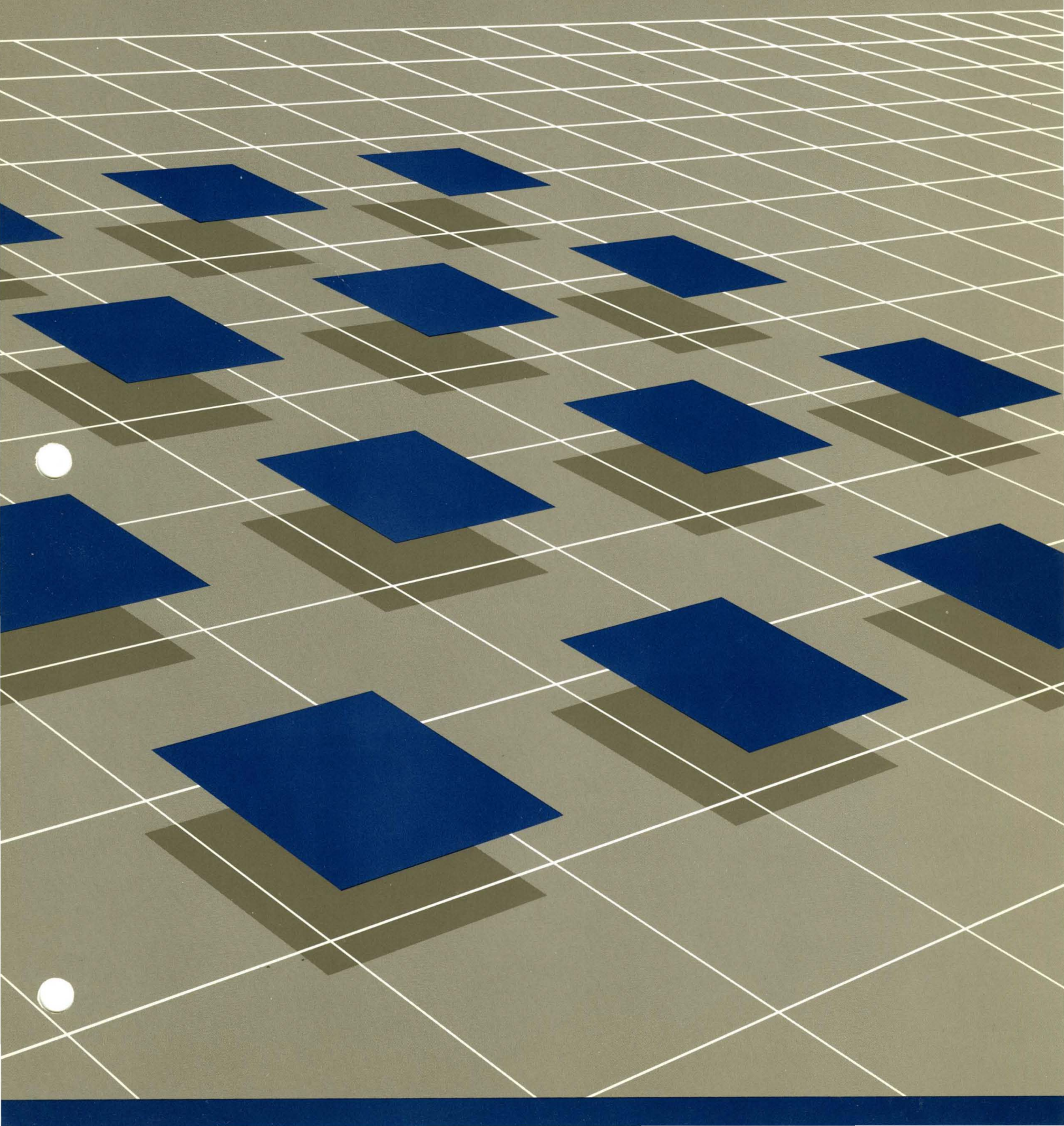
Bridge
Communications
Inc.

Ethernet System Product Line
Software Technical Reference Manual
Volume Three -- Drivers and Firmware



BRIDGE COMMUNICATIONS, INC.

ETHERNET SYSTEM PRODUCT LINE

SOFTWARE TECHNICAL REFERENCE MANUAL

VOLUME THREE -- DRIVERS AND FIRMWARE

09-0018-00

July, 1983

Bridge Communications, Inc., reserves the right to revise this publication, and to make changes in content from time to time without obligation on the part of Bridge Communications to provide notification of such revision or change.

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
09-0018-00	07/83	First Release	All

PREFACE

The Ethernet System Product Line (ESPL) Software Reference Manual provides the Bridge Communications customer with the information necessary to add software to a Bridge ESPL product.

The publication 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.

6. The reader should be familiar with the material presented in Volume One of this manual (particularly Section 5.Ø, the Kernel Interface).

Volume Three of the Software Technical Reference Manual is grouped in three major sections whose contents are as follows:

Section 1.Ø - Introduction: Provides an overview of the three volumes of the Software Technical Reference Manual and describes their purpose and scope.

Section 2.Ø - Data Link Service: Describes the Data Link Service, which provides a means of sending packets compatible with the Ethernet Version 1.Ø Specification. The interfaces with higher-level protocol layers are defined.

Section 3.Ø - Serial I/O Interface: Describes the Serial I/O (SIO) interface between the serial I/O driver and software on the Main CPU board.

Volume One of this manual describes the ESPL software architecture, development environment, MCPU monitor, operating system and floppy disk interface. Volume Two describes the high-level protocols used in the ESPL.

*UNIX is a Trademark of Bell Laboratories.

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 One and Two, 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 MEX68KECB/D2 (Motorola Corporation, 1982)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

The ESPL Software Reference Manual provides the Bridge Communications OEM-level 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 for the sophisticated user (e.g., the Network Manager).

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

The Software Technical Reference Manual is divided into three volumes. Volume One 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 (this manual) describes the ESPL drivers and firmware.

2.0 DATA LINK SERVICE

This section describes the Data Link Service. Section 2.1 provides an overview of the service and the functions it performs, and Section 2.2 describes the interface between the Data Link firmware (residing on the ESB board) and its agent (residing on the MCPU board). Section 2.3 describes the interfaces between the Data Link Service and its level one client protocols. Section 2.4 indicates the Ethernet peer protocols with which the Data Link Service communicates.

2.1 Overview

The data link layer is the lowest layer of protocol in the network. Its primary functions are transmitting and receiving packets, keeping statistics about network traffic, packet characteristics and errors, and supporting diagnostic aids (including power-on diagnostics and higher-level testing). The majority of the data link protocol resides as firmware on the Ethernet Shared Buffer (ESB) board. This firmware is known as the ESB firmware. In addition, an ESB Agent (EA) module residing on the MCPU acts as a driver for the Ethernet controller, and provides an interface to the data link layer for IDP or any other client layer.

2.1.1 ESB Firmware and Software

There are two operational units of the ESB firmware: the Command Unit (CU) and the Receive Unit (RU). The natural division into the two units is synchronous versus asynchronous. The CU handles the communication with the ESB agent, performs any control commands, and initiates any transmit requests. The RU handles packet reception from the Ethernet, providing separation of the data link header from the data portion of the packet. The RU performs data chaining on transmit as well as receive.

2.1.2 ESB Hardware

The FIFO register structure of the ESB DMA provides the capability to scatter-write during packet reception. This is exploited so that the header information can be placed in a separate location from the data. The data portion of the packet could be received into discontiguous memory segments, though it is typically received into a single contiguous block. The reception of back-to-back packets, however, is made possible with the FIFO register and scatter-write capability. Address filtering is performed by the Ethernet Transceiver Interface (ETI) hardware, unless the multicast bit in the address is set; all multicast filtering is done by the ESB firmware. The ESB may be set to receive promiscuously.

The FIFO register structure of the ESB DMA also provides the capability to gather-read on transmission. Packets destined for the Ethernet may be presented to the data link layer as an ordered collection of packet fragments, each fragment characterized by an address in the shared buffer memory plus a byte count. The ESB firmware loads the transmit portion of the FIFO register with the fragment characteristics. The DMA can then be commanded to transmit, at which point it initializes itself with the information about the first fragment and presents the first fragment to the Ethernet Transceiver Interface (ETI). As the data is being transferred, the DMA can prefetch the address and byte count for each fragment in turn, thus presenting the packet to the ETI as a contiguous stream of words.

2.1.3 ESB Control Structure

Commands and buffers are passed to the ESB firmware from the ESB Agent (EA) via the control structure called the System Control Block (SCB). The SCB contains pointers to lists of buffers and commands, as well as fields for error statistics (CRC, alignment, and resource errors). The EA presents new commands to the ESB firmware by acquiring a command block in the Command Block List (CBL), formatting it and logically chaining it to the end of the CBL. The free buffer pool is expanded by appending new buffers onto the end of the buffer list. Some commands are written directly into the SCB (control commands only, with no parameters or data associated with them).

