used to perform a raw read from the diskette. The "block" parameter specifies the starting block to be read from the diskette. Note that block Ø is used only for directory information. The "address" parameter specifies the memory location to which the transfer will be made. The "length" parameter specifies the length (in bytes) of the transfer.

4.1.24 Trace Command

Syntax: T <address>

Description: This command is used to single-step through code. The "address" parameter specifies the starting location of the single-step operation. If omitted, the monitor assumes as default the current value of the Program Counter (PC).

If a system diskette is in place, the instruction to be executed next is first disassembled and displayed; after the instruction is executed, the current contents of the register are dumped.

If the system diskette is not in place, no disassembly is performed and no instruction is displayed; however, after instruction execution the current contents of the registers are dumped.

To continue to the next instruction, enter a carriage return; to return to the monitor, enter the Quit command (q).

4.1.25 Write Command

Syntax: W <block> <address> <length>

Description: This command is used to perform a raw write to the diskette. The "block" parameter specifies the starting block of the area to be written; block Ø is reserved for directory information. The "address" parameter specifies the memory location where the transfer is to start. The "length" parameter specifies the length (in bytes) of the transfer.

Note that a partial-sector write (e.g., a Write command specifying a length less than one sector) causes the remainder of the specified sector to be written as all zeros. In addition, the Write command does not automatically update disk directory information, while the Put command does do so.

4.2 Monitor Error and Exception Messages

The error and exception messages which the monitor is capable of displaying on the console terminal (if one is attached to the ESPL unit's console port) are divided into three categories: bus errors, address errors and exceptions.

4.2.1 Bus Errors

Bus errors occur when an attempt is made to access a nonexistent location in Multibus memory or to write to the monitor's PROM space. Bus error messages have the following format:

```
Bus Error, addr: xx at yy
```

where "xx" is the address to which the erroneous read or write was attempted, and "yy" is the value in the program counter when the attempt was made.

Bus errors are fatal errors. If the MCPU automatic reboot option is enabled, the monitor performs a software reset and then automatically reboots the system. If the automatic reboot option is disabled, control returns to the monitor; no reset or reboot takes place. Refer to the appropriate ESPL product User's Guide for a description of the MCPU automatic reboot option.

4.2.2 Address Errors

Address errors occur when an attempt is made to perform a word or long access starting on an odd address boundary. Address error messages have the following format:

```
Address Error, addr: xx at yy
```

where "xx" is the address to which the erroneous read or write was attempted, and "yy" is the value in the program counter when the attempt was made.

Address errors are fatal errors. If the MCPU automatic reboot option is enabled, the monitor performs a software reset and then automatically reboots the system. If the automatic reboot option is disabled, control returns to the monitor; no reset or reboot takes place. Refer to the appropriate ESPL User's Guide for a description of the MCPU automatic reboot option.

4.2.3 Exceptions

The monitor initializes the microprocessor's 256 exception vector locations to various monitor routines. These routines display exception messages on the console terminal as exceptions occur, unless a user program takes control of the vectors used by the routines.

Exception messages have the following format:

    Exception: xx at yy

where "xx" is a two-character mnemonic for the applicable exception condition and "yy" is the value in the program counter when the exception condition occurred.

Table 4-1 provides a list of mnemonics and brief descriptions for each possible exception condition. For more detailed descriptions of the exceptions, see reference [10].

Fatal exceptions are identified by an asterisk following the mnemonic. The monitor treats fatal exceptions according to the setting of the MCPU automatic reboot option, as described in the previous subsections. When nonfatal exceptions occur, the applicable message appears on the console (if one is attached) and control returns to the monitor.

Note that the Bridge ESPL system code normally uses some of the exception vectors and thus, when these vectors are called, the monitor does not consider the event an exception condition. In the CS/1 product, for example, Multibus Interrupt 2 (M2) is used by the ESB board, and up to four Multibus Interrupts (beginning with M4) are used by the SIO board(s).

Table 4-1 MCPU Monitor Exception Conditions

Mnemonic	Description
II *	Illegal Instruction
ZD *	Zero Divide
Ch *	CHK Instruction
TV *	TRAPV Instruction
Pr *	Privilege Violation
U0 *	Undefined Opcode 1010
U1 *	Undefined Opcode 1111
M0	Multibus Interrupt 0
M1	Multibus Interrupt 1
M2	Multibus Interrupt 2
M3	Multibus Interrupt 3
M4	Multibus Interrupt 4
M5	Multibus Interrupt 5
M6	Multibus Interrupt 6
M7	Multibus Interrupt 7
CA	Channel Attention Interrupt
T2	Timer Channel 2 Interrupt
Tr	TRAP 2 through TRAP C Instruction Vector (TRAPs 1 and D through F are reserved for the monitor's internal use; refer to Section 4.3)
UN	Unknown (this message is used for the remainder of the 68000's vector space)

* - Fatal exception condition

4.3 Program Access to Monitor Trap Vectors

As indicated in Table 4-1, four trap vectors are reserved for the monitor's internal use (TRAP 1, TRAP D, TRAP E and TRAP F). These vectors may also be accessed by OEM software in order to perform various operations (e.g., I/O to the console or to the floppy, reboot or return to the monitor on error, etc.).

TRAP 1 is a break trap vector, called only by the monitor during the processing of a break instruction when a breakpoint is reached. This trap vector is not called by user code.

TRAP D is an automatic reboot trap vector, called when an event occurs which requires that the system reboot automatically. An example of TRAP D usage is provided in Section 4.3.1.

TRAP E is an exit trap vector, called when an event occurs which requires that the process exit and return control to the monitor. An example of TRAP E usage is provided in Section 4.3.2.

TRAP F is used for a seven different functions, defined by the trap type code passed as an argument when the trap is called. Table 4-2 lists the trap type codes and associated functions. Sections 4.3.3 through 4.3.9 provide examples of TRAP F usage.

Table 4-2 TRAP F Type Codes

Code	Function
1	Output to console
2	Get memory size
3	Input from console
4	Reserved for use by monitor only
5	Write to floppy
6	Read from floppy
7	Format floppy
8	Initialize floppy

4.3.1 TRAP D Usage

The following commented assembly-language procedure call illustrates the use of TRAP D to force an automatic system reboot.

```
        .text
        .globl reboot

    TRAPD = /ØD

    exit:
        trap    #TRAPD          | trap to monitor
```

4.3.2 TRAP E Usage

The following commented assembly-language procedure call illustrates the use of TRAP E to return control to the monitor.

```
        .text
        .globl exit

    TRAPE = /ØE

    exit:
        trap    #TRAPE          | trap to monitor
```

4.3.3 TRAP F Output to Console Usage

The following assembly-language procedure call illustrates the use of TRAP F to output a character to the console.

```
        .text
        .globl putchar

    TRAPF = /ØF
    EMT_PUTCHAR = 1             | trap type code

    putchar:
        movl    char,sp@-       | push arg: character (32 bits)
        pea     EMT_PUTCHAR     | push traptype: putchar
        trap    #TRAPF          | trap to monitor
        addql   #8,sp           | pop arg & traptype
        rts
```

4.3.4 TRAP F Memory Size Usage

The following assembly-language procedure call illustrates the use of TRAP F to obtain the address of the last long which can be put into the MPCU's onboard memory.

```
        .text
        .globl getmemsize

    TRAPF = /ØF
    EMT_GETMEMSIZE = 2

    getmemsize:
        pea     EMT_GETMEMSIZE    |push traptype: getmemsize
        trap    #TRAPF            |trap to monitor
        addql   #4,sp             |pop traptype
        rts                       |dØ contains memory end value
```

4.3.5 TRAP F Input from Console Usage

The following assembly-language procedure call illustrates the use of TRAP F to receive a character from the console.

```
        .text
        .globl getchar

    TRAPF = /ØF
    EMT_GETCHAR = 3

    getchar:
        pea     EMT_GETCHAR       |push traptype: getchar
        trap    #TRAPF            |trap to monitor
        addql   #4,sp             |pop traptype
        rts                       |dØ contains character value
```

4.3.6 TRAP F Floppy Write Usage

The following assembly-language procedure call illustrates the use of TRAP F to perform a write to the floppy disk unit. Note that the return codes are the same as those generated by the Floppy Disk I/O Service; refer to Table 6-2 in this manual.

```
        .text
        .globl write

    TRAPF = /ØF
    EMT_WRITE = 5

    write:
        movl    strtblk,sp@-   |push arg: starting block (32 bits)
        movl    ptr,sp@-       |push arg: data pointer (32 bits)
        movl    length,sp@-    |push arg: data length (32 bits)
        pea     EMT_WRITE      |push traptype: write
        trap    #TRAPF         |trap to monitor
        addl    #/1Ø,sp        |pop arg and traptype
        rts                    |dØ contains return code
                                (nonzero on error)
```

4.3.7 TRAP F Floppy Read Usage

The following assembly-language procedure call illustrates the use of TRAP F to perform a read from the floppy disk unit. Note that the return codes are the same as those generated by the Floppy Disk I/O Service; refer to Table 6-2 in this manual.

```
        .text
        .globl read

    TRAPF = /ØF
    EMT_READ = 6

    read:
        movl    strtblk,sp@-   |push arg: starting block (32 bits)
        movl    ptr,sp@-       |push arg: data pointer (32 bits)
        movl    length,sp@-    |push arg: data length (32 bits)
        pea     EMT_READ       |push traptype: write
        trap    #TRAPF         |trap to monitor
        addl    #/1Ø,sp        |pop arg and traptype
        rts                    |dØ contains return code (nonzero
                               on error)
```

4.3.8 TRAP F Floppy Format Usage

The following assembly-language procedure call illustrates the use of TRAP F to perform a floppy formatting operation. Note that the return codes are the same as those generated by the Floppy Disk I/O Service; refer to Table 6-2 in this manual.

```
        .text
        .globl format

    TRAPF = /ØF
    EMT_FORMAT = 7

    format:
        pea     EMT_FORMAT      |push traptype: format
        trap    #TRAPF          |trap to monitor
        addql   #4,sp           |pop traptype
        rts                     |dØ contains return code
                                 (nonzero on error)
```

4.3.9 TRAPF Floppy Initialization Usage

The following assembly-language procedure call illustrates the use of TRAP F to initialize the floppy disk unit. This procedure call must precede any of the other TRAP F floppy procedure calls (i.e., write, read or format). This call causes the monitor to check for the presence of the floppy controller, the floppy unit and a diskette, initializes the controller, and turns on the floppy unit motor. Note that the return codes are the same as those generated by the Floppy Disk I/O Service; refer to Table 6-2 in this manual.

