ETHERNET COMMUNICATIONS CONTROLLER PROGRAMMER'S REFERENCE MANUAL

Order Number: 121769-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

A433/981/7K CP

REV.	REVISION HISTORY	DATE
-001	Original issue.	8/81

This manual documents the programmer's interfaces for users of the iSBC 550 Ethernet* Communications Controller and users of the DS/E 675 Ethernet Development System.

Chapter 1 presents a non-technical overview of the programming aspects of these products. The rest of the manual, however, assumes that you are familiar with:

1. Data communications concepts and vocabulary.
2. Ethernet specifications.
3. PL/M or some similar high-level programming language, or 8080 or 8086 Assembly Language.

If you have the DS/E 675 Ethernet Development System and are already familiar with the Intellec Microcomputer Development System, you will be ready to write Ethernet programs after reading only Chapter 1, the first two sections of Chapter 2, and Appendix D. If you are not already familiar with program development on the Intellec Microcomputer Development System, these chapters refer you to information in other manuals.

If you use one of the iRMX operating systems and intend to use one of the iMMX 800 products to implement the Multibus Interprocessor Protocol (MIP), you need the information in Chapters 1 through 5 and Appendix A.

Appendix B and Appendix C help you if you wish to implement your own MIP facility.

Related Publications

For more information related to programming for the Ethernet Communications Controller and Ethernet Development System, refer to the following manuals:

- *The Ethernet—A Local Area Network—Data Link Layer and Physical Layer Specifications*, 121794.
- *iSBC 550 Ethernet Communications Controller Hardware Reference Manual*, 121746.
- *iMMX 800 Software Reference Manual and User's Guide*, 143808.
- *Intellec Series III Microcomputer Development System Product Overview*, 121575.
- *PL/M-80 Programming Manual*, 401700.
- *8080/8085 Assembly Language Programming Manual*, 401100.
- *MCS-80/85 Utilities User's Guide for 8080/8085-Based Development Systems*, 121617.

Notation

Hexadecimal numbers are used frequently throughout this manual. To distinguish from decimal numbers, the letter 'H' follows all hexadecimal numbers. A leading zero may be added to a hexadecimal number that does not begin with one of the digits 0 through 9. For example, the hexadecimal number 0FH has the same value as decimal 15.

*Ethernet is a trademark of the Xerox Corporation.

CONTENTS

TABLES

ILLUSTRATIONS

iSBC 550 Ethernet* Communications Controller

The Ethernet local area network provides a communication facility for high-speed data exchange among digital devices located within 2.5 kilometers of each other.

The iSBC 550 Ethernet Communications Controller gives you the means to connect a Multibus system to an Ethernet facility and begin evaluating Ethernet capabilities. Figure 1-1 illustrates how the Ethernet Controller is used in an Ethernet configuration.

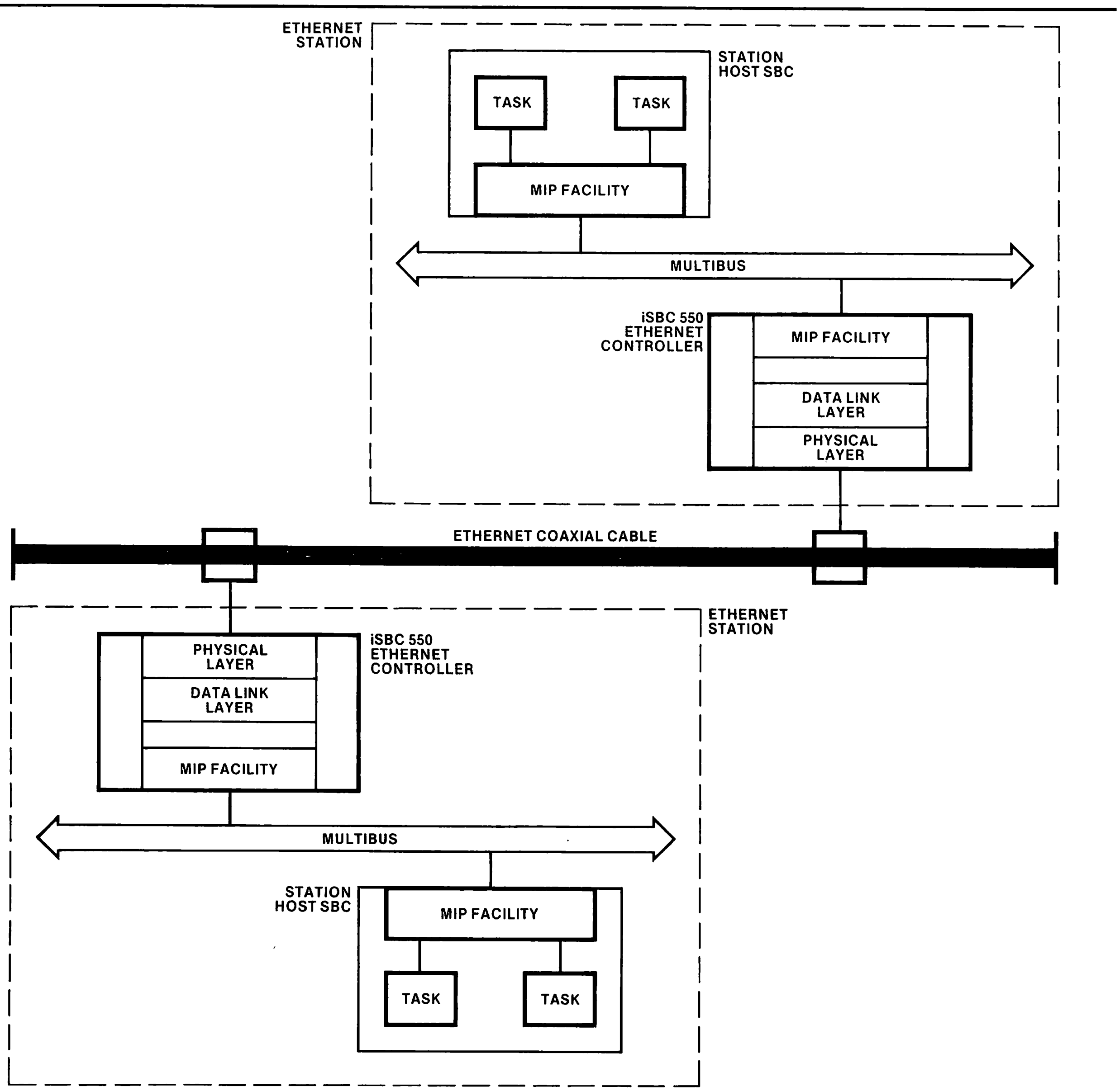

Figure 1-1. Configuration of an Ethernet Link.

*Ethernet is a trademark of the Xerox Corporation.

Each Ethernet station is a multi-computing system consisting (at a minimum) of the following hardware components:

- An Intel Multibus system bus.
- An iSBC 550 Ethernet Communications Controller.
- One or more station hosts.

A *station host* is any processor board (such as the iSBC 80/30, iSBC 88/40, or iSBC 86/12A) that runs application tasks that need to access an Ethernet network. The rate at which station hosts can transfer messages to and from the Ethernet Controller is one to two million bits per second. The Ethernet Controller, however, can transfer to and from the network at 10 million bits per second.

The Ethernet Communications Controller implements the Physical Layer and Data Link Layer of network architecture as defined in *The Ethernet Data Link Layer and Physical Layer Specifications*. The lowest layer, the Physical Layer, is concerned with the coaxial cable interface. It completely specifies the essential physical characteristics of the Ethernet network, such as data encoding, timing, and voltage levels. The Data Link Layer defines a medium-independent, link-level communication facility, built on the medium-dependent physical channel provided by the Physical Layer. The Data Link Layer supports packet framing, addressing, error detection, channel allocation, and collision detection.

The next higher level, the Client Layer, consists of the programs that you write using the Data Link Layer and Physical Layer functions provided by the Ethernet Controller. In addition to providing applications logic, it is the responsibility of the Client Layer to deal (where necessary) with the following functions:

- Packet aging
- Congestion control
- Processing identification
- Routing messages among processes
- Detection of message loss
- Recovery of lost messages
- Matching message flow among processes with available resources

User tasks running on the station host communicate with the Ethernet Controller by passing messages through shared memory. The protocol used to ensure reliable and efficient communication across the bus is the *Multibus Interprocessor Protocol* (MIP). Software that implements MIP is known as a *MIP facility*. The Ethernet Controller comes equipped with a ROM-resident MIP facility. However, for use at the station hosts, you have several choices:

- If you are using one of the iRMX operating systems, then Intel offers its iMMX 800 series of MIP facilities. (Refer to the *iMMX 800 Software Reference Manual and User's Guide*.)
- If you have purchased a DS/E 675 Ethernet Development System or a DS/E 677 Ethernet Development System Upgrade Kit, then you may use the Ethernet Data Link Library, which contains a MIP facility for use with the Ethernet Development System's 8085 processor. (Refer to Appendix D.)
- You may implement a custom version of MIP using the specifications presented in Appendix B. The example implementations in Appendix C may serve as a useful starting point.

Application tasks control the Ethernet Controller by means of a set of messages known collectively as the *External Data Link* (EDL). Fixed-format *request blocks*, sent to the Ethernet Controller under control of the MIP facilities, instruct it to perform such functions as:

- Transmit a packet.
- Receive a packet.
- Recognize certain packet types from the network.
- Recognize certain multicast addresses from the network.
- Read out network parameters.
- Read and clear network parameters.