The packets received with CRC or alignment errors are counted by the ESB agent, and then discarded. Resource errors occur when there are not enough buffers for packet reception. If this happens, the receiver shuts down and must be restarted by the agent when the supply of buffers increases again.

In addition, the ESB firmware keeps a count of the following statistics for use by Network Management:

1. Packets too short (Receive)
2. Packets too long (Receive)
3. Number of collisions (Transmit)
4. Packet size histogram (Receive and Transmit)
5. Retransmission histogram (Transmit)
6. Number of packets with CRC errors (Receive)
7. Number of resource errors (Receive)

2.1.4 Initialization

The ESB agent is started up by a client process (usually the IDP process) via the "eainit" procedure call described in Section 2.3.1. IDP obtains the required information from its Service Access Point Table, which contains an entry for every attached network (i.e., each agent with which IDP must be able to communicate) including the entry point for initialization. The table is described in more detail in Volume Two, Section 2.0.

After it has been initialized, the ESB agent initializes the System Control Block (SCB), the Receive Packet Area (RPA), and the placement of any initial commands in the Command Block List (CBL). The ESB firmware finds these data structures through the System Configuration Pointer (SCP), which points to the System Control Block. The SCP is stored at a known location so the firmware can always find it.

It is extremely important that the two processors be coordinated in the initialization effort. The order is agreed upon by the use of interrupt lines and state variables. When the ESB receives a hardware reset (or software reset command), it initializes its RAM and timers, etc. When all device initialization is complete, the firmware waits for a Channel Attention from the agent.

While the firmware waits, the agent initializes the SCP, SCB, RPA, and CBL to a ready state and then sends a Channel Attention to the ESB firmware. On receipt of the first Channel Attention, the ESB firmware reads the SCP from the known location, finds the SCB, and starts executing commands from the Command Block List (CBL).

The ESB 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 128K bytes of each 256K-byte block are in a straight access window, while the high 128K bytes are in a swapped access window (refer to Volume One, Section 5.1.5, for a description of the straight and swapped access windows). In a system containing two ESBs (e.g., an Ethernet-to-Ethernet bridge), from the MCPU's viewpoint ESB1 would have the range 100000-13FFFF, while ESB2 would have 140000-17FFFF.

The agent would initialize ESB1 by writing to 11FF1C and giving it a Channel Attention with a write to 11FFE0, and initialize ESB2 by writing to 15FF1C and giving it a Channel Attention with a write to 15FFE0. Each ESB, however, would look at 11FF1C (or 15FF1C, 19FF1C, 1DFF1C, depending on its ESB number) on receipt of the first Channel Attention to find its SCP and SCB.

The following steps summarize the initialization process:

1. The ESB firmware initializes variables and devices, and waits for the first Channel Attention from the agent.

2. The ESB agent sets up the ESB interrupt handler with the kernel and initializes state variables.

3. The ESB agent initializes the SCP (located at 11FF1C for ESB1, and at 15FF1C for ESB2), SCB, CBL and RPA, and issues the ESB firmware its first Channel Attention by writing to 11FFE0 (for ESB1) or 15FFE0 (for ESB2).

4. The ESB firmware reads the SCP, finds the SCB, performs any control commands in the SCB and possibly starts executing commands in the CBL.

2.1.5 Transmission

The data link layer transmits all packets destined for another station on the network. It must obey the rules of the contention scheme, and must present the data to the physical layer in a contiguous stream of bytes once access to the channel is acquired.

Some of the transmit functions are provided by the hardware and some by the software. The transmit functions provided by the hardware include generating the preamble to the packet, serializing the sequence of octets into a stream of bits, generating the CRC and appending it to the end of the packet, performing Manchester encoding on the bit stream, deferring, enforcing the interpacket gap, detecting collisions, and reinforcing collisions with jamming.

The functions performed by the ESB software during transmission include padding packet to minimum length, initiating transmission of a packet (data link header and data portion), keeping track of the number of retries, performing the truncated binary exponential backoff algorithm, and recording transmission errors.

2.1.6 Reception

Address recognition and carrier sense are hardware functions. The firmware Receive Unit is activated after a successful address match on an incoming packet (i.e., the destination address matches the host address). If the multicast bit in the destination address of the data link header is set, the packet will be received and the multicast addresses will be filtered by the firmware. True broadcast packets are always received.

The ESB agent is responsible for maintaining a pool of free buffers into which packets may be received, and for feeding the characteristics of these buffers to the FIFO register as needed by the DMA to continue reception of the incoming bit stream. The ESB firmware feeds the buffer addresses to the FIFO register, and records the count at the end of a packet full of data.

The ESB agent keeps the buffer pool well-stocked, replacing buffers as they are used up, and maintaining a reasonable margin of the number of buffers available at one time. The goal is to never run out of buffers if at all possible, without absorbing system resources in the process. This might not always be possible if traffic is very heavy; buffers may not be available at the time of the interrupt, and the firmware might need to stop receiving until there are more buffers. A process known as Data Link Network Manager (DLNM) assists the ESB agent in situations such as this. When the firmware reports to the agent (via the SCB field "esb_stat") that it has stopped receiving due to a shortage of buffers or packet descriptors, the interrupt routine tries to restock the pool and sends a message to DLNM to restart the receiver. DLNM then checks the buffer pool, replenishes it if necessary, and issues a control command (via the SCB field "esb_cmd") to resume the receive unit.

2.2 Communication Between Firmware and Agent

The ESB agent runs on behalf of the level one internetwork layer protocol (IDP or a customer's network layer protocol) and provides an interface to the ESB that is based partly on procedure calls (synchronous) and partly on messages from interrupt level (asynchronous).

There are three singly-linked lists the agent stocks: the PD list, the BD list, and the CBL list. The PD list is of finite length; the elements in the list are allocated once during initialization, but never given back to free storage. Instead, their contents (Ethernet header and pointer to first buffer descriptor) are copied into a message block and passed to the appropriate recipient. The PDs are always linked circularly, with the head being pointed to by the SCB and by another pointer held by the agent, and the tail pointed to by a status bit in the PD itself and by another pointer held by the agent. If the ESB firmware runs out of PDs, a resource error is noted in the statistics block of the SCB.

The BDs are allocated at initialization and every time a packet is successfully received by the ESB agent. The method for replenishing the ESB buffer pool is to allocate as many new buffers as are used each time a packet is received. There is no re-use of transmit buffers; they are freed when transmission terminates (whether successfully or not). If the ESB runs out of buffers, a resource error is noted in the statistics block of the SCB.

At any time, the statistics that the ESB firmware is keeping on the network traffic can be read by the Statistics Monitor, so no special communication between processors is necessary.

There are two types of commands to which the ESB firmware responds: control commands (resume, suspend, reset, acknowledgements) and block commands (transmit packet, read host address from prom, add multicast address for receive). The ESB firmware can report asynchronous events to the ESB agent via the status fields in the SCB and an interrupt line.

Figure 2-1 illustrates the flow of commands and messages between the ESB firmware, the ESB agent, and the agent's client processes.

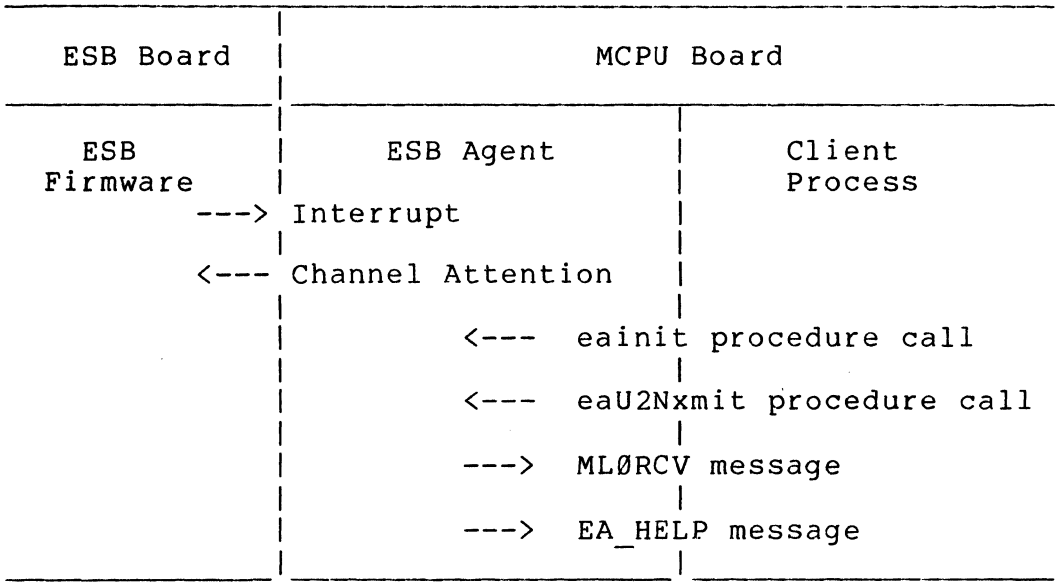


Figure 2-1 Command and Message Flow

2.2.1 Control Commands

Control commands are communicated to the ESB firmware via the SCB. They have no parameters. The ESB agent writes the command into the command field in the SCB and issues a Channel Attention to the ESB firmware. The ESB firmware acknowledges completion of any of these commands by writing a zero into the same field. Two commands (resume and suspend) apply individually to either unit. In addition, the reset command always applies simultaneously to both units.