It is the user's responsibility to turn off the floppy motor after the write, read or format operation is complete by writing the value 18ØØ (hexadecimal) to location CØØØØØ. Note that the exit trap (TRAP E) also turns off the floppy motor.

```
        .text
        .globl floppyinit

    TRAPF = /ØF
    EMT_FLOPPY_INIT = 8

    floppyinit:
        pea     EMT_FLOPPY_INIT |push traptype:  floppy init
        trap    #TRAPF          |trap to monitor
        addql   #4,sp           |pop traptype
        rts                     |dØ contains return code
                                 (nonzero on error)
```

4.4 System Generation

This section describes the Sysgen program, which is used to display, change and save sysgem generation parameter values.

System generation parameters differ from configuration parameters in that system generation parameters typically need only be changed once per ESPL product for any given installation; configuration parameters are changed dynamically, often on a per-port basis, and may need to be changed frequently, depending on the requirements of the customer application and the device attached to the port. This section describes the Sysgen program; Sysgen parameters for the ESPL utilities and for each ESPL service are described in the section(s) of this manual devoted to each service or utility. Configuration parameters are described in the User's Guides for each ESPL product.

The Sysgen program is executed from the MCPU monitor, and provides a simple, menu-driven means of performing the following operations:

1. View (display) the current values of Sysgen parameters,
2. View the recommended values of Sysgen parameters,
3. Alter current Sysgen parameter values, or
4. Save Sysgen parameter values on the diskette.

The following subsections briefly describe these operations.

4.4.1 Running Sysgen

To run the Sysgen program, enter the MCPU monitor command "GN". The Sysgen program executes as a monitor overlay routine, so a system diskette must be in place when the command is entered. The main Sysgen menu (a numbered list of options similar to the list above) is then displayed, followed by the prompt:

```
Command number ?
```

At this prompt, enter the number corresponding to the desired option. Note that no carriage return is needed to terminate the entry.

Depending on the number entered, the program either displays a secondary menu or returns to the MCPU monitor.

4.4.2 Displaying Current Sysgen Parameter Values

To display current Sysgen parameter values from the main Sysgen menu, enter the command "1". Note that no carriage return is necessary.

The program displays a numbered list of parameter types, and prompts the user to specify the desired parameter type. To select a parameter type, enter the number corresponding to the type (with no terminating carriage return).

Depending on the number selected, the program either displays all the parameters of the specified type or returns to the main Sysgen menu.

4.4.3 Displaying Recommended Sysgen Parameter Values

The optimum Sysgen parameter values may differ for each ESPL product, or even for each version of product code, depending on the number of ports present in the unit, the maximum number of sessions permitted per port, and the type of traffic supported by each port (e.g., interactive terminal-to-host session, host-to-host file transfer, or X.25 gateway). The "Recommended Settings" menu lists the optimum parameter values for the applicable configurations.

To obtain the list from the main Sysgen menu, enter the command "2" without a terminating carriage return. The program displays a table listing the parameters affected by the various possible combinations, and indicates how to return to the main Sysgen menu.

4.4.4 Altering Sysgen Parameter Values

To alter a Sysgen parameter value from the main Sysgen menu, enter the command "3" with no terminating carriage return. The program displays the secondary "Alter Parameter Values" menu, which is identical to the "View Current Values" menu except for its heading. To select a parameter type, enter the number corresponding to the desired type (with no terminating carriage return).

The program then prints a numbered list of the parameters appropriate to the selected type, prints instructions for returning to the main menu, and prompts the user to type a parameter number.

To alter the value of a parameter, enter the number corresponding to the desired parameter. The program prints the recommended range for the parameter, then prompts for a new value. To alter the current value, enter a new value followed by a carriage return. To leave the current value unchanged, enter a single carriage return; the program will then return to the "Alter Parameter Values" menu.

** NOTE **

In most standard ESPL product installations, there is no need to alter any Sysgen parameters except those listed in the "Recommended Settings" menu, in order to ensure that the basic software configuration is appropriate for the hardware configuration and the requirements of the application. The remaining Sysgen parameters should only be altered if a standard ESPL product has been modified to add custom software or interfaces to the system.

4.4.5 Saving Sysgen Parameter Values

To save altered Sysgen parameter values on the diskette from the main Sysgen menu, enter the command "4" with no terminating carriage return. Before performing the disk write, the program requests confirmation from the user. To save the changed parameters, first ensure that the diskette is in place in the floppy disk unit, then enter "y". The program prints a message confirming the disk write.

To ignore all changes made during the current run (or all changes made since a prior "Save" operation earlier in the current run), enter "n". (Note, however, that the "n" response will not undo a "Save" operation performed earlier in the same run.) The program confirms the fact that no disk write is performed, and returns to the main Sysgen menu.

4.5 Floppy Utilities

The floppy utilities available to the user fall into three categories: utilities accessible interactively via the MCPU monitor, and utilities accessible via procedure calls to monitor trap vectors, and utilities accessible via procedure calls to the Floppy Disk I/O Service. This section briefly describes the first category. Utilities accessible via monitor traps are described in Section 4.3, and utilities accessible via the Floppy Disk I/O Service are described in Section 6.0.

The floppy utilities which may be accessed interactively via the MCPU monitor include the following:

1. Booting the ESPL system code from a file on the diskette (refer to Section 4.1.2),

2. Making duplicate copies of ESPL system diskettes (refer to Section 4.1.6),

3. Formatting diskettes (refer to Section 4.1.13),

4. Copying a memory image of code onto the diskette (refer to Section 4.1.22), or

5. Performing a raw read from the diskette or a raw write to the diskette (refer to Sections 4.1.23 and 4.1.25, respectively). These two functions should only be used when patching code, and must be used with caution.

5.0 KERNEL INTERFACE

This section describes the system resource management provided by the ESPL kernel, and the access to kernel resources available to processes running in an ESPL product.

5.1 Overview

The kernel provides or manages system resources of several types:

- A flexible process management system
- A fast, efficient InterProcess Communication (IPC) facility
- A memory management system
- A centralized facility for the use of a real-time clock
- An interrupt service handler

These resources are available via procedure calls to a process running under the kernel. The data structures used by the kernel to manage resources are described in Section 5.2. The procedure calls made by a process to request resource management from the kernel are described in Sections 5.3 through 5.6. For each procedure call, a "C" procedure declaration is given, as well as definitions of all input and output parameter types, return values and possible errors.

5.1.1 System Initialization

When the system software is loaded into the main processor memory by the bootload device, the entrypoint is a global symbol called "main". The routine "main" initializes all system tables and buffer descriptors, and makes queues of available process control blocks, storage blocks, mailboxes, semaphores, etc. The routine then creates the single initial process.

This process, called "init", acts as the parent process and creates the first instances of the system processes, based on information contained in the system initialization table "sysinit". (Refer to Section 5.2.1 for a description of this data structure.) The init process registers the mailboxes allocated to the new system processes in the table of well-known mailboxes, then lowers its own priority and allows the system processes to begin execution. At this point, init becomes the idle process and runs when no other process has sufficient resources to run. It is the lowest priority process in the system, and is always on the ready list.

After initialization, the newly-created system processes may use library routines to insert mailbox names into the table of well-known mailboxes or to search the table. Processes may register different mailboxes with this table, differentiating them by string name. A mailbox can be looked up by its string name.

5.1.2 Process Scheduling

The scheduling algorithm used by the kernel is round-robin, prioritized scheduling with preemption based on availability of resources and presence of messages. When a process is created, it has a priority. This priority, together with a ready state, puts the process on a ready queue. The processes are dispatched from the highest priority ready queue that has any process linked to it.

Each process is either on the ready queue, waiting at a semaphore, or waiting for a message. When the resource associated with the semaphore becomes available (or the message arrives), the waiting process is given the resource and graduated to the end of the ready queue of processes of like priority. As the currently running process requests resources, it may be queued onto a semaphore wait queue or marked waiting for a message, and the scheduling of the next ready process of highest priority takes place. On the other hand, if the process gives up a semaphore or sends a message which makes a process of higher priority runnable, then that higher priority process will be run and the first process will be linked to the front of the ready queue of its own priority.

5.1.3 Mail Scheduling

When a message is sent to a mailbox, it is linked into a circular queue associated with the mailbox (in fact, part of the mailbox data structure). There are two priorities for messages, URGENT and NORMAL. An urgent message is inserted at the front of the queue, after other URGENT messages, and a normal message is inserted at the end of the queue. The mail is only delivered when the owner of the mailbox makes a receive request on the mailbox. The owner may be blocked waiting for the message, in which case the owner can be made ready at the time of the send; if the owner is of higher priority than the sender, the owner will be run and the sender returned to the front of the sender's ready queue.

Multiple mailboxes are implemented so that a process may demultiplex its messages based on the mailbox receiving the mail. A process can take advantage of this feature by setting up separate mailboxes for its communicants to send data and control messages, as well as a special mailbox for emergency messages.

Messages must always be built in memory obtained from free storage. Also, the same message should not be sent to more than one mailbox, since messages are linked rather than copied.

5.1.4 Memory Management

Within the Bridge kernel, memory is viewed in two ways: storage memory, and buffer memory. The major data processing tasks in the system are protocol processes. On reception of a packet, the protocol process need look only at the beginning of a packet, perform some function based on the header information, then pass the packet up to the next level of protocol with the header stripped off. The data portion of the packet is not of interest except in a few cases. During transmission of a packet, each protocol process needs to prepend a header to the data portion. The successive data encapsulation with headers can be done with a preallocated prologue at the beginning of the buffer, but the size and number of headers is not known at allocation time.

Copying data is something to be avoided. If a protocol layer guarantees reliable transmission, it must retain a copy of the packet it sends to the next lower layer. And if one network has a smaller maximum packet size, or a connection has a smaller packet size, then the splitting up of packets should be made an easy and centrally-controlled function in the system. In addition, each new packet resulting from a split needs its own header, and needs to be kept for retransmission if the peer protocol fails to acknowledge its reception. These considerations are motivation for the buffer management scheme described in this manual.

Storage memory comes in fixed sized blocks. Each block is a contiguous string of bytes, beginning at a word boundary. Storage memory can be accessed directly.

Memory is allocated whenever requested if there is enough free memory. If a process does not get the memory it requests, it may try to alleviate the buffer shortage problem by freeing up any buffers or storage of which it has control. Otherwise, if it has nothing else to do until memory is available, it can create an alarm to wake itself up to try again later.

5.1.5 ESB Shared Memory

The ESB shared memory addresses are in Multibus memory relative to the MCPU. As the MCPU views this memory, the ESB resides in a single 256K-byte block which is one of four 256K-byte block partitions of the address range from 1M to 2M. However, the ESB CPU only decodes the low 17 bits of the offset into Multibus memory, so to the ESB each 128K-byte block in the 1M to 2M range is identical to any other.