To create an Ethernet application using the Ethernet Controller, you need to become familiar with some or all of the following programmatic interfaces:

- Calling on the services of the MIP facility that resides at the station host (details of which depend on which MIP implementation you are using)
- Initializing the Ethernet Controller firmware (discussed in Chapter 2)
- Formatting EDL request blocks for the Ethernet Controller (explained in Chapters 3 and 4)

DS/E 675 Ethernet Development System

The Development System for Ethernet (DS/E 675) is a complete set of tools to help develop Ethernet communications software and applications. It combines the power of the Intellec Series III Microcomputer Development System and an Ethernet Communications Controller. All the software development aids of the Intellec Series III Microcomputer Development System are available. Refer to the *Intellec Series III Microcomputer Development System Product Overview* for a complete list of features.

In addition to Series III software support, the Ethernet Development System includes a diskette containing:

- The Ethernet Data Link Library
- An example Ethernet application

The Ethernet Data Link Library (file name EDL80.LIB) contains procedures that enable programs that run on the 8085 processor of the Ethernet Development System to easily and simply communicate with the network via the Ethernet Communications Controller. The procedures of the library hide the details of controller initialization and MIP facility interface, thereby permitting you to develop Ethernet software in minimal time. Complete information on the Ethernet Data Link Library is contained in Appendix D.

The example application on the diskette consists of the source code for the PL/M example presented in Chapter 5 and Appendix C of this manual. Print or display the file entitled EXAMPL.HLP for more information on how to use the example files.

Introduction to Terms and Concepts

The following terms and concepts are used frequently throughout the manual.

Data Link Addresses and Types

Data link addresses are 6 bytes long. A data link address is of one of two types:

1. *Physical Address*—The unique address associated with a particular station on the Ethernet network. Each iSBC 550 Ethernet Communications Controller contains a unique, hardware-determined address selected from the set of addresses assigned to Intel Corporation by the Ethernet Address Administration Office of Xerox Corporation.
2. *Multicast Address*—A multi-destination address associated with one or more stations on a given Ethernet network. There are two kinds of multicast address:
    - *Multicast-group address*—An address associated by higher-level convention with a group of logically related stations
    - *Broadcast address*—A distinguished, predefined multicast address that always denotes the set of all stations on a given Ethernet network

The first transmitted bit of a data link address (the low-order bit of the high-order byte) distinguishes physical from multicast addresses:

0 — physical address

1 — multicast address

The broadcast address consists of 48 one-bits. To obtain a block of multicast-group addresses for use by your organization, write to Xerox Corporation at the address shown below.

When considering the use of multicast addresses, be aware that network throughput may degrade significantly. While recognition of physical addresses is performed automatically by hardware, the presence of even one multicast address on the Ethernet cable causes every iSBC 550 Ethernet Communications Controller on the network to perform a firmware-level search of its multicast address table to determine whether it should respond to the packet containing that multicast address.

The *data link type* field is a two-byte item reserved for use by the Client Layer (in particular, to identify the Client Layer protocol associated with the packet). The type field is not interpreted by the Physical Layer or Data Link Layer.

The address and type fields are administered by Xerox Corporation. To obtain a multicast-group address or type field assignment, submit written requests to:

Xerox Corporation
Ethernet Address Administration Office
3333 Coyote Hill Road
Palo Alto, CA 94304

A nominal fee to cover administrative costs is charged.

Intel Corporation makes available to users of the iSBC 550 Ethernet Communications Controller one of the type codes assigned to Intel by Xerox, namely 5009H. You may use this type code without charge for the purposes of developing and testing systems that use the iSBC 550 Ethernet Communications Controller. However, for production systems, you must obtain your own unique type codes from Xerox Corporation.

MIP Concepts

Within an Ethernet station, MIP facilities aid communication among tasks that reside on various processor boards attached to a common Multibus system bus. The set of all such tasks, along with associated processor boards, operating systems, and MIP facilities, is called a *MIP system*.

The term *device* is used for each processor board in a MIP system. Each device has a *device-ID*, which is a number ranging from zero to the number of devices (less one) communicating in one MIP system. The assignment of device-ID's is up to you. The device-ID's assigned must be used consistently throughout the MIP system.

Communications are delivered to a task at a *MIP port*, which is a logical delivery mechanism that enables delivery in ''first-in, first-out'' (FIFO) order. (Do not confuse MIP ports with hardware I-O ports.) The actual implementation of a port depends on the operating system and MIP facilities involved. In some operating systems MIP ports are implemented as ''mailboxes'' or ''exchanges.'' The ports at a given device are identified by a *port-ID*, a number which ranges from zero to the number of ports (less one) at the device. Assign port-ID's for the devices that you program.

To provide system-wide addressability, a port is also identified by a *socket*, which is a pair of items in the form (D,P), where ''D'' is the device-ID and ''P'' is the port-ID.

CHAPTER 2
INITIALIZING THE ETHERNET CONTROLLER

Overview

Initialization consists of:

- Sending configuration parameters for the MIP facility that runs on the Ethernet Controller
- Running the firmware confidence tests and reporting the results
- Determining whether another station on the network is running

A bootstrap routine which runs in ROM on the Ethernet Communications Controller performs initialization when the system is powered-up or reset, or when interrupted by the station host.

Communicating With the Bootstrap Routine

The host processor communicates with the bootstrap routine by a system of interrupts and messages. The station host interrupts the Ethernet Controller by writing to a specific I-O port address known as the *wake-up port*. Plug-in jumpers on the controller determine which I-O port address the controller recognizes as a host interrupt. The jumper settings are defined in table 2-1 and figure 2-1. I-O port addresses are available in both 8-bit and 16-bit addressing ranges. The address you choose must not be used for any other function on the Multibus system bus.

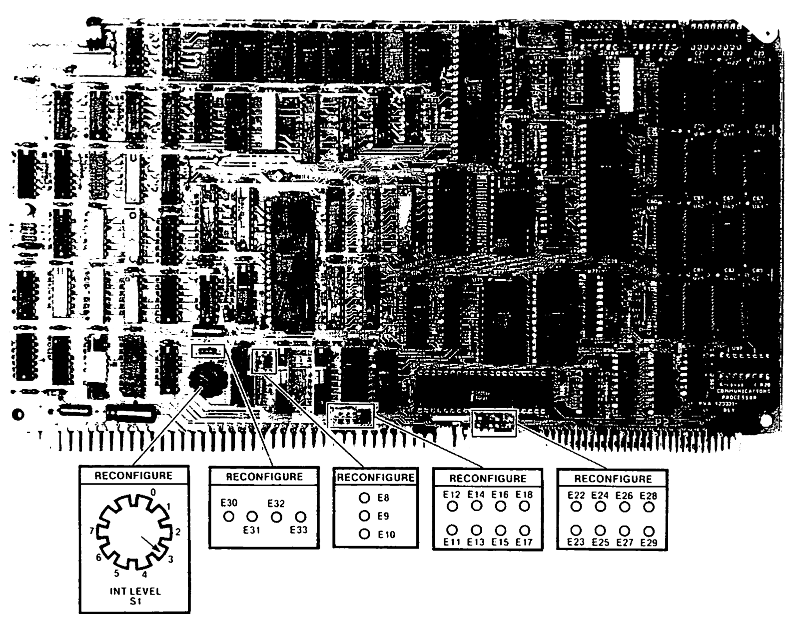

Figure 2-1. User-Configurable Switch and Jumpers. 769-2

Table 2-1. Controller Wake-Up I/O Port Address Jumpers

Port Address	Plug-In Jumpers							
	E8-E9	E9-E10	E11-E12	E13-E14	E15-E16	E17-E18	E30-E31	E32-E33
A4H	IN	OUT	OUT	OUT	OUT	IN	Jumper may be installed in either E30-E31 or E32-E33	
A5H	IN	OUT	OUT	OUT	IN	OUT		
A6H	IN	OUT	OUT	IN	OUT	OUT		
A7H	IN	OUT	IN	OUT	OUT	OUT		
8A4H	OUT	IN	OUT	OUT	OUT	IN	OUT	IN
8A5H	OUT	IN	OUT	OUT	IN	OUT	OUT	IN
8A6H	OUT	IN	OUT	IN	OUT	OUT	OUT	IN
8A7H	OUT	IN	IN	OUT	OUT	OUT	OUT	IN
9A4H	OUT	IN	OUT	OUT	OUT	IN	IN	OUT
9A5H	OUT	IN	OUT	OUT	IN	OUT	IN	OUT
9A6H	OUT	IN	OUT	IN	OUT	OUT	IN	OUT
9A7H	OUT	IN	IN	OUT	OUT	OUT	IN	OUT

The value written to the wake-up port determines the action taken by the firmware on the Ethernet Controller.

01H — Resets the controller and starts the bootstrap routine.

02H — During initialization, signals that a bootstrap command has been placed in the command area of memory; after initialization, signals to the MIP facility that a request for the Ethernet Controller has been placed in one of its input queues.

04H — After initialization, resets the interrupt level generated by the Ethernet Controller. (Refer to the Start Command in this chapter for more details on the use of this interrupt.)

Only interrupt values 01H and 02H are used during initialization. Value 01H should always be used to start the initialization process, even after power-up or system reset. After issuing this interrupt, the host processor must wait at least two seconds before issuing any more interrupts to the Ethernet Controller.