The command field in the SCB is also used by the ESB agent to acknowledge the following ESB events:

1. Command completed.

2. Packet received.

3. Command Unit becoming not ready.

4. Receive Unit becoming not ready.

Refer to Section 2.2.3.1 for the specific bit values used to specify commands and acknowledge events.

2.2.2 Block Commands

The block commands are chained together in shared memory, and the SCB contains a pointer to the first command in the chain. Each command is composed of two parts: the common portion with the standardized status, link, and command name fields; and the command specific portion, which is different for each command.

2.2.3 Data Structures

The data structures used by the ESB firmware and agent reside in shared ESB memory. Interprocessor communication is conducted via these structures (plus a Channel Attention or a Multibus interrupt, depending on the direction of information flow). The data structures include the System Control Block (SCB), the Command Block List (CBL), the Receive Packet Area (RPA), the Packet Descriptor (PD), the Buffer Descriptor (BD), respectively. In addition, the agent maintains in private memory a data structure called the NET table, which contains pointers into the shared data structures. These data structures are illustrated in Figure 2-2, and described in the subsequent subsections.

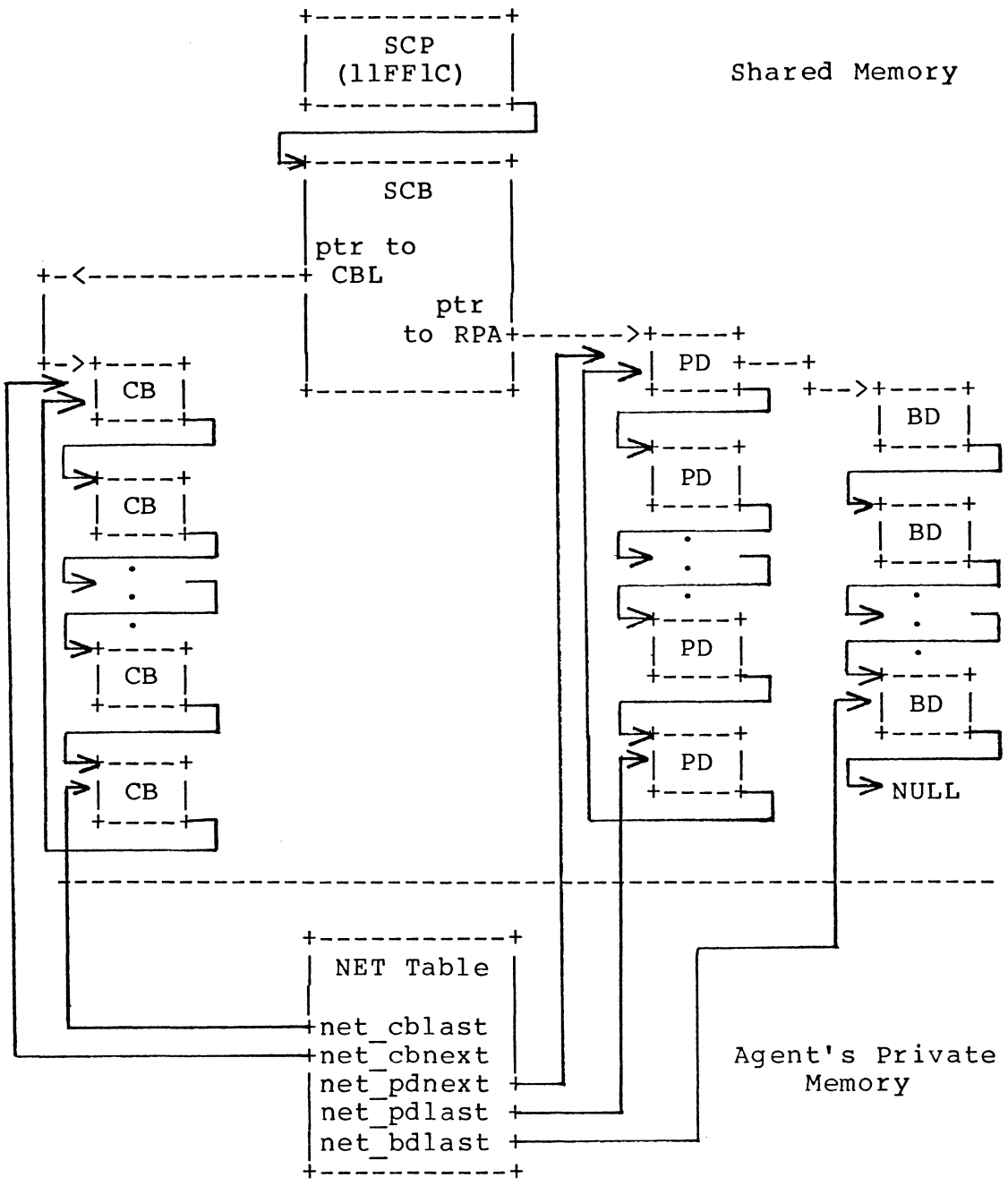


Figure 2-2 ESB Control Structure

2.2.3.1 System Control Block

The "C" representation of the SCB data structure is as follows:

```
#define SCB struct scb        /* definition of SCB                */
SCB {
        short   esb_stat;     /* status of ESB RU and CU
                               * bits  meaning
                               *
                               * 9-15  unused
                               * 8     reset
                               * 7     completed CBL command
                               * 6     packet received
                               * 5     CU not ready
                               * 4     RU not ready
                               * 3     CU Ready/Not Ready
                               * 2     RU Ready/Not Ready
                               * 1-Ø   unused
                               *
                               * bits 4-7 are valid only at the
                               * time of interrupt to agent
                               */
        short   esb_cmd;      /* command field, set by agent,
                               * cleared by ESB CPU
                               * bits  meaning
                               * 8-15  unused
                               * 7     ack command executed
                               * 6     ack packet received
                               * 5     ack CU not ready
                               * 4     ack RU not ready
                               * 3     suspend CU
                               * 2     resume CU
                               * 1     suspend RU
                               * Ø     resume RU
                               */
        CB      *esb_cbl;     /* ptr to command block list        */
        PD      *esb_rpa;     /* ptr to receive packet area       */
        short   esb_PktPend;  /* no. of processed packets (valid
                                 at time of intrpt to MCPU)       */
        short   esb_CmdCmplt; /* no. of cmds completed (valid
                                 at time of intrpt to MCPU)       */
        long    esb_crc;      /* no. crc err since powerup        */
        long    esb_algn;     /* no. align. err since p.u.        */
        long    esb_pderr;    /* no. PD rsrc err since p.u.       */
        long    esb_bderr;    /* no. BD rsrc err since p.u.       */
        long    esb_short;    /* no. pkt-2-short since p.u        */
        long    esb_long;     /* no. pkt-2-long err since p.u.    */
        long    esb_coll;     /* no. collisions since p.u.        */
        short   esb_MaxSize;  /* maximum size packet              */
        short   esb_MinSize;  /* minimum size packet              */
        long    esb_Then;     /* last time reading (not impl)     */
        long    esb_LastTic;  /* clock value at last tic
                               * increment (not implemented)      */
        long    esb_ticsz;    /* size of a tic                    */
```

```
        long    esb_tind;    /* index array esb_tic (wraps
                                around) (not implemented)      */
        long    esb_tic[MAXTIC];  /* pkts per tic (n.impl.)    */
        long    esb_size[MAXSZ];  /* hist of pkt sizes (n.i.)  */
        long    esb_rtry[MAXRETRY];  /* "" no. of retries      */
        long    esb_ipa[MAXGAPS];   /* "" intrpkt arrvls(n.i.) */
        short   esb_ipat;    /* interpkt arrival bin size      */
        short   esb_bytpkt;  /* byte-per-packet bin size       */
        short   esb_retry;   /* max retry value                */
        short   esb_opmode;  /* control to ETI                 */
};
```