From the MCPU point of view, the low-order 128K bytes of each 256K-byte block are in a straight access window, while the high-order 128K bytes are in a swapped access window.

The swapped access window is necessitated by the difference between the way bytes are normally ordered by the 68000 in memory and the order in which the DMA transfers bytes to and from the Ethernet. The 68000 normally orders bytes according to Motorola convention in increasing address order from most significant byte to least significant byte (whether singly, within a word or within a long word).

The DMA, on the other hand, uses the Intel convention for byte order. The DMA transfers 16-bit quantities at a time, starting on an even address. Within this 16 bits, the DMA transfers the bits individually, starting with the least significant bit of the byte residing in the higher address. Thus in order to ensure that the DMA transfers properly ordered bytes, a byte swap operation can be performed automatically by the 68000 by writing to memory via the swapped access window. On a read from the Ethernet, the same byte swap operation can be achieved by the 68000 by reading from the swapped access window.

Figure 5-1 illustrates the byte swapping operation. In the illustration, the DMA is shown writing data to ESB shared memory in the order in which it was received from the Ethernet. The 68000 is shown writing to or reading from ESB shared memory both via the straight access window and via the swapped access window, according to how the data is to be used.

Typically, the straight window addresses are used for interprocessor communication and the swapped window addresses are used for data transferred to or from the Ethernet. However, a customer-added OEM board in an ESPL system might need to use the straight access window rather than the swapped access window for Ethernet-bound data, depending on the type of processor and the byte-ordering scheme it uses.

```
==========Ethernet======= -->

-+---------+---------+
 | LSB     | MSB     |__
 | 7...Ø   | 15...8  |  |
-+---------+---------+  |
                        | <-- 16-Bit Transfer
                        |
                        |
                   +----+---+
                   |        |
                   |  DMA   |
                   |        |
                   +----+---+
                        |
      +---<-------->--+
      |
      |
      |
      |                      Swapped
+-----+--+--------+      Access Window       +---------+---------+
| LSB     | MSB     | <---------+-----------> | MSB     | LSB     |
| 7...Ø   | 15...8  |           |             | 15...8  | 7...Ø   |
+---------+---------+           |             +---------+---------+
|                   |       +----+----+       |                   |
|        ESB        |       |         |       |       68ØØØ       |
|      SHARED       |       |  68ØØØ  |       |       MEMORY      |
|      MEMORY       |       |         |       |                   |
|                   |       +----+----+       |                   |
+---------+---------+            |            +---------+---------+
| MSB     | LSB     |            |            | MSB     | LSB     |
| 15...8  | 7...Ø   | <---------+-----------> | 15...8  | 7...Ø   |
+---------|---------+        Straight         +---------+---------+
                          Access Window
```

Notes:

LSB - Least Significant Byte
MSB - Most Significant Byte

Figure 5-1 Byte Swapping in ESB Shared Memory

5.1.6 Semaphore Scheduling

When a process requests a semaphore, if the semaphore is not available then the process is linked into a wait queue whose header is part of the semaphore data structure. When a process is finished using the object protected by the semaphore, it releases the semaphore, thus allowing another process to use the object. At the time the semaphore becomes available, a process waiting for the semaphore gets it, and is linked onto the ready queue of processes of the same priority. If the newly readied process is of higher priority than the current process, the newly readied process runs, and the first process is returned to the front of its ready queue.

A semaphore has a depth associated with it which enumerates the number of similar objects that are governed by the semaphore. The semaphore count is not allowed to go to zero, and if a blocking request is made, the requestor is queued at the semaphore until the resource is again available.

5.1.7 Clock Scheduling

The MCPU contains a clock used as an interval timer. The resolution of the interval timer is one "tic" every 50 milliseconds. The kernel clock structure may be set or read, and alarm messages may be created for wake-up scheduling. The minimum interval that may be specified for an alarm message is one tic; however, a finer resolution of time may be measured using an elapsed time routine provided by the kernel. Alarm messages are created with a priority (URGENT or NORMAL). When the timer interrupt occurs, the interval count on all outstanding alarms is decremented, and the global clock updated. The interval count (or duration of an alarm) is specified as a 32-bit number of milliseconds in the range from 50 milliseconds to 25 days. If the count on an alarm reaches zero, the alarm message is sent to the caller's default mailbox.

The timer interrupt may be turned on/off for debugging purposes. An elapsed time structure, calibrated in seconds and cycles, may be read by the requesting process. A cycle is derived from the timer/counter input clock, divided by 10. Each such cycle is 2.5 microseconds, using the 10Mhz system clock. The kernel uses this elapsed time facility to measure the accumulated execution time in each process.

If the process using an alarm ceases to need the alarm before it has expired, the alarm may be cancelled.

5.1.8 Interrupt Services

Standard interrupt routines on the MCPU include the clock, one or more agents for the data link, and one or more agents for the serial link. An agent may be a process on its own, or it may be a set of subroutines running on behalf of a requesting process. All the agents mentioned above model well as a set of subroutines along with one (or two) interrupt routines. The part of the agent that runs under interrupt control must have an interrupt vector set up for it by the subroutine portion. Also, a mailbox must be set up for the interrupt routine to use for notification of an event and for data associated with it.

To this end, the general structure of an agent is a body of code that handles requests from a client. One of those requests is an initialization request, passing the ID of the mailbox for asynchronous events. Within the initialization code, the agent makes a kernel call to set up the interrupt vector so that dispatching can be done through a centralized facility which saves the machine state and keeps a nesting count. The nesting count is used by the kernel so it will refrain from scheduling decisions should an agent make a kernel call during an interrupt service routine (e.g., when sending a message). Also, the kernel knows which ID to put in the message header because the interrupts all passed through a common point. All interrupt routines are written in "C", use a common stack, and return from interrupt through a common point.

5.1.9 Well-known Mailboxes

THe kernel maintains a globally accessible table in which processes can register their "well-known" mailboxes for initial contact. The "init" process makes the first entries into the table. A table entry consists of a string name and the mailbox ID. Registration of a mailbox requires a string and a mailbox ID. Resolution of the name takes the string and returns the mailbox ID.

The mailboxes registered in this table should be stable (not transient).

5.2 Kernel Data Structures

The kernel uses several types of data structures, including the system initialization table "sysinit", Process Control Blocks (PCBs), mailboxes, semaphores, interrupt facility data structures, clock data structures, storage blocks and buffer descriptors (BDs). The following subsections briefly describe these structures.

5.2.1 System Initialization Table

The system initialization table (known as "sysinit") is read by the "init" parent process. The table contains an entry for each system process that init must create. The format for entries includes the same arguments as those used for the process creation procedure call (described in Section 5.3.1).

For the customer who wishes to add software to an ESPL product, the sysinit table is distributed as part of the ESPL software distribution kit, in the file "cslrlse/integ.test/csl.c". In order to add a new process and instruct init to start it up, the customer must edit and recompile this source file. In order to disable an existing system process (e.g., the Statistics Monitor or the Echo Protocol), the customer must delete or comment out the corresponding entry in the table and recompile the source file.

If a new process is added, it should be added at the end of the table. The order in which existing entries appear in the table is critical, and should not be altered. The following rules must be observed:

1. The entry for the IDP process (idinit) must appear before the entry for the Parent VT process.

2. The entry for the Data Link Network Manager process (eanminit) must appear before the entry for the Statistics Monitor process (sminit).

3. The entry for the Statistics Monitor (SM) process (sminit) must appear before the entries for the Parent VT and Parent SPP processes (pvinit and psinit, respectively).

In addition, note that neither the Ethernet Agent nor the SIO Agent is started up by the init process. Instead, the Ethernet Agent is started by the IDP process, and the SIO Agent is started by the Parent VT process.

5.2.2 Process Control Block

A Process Control Block (PCB) describes a process. The table "p_lookup" is an array of pointers to the PCBs of all the processes that exist on the system.

A PCB is initialized for a process at process creation time. the process is awarded a unique identifier, which is a hybrid structure composed of the p_lookup index of the PCB and a number drawn from an ever-incrementing counter. The process also gets or shares a stack, and the process state is set to "suspended". As the process becomes ready to run, it is graduated to a ready list designated for processes of its priority, and subsequently scheduled to run.

As a process requests a resource, it may choose to block until the resource is available, allowing another process to run. The other process could conceivably release the resource that the blocking process needs. A process can be blocked waiting for a message from one or more mailboxes, or waiting for access to a data structure (semaphore). If the process is waiting for a semaphore, it is linked into a wait queue in the same way it was linked into the ready list. As the resource becomes available, the process is graduated to the ready list and scheduled according to priority.

During its lifetime, a process allocates memory for itself, gives some of this memory away to other processes, and keeps some for private tables, etc. When a process terminates, the kernel knows about mailboxes belonging to the process and about the process stack segments. In order to free all resources absorbed by the terminating process, the kernel must know about any dynamically allocated memory still held by the process. To this end, in the development phase a queue header for a linked list of memory blocks and one for buffer descriptors is kept in the PCB.

5.2.3 Mailbox Data Structures

The kernel's mailbox scheme uses several data structures, including mailboxes, mailbox lists, messages and the well-known mailbox directory.

A mailbox belongs to exactly one process. Associated with the mailbox are a queue of messages, a queue depth and a message count. The sendmsg, receive, and testmbox procedure calls (refer to Section 5.4) access the mailboxes directly. The sendmsg call allows for two priorities of messages (URGENT and NORMAL), with the additional requests of MUSTDELIVER and FAST.

5.2.4 Semaphore Data Structures

When a data structure is shared between two or more processes, there is a need for some way to ensure mutual exclusion on the data structure. If one process tests a variable in the structure and performs some action based on its value, the process must be assured that the value hasn't changed between the time the structure was tested and the time the action was taken. This exclusion is accomplished through the use of semaphores.

If a process wants to share a data structure, it defines a field within the structure to hold the ID of a semaphore, and requests a semaphore from the kernel during runtime. When access to the protected structure is required, the requesting process waits at the semaphore. The wait call will disable interrupts to make a check on the availability of the structure. If the structure is available, the process instructs the kernel to mark the semaphore in use. Interrupts are enabled and the process continues. If the structure is not available, the process may be blocked, queued on a waiting list at the semaphore until the structure is released by the process currently holding the semaphore.

5.2.5 Interrupt Facility Data Structures

Both the interrupt facility and the context switching mechanism use a data structure called a FRAME in which to record the context of a process. Most processor registers are recorded in the FRAME at context switch time and when the process is interrupted. In addition, when a process is interrupted, the temporary registers aØ, al, dØ and dl are saved on the current stack (usually the process's stack). Another data structure used by the interrupt facility is the ITABLE of INTPTR structures. Each structure contains an interrupt handler for one type of interrupt.