Once started, the bootstrap routine responds to commands issued by the station host processor. The host places a bootstrap command with the proper parameters at a fixed location in memory and generates an interrupt with a value of 02H. (A MIP facility is not used for communication with the bootstrap since one of the functions of the bootstrap is to start the MIP facility on the controller.) Figure 2-2 illustrates the general format of a command block. The meanings of the fields in this block are defined below:

- COMMAND. Fill this item with the identifier of the bootstrap function to be performed.
- RESPONSE. Fill this item with zero. The bootstrap routine changes RESPONSE to a non-zero value upon completion of the function.
- PARAMETERS. The contents and length of this area depend upon the function to be performed. The various commands are described in detail in the following section.

The location of the communication area depends on the configuration of your system. In different configurations, the host processor and the Ethernet Controller share different memory areas. Plug-in jumper settings on the Ethernet Controller tell which location to use for communication. Refer to table 2-2 and figure 2-1 for the communication area addresses and plug-in jumper settings to be used with various system configurations.

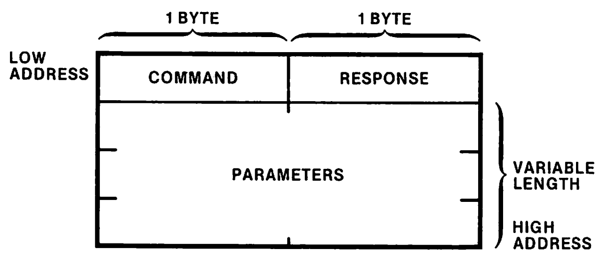

Figure 2-2. Command Block. 769-3

Table 2-2. System Compatibility Selection Jumpers.

Host System	Communications Area Starting Address	Plug-In Jumpers			
		E22-E23*	E24-E25	E26-E27	E28-E29
Series II/800	0F690H	OUT	IN	IN	IN
Series III—8085	0F690H	OUT	IN	IN	IN
Series III—8086	1F000H	OUT	IN	IN	OUT
(reserved)	to be defined	OUT	IN	OUT	IN
User 1	1000H	OUT	IN	OUT	OUT
User 2	8000H	OUT	OUT	IN	IN
User 3	10000H	OUT	OUT	IN	OUT
User 4	20000H	OUT	OUT	OUT	IN
User 5	2F000H	OUT	OUT	OUT	OUT

*A jumper installed in E22-E23 causes the firmware diagnostic to loop repeatedly on power-up or reset.

After placing a command in the communication area and writing the value 02H to the wake-up port, the host waits for the RESPONSE field to become non-zero. The bootstrap responds within two seconds to all commands. If it does not respond within that time, it is either not present, misconfigured, or not functioning.

If you are using the Ethernet Data Link Library to facilitate your interface with the Ethernet Controller, you may wish to skip the rest of this chapter. The Ethernet Data Link Library automatically handles these initialization functions through its CQSTRT routine. (Refer to Appendix D for a complete description of the Ethernet Data Link Library.)

Configuring the MIP Facility

One function of initialization is to tell the MIP facility at the Ethernet Controller what the configuration of the station is. The MIP facility needs two kinds of information:

- Each device attached to the bus
- Memory that can be shared among devices attached to the bus

Device Information

The physical communication mechanism between devices at a station is a fixed size, unidirectional, FIFO queue called a *Request Queue*. An element in a Request Queue is known as a *Request Queue Entry* (RQE). Each Request Queue is managed by a *Request Queue Descriptor* (RQD). An RQD and associated RQE's forming one queue occupy a contiguous block of memory, as illustrated in figure 2-3.

Two-way communication between the Ethernet Controller and another device at the same station is implemented by a pair ot Request Queues, known together as a *channel*. Figure 2-4 shows a conceptual diagram of a channel.

To communicate with the MIP facility on a station host, the MIP facility at the Ethernet Controller must have the starting address of each of the Request Queues of the channel. This information is sent to the Ethernet Controller via the Start Command, as explained in the "Bootstrap Commands" section below. Just how you obtain the queue addresses depends on what MIP facilities you are using at the host processors.

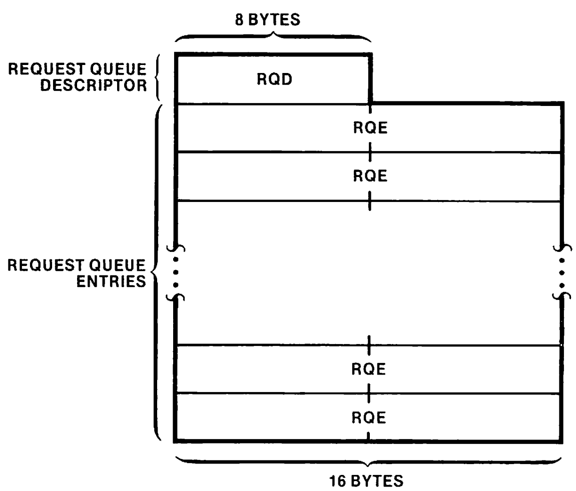

Figure 2-3. Format of a Request Queue. 769-4

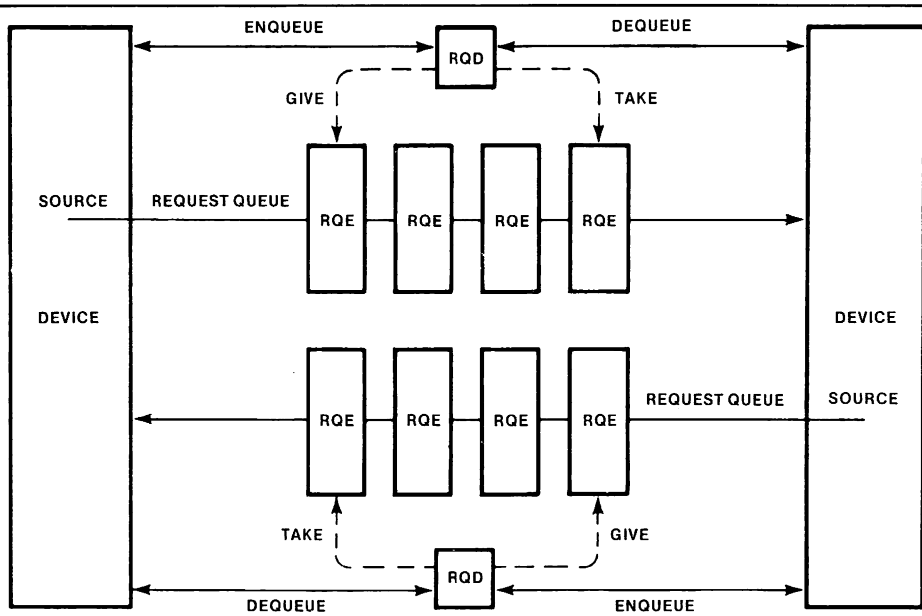

Figure 2-4. Conceptual Structure of a Channel. 769-5

Memory Configuration

The Ethernet Controller communicates with the station host or hosts via shared memory. The abilities of the devices to access the memory available on the Multibus system bus can be used to define a partition of that memory. MIP partitions all of memory into non-overlapping segments such that, for any segment and any device, either:

- The segment is continuously addressable within the address space of the device, or
- The device cannot address any of the segment.

Each segment that can be shared among devices is called an *inter-device segment* (IDS) and is identified by an *IDS-ID* (a number ranging from zero to the number of IDS's (less one) in the station).

Figure 2-5 presents a hypothetical memory configuration and shows how the address space is partitioned. Processor A and processor C can communicate through IDS 1. Processor B and processor C can communicate through IDS's 0, 1, and 3. IDS 3, however, is a segment of dual-ported memory; it is accessed by processor B using a different range of addresses than processor C uses. Memory segments A, B, and C cannot be used for inter-device communication.

Table 2-3 summarizes the memory configuration shown in figure 2-5. The table shows the lowest address (the *base address*) by which each device can access each IDS.

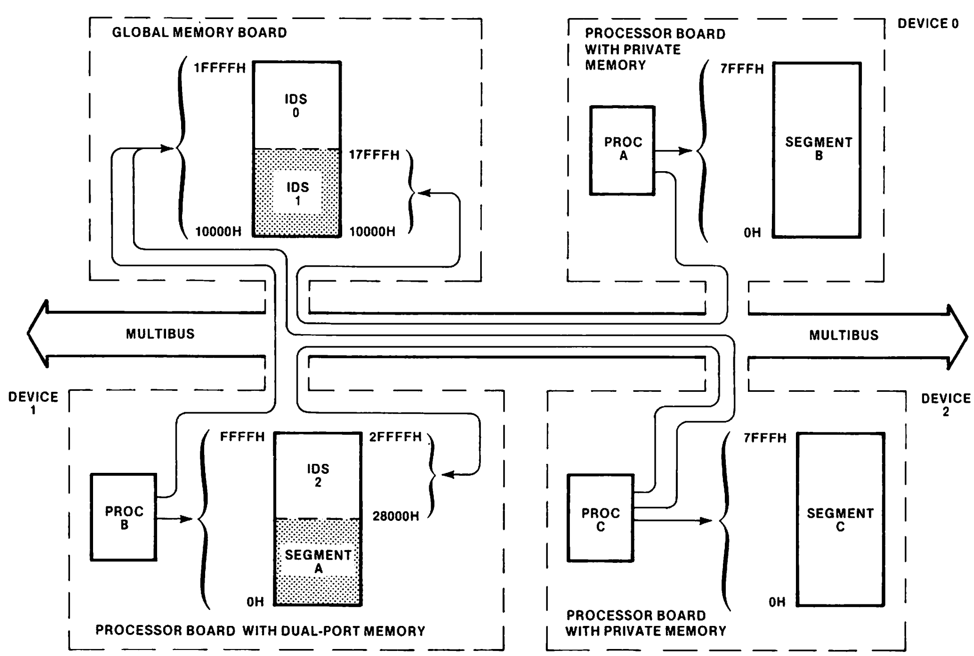

Figure 2-5. Example of Inter-Device Memory Segments. 769-6

Table 2-3. System Inter-Device Segment Table.