2.2.3.2 Command Block List

The Command Block List (CBL) is a singly-linked list of command blocks awaiting execution by the ESB CPU. The header for the list is in the SCB. When the Receive Unit is quiet, the Command Unit is executing commands from the CBL, or waiting for a Channel Attention. The "C" structure of a command block is as follows:

```
#define CB struct cb  /* definition of cmd block    */
CB {
    short   cb_stat;  /* command block status
                       *  bit      meaning
                       *  15-8  unused
                       *  7     currently executing
                       *  6     completed execution
                       *  5,4   reserved
                       *  3-Ø   status:
                       *        Ø   no error (TU, RU)
                       *        1   invalid cmd (TU only)
                       *        2   MAX_RETRY reached
                                    (TU only)
                       *        3   transmit too long
                                    (TU only)
                       */
    short   cb_cmd;   /* command
                       *  bit      meaning
                       *  15-8  unused
                       *  7     last cb on CBL
                       *  6     suspend CU at end
                       *  5     interrupt CPU at end
                       *  4     reserved
                       *  3-Ø   command:
                       *        Ø   no op
                       *        1   READ_HOST
                       *        2   CONFIGURE
                       *        3   MULTICAST_ADDRESS
                       *        4   XMIT
                       *        5   DIAGNOSTIC
                       */
```

```
    CB      *cb_link; /* offset to next cb in list  */

    union  {   /* structures for command-specific parameters */
       READHOST_CB   readhost; /* read host address in prom  */
       CONFIG_CB     confg;   /* config the network intrface */
       MC_ADDR_CB    maddr;  /* rcv pkts w-this m.cast addrs */
       TRANS_CB      trans;            /* transmit packet    */
       DIAG_CB       diag;             /* diagnostic command */
    } cb_parms;


#define READHOST_CB struct readhost_cb
READHOST_CB {
        HOSTADD h_addr;   /* 48-bit station address     */
};

#define CONFIG_CB struct config_cb
CONFIG_CB {
        HOSTADD cf_addr;  /* 48-bit station address     */
};

#define MC_ADDR_CB struct mc_addr_cb
MC_ADDR_CB {
        HOSTADD mc_addr;  /* 48-bit multicast address   */
};

#define TRANS_CB struct trans_cb
TRANS_CB {
        short   trans_bbhdptr; /* DMA-style address     */
        HOSTADD trans_dest;    /* destination of packet */
        HOSTADD trans_src;     /* source of packet      */
        short . trans_type;    /* blue book hdr type    */
        BD      *trans_bdptr;  /* ptr to BD with data   */
};

#define DIAG_CB struct diag_cb
DIAG_CB {
        int (*diag_routine)(); /* ptr to diag. routine  */
};
```

2.2.3.3 Receive Packet Area

The RPA is a linked list of Packet Descriptors (PDs) with an associated list of Buffer Descriptors (BD). The PDs are initially all linked together and empty, except that the first PD has a pointer to the first free BD. The BDs are also linked together. Before reception starts, a pointer to the Ethernet header field in the first PD is loaded into the first FIFO register for receive, and a pointer to the buffer pointed to by the first free BD is loaded into the second FIFO register. When reception begins, the Ethernet header is received into the PD, and the data portion is received into the buffer pointed to by the first BD. If there is more data than the first buffer can hold, the next BD in the linked list is used; the next buffer pointer is loaded into the FIFO register immediately after the DMA fetches the address/count pair, so that it can be ready for a back-to-back receive in which the first packet is a "runt" packet.

2.2.3.4 Packet Descriptor

The structure of the Packet Descriptor is customized for the FIFO register hardware, so that it can be loaded quickly from the structure.

```
#define PD struct pd          /* definition of PD                */
PD {
        short   pd_busy;      /* 0/1 busy word                   */
        short   pd_bbhd;      /* word ptr to blue book hdr       */
        BD      *pd_bdptr;    /* pointer to first BD             */
        short   pd_end;       /* marked 1 => last free PD        */
        PD      *pd_link;     /* pointer to next PD              */
        short   pd_stat;      /* status field
                               * bit   meaning
                               *  0  invalid multicast addr
                               *  1  ran out of buffer space
                               *  2  CRC error
                               *  3  alignment error
                               *  4  packet too long
                               *  5  runt packet
                               *  8-15 unused
                               */
        short   pd_comp;      /* reception completed             */
        short   pd_length;    /* total length of packet          */
        HOSTADD pd_dest;      /* destination address             */
        HOSTADD pd_src;       /* source address                  */
        short   pd_type;      /* packet type field               */
};
```

2.2.3.5 Buffer Descriptor

The data structure used on the ESB to keep track of, link together, split in two and grow buffers is the buffer descriptor (BD). Some conversions and derivations must be done by the ESB agent before passing a packet to the ESB firmware for transmission, and conversely, some fields in the buffer descriptors of the received packets must be initialized by the ESB agent.

The "C" representation of the buffer descriptor data structure is as follows:

```
typedef struct bufdesc {
     union {
             struct esbuffarea bd_esb;
             struct siobuffarea bd_sio;
             struct sppbuffarea bd_spp;
             struct uibuffarea bd_ui;
     } bd_uarea;
     struct bufdesc *bd_next;    /* next BD in chain          */
     caddr_t         bd_address; /* buffer address            */
     BUFP           *bd_buf;     /* pointer to buffer header  */
     short           bd_length;  /* length of this segment    */
     short           bd_flags;   /* flags on buffer descriptor */
} BD;

struct esbuffarea {
     short           use;        /* buffer in use by esb      */
     short           addr;       /* shifted offset into 128K  */
     short           last;       /* marked 1 => last free BD  */
     struct bufdesc *next;       /* ptr to next BD in chain   */
     short           eop;        /* esb, end of packet flag   */
     short           count;      /* count for ESB datalink rcv */
};
```

The ESB DMA uses a word pointer, which supplies the lowest address line as zero, and the highest address line as a one, so the address of the data must be shifted right and truncated to 16 bits to feed the DMA.

```
     bd_esb.addr = (short) ( ((long) bd_address)
```

2.2.3.6 Network Table

The ESB agent maintains an array of data structures containing initialization parameters, network address, client table and various pointers into shared data structures (everything it needs to keep track of the controller). These data structures are called NET structures, whose "C" representation is given below, along with the representation of the subsidiary CLIENT structure, which consists of an array used to store client and type identifiers.

```
#define NET struct net          /* one structure per network    */

NET {
     INTID   net_ident;         /* put on stack for handler     */
     SCB     *net_scbptr;       /* ptr to shared SCB structure  */
     CB      *net_cblast;       /* last formatted               */
     CB      *net_cbnext;       /* next to be recycled          */
     PD      *net_pdlast;       /* last to be processed         */
     PD      *net_pdnext;       /* next to be processed         */
     BD      *net_bdlast;       /* last free                    */
     CLIENT  *net_clients;      /* array of (client,type)       */
     short   net_NextClient;    /* index into client block      */
     HOSTADD net_hostadd;       /* station address              */
     short   net_vec;           /* 68000 vector number          */
     long    net_offset;        /* addr offset for this network */
     short   net_opmode;        /* receive mode                 */
     short   net_cbs;           /* number of CBs                */
     short   net_pds;           /* number of PDs                */
     short   net_bds;           /* number of BDs                */
     short   net_tic;           /* size of basic tic            */
     short   net_max;           /* size of maximum packet       */
     short   net_min;           /* size of minimum packet       */
     short   net_ipg;           /* size of interpkt arrival bin */
     short   net_bpp;           /* size of byte per packet bin  */
     short   net_rtry;          /* maximum retry count          */
     short   net_bdcount;       /* current count of buffers     */
     short   net_EsbCmd;        /* private copy of acks         */
};
#define CLIENT struct client    /* array of (client,type)       */

CLIENT {
     MBID    client_name;       /* client mailbox for packets   */
     short   client_type;       /* this client type             */
};
```

2.3 Client Interface to ESB Agent

Communication between the ESB Agent and its client consists of two procedure calls and two IPC messages, described in the following subsections.

2.3.1 The Eainit Procedure Call

The eainit procedure call must be used by all clients of the ESB Agent The first time it is issued, the data structures are allocated and initialized, and the first Channel Attention is sent to the firmware. Also, on the first and any subsequent calls to this routine, the client-type data is entered in a table. Any received packet of the same specified type will be sent to the corresponding client mailbox.

If the ESB being initialized by this call is the first ESB to be initialized, the ESB's host ID PROM is also read and that address used as the station address. If another ESB is subsequently initialized, its host ID PROM will not be read.

"C" Declaration:

```
eainit ( ident, client, type, ret_parm )
INTID   ident;
MBID    client;
short   type;
long    ret_parm;
```

Input Parameters:

ident	Identifies this attached network.
client	Mailbox to which to send received packets.
type	All packets of this data link type go to client.
ret_parm	This value is passed to the level 1 client with the MLØRCV message as the parameter "earcv_ident".

Output Parameters: None

Error Codes:

NoError	No error detected.
Error	Not enough memory, or invalid interrupt ID.