Each interrupt routine is "registered" with the kernel, at which time the kernel-supplied interrupt handler is bound to the user-supplied interrupt handler. The interrupt is armed by storing the location of the kernel-supplied interrupt handler into the specific exception vector in low memory. All interrupt handlers are dispatched through a central place in the kernel, so nesting and stack usage can be carefully controlled.

5.2.6 Alarm Messages and Real-Time Clock Data Structures

The real-time clock facility uses a structure called an alarm message when a process requests a "wakeup" service. An alarm is created with a delay interval and a priority. The delay is specified as a 32-bit number of milliseconds, in the range from 50 milliseconds to 25 days of delay. There is an interval timer which interrupts the system every 50 milliseconds (20Hz), at which time the alarm counters are decremented. When one of the counts descends to zero during the decrementation, the message is sent to the requestor's default mailbox.

A process can put itself to sleep for a specified period of time by first requesting an alarm, then blocking on reception of a message from its default mailbox. If a process is waiting for a message, and wants to give up on the message if it doesn't come within a specified time period, the process may wait on two mailboxes. Then, if the expected message is received before the alarm goes off, the alarm may be cancelled.

The current time can be read with two routines, each providing a different accuracy. One routine returns (and another routine sets) a 32-bit number of seconds. The current time is maintained in a "timeb" structure similar to the UNIX timeb structure. In addition, another routine reads the system timer-counter chip (the source of the 20Hz interrupts) and determines the elapsed time, accurate to the cycle (2.5 microseconds). Procedure calls to compute the difference between two exact times are provided, as well as calls to convert time to more meaningful structures or strings.

5.2.7 Storage Block Data Structure

Normal memory is allocated in blocks. At system generation time, free memory is broken up into a reasonable number of blocks of reasonable size and made available to the processes in the system. The number and sizes are sysgenable numbers. There is a header array for the two kinds of memory (private and shared), and the elements in these arrays are structures containing a pointer to the beginning of a FIFO list of storage blocks, plus the size, count, and HIGH/LOW water marks, etc., for the list. The memory blocks themselves will have headers and a pointer back to the free list to which the block belongs.

When a process needs memory for tables or working parameters, it allocates normal private storage from the kernel. If a process wants to send a message to another process, the storage for the message should be allocated from normal private storage. All memory needs aside from network-bound data and headers should be allocated from normal private storage.

5.2.8 Buffer Descriptor Data Structure

A buffer descriptor is used to logically associate several discontiguous segments of a packet, or to define a subsegment of a larger buffer. The buffer itself has a use count, which is incremented every time another buffer descriptor is created that points to the buffer or to any fragment of the buffer. A buffer is fully described by a linked list of these buffer descriptors, which are pointer structures. Each pointer structure has an address and a length, pointing to a contiguous memory block of at least the recorded length. Collectively, the pointer structures in the buffer descriptor define the logical buffer.

Buffer descriptors are required because the data link layer can perform a gather read; from the buffer descriptor information, the data link layer can locate and deliver the discontiguous pieces to the physical layer as an uninterrupted stream of bits.

Copying data is a costly task which downgrades performance. The ability to chain headers onto data at each layer of protocol during transmission is key to the notion of buffer descriptors, and the ability of reliable protocol layers to retain an image of the packet (in case a retransmission is necessary) depends on this logical view of a buffer. The kernel is aware of buffer descriptors, and can make a copy of an original buffer descriptor for a process if necessary.

The buffers themselves are allocated from storage by the kernel, and ownership is transferred to the buffer allocator by linking the buffer into a circular list in the structure containing the buffer descriptor.

5.3 Process Management Procedure Calls

The ESPL architecture views protocol entities as separate processes. For some protocols (e.g., IDP), a single process is sufficient; for other protocols, there must be a separate process for each session. As sessions are established and disconnected, processes are dynamically created and deleted. Processes of the same protocol type share code, but each possesses its own IPC mailboxes (maintained on its behalf by the kernel) and dynamic data structures. There are two types of processes: those that share a stack with all other processes of the same priority, and those that have their own stack.

The following subsections describe the procedure calls used in process management.

5.3.1 The Procreate Call

The procreate procedure call is used for dynamic creation of processes. The parameter "initentry" is the address at which this process will start execution when it becomes ready. The parameter "initarg" can be either a parameter passed by value, or a pointer to a parameter list. The specified process name is placed in the PCB for this process, and the specified priority will be assigned to it. The parameter "mode" specifies whether the process has user or supervisor privilege, and optionally that the process is a shared-stack process. Privilege is a 68000-dependent security mechanism. Supervisor privilege allows a process to execute all instructions and operations; user privilege restricts a process to a subset of instructions and operations. A detailed description of user and supervisor privilege is provided in reference [10], Section 5.3.

Shared stack processes have two entry points: an initial entry point specified by "initentry", and a main entry point specified by "mainentry". The routine "mainentry" has the following arguments:

```
mainentry( msgptr, mboxid )
MSG      *msg;
MBID     mboxid;
```

Shared-stack processes must never issue blocking kernel calls, such as semawait, breceive or sched, even during initialization. Shared-stack processes return back to the kernel on completion of both initialization and message processing, and the kernel calls them again at mainentry to process the next message.

Non-shared-stack processes have a single entry point, specified by the parameter "initentry". These processes never return back to the kernel from initentry until they are ready to exit.

All processes are created in the suspended state.

The kernel returns either a pointer to the new process's PCB, or an error code if the request fails. Each new process is created with one default mailbox, whose mailbox ID may be obtained via the MYMBID procedure call (refer to Section 5.3.6).

"C" Declaration:

```
PCB *    procreate( initentry, initarg, p_name,
                    priority, mode, mainentry )
int      (*initentry)();
long     initarg;
char *   p_name
ushort   priority;
short    mode
int      (*mainentry)();
```

Input Parameters:

initentry Initial entry point of process.

initarg An argument or argument pointer to initentry.

p_name Pointer to a string consisting of the process name followed by a zero.

priority Process priority, in the range Ø-7 (Ø is highest).

mode Mode (USER/SUPER + SHARESTACK).

mainentry
Main entry point, applicable only if this is a shared-stack process.

Output Parameters:

PCB Pointer to PCB of new process. On error, a NULL pointer is returned.

5.3.2 The Prorun Call

The prorun procedure call causes the specified suspended process to move to the ready list.

"C" Declaration:

```
short   prorun( p )
PCB *   p;
```

Input Parameters:

p Pointer to PCB of process to be made runnable.

Error Codes:

NoError No error detected (Ø).

InvPCB Process does not exist (-1).

ProcWaiting
Process is waiting (-2).

5.3.3 The ProPriority Call

The propriority call changes the priority of the specified process. The highest-priority ready process is then resumed.

A process may use this call to deschedule itself. For example, the SPP parent process (running at a specific priority) can create a child process with the same priority and then lower its own priority, thus effectively descheduling itself and relinquishing the CPU to the child process.

"C" Declaration:

```
short    propriority( p, priority )
PCB *    p;
ushort   priority;
```

Input Parameters:

p — Pointer to the PCB of the process.

priority — New priority of process, in the range Ø through 7.

Error Codes:

NoError — No error detected (Ø).

InvPCB — Invalid PCB (-1).

InvOp — Invalid operation if called by a shared-stack process (-2).

InvPriority — Invalid priority (-2).

5.3.4 The Sched Call

The sched procedure call is used when a process wishes to relinquish the CPU to the next process of the same priority. If no process of the same priority is currently ready, no context switch will occur.

"C" Declaration:

```
sched()
```

Input Parameters: None

Error Codes: None

5.3.5 The MYPID Macro

The MYPID call supplies the calling process with its own process ID. This call is implemented as a macro for efficiency.

"C" Declaration:

```
PID     MYPID()
```

Input Parameters: None

Output Parameters:

PID — Process ID of current process.

Error Codes: None

5.3.6 The MYMBID Macro

The MYMBID call supplies the calling process with the ID of its default mailbox. The call is implemented as a macro for efficiency.

"C" Declaration:

```
MBID    MYMBID()
```

Input Parameters: None

Output Parameters:

mbid — Default mailbox ID of current process.

Error Codes: None

5.3.7 The SETDATA and MYDATA Macros

When a protocol consists of multiple processes, all processes of the same type share the same code. However, if a process needs to have separate data sections, it needs a way of associating the data with its process id rather than with the code space. To do this, a process allocates storage from the kernel via the "allocate" procedure call, then informs the kernel that the storage is a "global" data area. Then, when the process resumes after blocking it can orient itself towards the data by requesting the value of its global data pointer from the kernel.

Two procedure calls are used; one call sets the pointer, and the other returns an already set pointer. These calls are implemented as macros for efficiency.

Set Data Pointer "C" Declaration:

```
SETDATA( dataptr )
ADDRESS dataptr;
```

Input Parameters:

dataptr   Pointer to global data area.

Output Parameters: None

Error Codes: None

Return Data Pointer "C" Declaration:

```
ADDRESS MYDATA()
```

Input Parameters: None

Output Parameters:

dataptr   Pointer to global data area.

Error Codes:

None   Macros typically do not return error codes. Note that if the SETDATA macro has not previously been called, the pointer returned by the MYDATA macro will be invalid.

5.3.8 The Mexit Call

The mexit call is used for voluntary surrender of existence. The kernel will reclaim any dynamic resources belonging to the process (mailboxes, queued messages and stack). Any dynamic memory must be freed by the process before calling mexit.

The process may inform its communicants of its termination; the kernel does not do so. Any process which continues to send the terminated process messages will know by the return code from the sendmsg call that the process no longer exists.

"C" Declaration:

```
mexit()
```

Input Parameters: None

Output Parameters: None

Error Codes: None

5.4 Interprocess Communication Procedure Calls

Processes communicate with the kernel via procedure calls, but they communicate with other processes via messages to a mailbox. Any process may send a message to a mailbox, but only the owner of a mailbox can receive a message from the mailbox.

All messages have a standard message header, which may be followed by any number of bytes of data. The format of the header is as follows:

```
#define MSG struct MSG
MSG {
        MSG     *m_fwd;         /* kernel queue pointers     */
        MSG     *m_bwd;
        PID     m_sender;       /* process ID of sender      */
        BD      *m_bufdes;      /* ptr to buffer descriptor */
        short   m_prio;         /* message priority          */
        short   m_type;         /* user message type         */
```

The following subsections describe the procedure calls used for interprocess communication.

5.4.1 The Mboxcreate Call

The mboxcreate call requests that a mailbox be created for the process. The kernel returns the mailbox identifier. Mailboxes are always created with a state of "on"; in order to not receive from the newly created mailbox, the requestor must issue an mboxoff call.