IDS	Length	Base Addresses		
		Device 0	Device 1	Device 2
0	8000H	—	18000H	18000H
1	8000H	10000H	10000H	10000H
2	8000H	—	8000H	20000H

The Start Command (discussed below) provides the MIP facility at the Ethernet Controller with the IDS information it needs to communicate with the station host or hosts.

Bootstrap Commands

The bootstrap routine provides three functions to the host processor:

1. Presence
2. Echo
3. Start

For an example of how these commands are used in a program, refer to Chapter 5.

Presence Command

This must always be the first command executed after resetting the Ethernet Controller with an interrupt value of 01H. The Presence Command determines whether the Ethernet Controller is present. If it is present and functioning, the Presence Command returns the version number of the firmware and the result of the most recent execution of the confidence test. (The bootstrap executes the confidence test as soon as it receives the first command from the station host and before it returns a response.) Figure 2-6 illustrates the format of the Presence Command.

1. RESPONSE. A value of one is returned within two seconds if the Ethernet Controller is present and functioning.
2. TEST RESULT. The bootstrap inserts the result of the most recent execution of the confidence test. Refer to Appendix A for a summary of the possible result codes.
3. VERSION. The bootstrap routine fills in the version number of the firmware residing on the Ethernet Controller. The version number has the form *X,Y* where *X* is binary value stored in the high-order four bits and *Y* is a binary value stored in the low-order four bits.

COMMAND = 01H	RESPONSE
TEST RESULT	VERSION

Figure 2-6. Presence Command Block. 769-7

Echo Command

This command causes the bootstrap to transmit an echo request packet to another station on the network and wait for a reply. The bootstrap routine waits for up to 10 seconds before concluding that no echo has occurred. Refer to figure 2-7 for the format of the Echo Command.

- RESPONSE. Bootstrap returns 01H if an echo is received, 02H if no echo is received within 10 seconds.
- DESTINATION ADDRESS. Enter the data link address of the station to be tested.
- SEND DATA. The value you enter in this field is transmitted to the destination station.
- ECHO DATA. This field is filled from the echo response. If an echo is received, ECHO DATA should be the same as SEND DATA. If no echo is returned, the content of this item is not defined.

The Ethernet Controller at the destination address responds to an echo request packet if it has been initialized. It will also respond when it receives the echo request packet during initialization, but only if it has already processed a Presence Command from its host.

The format of the echo request packet is illustrated in figure 2-8 to aid users of telecommunications monitoring equipment.

Start Command

This command performs two functions:

- It supplies a description of the system environment for use by the MIP facility that runs on the Ethernet Controller.
- It starts execution of the MIP facility and other communications firmware on the controller.

Once the Start Command is successfully executed, the initialization process is over. A portion of the bootstrap routine becomes part of the running software on the Ethernet Controller so that it can respond to echo commands from other stations on the network. However, attempts to execute bootstrap commands from the local station host are ignored. After initialization, writing a value of 02H to the wake-up port is interpreted as a signal to the MIP facility that runs on the Ethernet Controller.

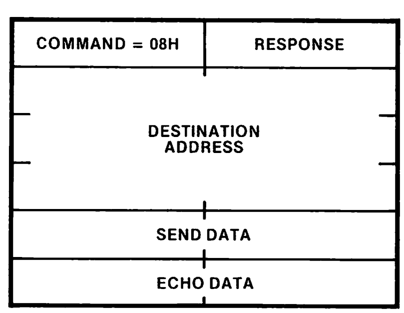

Figure 2-7. Echo Command Block. 769-8

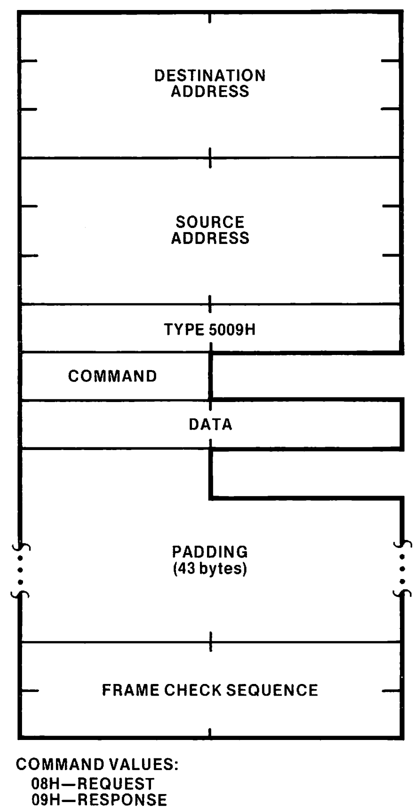

Figure 2-8. Echo Packet. 769-9

The MIP request queue from the Ethernet Controller to the host must already be initialized before executing the Start Command. Contrary to the MIP specifications in Appendix B, the MIP facility on the Ethernet Controller does not initialize its outgoing request queues.

The Ethernet Controller does no error checking on the values of the fields in the Start Command block. Incorrect values may cause the controller to malfunction, so take care to supply proper values.

The format of the Start Command is shown in figure 2-9. The format has three parts:

- The fixed-length header
- The variable-length IDS section. The number of entries here must correspond to the value in IDS COUNT.
- The variable-length device section. The number of entries in this section must correspond to the value in DEVICE COUNT.

The fields of the Start Command are explained below:

- RESPONSE. The bootstrap returns 01H for a successful load and go; 0FFH if an illegal command is entered.
- (RESERVED). These areas are reserved for future expansion.

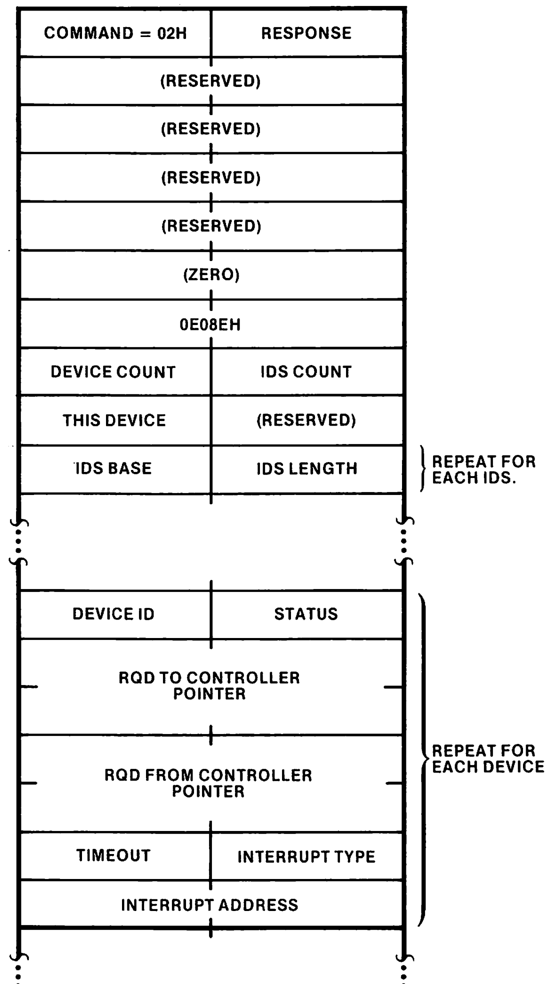

Figure 2-9. Start Command Block. 769-10

- (ZERO). Fill this item with zeros.
- '0E08EH'. This address value must be entered exactly as shown.
- DEVICE COUNT. Enter the number of other devices in the MIP system. The maximum number of devices with which the Ethernet Controller can communicate is six.
- IDS COUNT. Enter the number of IDS's in the MIP system. The maximum number is eight, the minimum is one.
- THIS DEVICE. Enter the device-ID assigned to the Ethernet Controller.
- IDS BASE. The starting address of an IDS must be evenly divisible by 4096 (1000H). Enter the starting address of the IDS less low-order 12 bits. This address is multiplied by 1000H (4096) to arrive at the actual starting address. For example, if the actual starting address is 3000H, enter 3.
- IDS LENGTH. Enter the number of 4096-byte (1000H) segments of memory in this IDS.
- DEVICE ID. Enter the device ID of the device to which this entry applies. Device ID's may range from 0 to 7.
- STATUS. Enter 0FFH.

- RQD TO CONTROLLER POINTER. Enter an 8086-style pointer to the RQD of the MIP queue for passing requests to the Ethernet Controller from this device. (The format of an 8086-style pointer is illustrated in figure 2-10.) Queues must be contained within in the range 800H through EFFFFH (2K to 960K), the Multibus addressing range of the Ethernet Controller.
- RQD FROM CONTROLLER POINTER. Enter an 8086-style pointer to the RQD of the MIP queue for passing requests from the Ethernet Controller to this device. (See figure 2-10.) Queues must be contained within the range 800H through EFFFFH (2K to 960K), the Multibus addressing range of the Ethernet Controller.
- INTERRUPT TYPE. Enter a code for the type of interrupt the MIP facility on the Ethernet Controller should use when signalling the MIP facility on this device. The valid codes are:

  0H — No interrupt; the device polls the RQD. This technique is suitable if a processor is running only one task.

  1H — I-O mapped. Some devices (such as the iSBC 550 Ethernet Communications Controller) recognize a write to a specific I-O port address as an interrupt. This is a highly reliable technique; it should be used when available. The I-O port address is specified in the INTERRUPT ADDRESS field. The value written to this port is 02H.