2.3.2 The EaU2Nxmit Procedure Call

The eaU2Nxmit procedure call is used by clients of the ESB Agent to instruct the Agent to transmit a packet. Assuming the parameter "ident" is valid and there is a free command block in the CBL, the Agent will build a header in the command block using the values specified by the "dest", "src", and "type" parameters in the message, format a transmit command block for the ESB firmware, and issue a Channel Attention to the firmware.

If the destination host is this host, the ESB agent will not send the message out over the Ethernet, but will instead send a message to the proper recipient within the local node.

"C" Declaration:

```
short   eaU2Nxmit (m)
```

Input Parameters:

m	Pointer to message (message type EA_XMITPKT).

The message pointed to by "m" has the format:

```
EA_XMITMSG {
     MSG       eaxmit_sysmsg;
     INTID     eaxmit_ident;
     HOSTADD   eaxmit_eadest;
     HOSTADD   eaxmit_easrc;
     short     eaxmit_eatype;
} ;
```

Message Parameters:

eaxmit_sysmsg	System portion of message (includes BD).
eaxmit_ident	Distinguishes between networks in a bridge.
eaxmit_eadest	Destination address.
eaxmit_easrc	Source address.
eaxmit_eatype	Protocol type.

Output Parameters: Error code

Error Codes:

NoError	No error detected.
Error	No command block available, or invalid ident.

2.3.3 The MLØRCV Message

If a packet has been received when the ESB firmware issues an interrupt to the agent, the interrupt routine will format an MLØRCV message and send it to the appropriate recipient. The appropriate recipient is either a process which called the eainit procedure call with a type parameter matching this packet's type field, or DLNM if the packet is unclaimed. Any buffer that was consumed by the received packet is replaced with a new one at this time, and the replacement is linked to the end of the BD list. The PD with the Ethernet header will be recycled.

"C" Declaration:

```
MLØRCV {
    MSG       earcv_sysmsg;
    long      earcv_ident;
    HOSTADD   earcv_eadest;
    HOSTADD   earcv_easrc;
    short     earcv_eatype;
}
```

Message Parameters:

earcv_sysmsg
: System portion of message (message type MLØRCV).

earcv_ident
: Distinguishes between networks in a bridge. Derived from the eainit procedure call parameter"ret_parm".

earcv_eadest
: Destination address.

earcv_easrc
: Source address.

earcv_eatype
: Protocol type.

2.3.4 The Receiver Restart Message

When the Receive Unit detects a resource error, it shuts down in a graceful manner and must be restarted by a higher level process after the higher-level process has unloaded the full buffers by shipping them off to their appropriate destinations. The agent interrupt service routine makes the initial steps to start the receiver back up by recycling used packet descriptors and replenishing used buffers. Some recoveries are simple, while other recoveries are more complex. For instance, if the last buffer is used, a new "last" buffer must be allocated to prime the list.

There are four fields in the PD and one field in the BD which determine the restart algorithm. The four PD fields pertain to the PD following the last complete packet. The four fields are pd_busy, pd_comp, pd_bdptr, and pd_last. The BD field is bd_lastfree. Table 2-1 represents a simple boolean table which sorts the various scenarios into a small number of cases.

The bd_lastfree field doubles this table and is sampled on the last BD that is used; if bd_lastfree is TRUE, then a new BD must be allocated to start the new chain of free buffers.

The agent first appraises the situation, then sets the pointers correctly, clears fields in partially used PDs and BDs, and sends a message to the higher level process. The higher level process checks the count of BDs on the list, replenishes it if necessary, and sends a control command (RU_RESUME) to the ESB firmware to resume the Receive Unit. The message type for the message is EA_HELP.

Table 2-1 Receiver Restart Algorithm

pd_busy	pd_bdptr	pd_comp	pd_last
FALSE	xxxx	xxxx	xxxx
TRUE	NULL	xxxx	xxxx
TRUE	bdptr	FALSE	xxxx
TRUE	bdptr	TRUE	TRUE

Note: xxxx = don't care.

2.4 Peer Protocol

The Data Link Service is compatible with the data link protocol described in the Ethernet Specification, Version 1.Ø (reference [4]).

3.0 SERIAL I/O MODULE

This section describes the Serial I/O (SIO) Module, which provides communication between an agent process and other processes running on the MCPU, and communication between the agent process and the firmware on the SIO board.

3.1 Overview

The Serial I/O Module is a front-end processor board which provides eight RS-232-C/RS-423 full duplex lines for connecting serial devices to a CS/1. Each line can be independently configured for the type of serial device attached.

The purpose of the SIO board is to alleviate the burden of character interrupt handling in the MCPU, so that the MCPU can be devoted mostly to high-level protocol-processing tasks. Instead of interrupting on each character, the MCPU is interrupted on a buffer of characters. On output, when a buffer of characters has been transmitted, an interrupt is generated to the MCPU. On input, the particular conditions which cause the interrupt to the MCPU depend on the line discipline.

In addition to the basic read and write commands, the SIO board responds to commands for setting the parameters of a line, dynamically changing these parameters, and flow-controlling the line.

The following subsections describe the SIO software and its interfaces in detail.

3.1.1 SIO Software

The SIO software is comprised of firmware residing on the SIO board and an SIO agent (SA) which resides on the MCPU and communicates with the SIO firmware on behalf of the agent's client, normally the Virtual Terminal Process.

The firmware maintains character flow over eight serial lines in a manner which is fair to all lines. It receives requests from the SA, and reports completion of requested actions as well as exceptions to the agent. The agent maintains shared data structures used for communication and control between firmware and agent, allocates memory, and acts as an interface to the SIO's client processes. The agent is capable of interfacing to SIO firmware on up to four SIO boards; demultiplexing is based on port numbers, which are mapped to physical board and line numbers as described in Table 3-1.

3.1.2 SIO Hardware

The SIO board is described in detail in reference [2]. Note that the SIO firmware and agent described in this manual do not utilize all the configuration features supported by the SIO board. These modules' current scope is limited to asynchronous device support, including terminals, host computers and modems.

The first 32 SIO ports, beginning with Ø, are mapped to physical SIO lines as indicated in Table 3-1.

Table 3-1 SIO Line Number/Port Mapping

Port No.	SIO Board No.	Line Number
Ø	1	Ø
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	1	6
7	1	7
8	2	Ø
9	2	1
1Ø	2	2
11	2	3
12	2	4
13	2	5
14	2	6
15	2	7
16	3	Ø
17	3	1
18	3	2
19	3	3
2Ø	3	4
21	3	5
22	3	6
23	3	7
24	4	Ø
25	4	1
26	4	2
27	4	3
28	4	4
29	4	5
3Ø	4	6
31	4	7

3.1.3 Agent/Firmware Control Structure

The firmware is initially set up with an SIO Control Block (SIOCB), a Command Block List (CBL), and a Character Receive Area (CRA). The CRA receives characters into a supply of buffers whose BDs are singly linked in chains. The SIOCB contains pointers to one LINE structure for each port; each LINE structure points to the first BD in a chain. The CBL is a circular list of singly-linked Command Blocks (CBs).

When a buffer is completed with a termination condition commensurate with the particular line discipline, the termination condition is noted in the structure associated with the buffer (BD) and the event is marked in the SIO's private copy of the status for that line. When a command from the CBL is completed, the completion status is noted in the CB and likewise marked in the SIO's private copy of the status.

If the MCPU has acknowledged all previous events from the SIO board (any of the lines may report concurrently), then the firmware can report a new event to the MCPU by copying the status and associated count fields to the SIOCB in shared memory, and issuing a Multibus interrupt to the MCPU. This invokes the SIO agent (SA), which reports these events to the appropriate client and restocks the CRA.

If a BREAK condition is detected, the current read is completed and the status BREAK is noted in the BD associated with the buffer. The same is true of a change in the carrier sense or a receive error condition.

The SIO Agent (SA), residing on the MCPU board, initializes the shared data structures for the firmware by setting up the two list structures used for character handling (the CRA and the CBL) and linking them to the SIOCB. The agent then writes the address of the SIOCB in a prearranged location in shared memory (the range 11FF00 through 11FF0F) for the SIO firmware to read, and issues a Channel Attention to the SIO board (the GO signal).

Upon request of its client (usually VT), the agent formats request blocks for the SIO firmware using a CB in the CBL, and interrupts the firmware via a Channel Attention.

The interrupt-driven portion of the SA handles completed requests from the SIO firmware. If a write request is completed, the agent frees the buffer. For a completed read buffer, the agent sends a message to the appropriate client process and restocks the CRA.

3.1.4 Initialization

The first code that executes on both the SIO board and the MCPU is a multiprocessor power up diagnostic. In addition to memory and devices, the diagnostic tests mutual interrupt capability between the two processors, and in doing so determines both the presence of each SIO board and the firmware type assigned to the higher level code controlling the board. This information is stored on the MCPU in a globally accessible structure. If an OEM customer replaces the higher level code of the SIO firmware and develops a new agent to run with some or all of the existing ESPL software, a different type code must be assigned to the new firmware.