To create a mailbox with an infinite depth, set the parameter "qdepth" equal to zero.

"C" Declaration:

```
MBID    mboxcreate( qdepth )
ushort  qdepth;
```

Input Parameters:

qdepth   Depth of message queue.

Output Parameters:

mbid   ID of newly created mailbox (or NULL if no more mailboxes are available).

5.4.2 The Mboxdelete Call

The mboxdelete call deletes the specified mailbox. The requesting process must be the owner of the mailbox. Any queued messages will be freed.

"C" Declaration:

```
short   mboxdelete( mboxid )
MBID    mboxid;
```

Input Parameters:

mboxid Identifies the mailbox to be deleted.

Output Parameters:

Error code

Error Codes:

NoError No error detected (Ø).

InvMbox No such mailbox (-1).

NotYourMbox
Requesting process is not mailbox owner (-2).

5.4.3 The Sendmsg Call

The sendmsg call sends the message pointed to by "msgptr" to the mailbox specified by "mboxid".

There are two priorities of message (URGENT and NORMAL), plus two independent delivery requests called MUSTDELIVER and FAST. The priority is specified in the field m_prio as the binary OR of the queue priority and the delivery requests. The MUSTDELIVER request means that even if the mailbox is exactly full, the message must be delivered. However, a subsequent MUSTDELIVER message will fail if the mailbox is still over-full. The FAST request means that the receiving process is queued at the front of its run queue, so that it will be the next process of its priority to be run.

A buffer may be passed to the receiving process at the same time as the message by passing the buffer descriptor of the buffer in the m_bufdes field. Note that this method must not be used to transfer ownership of storage allocated using the "allocate" call; it may only be used to transfer buffer memory referenced by buffer descriptors. If a process wants to inform another process of the location of allocated storage, it must pass the pointer in the text of the message.

If no buffer is passed, the m_bufdes pointer should be set to NULL.

"C" Declaration:

```
short   sendmsg( msgptr, mboxid )
MSG *   msgptr;
MBID    mboxid;
```

Input Parameters:

msgptr   Pointer to the message being sent.

mboxid   ID of mailbox to which message is being sent.

Output Parameters:

Error code

Error Codes:

NoError No error detected (Ø).

InvMbox Invalid mailbox (-1).

InvBD Invalid BD (-2).

InvPriority
Invalid priority (-3).

MBFull Mailbox already full (-4).

5.4.4 The Mboxon Call

The mboxon call turns on the specified mailbox, indicating that the process is willing to receive messages from the mailbox.

"C" Declaration:

```
short   mboxon( mboxid )
MBID    mboxid;
```

Input Parameters:

mboxid Id of mailbox to be turned on.

Output Parameters:

Error code

Error Codes:

NoError No error detected (Ø).

InvMbox Invalid mailbox (-1).

NotYourMbox
Requesting process is not mailbox owner (-2).

5.4.5 The Mboxoff Call

The mboxoff call turns off the specified mailbox, indicating that the process is not willing to receive messages from the mailbox.

"C" Declaration:

```
short   mboxoff( mboxid );
MBID    mboxid;
```

Input Parameters:

mboxid — Id of mailbox to be turned off.

Output Parameters:

Error code

Error Codes:

NoError — No error detected (0).

InvMbox — Invalid mailbox (-1).

NotYourMbox — Requesting process is not mailbox owner (-2).

5.4.6 The Receive Call

The receive call dequeues the first message found in any of the process's mailboxes that are turned on, and returns a message pointer. If no messages are queued, the message "NoMessage" is returned.

In "C", variables may be assigned registers for efficiency. However, a pointer to a register is always NULL; when this call is used, a "msgptr" and a "mboxid" may point to memory (either stack or local) but not to a register.

The message header contains the process ID of the sending process, the priority of the message and the buffer descriptor pointer.

"C" Declaration:

```
short   receive( amsgptr, amboxid )
MSG     **amsgptr;
MBID    *amboxid;
```

Input Parameters:

amboxid   Address for returning message pointer.

amsgptr   Address for returning mailbox ID.

Output Parameters:

Error code

Error Codes:

NoError   No error detected (Ø).

NoMessage No message in mailbox (-3).

5.4.7 The Breceive Call

The breceive call waits for a message to arrive at any of the process's mailboxes which are set to "on", and returns message pointer and mailbox ID. The process is suspended until a message is received.

The error code "InvOp" is returned if a shared-stack process issues this call. The pointers "amsgptr" and "amboxid" must point to memory, either stack or local, and not to a register.

The message header contains the process ID of the sending process, the priority of the message and the buffer descriptor pointer.

"C" Declaration:

```
short   breceive( amsgptr, amboxid )
MSG     **amsgptr;
MBID    *amboxid;
```

Input Parameters:

amsgptr   Address for returning message pointer.

amboxid   Address for returning mailbox ID.

Output Parameters:

Error code

Error Codes:

NoError   No error detected (Ø).

InvOp   Invalid operation (-2).

5.4.8 Blocking Message Reception, Shared Stack Processes

Whenever a shared-stack process returns from an entry point back to the kernel, logic similar to the breceive call is executed to wait for a message to arrive at any of the process's mailboxes that are set to "on". The process is resumed by calling its main entry point with arguments as follows:

```
mainentryt( msgptr, mboxid )
MSG     *msgptr;
MBID    mboxid;
```

This call returns a pointer to the received message, and the id of the mailbox from which "msgptr" was dequeued.

5.4.9 The Notifynfull Call

The notifynfull call is used after a sendmsg has failed because the specified mailbox is full. The call saves the message pointed to by the parameter "msgptr" and sends it to the caller's default mailbox when the mailbox specified by "mboxid" is no longer full.

The message header must contain valid priority and buffer descriptor fields.

If the specified mailbox has a depth of zero (infinite depth), the error code "MBNFull" is returned.

"C" Declaration:

```
short   notifynfull( msgptr, mboxid )
MSG     *msgptr;
MBID    mboxid;
```

Input Parameters:

msgptr — Pointer to message to be sent when mailbox becomes not full.

mboxid — Id of full mailbox.

Output Parameters:

Error code

Error Codes:

NoError — No error detected (Ø).

InvMbox — Invalid mailbox (-1).

InvBD — Invalid buffer descriptor (-2).

MBNFull — Mailbox not full (-3).

5.4.1Ø The Stopnfull Call

The stopnfull call cancels a previous notifynfull request by dequeuing the message from the specified mailbox's notifynfull list, or from the caller's default mailbox if the notification message has already been sent.

If no such message is found, the message "NoMessage" is returned.

"C" Declaration:

```
short   stopnfull( msgptr, mboxid )
MSG *   msgptr;
MBID    mboxid;
```

Input Parameters:

msgptr   Pointer to message to be sent when mailbox becomes not full.

mboxid   Id of full mailbox.

Output Parameters:

Error code

Error Codes:

NoError   No error detected (Ø).

InvMbox   Invalid mailbox (-1).

NoMessage No message found (-3).

5.4.11 The Testmbox Macro

The testmbox macro tests the mailbox specified by "mboxid" for any messages queued at it. The kernel returns the count of messages queued at the box. If a query is made of a mailbox that does not belong to the calling process, the count returned will be negative.

"C" Declaration:

```
short   testmbox( mboxid )
MBID    mboxid;
```

Input Parameters:

mboxid   ID of mailbox being tested.

Output Parameters:

count   Number of messages queued at the mailbox. A negative count is returned on error.

Error Codes:

InvMbox   Invalid mailbox (-1).

NotYourMbox
    Requestor is not the owner of the mailbox (-2).

5.4.12 The Regmbox Call

The regmbox call registers a mailbox ID under a string name in the directory of well-known mailboxes.

"C" Declaration:

```
short   regmbox( pname, mbid )
char *  pname;
MBID    mbid;
```

Input Parameters:

pname — Process name (zero-terminated string, no more than seven characters).

mbid — Mailbox ID.

Output Parameters:

Error code

Error Codes:

NoError — No error detected (0).

InvMbox — Invalid mailbox ID (-1).

NoRoom — No room in registration table (-2).

TooLong — Name too long (-3).

5.4.13 The Resolve Call

The resolve call is used to obtain the mailbox ID represented by a string name in the directory of well-known mailboxes.

"C" Declaration:

```
MBID    resolve( pname )
char *  pname;
```

Input Parameters:

pname — Name to resolve, null terminated string no more than seven characters long.

Output Parameter:

mboxid — Mailbox ID corresponding to resolved name, or error code.

Error Codes:

NoEntry — Mailbox entry not found (-1).

5.5 Semaphore Procedure Calls

Semaphores are used to guarantee a process exclusive access to a shared data structure. Refer to Section 5.2.3 for a description of the use of semaphores.

5.5.1 The Semacreate Call

The semacreate call creates a semaphore. The integer returned is an identifier for the semaphore. The identifier should be stored as part of the data structure it is protecting, so that each process interested in the structure can know the ID of the semaphore protecting it. The kernel manages a queue of processes blocked waiting for the semaphore.

"C" Declaration:

```
SEMAID  semacreate( count )
ushort  count;
```

Input Parameters:

count — Number of processes allowed access to the semaphore (typically one).

Output Parameter:

SemaID — Nonzero semaphore identifier, or error code.

Error Codes:

Error — No semaphores available (-1).

5.5.2 The Sematest Call

The sematest call returns the availability of the semaphore. The result is not guaranteed to remain accurate. The count at the semaphore may change after the semaphore is tested, since there is no semaphore on the semaphore.

"C" Declaration:

```
BOOL    sematest( semaid )
SEMAID  semaid;
```

Input Parameters:

SemaId ID of semaphore.

Output Parameters:

Result Result of test (true = 1, false = Ø).

Error Codes: None

5.5.3 The Semawait Call

The semawait call tests the availability of a semaphore and also blocks the requesting process if necessary. If the semaphore use count is zero, the process is blocked and queued at the semaphore. Otherwise, the semaphore count is decremented and the process continues with access to the data structure granted.

"C" Declaration:

```
short   semawait( semaid )
SEMAID  semaid;
```

Input Parameters:

SemaId ID of semaphore.

Output Parameters:

Error code

Error Codes:

NoError No error detected (Ø).

NoSuchSema Specified semaphore does not exist (-1).

InvOp Invalid if called by shared-stack process (-2).

InvSema Invalid semaphore identifier (-3).

5.5.4 The Semarelease Call

The semarelease call allows other processes waiting at the semaphore to get the semaphore and thus become ready.

"C" Declaration:

```
short   semarelease( semaid )
SEMAID  semaid;
```

Input Parameters:

SemaId   ID of semaphore.

Output Parameters:

Error code

Error Codes:

NoError   No error detected (Ø).