  2H — Memory mapped. Some devices (such as the iSBC 544 Intelligent Communications Controller) recognize a write to a specific memory address as an interrupt. This is also a reliable technique. The memory paragraph to be written is specified in the INTERRUPT ADDRESS field. The value written to this address is 02H.

  3H — Edge level. The Ethernet Controller raises one of the Multibus interrupt lines after lowering it briefly. The rising edge triggers a processor interrupt. This technique is available on most current Intel processor boards, such as the 80/30, 80/24, and 86/12. The Multibus interrupt line is selectable by the rotary INT LEVEL switch S1 on the Ethernet Controller board as shown in table 2-4 and figure 2-1.

  4H — Pure level. The Ethernet Controller asserts one of the Multibus interrupt lines for 100μs. If the host processor has interrupts enabled and is not busy processing other interrupts during this time, an interrupt is triggered. The Multibus interrupt line is selectable by the rotary INT LEVEL switch S1 on the Ethernet Controller board as shown in table 2-4 and figure 2-1. To cause the Ethernet Controller to drop the interrupt line before 100μs, the MIP facility at the host must write a value of 04H to the controller's wake-up port before servicing the interrupt. To guard against missed interrupts, the MIP facility at the host should periodically poll the signals in its incoming request queues.
- TIMEOUT. Enter the time (in 52 millisecond units) that the Ethernet Controller should wait for a response when signalling this device. If the device does not respond within this time, the Ethernet Controller assumes that the device is dead. The value in this field must be greater than zero. A value of 0FFH indicates that the Ethernet Controller should wait forever. The only time a value of 0FFH should be used, however, is when only one device is communicating with the controller, since a failure of one device prevents the Ethernet Controller from servicing any other devices.
- INTERRUPT ADDRESS. Enter the interrupt address as specified above for INTERRUPT TYPE. If INTERRUPT TYPE is 0, 3, or 4, then the INTERRUPT ADDRESS field is not used.

Table 2-4. Interrupt Priority Level Selection.

S1 Switch Position	Interrupt Level	Priority
0	INT0/	Highest
1	INT1/	
2	INT2/	
3	INT3/	
4	INT4/	
5	INT5/	
6	INT6/	
7	INT7/	Lowest

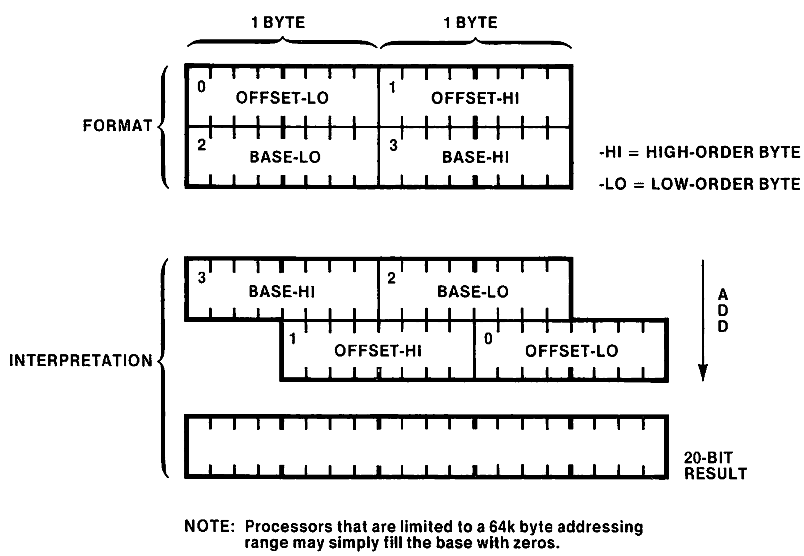

Figure 2-10. Format of 8086-Style Pointer 769-11

CHAPTER 3
EXCHANGING MESSAGES OVER AN ETHERNET NETWORK

Introduction to External Data Link

The External Data Link (EDL), a task that runs on the Ethernet Controller, is provided to enable tasks running on the station host to control some of the Data Link Layer functions of the Ethernet Communications Controller. A host task makes a request of the Ethernet Controller by transferring a fixed format *request block* to EDL through the MIP facilities. EDL receives request blocks at MIP port 01H. EDL interprets a request block, performs the request, and then returns the request block to the MIP port specified in the RESPONSE SOCKET field.

Details of how request blocks are transferred from the station host to the Ethernet Controller depend on which implementation of MIP you are using at the station host. The MIP facility at the station host must follow two conventions in communication with the MIP facility at the Ethernet Controller:

- The MIP facility at the host must initialize the request queue from the controller to the host before the controller is initialized (contrary to the MIP specifications in Appendix B).
- The MIP facility at the host must signal any change in queue status (full to not full, or empty to not empty) by writing a value of 02H to the Ethernet Controller's wake-up port.

The general format of a request block is illustrated in figure 3-1. The fields shown in figure 3-1 are explained below:

- (RESERVED). The first 12 bytes of every request block are reserved for use by the Ethernet Controller.
- COMMAND. Fill this with the identifier of the function requested of EDL.
- RESULT. Filled by EDL to indicate the outcome of the request. Always be sure to check this field when the request block is returned.
- RESPONSE SOCKET. Enter the address of the MIP port to which EDL should return the request block when finished executing the request.
- BODY OF REQUEST. The length and meaning of this area depend on the contents of the command field.

The EDL requests available are:

1. CONNECT.
2. DISCONNECT.
3. ADDMCID.
4. DELETEMCID.
5. TRANSMIT.
6. SUPPLYBUF.
7. READ.
8. READC.

Before using the network, you must tell the Ethernet Controller which type codes and multicast addresses to accept. Type codes are not interpreted by the Data Link Layer; they are used to identify the Client Layer protocols associated with each frame. A multicast address associates one station with a group of other stations that

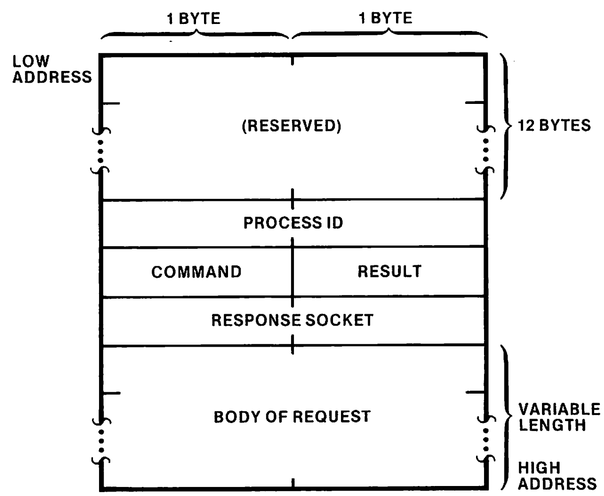

Figure 3-1. General Format of a Request Block. 769-12

have the same multicast address. Type codes are specified by the CONNECT function, and multicast addresses are specified by the ADDMCID function.

The Ethernet Controller has no memory that the host can access; therefore, to receive packets from the network, you must supply buffer space by using the SUPPLYBUF function. When a packet is received, EDL returns the buffer containing that packet. The SUPPLYBUF request effectively implements the ReceiveFrame function of the Ethernet Specifications.

To transmit a packet, send a TRANSMIT request to EDL. The TRANSMIT request effectively implements the TransmitFrame function of the Ethernet Specifications.

The DISCONNECT and DELETEMCID functions tell EDL to stop accepting certain type codes and multicast addresses.

The READ and READC functions allow you to access and reset certain network parameters. These two requests are discussed in Chapter 4. The previously mentioned requests are defined in more detail in the following sections.

CONNECT

The CONNECT request informs EDL which data link packet types to route to the station host. Note that, when there is more than one host at a station, EDL does not distinguish between type codes specified in CONNECT requests from different hosts. Therefore, any host may receive packets containing type codes specified by any other host at the same station. Up to eight types may be active at one time; however, EDL uses two type codes, leaving you space for only six. The format of a CONNECT request is shown in figure 3-2, and the fields in the request are clarified below:

- RESULT. EDL fills this with a zero if the operation is successful; with a one if the limit of eight types is exceeded.
- TYPE. Enter the data link type code for which the Data Link Layer should start looking.

DISCONNECT

The DISCONNECT request causes the Data Link Layer to cease forwarding those packets identified by a specific type code. See figure 3-3 for the format of the DISCONNECT request.

- RESULT. This item always returns zero.
- TYPE. Enter the data link type code for which the Data Link Layer should stop looking.

ADDMCID

The ADDMCID request tells the Data Link Layer which multicast addresses to recognize. Note that, when a station has more than one host processor, EDL does not distinguish between multicast addresses specified in ADDMCID requests from different processors. Therefore, any host may receive packets containing multicast addresses specified by other hosts at the same station. Up to eight multicast addresses may be active at one time. Figure 3-4 shows the format of the ADDMCID request.

- RESULT. EDL fills this with zero if the operation is successful, with one if the limit of eight multicast addresses is exceeded.
- MULTICAST ADDRESS. Enter the multicast address for which the Data Link Layer should start looking.