** NOTE **

> Firmware type codes are two-byte codes allocated by Bridge Communications to uniquely identify each different set of higher-level code controlling an SIO board. To obtain a new firmware type code, contact Document Control, Bridge Communications, Inc., 10440 Bubb Road, Cupertino, CA 95014.

Additionally, the OEM firmware and agent must not use the addresses in the range 11FF00 to 11FF0F for the equivalent of the SIOCB pointer. Instead, the addresses from 11FEF0 to 11FEFF should be used as an array of four pointers to the shared structure.

The SIO firmware initializes its own private structures, then waits for the first Channel Attention from the SIO agent.

For each SIO board, the agent allocates all the shared data structures (CBL, CRA, and SIOCB), writes the address of the SIOCB in a known location, then sends the first Channel Attention to the SIO firmware.

3.1.5 Line Configuration

Upon client request (e.g., VT), the agent allocates memory in shared memory, copies the relevant parameters from the UI parameters into the shared memory block, gets the first free CB in the CBL for the SIO line, fills in the command field and parameter field (pointer to the shared memory with the parameters), and issues the Channel Attention to the SIO board.

The SIO firmware interprets the command on the CBL and copies the relevant parts into the Line Control Block (LCB) for that line. Then the UART is initialized according to the parameters that control the physical device. The SIO line is now ready for operation.

3.1.6 Transmission

The client requests transmission by specifying the SIO line number and the buffer descriptor with the data.

The agent gets the pointer to the first free CB in the CBL of the SIO line, and fills in the fields.

The agent uses the control command field in the SIOCB to inform the SIO firmware of the presence of a CB on the list and issues a Channel Attention to the SIO board to draw its attention to the command. The SIO firmware is now responsible for the transmission.

3.1.7 Reception

The agent sets the firmware up with a list of BDs in which to receive characters before the UARTs are even initialized.

The interrupt service portion of the agent receives the Multibus interrupt from the SIO firmware and notes that a read has been completed. The agent sends the buffer to the appropriate data mailbox.

3.2 Communication Between Firmware and Agent

The MCPU and the SIO are loosely coupled processors, each with a private memory space, I/O space and Multibus mastership capability. They exchange information in shared data structures contained in ESB memory. The control mechanisms that govern the exchange consist of a mutual hardware interrupt capability, software control flags and commands. This section describes these mechanisms and the way they are used to achieve processor synchronization.

3.2.1 Channel Attention

The Channel Attention interrupt is used by the SA to interrupt the SIO firmware in order to pass a new command (or set of commands) or to acknowledge receipt of a command completion notice. The Channel Attention is a decoded address interrupt and does not consume a Multibus interrupt level. It is automatically cleared by the SIO board in hardware. There is one Channel Attention interrupt per SIO board. The SA generates a Channel Attention by writing to the locations indicated in Table 3-2.

3.2.2 Multibus Interrupt

The Multibus interrupt is used by the SIO firmware to interrupt the SA upon completion of a command (or set of commands). As its name indicates, it is a real Multibus interrupt. It is cleared by the SIO agent on the MCPU. There is one Multibus interrupt for all SIO boards, and one Multibus interrupt clear address for each SIO board (see Table 3-2).

3.2.3 SIO Reset

The SIO board can be reset by the MCPU. Once reset, the board is held in reset mode until a reset clear is received. The addresses written to for reset and reset clear are listed in Table 3-2.

Table 3-2 SIO Channel Attention, Multibus Interrupt and Reset Addresses

Function	SIO 1 Address	SIO 2 Address	SIO 3 Address	SIO 4 Address
Channel Attention	1E0000	1E8000	1F0000	1F8000
Multibus Interrupt *	E00000	E00000	E00000	E00000
Multibus Interrupt Clear	1E2000	1EA000	1F2000	1FA000
Reset	1E4000	1EC000	1F4000	1FC000
Reset Clear	1E6000	1EE000	1F6000	1FE000

* The same address is used by all boards; SIO jumper E61 indicates which of eight Multibus interrupts will be generated (see reference [2] for jumper information).

3.2.4 Software Control Flags

The SIO firmware maintains two software flags for synchronization. The flag "rcv" indicates how many lines have a control command outstanding, while the flag "sent" indicates whether or not the SIO firmware is expecting an acknowledgment from the MCPU. Both flags are read and written by the SIO firmware only. Together with a few rules, they determine how the two processors become synchronized and achieve mutual exclusion of access to shared data structures.

The "rcv" flag is zero when the SIO firmware is in its initial state, and is incremented in the Channel Attention interrupt service routine when the MCPU issues a control command to the SIO firmware. The flag value is decremented at the time a control command is interpreted and acted upon.

The "sent" software control flag value is false when the SIO firmware is in its initial state, and is set to true immediately after a Multibus interrupt (command completion notification) has been issued to the MCPU. It is set to false in the Channel Attention interrupt service routine when the MCPU issues an acknowledgement to the SIO CPU.

3.2.5 Commands

There are two types of commands to which the SIO firmware responds: control commands (resume, suspend, acknowledgements, and send break) and block commands (set parameters, transmit, change a parameter, and send break). The SIO firmware reports asynchronous events to the SIO agent on the MCPU via status fields in the LINE structure and the interrupt line.

Control commands are communicated to the SIO firmware via the LINE structure, and have no parameters. The MCPU writes the command into the command field in the LINE structure and issues a channel attention to the SIO CPU. The SIO CPU acknowledges completion of any of these commands by writing a zero into the same command field and clearing the Channel Attention.

Block commands (set parameters, transmit, change a parameter, send break) are received by the agent from the client in a message passed to one of the agent's procedures. The agent formats them into a CB and chains them into the CBL for the particular line on the particular SIO board. A channel attention is then issued with the control command "resume the Command Unit".

3.2.6 Synchronization

The following rules must be observed to achieve synchronization:

1. The Agent must wait for the command fields in the LINE structure to be reset by the SIO firmware before setting them again to issue a control command.

2. The Agent must write the command field into the LINE structure before issuing a Channel Attention interrupt.

3. While Sent is true, the SIO firmware cannot modify the status fields of the LINE structure.

4. The Sent and Rcv flags should change state after the occurrence of the event rather than before.

The steps necessary to achieve synchronization are illustrated in Figure 3-1.

Agent	Firmware
	Initial State Rcv = Ø Sent= F
Wait for command fields to be cleared Format LINE structure Channel Attention ===>	
	Rcv --> +=1 Sent= F
	Rcv --> -=1 Clear command fields
New commands and Channel Attention allowed	
	<=== Multibus Interrupt Sent --> T
Clear Multibus Interrupt Read LINE structure status Issue acknowledgement Channel Attention ===>	
	Sent --> F Clear acknowledgement If cmd pending, Rcv --> +=1

Figure 3-1 Synchronization Control

3.2.7 Data Structures

The SIO agent and firmware use ESB shared memory for the SIOCB, the CBL and the CRA. The following subsections describe these shared data structures. Also described are the data structures CS and SIOPORT, which are kept by the agent in private memory and are used to store information on a per-board and per-line basis. These data structures are included here because they contain pointers into the CBL and CRA that reflect current CB and buffer utilization.

The data structures used by the SIO firmware and agent are illustrated in Figure 3-2.