NoSuchSema
Specified semaphore does not exist (-1).

InvSema   Invalid semaphore identifier (-3).

5.6 Memory Management Procedure Calls

Within the Bridge kernel, memory is viewed in two ways: storage memory and buffer memory. The following subsections describe the procedure calls used to manipulate these types of memory.

5.6.1 The Allocate Call

The allocate procedure call requests a block of memory of at least "nbytes" in length from the list of free blocks. The parameter "area" indicates whether the memory is to come from private memory or shared memory.

"C" Declaration:

```
caddr_t allocate( nbytes, area )
short   nbytes;
short   area;
```

Input Parameters:

nbytes	Number of bytes to allocate.
area	Type of memory (PRIVATE or SHARED).

Output Parameters:

ptr	Pointer to memory. A NULL pointer is returned if no free memory is available.

5.6.2 The Mfree Call

The mfree call returns to the list of free blocks the block pointed to by the parameter "memptr".

"C" Declaration:

```
mfree( memptr )
caddr_t memptr;
```

Input Parameters:

memptr   Pointer to the block being freed.

Output Parameters: None

Error Codes: None

5.6.3 The BLOCKLEN Macro

The BLOCKLEN macro returns the length of the specified memory block. A scalar variable "x" is assigned the block length of the block to which "p" points.

"C" Declaration:

```
short   BLOCKLEN( x, p )
short   x;
caddr_t p;
```

Input Parameters:

x   Scalar variable.

p   Pointer.

Output Parameters:

length   Length of block to which "p" points.

Error Codes: None

5.6.4 The Getbuf Call

The getbuf call allocates a buffer of at least the specified length from the list of free buffers. The kernel sets up a buffer descriptor for the buffer and returns a pointer to the descriptor.

If there are no buffers or buffer descriptors, the pointer returned is NULL.

"C" Declaration:

```
BD *     getbuf( length )
short    length;
```

Input Parameters:

length Size of buffer to get.

Output Parameter:

ptr Pointer to the buffer descriptor. A NULL pointer is returned if no buffers or buffer descriptors are available.

5.6.5 The Joinbuf Call

The joinbuf call logically appends buffer 2 (bd2) to buffer 1 (bd1) by pointing the last segment pointer of bd1 at bd2. This makes bd1 point to the entire buffer. The descriptor bd2 must never be used to refer to memory again and should be freed using the freebuf call (refer to Section 5.6.12).

"C" Declaration:

```
short   joinbuf( bd1, bd2 )
BD      *bd1, *bd2;
```

Input Parameters:

bd1 Buffer descriptor for first buffer.

bd2 Buffer descriptor for second buffer.

Output Parameters:

Error code

Error Codes:

NoError No error detected (Ø).

Error Invalid parameter (-1).

5.6.6 The Prependbuf Call

The prependbuf call attempts to add to the physical beginning of the specified buffer the specified number of bytes. This can only be done if there are "length" unused bytes at the beginning of the buffer. If it is impossible to allocate contiguous memory, the kernel will link a buffer of "length" bytes to the current buffer using the getbuf and joinbuf calls to logially add "length" bytes to the beginning of the buffer.

The kernel returns a pointer to the new BD, which may be the same as the old BD, with address and length fields updated.

"C" Declaration:

```
BD *    prependbuf( bd, length )
BD *    bd;
short   length;
```

Input Parameters:

bd	Pointer to buffer descriptor of original buffer.
length	Required additional length to be prepended.

Output Parameters:

ptr	Pointer to new buffer descriptor. A NULL pointer is returned if no buffers or BDs are available.

5.6.7 The Appendbuf Call

The appendbuf call logically appends a buffer by adding space at the end. The kernel attempts to add "length" bytes to the physical end of the buffer. This can only be done if there are "length" unused bytes at the end of the buffer. If it is unable to allocate contiguous memory, the kernel will link a buffer of "length" bytes to the current buffer using the getbuf and joinbuf calls.

In either case, the original buffer descriptor pointer still points to the extended buffer.

"C" Declaration:

```
short   appendbuf( bd, length )
BD *    bd;
short   length;
```

Input Parameters:

bd — Pointer to buffer descriptor of original buffer.

length — Required additional length to be prepended.

Output Parameters:

Error Code

Error Codes:

NoExtend — Bad parameter(s), or no buffers available (-1).

CheapExtend — Append was contiguous (Ø).

ExpensiveExtend — Append required a joinbuf (1).

5.6.8 The Padbuf Call

The padbuf call logically pads a buffer by adding one byte to the length. This can only be done if there is one unused byte at the end of the buffer.

In the current implementation, this call is used only by IDP to pad a buffer to even length.

"C" Declaration:

```
short   padbuf( bd )
BD *    bd;
```

Input Parameters:

bd — Pointer to buffer descriptor of original buffer.

Output Parameters:

Error code

Error Codes:

CheapExtend — Padbuf call was successful, no error detected (Ø).

NoExtend — Bad parameter, or call failed because no buffer space was available (-1).

5.6.9 The Copybuf Call

The copybuf call logically copies the buffer described by "bd" by creating a new buffer descriptor that points to it. This increments the usage count on the buffer pieces. The pointer to the new BD is returned.

If there are no buffer descriptors available, the pointer returned will be NULL.

"C" Declaration:

```
BD *    copybuf( bd )
BD *    bd;
```

Input Parameters:

bd — Pointer to buffer descriptor of original copy.

Output Parameters:

ptr — Pointer to buffer descriptor of new copy. A NULL pointer is returned if no buffers or BDs are available.

5.6.10 The Unprependbuf Call

The unprependbuf call logically deletes "length" bytes from the front of a buffer. The kernel returns the pointer to the new Bd, which may be the same as the input BD.

"C" Declaration:

```
BD *    unprependbuf( bd, length )
BD *    bd;
short   length;
```

Input Parameters:

bd — Pointer to the buffer descriptor.

length — Number of bytes to be deleted.

Output Parameters:

ptr — Pointer to resulting BD (may be same as old BD).

5.6.11 The Unappendbuf Call

The unappendbuf call logically deletes "length" bytes from the end of a buffer.

"C" Declaration:

```
BD *     unappendbuf( bd, length )
BD *     bd;
short    length;
```

Input Parameters:

bd — Pointer to the buffer descriptor.

length — Number of bytes to be deleted.

Output Parameters:

Error code

Error Codes:

NoError — No error detected (Ø).

ErrParm — Bad parameter (-1).

5.6.12 The Freebuf Call

The freebuf call logically frees the buffer by freeing the BD. If this was the only BD with a link to the buffer, the buffer is also released to the list of free buffers.

"C" Declaration:

```
freebuf( bd )
BD *     bd;
```

Input Parameters:

bd — Pointer to the descriptor of the buffer to be freed.

Output Parameters: None

Error Codes: None

5.6.13 The Bufinfo Call

The bufinfo call returns information about the specified buffer at the specified offset into the buffer. The returned status flag indicates whether or not this block is the last block of the buffer.

"C" Declaration:

```
short    bufinfo( bd, offset, addrp, lenp )
BD *     bd;
short    offset;
caddr_t *addrp;
short    *lenp;
```

Input Parameters:

bd — Pointer to the buffer descriptor.

offset — Point in the buffer at which to resolve physical address.

addrp — Address to which to return physical address information.

lenp — Address to which to return length information.

Output Parameter:

status — Indicates whether or not this block is last in buffer (e.g., LASTSEG or NOTLASTSEG), or error code.

Error Codes:

Error — Invalid parameter(s) (-1).

5.6.14 The Buflen Call

The buflen call extracts the buffer segment lengths from all buffer descriptors in the chain of buffer descriptors that start with "bd", and returns the sum of these lengths.

"C" Declaration:

```
short   buflen( bd )
BD *    bd;
```

Input Parameters:

bd — Pointer to a buffer descriptor.

Output Parameters:

number — Total number of bytes in all buffer segments; returns length = zero on error.

Error Codes: None

5.6.15 The BUFADDR, BUFLENC and BUFCONT Macros

Three macros are provided for retrieving buffer information.

The BUFADDR macro returns the address of the start of the buffer.

"C" Declaration:

```
caddr_t BUFADDR( bd )
BD *    bd;
```

The BUFLENC macro returns the length of the first contiguous segment in the buffer.

"C" Declaration:

```
short   BUFLENC( bd )
BD *    bd;
```

The BUFCONT macro returns a boolean variable in answer to the query "Is this buffer contiguous?".

"C" Declaration:

```
BOOL    BUFCONT( bd )
BD *    bd;
```

5.7 Interrupt Service Procedure Calls

The kernel provides the actual interrupt handler for any armed interrupts. However, the user process can register an interrupt, and thereby bind a "C" routine to the interrupt server. The following subsections describe interrupt service procedure calls.

5.7.1 The Disable Call

The disable call disables interrupts on the MC68000.

"C" Declaration:

```
int         disable()
```

Input Parameters: None

Output Parameters:

imask Previous mask value of the SR register.

5.7.2 The Enable Call

The enable call re-enables interrupts, using as an imask value the function return value from a previous disable call.

"C" Declaration:

```
enable( imask )
short      imask;
```

Input Parameters:

imask The interrupt mask assigned to the SR register.

5.7.3 The Regintrpt Call

The regintrpt call registers an interrupt by vector and ID. An interrupt handler is bound at call time to the specified function. Because an ID is established and used while the interrupt handler is running, these routines can use system services.

If messages are sent (using sendmsg) while this interrupt is being served, the message sender is identified as this "intid".

The actual interrupt handler is built in the kernel's ITABLE.

"C" Declaration:

```
regintrpt( intid, funcp, vector )
INTID   intid;
int     (*funcp)();
caddr_t *vector;
```

Input Parameters:

intid	The unique ID of this interrupt.
funcp	The "C" function bound to the interrupt server.
vector	The hardware interrupt vector address (see reference [10], Table 5-2, for a list of legal values).

Output Parameters: None

5.7.4 The MYINTID Macro

This macro is used to obtain the exact interrupt ID recorded by the interrupt dispatch routine. The macro is typically called by interrupt handlers (e.g., the SIO Agent interrupt code) in cases where multiple SIO agents exist in a single system. The resulting ID is used as an index into a shared table, assuring that an agent locates its own entry, not one belonging to another agent.

"C" Declaration:

```
long      MYINTID()
```

Input Parameters: None

Output Parameters:

intid	ID of interrupt currently being serviced.

5.8 Real-Time Clock Procedure Calls

The MCPU contains a real-time clock. The following subsections describe the procedure calls used for time of day and timeout facilities. These calls are typically used only for testing; most normal timer-related functions can be performed by alarm messages.

5.8.1 The Time of Day Macros

Five macros are provided for time of day clock functions.