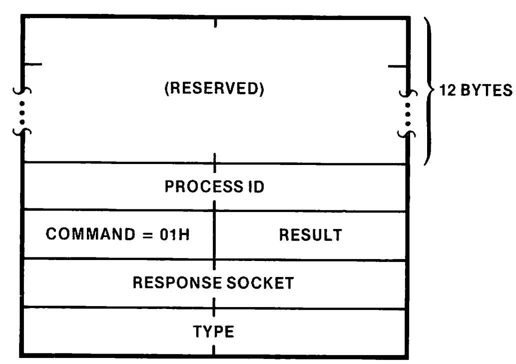

Figure 3-2. CONNECT Request Block. 769-13

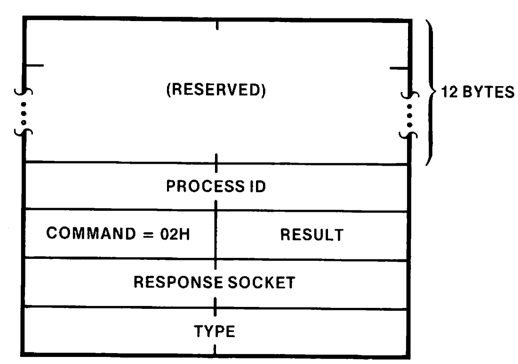

Figure 3-3. DISCONNECT Request Block. 769-14

DELETEMCID

The DELETEMCID request causes the Data Link Layer to stop recognizing a specific multicast address. Figure 3-5 shows how to format the DELETEMCID request.

1. RESULT. EDL always returns zero.
2. MULTICAST ADDRESS. Enter the multicast address for which the Data Link Layer should stop looking.

TRANSMIT

The TRANSMIT request is used to transmit a packet over an Ethernet network. You have two options in formatting the TRANSMIT request block: either the entire request block is one contiguous area or a pointer in the request block points to a portion of the request block information that is located elsewhere in memory. Any por-

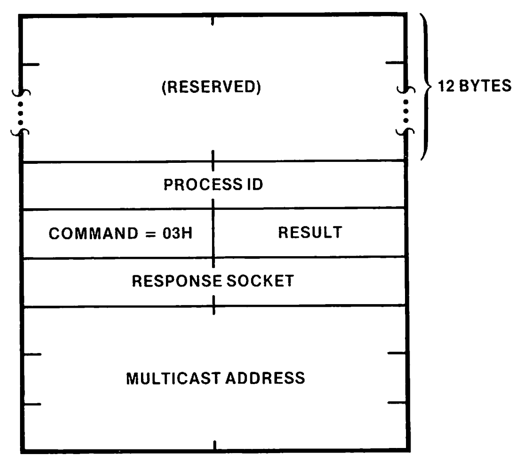

Figure 3-4. ADDMCID Request Block. 769-15

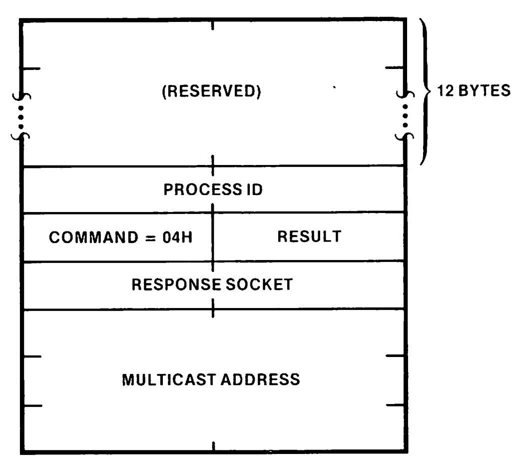

Figure 3-5. DELETEMCID Request Block. 769-16

tion of the request block after the EXTENSION LENGTH field may reside in this extension area. EDL effectively concatenates the extension area at the end of the contiguous portion of the request block. See figure 3-6 for details of the format.

- RESULT. EDL returns zero if the packet is transmitted, one if not transmitted. A packet is not transmitted if the data area contains less than 46 bytes or more than 1500 bytes.
- LENGTH. Enter the length (in bytes) of the contiguous portion of the packet, counting from the end of the EXTENSION LENGTH field.
- EXTENSION POINTER. Enter a 24-bit IDS pointer to an extension of the request block. Note that the high-order 8 bits of this address are stored separately from the low-order 16 bits. If EXTENSION LENGTH is zero, this pointer is ignored and the request block must lie in one continuous area of memory.
- IDS-ID. Enter the identifier of the inter-device segment in which the extension area is located.
- EXTENSION LENGTH. Enter the length in bytes of the extension or enter zero if the request block lies in one continuous area of memory.
- DESTINATION ADDRESS. Enter the data link address or multicast address of the Ethernet station or stations to which you wish to send the packet. Fill this field before sending the TRANSMIT block to EDL.
- SOURCE ADDRESS. EDL fills this item with the data link address of the sending station.
- TYPE. Fill with a data link type code before sending the request.
- DATA. Enter 46 to 1500 bytes of user data. To meet Ethernet minimum packet size requirements, you must pad smaller messages to make them at least 46 bytes long.

SUPPLYBUF

The SUPPLYBUF request provides a buffer in which to place a packet received from the Ethernet network. When EDL receives a packet, it copies it into this buffer and returns the buffer to RESPONSE SOCKET. The data area of the buffer should be at least 1500 bytes long to ensure that a maximum-length packet does not overflow the end of the buffer. SUPPLYBUF may be used several times in succession, thereby making several buffers available for receipt of packets. Make sure that the number of buffers supplied is great enough to receive all the packets that might arrive before more buffers can be supplied. If the Ethernet Controller receives a packet but does not have a user buffer in which to place it, the packet is discarded.

Note that, when there is more than one host at a station, EDL does not distinguish between buffers supplied by different hosts. Any buffer may be returned to any host.

See figure 3-7 for the format of the SUPPLYBUF request.

- RESULT. Always zero when a buffer is returned.
- LENGTH. The length in bytes of the received packet, counting from the beginning of the destination address through the end of the data area.
- DESTINATION ADDRESS. The physical address of the receiving station or a multicast address.
- SOURCE ADDRESS. The Data Link address from which the packet came.
- TYPE. The Data Link type code. This can only be one of the types specified in a previous CONNECT request.
- DATA. Filled with 46 to 1500 bytes of received data.

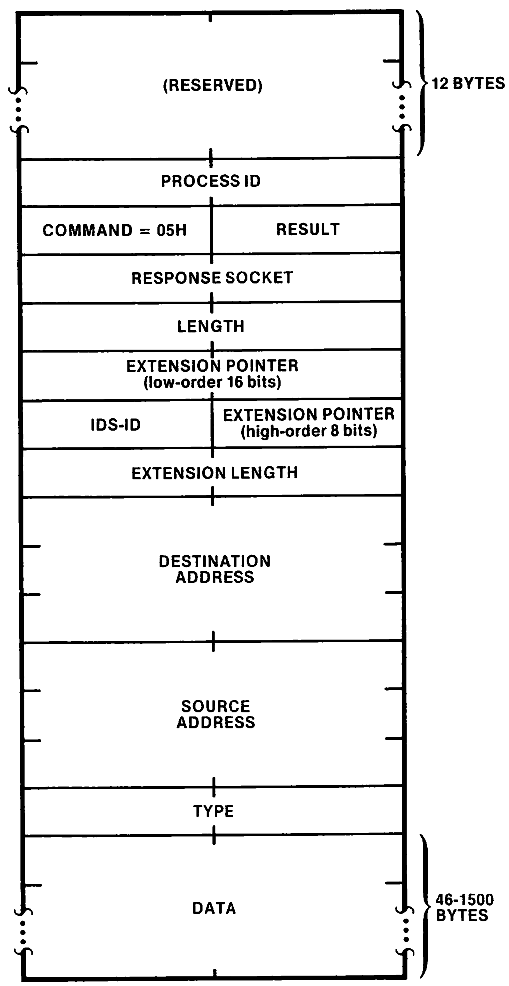

Figure 3-6. TRANSMIT Request Block. 769-17

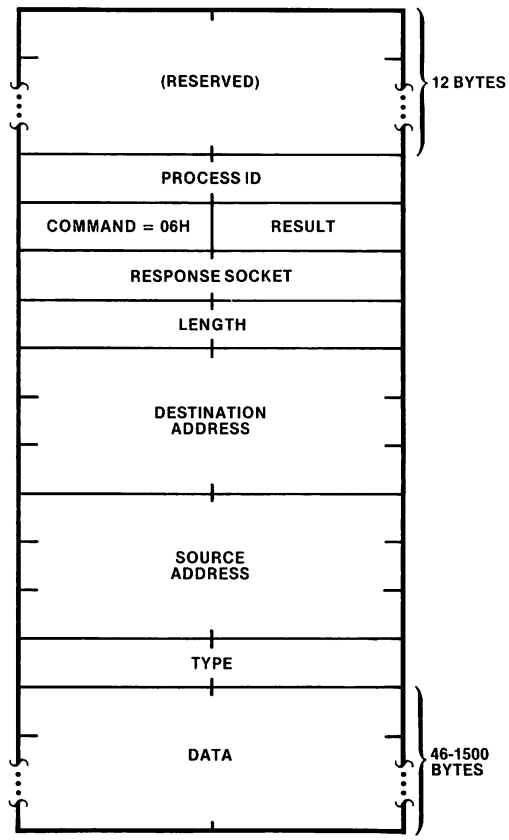

Figure 3-7. SUPPLYBUF Request Block. 769-18

CHAPTER 4 BASIC NETWORK MANAGEMENT FUNCTIONS

Data Link Objects

In addition to the message exchange functions explained in the previous chapter, the External Data Link (EDL) provides access to certain items of information about the network that are held by the Data Link Layer. The accessible data link objects are explained below:

0. TOTAL SENT—the total number of packets sent from this station
1. PRIMARY COLLISIONS—the number of transmit packets that have encountered at least one collision
2. SECONDARY COLLISIONS—the number of collisions (after the first collision) that transmit packets have encountered.