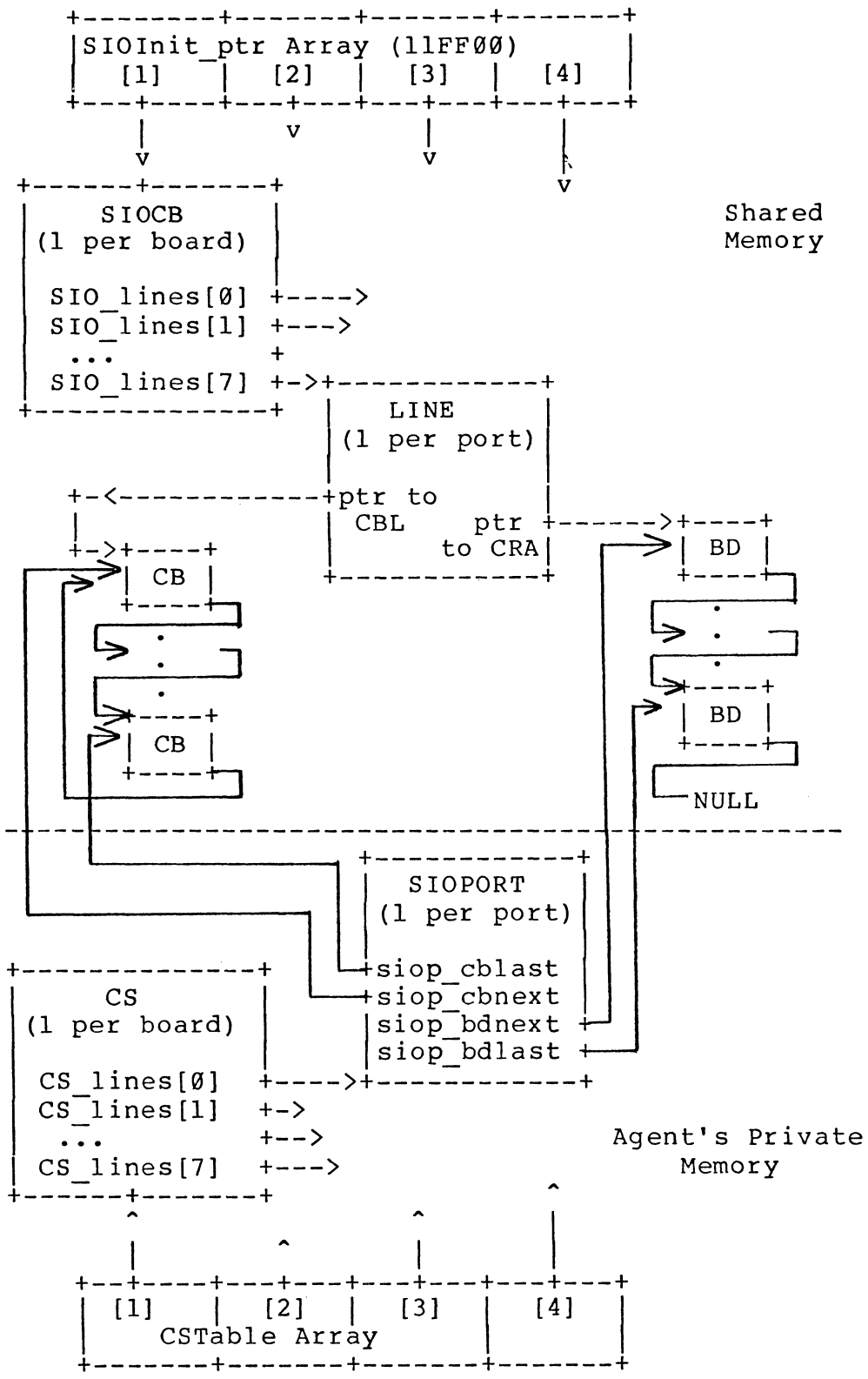


Figure 3-2 SIO Data Structures

3.2.7.1 SIO Control Block

The SIOCB is a shared data structure containing the fields for interprocessor communication. There are eight sources/destinations for characters, so the body of the structure is an array structure; but there is an overall structure for the entire board which facilitates the search for the interrupting line, and enables information about several lines to be exchanged in one interrupt. There is a LINE structure for each of the eight ports, containing pointers to the CRA and the CBL as well as fields for error and statistic information.

The SIOCB and LINE structures have the following format:

```
#define SIOCB struct siocb

SIOCB {
        short   sio_statmask;   /* bit mask - Each line is
                                   assigned a bit. The assigned
                                   bit is ON if the interrupt
                                   to the MCPU has information
                                   from that line. The obvious
                                   bit assignment is made, i.e.
                                   bit #n is used for line #n.
                                */
        short   sio_cmdmask;    /* bit mask - The assigned
                                   bit is ON if the channel
                                   attention from the MCPU has
                                   information for that line,
                                   acknowledgement or new cmd.
                                   Again, the obvious bit
                                   assignment has been made.
                                */
        LINE    *sio_lines[8];  /* pointers to the structures
                                   of the individual lines. A
                                   pointer is picked from here
                                   when it is determined (by
                                   checking the masks) which line
                                   has new information.
                                */
} ;
```

```
#define LINE struct line

LINE {
        short   line_stat;      /*
                                  bit   meaning
                                  8-15  unused
                                  7     command(s) completed
                                  6     read(s) completed
                                  5     CU gone not ready
                                  4     RU gone not ready
                                  3     CU ready/not ready
                                  2     RU ready/not ready
                                  1-Ø   unused
                                */
        short   line_cmd;       /*
                                  bit   meaning
                                  9-15  unused
                                  8     out of band break
                                  7     ack command complete
                                  6     ack read complete
                                  5     ack CU not ready
                                  4     ack RU not ready
                                  3     suspend CU
                                  2     resume CU
                                  1     suspend RU
                                  Ø     resume RU
                                */
        BD      *line_cra;      /* ptr to char rcv area   */
        CB      *line_cbl;      /* ptr to first CB        */
        short   line_cmdcmplt;  /* cnt of complete CBs    */
        short   line_readcmplt; /* cnt of complete reads  */
        short   line_parityerr; /* count of parity errors */
        short   line_bderr;     /* count of characters lost
                                 * due to lack of buffers */
};
```

3.2.7.2 Command Block

The SIO Command Block (SIOCB) is a shared data structure. A command (e.g., initialize, write or change parameter) is given to the SIO firmware by chaining it into the Command Block List (CBL). The CBL is a circular, singly-linked list of blocks, the last of which is marked by the absence of a command in the command type field. Each line has its own CBL, and keeps a pointer to the beginning of the CBL in the SIOCB.

```
#define CB struct cb

CB {
        short   cb_stat;
        short   cb_cmd;
        CB      *cb_link;
        union {
                SIO_INIT        sio_init;
                SIO_WRITE       sio_write;
                SIO_CHGPARM     sio_chgparm;
                SIO_CONNECT     sio_connect;
                SIO_DISCONNECT  sio_disconnect;
                SIO_DIAG        sio_diag;
        } cb_parms;
};

SIO_INIT {
        UIBLK   *i_ptr;
};

SIO_WRITE {
        BD      *w_ptr;
        short    w_eom;
};

SIO_CHGPARM {
        short   chg_parmno;
        short   chg_parmval;
};

SIO_CONNECT {
        BD      *c_bufdes;
        char    *c_ptr;
        short   c_len;
};

SIO_DISCONNECT {
};

SIO_DIAG {
        int     (*diag_routine)();
};
```

3.2.7.3 SIO Agent Private Data Structures

The private structures CS and SIOPORT are used by the agent and network management. These structures contain configuration variables, pointers for tracking the lists of CBs and buffers, and line or board identification such as interrupt identifiers.

```
#define CS struct cs            /* per-board structure          */

CS {
    INTID    cs_ident;          /* interrupt id for board       */
    SIOCB    *cs_siocbptr;      /* ptr to SIOCB for this board */
    short    cs_vec;            /* vector offset                */
    SIOPORT  *cs_lines[8];      /* individual lines             */
};

#define SIOPORT struct sioport  /* per-line structure           */

SIOPORT {
    short    siop_index;        /* index in table               */
    short    siop_state;        /* connected or not             */
    MSG      *siop_safull;      /* msg kept when mbox full      */
    MBID     siop_cmbox;        /* client control mailbox       */
    MBID     siop_dmbox;        /* client data mailbox          */
    CB       *siop_cblast;      /* first free CB                */
    CB       *siop_cbnext;      /* next CB to process           */
    BD       *siop_bdnext;      /* next BD to process           */
    BD       *siop_bdlast;      /* last BD in receive area      */
    short    siop_qcount;       /* no. of unused buffers        */
    short    siop_cbs;          /* number of CBs per line       */
    short    siop_bds;          /* number of BDs per line       */
    short    siop_bdcount;      /* no. of buffers in chain      */
    short    siop_bdsz;         /* size of buffer               */
};
```

3.3 Client Interface to SIO Agent

The agent receives requests in the form of procedure calls directed to a specific line number and gives notification in the form of IPC messages to the client's control mailbox. The procedure calls all take message pointers as parameters. Since the BD parameter is already in the message, the message block can easily be reused, thus reducing overhead.

The SA keeps track of the connection state of each line. The state of being connected or disconnected is associated with the establishment of a Virtual Terminal process for an SIO line. When the SA receives any data while in the disconnected state, it sends a connect request message to the Parent Virtual Terminal process. Once a VT process is established, it notifies the SA of data and control mailboxes to which to send messages.

For the IPC portion of the interface, the agent uses a standard message header (SAMSG) which has a standard data structure as follows:

```
SAMSG {
        MSG     sa_msg;         /* the system message  */
        ushort  sa_portid;      /* the SIO line number */
};
```

3.3.1 Connect Request Procedure Call and Message

The connect request procedure call is issued by a VT process to connect itself to an SIO line. The connect request message is sent by SA to the Parent VT to request the establishment of a VT process for an SIO line. The message has the same format as that passed as a parameter in the procedure call. A connection request may be rejected by sending a disconnected message back to the requestor.

Procedure Call "C" Declaration:

```
saV2Sconnect (m)
MSG *m;
```

Input Parameters:

m Message pointer.