The GETTIME_secs macro returns the value of the time of day clock, measured in seconds.

"C" Declaration:

```
long        GETTIME_secs()
```

The SETTIME_secs macro sets the time of day clock to the specified time, measured in seconds.

"C" Declaration:

```
SETTIME_secs( time )
long        time;
```

The GETTIME_msec macro returns the value of the millisecond field of the time of day clock. This is an integer in the range Ø-999.

"C" Declaration:

```
short       GETTIME_msec()
```

The GET_MSEC_COUNTER macro returns the number of milliseconds since the system was booted.

"C" Declaration:

```
long        GET_MSEC_COUNTER()
```

The SET_MSEC_COUNTER macro sets the number of milliseconds since the system was booted.

"C" Declaration:

```
SET_MSEC_COUNTER( new )
long        new;
```

5.8.2 The Getetime Call

The getetime call fills in the specified elapsed time (ETIME) structure. A hardware source interval timer is used to provide the highest possible resolution.

Getetime "C" Declaration:

```
getetime( timer )
ETIME   *timer;
```

Input Parameters:

timer   Pointer to the ETIME structure.

Output Parameters: None

The "C" representation of the ETIME structure is as follows:

```
typedef struct etime {
     long    et_seconds;
     long    et_cycles;
     ushort  et_amd2;
     short   et_pad;
} ETIME;
```

The fields in the structure are as follows:

et_seconds Elapsed time, in seconds.

et_cycles Elapsed time, in cycles. There are 20000 cycles per tic, and 400000 cycles per second.

et_amd2 The exact timer count reading from the AMD9513 timer/counter chip, channel 2.

et_pad Pad to make the structure 10 bytes long.

5.8.3 The Delta timer Call

The delta_timer call is used to obtain the difference between two specified timers.

"C" Declaration:

```
delta_timer( dt, timer1, timer2 )
ETIME   *dt, *timer1, *timer2;
```

Input Parameters:

dt — Pointer to the location into which the kernel is to write the resultant delta time, calculated as timer2 minus timer1.

timer1 — Pointer to the first ETIME structure.

timer2 — Pointer to the second ETIME structure.

Output Parameters: None

Error Codes: None

An event can be timed and reported as follows, where e() is the event:

```
{
     ETIME     dt, t1, t2;
     getetime( &t1 );
     e();
     getetime( &t2 );
     delta_timer( &dt, &t1, &t2 );
     print_timer( "time to do e() is :", &dt );
}
```

5.8.4 The Sum timer Call

The sum_timer call adds the elapsed time since the time stored in "timer1" to the time stored in "ttimer".

"C" Declaration:

```
sum_timer( ttimer, timer1 )
ETIME   *ttimer, *timer1;
```

Input Parameters:

ttimer — The total accumulated elapsed time.

timer1 — The current elapsed time.

Output Parameters: None

Error Codes: None

5.8.5 The Print timer Call

The print_timer call displays the contents of the specified timer on the monitor screen. Refer to Section 5.8.3 for an example of how print_timer is used.

```
print_timer ( s, timer )
char *  s;
ETIME * timer;
```

Input Parameters:

s — Descriptive string to be printed.

timer — Elapsed time (in the format hh:mm:ss.m.u).

Output Parameters: None

Error Codes: None

5.8.6 The Setalarm Call

The setalarm message passes a pointer to an alarm message which is sent to the requestor's default mailbox when the alarm goes off.

"C" Declaration:

```
short   setalarm( msgptr )
AMSG    *msgptr;
```

Input Parameters:

msgptr — Pointer to alarm message, or error code.

Error Codes:

NoError — No error detected (Ø).

Error — Invalid timeout interval (-1).

InvBD — Invalid buffer descriptor (-2).

The message itself must have the format:

```
#define AMSG struct AMSG
AMSG {
     MSG        a_msg;          /* message header      */
     long       a_timer;        /* timeout, in msec.   */
}
```

Any amount of data may follow the alarm message header. The message header fields "m_prio" and "m_bufdes" must be valid.

5.8.7 The Testalarm Call

The testalarm call returns the number of milliseconds until the alarm goes off.

"C" Declaration:

```
short   testalarm( msgptr )
AMSG    *msgptr;
```

Input Parameters:

msgptr   Pointer to alarm message.

Output Parameters:

time   Remaining time on alarm.

5.8.8 The Stopalarm Call

The stopalarm call dequeues the specified alarm message from the pending alarm list, or from the client's default mailbox if the alarm message has already been sent.

If no such message is found, an error code is returned.

"C" Declaration:

```
short   stopalarm( msgptr )
AMSG    *msgptr;
```

Input Parameters:

msgptr   Pointer to alarm message.

Output Parameters:

Error code

Error Codes:

NoError   No error detected (0).

NoMessage No such alarm message (-3).

5.8.9 The Clockon, Clockoff and Clockrestore Calls

Three procedure calls provide the ability to turn the interval timer on and off and to restore the previous timer status.

The clockon call turns on the 50-millisecond interrupt mechanism. The clock is initially on. A process need not use this function unless clockoff has been previously called.

"C" Declaration:

```
clockon()
```

The clockoff call turns off the 50-millisecond interrupt mechanism. Pending alarms will not age, and the kernel will not increment the real-time clock.

"C" Declaration:

```
BOOL
clockoff()
```

Input Parameters: None

Output Parameters:

on/off Previous setting of the clock.

The clockrestore call turns the 50-millisecond interrupt mechanism either on or off, depending on the argument passed.

"C" Declaration:

```
clockrestore( onoff )
BOOL        onoff;
```

Input Parameters:

on/off The setting of the clock.

Output Parameters: None

This call is used as follows:

```
csav = clockoff(); {
...
} clockrestore( csav );
```

5.9 Kernel Sysgen Parameters

This section describes the system generation parameters that apply to the kernel and to kernel functions.

5.9.1 Maximum Number of Processes

This parameter specifies the maximum number of separate processes in the system. In the CS/1, this is typically based on 32 VT processes, 48 SPP processes, and one each of the IDP, Parent VT, Parent SPP, Error, Echo, Statistics Manager, DISKIO, Data Link Manager (DLNM), and Clearinghouse processes.

5.9.2 Maximum Number of Mailboxes

This parameter specifies the maximum number of mailboxes the kernel can create. For the CS/1, the default is based on the the maximum number of processes, plus the following additional mailboxes:

2 (each VT process)
2 (each SPP process)
2 (IDP)
1 (Parent SPP)
1 (Error)
2 (Echo)
1 (Statistics Manager)
1 (Clearinghouse

5.9.3 Buffer Allocation

This Sysgen menu allows the user to specify the size and quantity of memory blocks allocated to private memory and to shared memory.

5.9.4 Statistics Manager Sample Interval

This parameter specifies the length (in seconds) of the interval between statistics samples. In the Sysgen menu display, this parameter is listed under the heading "Miscellaneous Parameters".

6.0 FLOPPY DISK I/O SERVICE

This section describes the floppy disk I/O services available to processes running in an ESPL system.

6.1 Overview

The DISKIO module provides an interface between client processes (e.g., VTP/UI, Clearinghouse or Network Management) and the physical disk driver.

The DISKIO module includes a queueing mechanism which ensures that disk requests from multiple client processes are handled one at a time. This prevents race conditions caused by simultaneous read/write requests to the same disk record or file.

6.2 Floppy Disk Interface

Communication between the DISKIO module and client processes is accomplished via four IPC messages. Three of the messages are sent by the client to DISKIO's default mailbox. These messages may contain any one of fourteen I/O-related requests, which are distinguished by message type. The DISKIO process performs the requested function and sends an acknowledgement message to the mailbox specified in the requesting message.

Sections 6.2.1 through 6.2.3 describe the three request messages; Section 6.2.4 describes the acknowledgement message.

Sections 6.2.5 through 6.2.19 describe the fourteen I/O-related requests recognized by DISKIO. The requests fall into three classifications: low-level requests, which deal with physical disk drive activities (e.g., turning the motor on/off, reading/writing sectors); mid-level requests, which deal with reading/writing disk files and records; and high-level requests, which deal primarily with User Interface-related requests (e.g., reading/writing port configuration tables, macros and directories of tables and macros). Note that the message structures used for the requests vary depending on the information needed by DISKIO. Table 6-1 summarizes the requests and the corresponding message types and message structures.

Each message structure uses the common IPC message header, which contains fields for forward and packward pointers, requestor's process ID, pointer to a buffer descriptor, message priority and message type.

In addition, each message structure contains a "dResult" field, which is filled in by DISKIO and supplies a return code for the requested operation. Table 6-2 contains a list of all possible return codes.

Table 6-1 DISKIO Request Summary

Request	Message Type	Message Format
MOTORON	MDI_MOTORON	diskiollmsg
MOTOROFF	MDI_MOTOROFF	diskiollmsg
RSECTOR	MDI_RSECTOR	diskiollmsg
WSECTOR	MDI_WSECTOR	diskiollmsg
OPENFILE	MDI_OPENFILE	diskioopenmsg
CLOSEFILE	MDI_CLOSEFILE	diskiomsg
RRECORD	MDI_RRECORD	diskiomsg
WRECORD	MDI_WRECORD	diskiomsg
RCONF	MDI_RCONF	diskioopenmsg
WCONF	MDI_WCONF	diskioopenmsg
RCONFDIR	MDI_RCONFDIR	diskiomsg
RMACRO	MDI_RMACRO	diskioopenmsg
WMACRO	MDI_WMACRO	diskioopenmsg
RMACRODIR	MDI_RMACRODIR	diskiomsg

Table 6-2 DISKIO Return Code Summary

Return Code	Meaning
0	NoError
1	Replaced
-1	IllegalCmd
-2	SeekError
-3	ReadError
-4	WriteError
-5	NotPresent
-6	WriteProtected
-7	NoMemory
-8	NoFile
-9	DirFull

6.2.1 Diskiomsg Message

This structure is used for the MDISKIOACK message and for the mid- and high-level RRECORD, WRECORD, CLOSEFILE, RCONFDIR and RMACRODIR requests.

"C" Declaration:

```
     struct diskiomsg {
              MSG    dMsg;
              MBID   dReplyMbox;
              short  dResult;
              short  dRecord;
              short  dFileId;
     };
```

Message Parameters:

dMsg
: System portion of message, containing the standard message fields identifying forward and backward message pointers, sending process, applicable BD, message priority and message type.

dReplyMbox
: Mailbox of requesting process.

dResult
: Return code (see Table 6-2).

dRecord
: Disk file record number.

dFileId
: Disk file identifier.

6.2.2 Diskiollmsg Message

This structure is used for the low-level MOTORON, MOTOROFF, RSECTOR, and WSECTOR requests.