   The average number of collisions for packets encountering collisions is given by the formula:

$$\frac{\text{PRIMARY COLLISIONS} + \text{SECONDARY COLLISIONS}}{\text{PRIMARY COLLISIONS}}$$

3. EXCEEDED COLLISIONS—the number of packets that are discarded in transmission because they encounter more than the maximum number (16) of collisions
4. TRANSMIT TIMEOUTS—the number of packets that trip the transmit watchdog timer. Under normal conditions, this counter should register zero.
5. TOTAL RECEIVED—the number of packets forwarded to the station host
6. CRC ERRORS—the number of packets discarded because of failure of the cyclic redundancy check (CRC)
7. FRAMING ERRORS—the number of incoming packets discarded because they contain more than 1500 bytes of user data
8. RESOURCE ERRORS—the number of incoming packets discarded because the Data Link Layer does not have enough space in its internal receive buffers (does not include packets discarded due to lack of user buffers)
9. HOST ADDRESS—the hardware-defined address of this station. (READC can read this value but not change it.)
10. LOADING—the moving average of the time the Ethernet cable is carrying traffic. This item is interpreted as the fractional part of a real number; that is, the part to the right of the binary point. The value ranges from 0.0 to 0.FFFFH. (READC reads this value, but does not change it.)

Each accessible data link object is identified by a number. Objects that are counters may be of two types:

- W (wrap-around). These counters, upon reaching their maximum value, automatically reset to zero and continue counting.
- S (''sticky''). These counters stop when they reach their maximum value.

Table 4-1 shows, for each accessible data link object, its identifying number, type, and size.

Table 4-1. Data Link Objects.

ID	Name	Length in Bits	Type
0	Total Sent	32	W
1	Primary Collisions	16	S
2	Secondary Collisions	16	S
3	Exceeded Collisions	16	S
4	Transmit Timeouts	16	S
5	Total Received	32	W
6	CRC Errors	16	S
7	Framing Errors	16	S
8	Resource Errors	16	S
9	Host Address	48	
10	Loading	16	

READ

The READ request reads the value of an accessible data link object. The format of a READ request is illustrated in figure 4-1. (Refer to Chapter 3 for the general format of EDL request blocks.)

- RESULT. This item always returns zero.
- DATA LINK OBJECT. Enter the identifying number of the object to be read. If the number put in this field is not a valid object identifier, this request has no effect.
- RETURN VALUE. EDL returns the value of the object here. If the object is less than six bytes in length, EDL uses only enough of this field to hold the value (justified left).

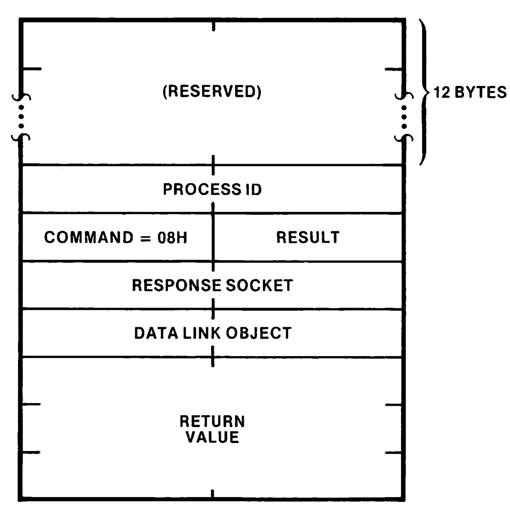

Figure 4-1. READ Request Block. 769-19

READC

The READC request reads an accessible data link object and, if the object is a counter, clears it to zero after it has been read. If the object is not a counter, READC functions the same as READ. Reading and clearing in a single operation avoids the "race" or contention condition that might result if reading and clearing were done in separate operations. Figure 4-2 shows the format of the READC request.

- RESULT. This item always returns zero.
- DATA LINK OBJECT. Enter the identifying number of the object to be read and cleared. If the number you place in here is not a valid object identifier, this request has no effect.
- RETURN VALUE. EDL returns the former value of the object here. If the object is less than six bytes in length, EDL uses only enough of this field to hold the value (justified left).

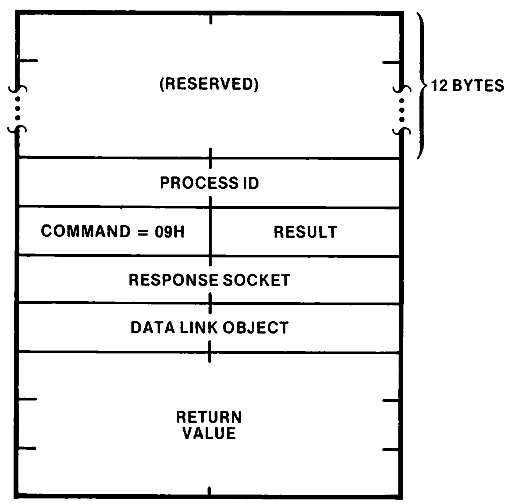

Figure 4-2. READC Request Block. 769-20

Overview

This chapter presents a simple remote printing utility to illustrate how to use the Ethernet Controller, EDL, and a MIP facility. The application consists of two programs that enable files located at one station of an Ethernet network to be printed at another station. The Print Server program (PRINSE) runs continuously at a station that has a printer. The command tail of the command that starts PRINSE must contain the pathname of the device on which to print; for example:

```
PRINSE :LP:
```

PRINSE accepts files transmitted over the network and prints them on the specified device.

The Remote Print program (REPRIN) can be executed at any station to transfer a file to PRINSE. The command tail of the command that invokes REPRIN must contain the pathname of the file to be transferred; for example:

```
REPRIN :F1:MYPROG.LST
```

When finished transferring the file, REPRIN displays a summary of any communications errors that may have occurred during transmission.

The example MIP facility presented in Appendix C (XMX) is used by both programs to communicate with the Ethernet Controller.

In the interests of simplicity, these programs omit much of the error checking and error recovery logic that is the responsibility of the Client Level. A real application of this type should provide (for example) for the possibility that a communications error causes loss of a packet. This example merely illustrates how to use the Ethernet Controller.

Remote Print Library Module

This submodule is a library of utility routines used by both PRINSE and REPRIN.

```
REMOTE$PRINT$LIB:  DO;

DECLARE WORD       LITERALLY 'ADDRESS',
        CONNECTION LITERALLY 'WORD',
        TRUE       LITERALLY '0FFFFH',
        FALSE      LITERALLY '0000H';
```

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

                /*  ISIS System Calls  */

WRITE:
    PROCEDURE (CONN, BUF$P, COUNT, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            COUNT    WORD,
            (BUF$P, STATUS$P) ADDRESS;
    END WRITE;
EXIT:
    PROCEDURE EXTERNAL;
    END EXIT;

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

DECLARE ASCII (16) BYTE INITIAL ('0123456789ABCDEF');

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

DECIMAL:  PROCEDURE (BINARY, OUT$P, WIDTH)       PUBLIC;

      /*  Converts a binary word into a right-     */
      /*  justified ASCII decimal representation.  */

DECLARE BINARY   WORD,                 /* Input. */
        OUT$P    ADDRESS,
        WIDTH    BYTE;

DECLARE CHAR$P   ADDRESS,              /* Local. */
        CHAR BASED CHAR$P BYTE;

 CHAR$P = OUT$P + WIDTH - 1;
 DO WHILE OUT$P <= CHAR$P;
  IF BINARY = 0
   THEN CHAR = ' ';
   ELSE DO;
    CHAR = ASCII(BINARY MOD 10);
    BINARY = BINARY / 10;
   END /* ELSE */;
  CHAR$P = CHAR$P - 1;
 END /* DO */;

 CHAR$P = OUT$P + WIDTH - 1;
 IF CHAR = ' ' THEN CHAR = '0';

END DECIMAL;

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

HEX:  PROCEDURE (IN$P, IN$LENGTH, OUT$P)          PUBLIC;

      /*  Converts a binary byte string of given  */
      /*  length into an ASCII hexadecimal        */
      /*  representation of twice the length.     */

DECLARE IN$P                   ADDRESS,       /* Input. */
        IN$BYTE BASED IN$P     BYTE,
        OUT$P                  ADDRESS,
        OUT$CHAR BASED OUT$P BYTE,
        IN$LENGTH              BYTE;
```

```
DECLARE STOP$P                ADDRESS;      /* Local. */

 STOP$P = IN$P + IN$LENGTH - 1;

 DO IN$P = IN$P TO STOP$P;
  OUT$CHAR = ASCII(SHR(IN$BYTE, 4));
  OUT$P = OUT$P + 1;
  OUT$CHAR = ASCII(SHR(SHL(IN$BYTE, 4), 4));
  OUT$P = OUT$P + 1;
 END /* DO */;

END HEX;

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

HEX$WORD:  PROCEDURE (BINARY, OUT$P)            PUBLIC;

  /*  Converts a binary word to an ASCII hexadecimal  */
  /*  representation (reversing order of bytes).      */

DECLARE BINARY WORD,                 /* Input. */
        OUT$P  ADDRESS;

 CALL HEX (.BINARY + 1, 1, OUT$P);
 CALL HEX (.BINARY,     1, OUT$P + 2);

END HEX$WORD;

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

COM$ERROR: PROCEDURE (MNEMONIC, EDL$STATUS)      PUBLIC;

DECLARE MNEMONIC       WORD,      /* Input. */
        EDL$STATUS     WORD;

DECLARE STATUS         WORD,      /* Local. */
        HEX$STATUS (4) BYTE,
        BACKWORDS (2)  BYTE;