The format of the connect request message (whether pointed to by "m" in the procedure call or sent as an IPC message) is as follows:

```
SACONNECT {
        SAMSG   sac_msg;
        MBID    sac_cmbox;
        MBID    sac_dmbox;
};
```

The parameter "sac_msg" has the standard SAMSG format, and contains the system message, a line number and a message type of MSACONN. When passed as a parameter in a procedure call, the other parameters of the message block indicate the control mailbox and data mailbox of the requesting VT process.

3.3.2 Disconnect Request Procedure Call and Message

The disconnect request procedure call is issued by a VT process to disconnect itself from an SIO line. The disconnect request message is sent by SA to a VT process to request disconnection from an SIO line. Disconnection invalidates the VT data and control mailboxes, so that any further data from the line causes a connect request message to be sent to the Parent VT process. Disconnection also flushes input and output, and drops DCD to the device if the UseDCDout parameter is set to "OnConnection".

Procedure Call "C" Declaration:

```
saV2Sdisconnect (m)
MSG *m;
```

Input Parameters:

m Message pointer.

The format of the disconnect request message is as follows:

```
SADISCONNECT {
        SAMSG           sad_msg;
        ushort          sad_reason;
};
```

The parameter "sad_msg" has the standard SAMSG format, with a message type of MSADISCONNECT. The "sad_reason" parameter values are passed to SA by VT and are defined in Section 5.3.9.

3.3.3 Connected Procedure Call

The connected procedure call is issued by a VT process to SA in order to inform SA of the establishment of a connection.

Procedure Call "C" Declaration:

```
saV2Scnctd (m)
MSG *m;
```

Input Parameter: Message pointer

The format of the connected message is as follows:

```
SACNCTD {
        SAMSG           sacd_msg;
        MBID            sacd_cmbox;
        MBID            sacd_dmbox;
};
```

The parameter "sacd_msg" has the standard SAMSG message block format, with a message type of MSACNCTD. The other parameters are the VT command and data mailboxes.

3.3.4 Disconnected Procedure Call and Message

The disconnected procedure call is issued by VT to SA to terminate a connection (e.g., when VT wishes to notify SA that a connection has been terminated at the remote end). The disconnected message is sent by SA to VT to terminate a connection (e.g., when SA wishes to advise VT that carrier is lost on an SIO line).

Procedure Call "C" Declaration:

```
saV2Sdscnctd (m)
MSG *m;
```

Input Parameter: Message pointer

The format of the disconnected message is as follows:

```
SADSCNCTD {
        SAMSG           sadd_msg;
        ushort          sadd_reason; };
```

The parameter "sadd_msg" has the standard SAMSG format, with a message type of MSADSCNCTD. The "sadd_reason" parameter values are passed to SA by VT and are defined in Section 5.3.9.

3.3.5 Board Initialization Procedure Call

This procedure call is issued by the Parent Virtual Terminal process once per SIO board at initialization time to request the SA to initialize the SIOCB, CRA and CBL for each line of the board according to the initial configuration state (which may say DO_NOT_INITIALIZE).

Procedure Call "C" Declaration:

```
saInitCBlck (board);
```

3.3.6 Set Parameters Procedure Call

The client (e.g., VT) issues this procedure call on a per-line basis to specify the parameters of the SIO line according to the set negotiated between the user interface and the user, or in response to a Rmtset command. All the parameters are affected, and the SIO line is initialized.

Procedure Call "C" Declaration:

```
saV2Ssparm (m)
MSG *m;
```

Input Parameters: Message pointer

Output Parameter: Error code

Error Codes:

NoError	No error detected (Ø).
Error	No memory blocks available for a copy of parameters (-1).
SAFULL	No command block available for requested operation (1).

The message block pointed to by "m" has the following format:

```
SASPARM {
        SAMSG           sas_msg;
        UIBLK           *sas_parmset;
};
```

The parameter "sas_msg" has the standard SAMSG format, and contains a line number and a message type of MSASPARM. The parameter "sas_parmset" points to the block of UI parameters which determine the specified line's parameters.

3.3.7 Change Parameter Procedure Call

This procedure call is used to change a single parameter in the parameter list.

Procedure Call "C" Declaration:

```
saV2Schgp (m)
MSG *m;
```

Input Parameters: Message pointer

Output Parameter: Error code

Error Codes:

NoError No error detected (Ø).

SAFULL No command block available for requested operation (1).

The SIO line, the parameter number and the new value are parameters in the message block, which has the format:

```
SACHGP {
        SAMSG           sach_msg;
        ushort          sach_parmno;
        ushort          sach_value;
};
```

The parameter "sach_msg" has the standard SAMSG format, with a message type of MSACHGP.

3.3.8 Flow Control Procedure Call

When SA cannot send received data to VT because VT's data mailbox is full, SA changes the message type to MSAFLCTRL and sends the message to VT's control mailbox instead. When VT receives this message, it passes the message as a parameter to the procedure call saV2Sflctrl, which again attempts to forward the data to VT's data mailbox. The cycle repeats if the data mailbox is full.

Procedure Call "C" Declaration:

```
saV2Sflctrl(m)
MSG *m;
```

Input Parameter:

m Message pointer.

The message block pointed to by "m" has the format:

```
SAFLCTRL {
        SAMSG   safl_msg;
        ushort  safl_ctrl;
};
```

The parameter "safl_msg" has the standard SAMSG format, with a message type of MSAFLCTRL.

3.3.9 Restart Line Procedure Call and Message

When the SIO firmware needs to flow control a line, it uses this message. The line may have run completely out of buffers if no flow control mechanism is used or if the client's data mailbox is full for a long time (e.g., if the remote end of the connection is flow-controlled). In this case, the interrupt routine sends an SAMSG to the Parent Virtual Terminal process, which in turn issues the restart line procedure call with the same message to initiate reception again. The cycle repeats if the line does not have an adequate supply of buffers.

Procedure Call "C" Declaration:

```
saV2Srestart (m)
MSG  *m;
```

Input Parameter:

m Message pointer.

The restart message block has the standard SAMSG format, and a message type of MSARESTART.

3.3.10 Send Data Procedure Call

This procedure call is used to transmit data to the SIO line. The buffer descriptor and the SIO line are passed in the message as parameters. The parameter "sada_reason" indicates whether or not an EOM signal accompanies the buffer. The EOM is mapped according to the parameter set for the port. If the data cannot be chained into the Command Unit of the SIO line, the return code indicates the reason.

Procedure Call "C" Declaration:

```
short
saV2Sdata (m)
MSG *m;
```

Input Parameter:

m Message pointer.

Output Parameter:

Error code

Error Codes:

NoError No error detected (0).

SAFULL No command block was available (1).

The send data message block pointed to by "m" has the format:

```
SADATA {
        SAMSG           sada_msg;
        ushort          sada_reason;
};
```

The parameter "sada_msg" has the standard SAMSG format, and a message type of MSADATA.

3.3.11 Receive Data Message

This message is sent from interrupt level to the client data mailbox when a read is completed on the line. The reason for completion is passed in the message, and comes from the status field of the BD for the received buffer.

"C" Declaration:

```
SADATA {
        SAMSG           sada_msg;
        ushort          sada_reason;
};
```

The parameter "sada_msg" has the standard SAMSG format, and a message type of MSADATA.

3.3.12 Send Attention Procedure Call

The attention procedure call is called when the VT process needs to send a BREAK to the SIO line. The attention message is sent from the interrupt level of SA to the VT control mailbox when a BREAK is detected on the SIO line.

Procedure Call "C" Declaration:

```
saV2Sattn (m)
MSG *m;
```

Input Parameter:

m — Message pointer.

Output Parameter:

Error code

Error Codes:

NoError — No error detected (Ø).

SAFULL — No command block was available for requested operation (1).

The format of the attention message is as follows:

```
SAATTN {
        SAMSG           saa_msg;
        ushort          saa_signal;
};
```

The parameter "saa_msg" has the standard SAMSG format, and a message type of MSAATTN. The message parameter "saa_signal" indicates whether the BREAK should be sent in-band or out-of-band. When directed to an SIO line by VT, the in-band BREAK is sent in a Command Block and follows any outstanding commands, while the out-of-band BREAK is issued as a control command and so occurs immediately.

3.3.13 Full CBL Message

Each SIO line has a small number of preallocated command blocks. If all of a line's command blocks are full, and a procedure call is received which requires a command block, the agent generates a return code of SAFULL and saves the pointer to the message used in the procedure call. When a command block becomes available, SA sends the saved message back to the client with a message type of MTMSAFULL. It is the responsibility of the client to remember the original message type and to reissue the procedure call.

The procedure calls which require a command block are the send data procedure call, the attention procedure call, the set parameters procedure call, and the change parameters procedure call.

Bridge
Communications
Inc.

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