"C" Declaration:

```
     struct diskiollmsg {
            MSG    dMsg;
            MBID   dReplyMbox;
            short  dResult;
            short  dSector;
            };
```

Message Parameters:

dMsg — System portion of message, identifying forward and backward message pointers, sending process, applicable BD, message priority and type.

dReplyMbox — Mailbox of sending process.

dResult — Return code (see Table 6-2).

dSector — Disk sector number.

6.2.3 Diskioopenmsg Message

This structure is used in the mid- and high-level OPENFILE, RCONF, WCONF, RMACRO and WMACRO requests.

"C" Declaration:

```
struct diskioopenmsg {
        MSG    dMsg;
        MBID   dReplyMbox;
        short  dResult;
        short  dRecordSize;
        short  dFirstSector;
        char   dFileName[14];
};
```

Message Parameters:

dMsg — System portion of message.

dReplyMbox — Sender's mailbox.

dResult — Return code (see Table 6-2).

dRecordSize — Size of record written or read.

dFirstSector — Number of first sector written or read.

dFileName — Disk file name.

6.2.4 Mdiskioack Acknowledgement Message

This acknowledgement message is sent from the DISKIO process to the default mailbox of the requesting process upon completion of the requested function.

The acknowledgement message uses the same message block as the request; however, only the m_type field, the dResult field, and sometimes the m_bufdes field are utilized. The possible return codes contained in the dResult field are listed in Table 6-1.

The acknowledgement message uses the diskiomsg message structure.

6.2.5 MOTORON Request

This request is used to turn on the motor that rotates the floppy. The request uses the diskiollmsg message structure. The m_bufdes field should be NULL, the mReplyMbox field should be filled in and the dSector field is unused. The head performs a recalibrate operation, the motor is turned on, and an acknowledgment message is returned.

6.2.6 MOTOROFF Request

This request is used to turn off the motor that rotates the floppy. The request uses the diskiollmsg message structure. The m_bufdes field should be NULL, the mReplyMbox field should be filled in and the dSector field is unused. The motor is turned off, and an acknowledgment message is returned.

6.2.7 RSECTOR Request

This request is used for a raw read of an arbitrary-length section of the disk. The request uses the diskiollmsg message structure. The m_bufdes field should point to a buffer descriptor; the data is read into the buffer pointed to by the buffer descriptor. The data length is derived from the length of the first buffer pointed to by the buffer descriptor; data cannot be read into chained BDs.

The mReplyMbox field should be filled in. The dSector field contains the sector number to read, and should be between Ø and 639. The motor is turned on (if it is off), the necessary seek is performed, and the data is read. The returned acknowledgement message passes the buffer descriptor back to the requestor.

6.2.8 WSECTOR Request

This request is used to do a raw write to an arbitrary-length section of the disk. The request uses the diskiollmsg message structure. The m_bufdes field should point to a buffer descriptor; the data is written from the buffer pointed to this buffer descriptor. The data length is derived from the length of the first buffer pointed to by the buffer descriptor; data cannot be written into chained BDs. The m_ReplyMbox field should be filled in. The dSector field contains the sector number to read, and should be between 0 and 639. The motor is turned on (if it is off), the necessary seek is performed, and the data is written. The returned acknowledgement message passes the buffer descriptor back to the requestor.

6.2.9 OPENFILE Request

This request opens a file on which record I/O will later be performed. The request uses the diskioopenmsg message structure. The m_bufdes field should be NULL. The mReplyMbox field should be filled in.

The dRecordSize field contains the record size in bytes. If the record size is between 1 and 256 bytes, more than one record is packed in a sector. If the record size is between 257 and 512 bytes, each record takes up one sector. Record sizes greater than 512 bytes are not implemented.

The dFirstSector field contains the sector number of the first record. It is responsibility of the requestor to properly use a first sector number (an enumerated constant in diskio.h), and limit the range of record numbers to fit in the preallocated areas on the disk (refer to Table 3-1).

The dFilename field will not be used in the initial implementation.

The information is recorded in a private data structure, and a FILEID is assigned. An acknowledgement message is returned, passing the return code in dResult and the FILEID in dFileid.

6.2.10 CLOSEFILE Request

This request is used to close a file. THe request uses the diskiomsg message structure. The m_bufdes field should be NULL, and the dReplyMbox field should be filled in. The dFileid field is the FILEID of the file being closed. The private file structure is deallocated, and an acknowledgement message is returned.

6.2.11 RRECORD Request

This request is used to read a random, single record from an open file. The request uses the diskiomsg message structure. The m_bufdes field should point to a buffer descriptor. The data is read into the buffer pointed to by the buffer descriptor. The data length copied is the record length, but the length of the buffer is not checked. The dReplyMbox field must be filled in. The dRecord field is the record number, and the dFileid field is the FILEID. The file must already be open.

A one-sector cache is maintained; therefore, subsequent sequential reads of records packed in the same sector will not cause disk reads. The data is either found in the cache or read into the cache, then copied into the requestor-supplied buffer. The returned acknowledgement message passes the buffer descriptor back to the requestor.

6.2.12 WRECORD Request

This request is used to write a random, single record into an open file. The request uses the diskiomsg message structure. The m_bufdes field should point to a buffer descriptor and the dReplyMbox field must be filled in. The data is written from the buffer pointed to by the buffer descriptor. The data length copied is equal to the record length, but the length of the buffer is not checked. The dRecord field is the record number, the dFileid field is the FILEID. The file must already be open.

All record writes are "write-through" the cache (i.e., the disk write is always done). If two or more records are packed in each sector, the sector is read into the cache (if not already there), the record is copied from requestor's buffer to the cache, and the sector is written.

After the write is complete, an acknowledgement message is returned, passing the buffer descriptor back to the requestor.

The following example clarifies the limited effectiveness of the one-sector cache in the case of writing sequential records.

```
FileDescriptor
    RecordSize = 128
    FirstSector = 300
```

Request	Action
write record 0	read sector 300, write sector 300
write record 1	write sector 300
write record 2	write sector 300
write record 3	write sector 300
write record 4	read sector 301, write sector 301
write record 5	write sector 301
write record 6	write sector 301
write record 7	write sector 301
write record 8	read sector 302, write sector 302
write record 9	write sector 302
write record 10	write sector 302

6.2.13 RCONF Request

This request is used to read a configuration table. The request uses the diskioopenmsg message structure. The m_bufdes field should be NULL and the dReplyMbox field should be filled in. The dFileName field contains the name of the configuration to be read.

The configuration tables are stored in a record-based (always open) file. The first two sectors of the file are actually a directory of names and starting record numbers.

First, a buffer is allocated; then the configuration table is read and finally copied into the buffer. The returned acknowledgement message passes the buffer descriptor back to the requestor.

6.2.14 WCONF Request

This request is used to write a configuration. The request uses the diskioopenmsg message structure. The m_bufdes field should point to a buffer descriptor and The data is written from the buffer pointed to by the buffer descriptor. The data length copied is the size of a UIBLOCK structure; the length of the buffer is not checked. The dReplyMbox field should be filled in. The dFileName field contains the name of the configuration table to be written.

The configuration tables are stored in a record-based (always open) file. The first two sectors of the file are actually a directory of names and starting record numbers.

The configuration directory is inspected and updated if necessary. The data is written, the buffer is freed, and an acknowledgement message is returned.

6.2.15 RCONFDIR Request

This request is used to read the configuration directory. The request uses the diskiomsg message structure. The m_bufdes field should be NULL and the dReplyMbox field should be filled in. Except for the m_type field (see Table 6-1), the remaining fields in the diskiomsg format are not used.

Buffers are allocated for each of the two directory sectors. The sectors are read and copied into the buffers, and the buffers are joined together. The returned acknowledgement message passes the linked buffer descriptor back to the requestor.

6.2.16 RMACRO Request

This request is used to read a macro. The request uses the diskioopenmsg message structure. The m_bufdes field should be NULL and the dReplyMbox field should be filled in. The dFileName field contains the macro name to be read.

The macros are stored in a record-based (always open) file. The first two sectors of the file are actually a directory of names and starting record numbers.

A buffer is allocated, the macro records are read, then copied into the linked buffers buffer, and an acknowledgement message is returned, passing the buffer descriptor back to the requestor.

6.2.17 WMACRO Request

This message is used to write a macro. The request uses the diskioopenmsg message structure. The m_bufdes field should be a pointer to a buffer descriptor; the data is written from the linked buffers pointed to by the buffer descriptor. The data length copied is the record size or buffer length (null-padded to the record size), which ever is less. The dFileName field contains the macro name to be written. The dReplyMbox field should be filled in.

The macros are stored in a record-based (always open) file. The first two sectors of the file are actually a directory of names and starting record numbers.

The macro directory is inspected and updated if necessary. The data records are written, the buffer is freed, and an acknowledgement message is returned.

6.2.18 RMACRODIR Request

This request is used to read the macro directory. The request uses the diskiomsg message structure. The m_bufdes field should be NULL and the dReplyMbox field should be filled in. Except for the m_type field, the remaining fields of the diskiomsg format are not used.

Buffers are allocated for each of the two directory sectors. The sectors are read and copied into the buffers, and the buffers are joined together. The returned acknowledgement message passes the linked buffer descriptor back to the requestor.

12:30:29 25 JAN 1984

BR........	IS:......................................
BDE	BRIDGED DEVELOPMENT ENVIRONMENT PACKAGE
CC68	MODIFIED FOR 68000, UNIX 'C' COMPILER
CCP	CENTRAL COMM PROCESSOR
EBA	ETHERNET BACKPLANE ATTACHMENT
EDP	ETHERNET DATALINK (XNS LEVEL 0)
ESB	ETHERNET SHARED BUFFER
ESPL	ETHERNET SYSTEM PRODUCT LINE
ETI	ETHERNET TRANSCEIVER INTERFACE
FDC	FLEXIBLE DISK CONTROLLER
ICE	IN-CIRCUIT EMULATOR
IDP	INTER-NETWORK DATAGRAM PROTOCOL (XNS LEV EL 1)
MCPU	MAIN CPU
PI	PROGRAM INTERFACE
SBA	SERIAL BACKPLANE ATTACHMENT
SDD	SERIAL DEVICE DRIVER
SIO	SERIAL I/O
SPP	SEQUENCED PACKET PROTOCOL (XNS LEVEL 2)
UI	USER/HOST INTERFACE
VTM	VIRTUAL TERMINAL MONITOR
VTP	VIRTUAL TERMINAL PROTOCOL
XNS	XEROX NETWORK SYSTEM PROTOCOL

21 ITEMS LISTED.

Bridge
Communications
Inc.

Bridge Communications, Inc.
10440 Bubb Road
Cupertino, California 95014
408/446-2981