 CALL WRITE (0, .('COMMUNICATIONS ERROR '), 21, .STATUS);
 BACKWORDS (0) = HIGH (MNEMONIC);
 BACKWORDS (1) =  LOW (MNEMONIC);
 CALL WRITE (0, .BACKWORDS, 2, .STATUS);
 CALL WRITE (0, .(' '), 1, .STATUS);
 CALL HEX$WORD (EDL$STATUS, .HEX$STATUS);
 CALL WRITE (0, .HEX$STATUS, 4, .STATUS);
 CALL WRITE (0, .('H', 0DH, 0AH), 3, .STATUS);
 CALL EXIT;

END COM$ERROR;
```

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

EQUAL$ADDRESS:  PROCEDURE (NET$ADR1$P,
                           NET$ADR2$P) WORD      PUBLIC;

       /*  Compares two data link addresses.  */

DECLARE NET$ADR1$P                ADDRESS,   /* Input. */
        BYTE1 BASED NET$ADR1$P    BYTE,
        NET$ADR2$P                ADDRESS,
        BYTE2 BASED NET$ADR2$P    BYTE;

DECLARE STOP$P                    ADDRESS;   /* Local. */

 STOP$P = NET$ADR1$P + 5;

 DO NET$ADR1$P = NET$ADR1$P TO STOP$P;
  IF BYTE1 <> BYTE2
   THEN RETURN FALSE;
  NET$ADR2$P = NET$ADR2$P + 1;
 END /* DO */;

 RETURN TRUE;

END EQUAL$ADDRESS;

END REMOTE$PRINT$LIB;
```

Controller Initialization Module

This submodule is used by both programs to initialize the Ethernet Controller.

```
CONTROLLER$INIT:  DO;

          /*  Ethernet Communications Controller  */
                   /*  Initialization  */

DECLARE WORD            LITERALLY 'ADDRESS';

DECLARE WAKE$UP$PORT    LITERALLY '0A4H',
        RESET$LOOPS     LITERALLY '200',  /*  2 seconds. */
        COMMAND$LOOPS   LITERALLY '200',  /*  2 seconds. */
        ECHO$LOOPS      LITERALLY '1000', /* 10 seconds. */
        NULL$PTR        LITERALLY '0';

DECLARE OUT$RQD (8) BYTE EXTERNAL,
        IN$RQD (8)  BYTE EXTERNAL;

DECLARE COMMAND  STRUCTURE
         (ID                 BYTE,
          RESPONSE           BYTE,
          BODY (32)          BYTE)  AT  (0F690H);

DECLARE PRESENCE  STRUCTURE
         (TEST$RESULT        BYTE,
          VERSION            BYTE)   AT (.COMMAND.BODY);
```

```
DECLARE ECHO  STRUCTURE
         (DESTINATION$ADDRESS (6) BYTE,
          SEND$DATA               WORD,
          ECHO$DATA               WORD) AT (.COMMAND.BODY);

DECLARE START  STRUCTURE
         (RSRVD (8)        BYTE,
          ZERO             WORD,
          ADR              ADDRESS,
          DEVICE$COUNT     BYTE,
          IDS$COUNT        BYTE,
          THIS$DEVICE      BYTE,
          RSVRD            BYTE,
          IDS$BASE         BYTE,
          IDS$LENGTH       BYTE,
          DEVICE$ID        BYTE,
          STATUS           BYTE,
          TO$CONT$OFFSET   ADDRESS,
          TO$CONT$BASE     WORD,
          FROM$CONT$OFFSET ADDRESS,
          FROM$CONT$BASE   WORD,
          INTR$TYPE        BYTE,
          TIMEOUT          BYTE,
          INTR$ADDRESS     WORD)
    INITIAL
         (0,0,0,0,0,0,0,0,  0, 0E08EH,       /* Fixed. */
          1, 1, 00H, 0,
          0, 16,                              /* IDS. */
          01H, 0FFH,                         /* Device. */
          .OUT$RQD, 0, .IN$RQD, 0,
          0, 0FFH, 0 );

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

SEND$TO$BOOT:  PROCEDURE (BOOT$LOOPS, STATUS$P);

DECLARE BOOT$LOOPS  WORD;              /* Input. */

DECLARE STATUS$P    ADDRESS,           /* Output. */
        STATUS BASED STATUS$P WORD;

DECLARE I           WORD;              /* Local. */

 COMMAND.RESPONSE = 0;
 OUTPUT(WAKE$UP$PORT) = 02H;
 DO I = 0 TO BOOT$LOOPS;
  CALL TIME (250);
  IF COMMAND.RESPONSE <> 0
   THEN DO;
    STATUS = COMMAND.RESPONSE;
    RETURN;
   END /* THEN */;
 END /* DO */;
 STATUS = 80H;  /* No response. */

END SEND$TO$BOOT;
```

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

ETHER$INIT:  PROCEDURE (ECHO$ADDRESS$P,
                        TEST$RESULT$P,
                        STATUS$P)        PUBLIC;

DECLARE ECHO$ADDRESS$P          ADDRESS,     /* Input. */
        ECHO$ADDRESS BASED ECHO$ADDRESS$P (12) BYTE;

DECLARE TEST$RESULT$P           ADDRESS,     /* Output. */
        TEST$RESULT BASED TEST$RESULT$P BYTE,
        STATUS$P                ADDRESS,
        STATUS BASED STATUS$P   WORD;

DECLARE I                       WORD;         /* Local. */

 OUTPUT (WAKE$UP$PORT) = 01H;  /* Reset the controller. */
 DO I = 1 TO RESET$LOOPS;     /* Give the controller */
  CALL TIME (250);            /* time to reset.      */
 END;

 COMMAND.ID = 01H;   /* Presence Command. */
 CALL SEND$TO$BOOT (COMMAND$LOOPS, STATUS$P);
 IF STATUS > 1 THEN RETURN;
 TEST$RESULT = PRESENCE.TEST$RESULT;

 IF ECHO$ADDRESS$P <> NULL$PTR
  THEN DO;
   COMMAND.ID = 08H;  /* Echo Command. */
   CALL MOVE (6, ECHO$ADDRESS$P,
              .ECHO.DESTINATION$ADDRESS);
   ECHO.SEND$DATA = 0F0F0H;
   CALL SEND$TO$BOOT (ECHO$LOOPS, STATUS$P);
   IF STATUS = 1
    THEN IF ECHO.SEND$DATA <> ECHO.ECHO$DATA
     THEN STATUS = 81H;
   IF STATUS > 1 THEN RETURN;
  END /* THEN */;

 COMMAND.ID = 02H;  /* Start Command. */
 CALL MOVE (32, .START, .COMMAND.BODY);
 CALL SEND$TO$BOOT (COMMAND$LOOPS, STATUS$P);
 RETURN;

END ETHER$INIT;

END CONTROLLER$INIT;
```

Remote Print Program

```
REMOTE$PRINT:  DO;

DECLARE WORD       LITERALLY 'ADDRESS',
        CONNECTION LITERALLY 'WORD',
        TRUE       LITERALLY '0FFFFH',
        FALSE      LITERALLY '0000H';
```

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

                 /*  ISIS System Calls.  */

OPEN:
    PROCEDURE (CONN$P, PATH$P, ACCESS,
                    ECHO, STATUS$P) EXTERNAL;
    DECLARE (CONN$P, PATH$P, STATUS$P) ADDRESS,
            ACCESS   WORD,
            ECHO     CONNECTION;
    END OPEN;

READ:
    PROCEDURE (CONN, BUF$P, COUNT,
                ACTUAL$P, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            COUNT    WORD,
            (BUF$P, ACTUAL$P, STATUS$P) ADDRESS;
    END READ;
WRITE:
    PROCEDURE (CONN, BUF$P, COUNT, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            COUNT    WORD,
            (BUF$P, STATUS$P) ADDRESS;
    END WRITE;
CLOSE:
    PROCEDURE (CONN, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            STATUS$P ADDRESS;
    END CLOSE;
EXIT:
    PROCEDURE EXTERNAL;
    END EXIT;
ERROR:
    PROCEDURE (ERRNUM) EXTERNAL;
    DECLARE ERRNUM WORD;
    END ERROR;

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

                    /*  XMX calls.  */

XMX$SEND:
    PROCEDURE  (BUFFER$PTR, BUFFER$LENGTH,
                SOCKET, STATUS$P)                 EXTERNAL;
    DECLARE BUFFER$PTR          ADDRESS,
            BUFFER$LENGTH       WORD,
            SOCKET              WORD;
    DECLARE STATUS$P            ADDRESS;
    END XMX$SEND;
XMX$RECEIVE:
    PROCEDURE  (STATUS$P) ADDRESS                 EXTERNAL;
    DECLARE STATUS$P            ADDRESS;
    END XMX$RECEIVE;
ETHER$INIT: PROCEDURE
        (ECHO$ADDRESS$P, TEST$RESULT$P, STATUS$P) EXTERNAL;
    DECLARE ECHO$ADDRESS$P      ADDRESS;
    DECLARE TEST$RESULT$P       ADDRESS,
            STATUS$P            ADDRESS;
    END ETHER$INIT;
```

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

            /*  Remote Print Library Calls  */

DECIMAL:
    PROCEDURE (BINARY, OUT$P, WIDTH)                EXTERNAL;
    DECLARE BINARY           WORD,
            OUT$P            ADDRESS,
            WIDTH            BYTE;
    END DECIMAL;
HEX:
    PROCEDURE (IN$P, IN$LENGTH, OUT$P)              EXTERNAL;
    DECLARE IN$P             ADDRESS,
            OUT$P            ADDRESS,
            IN$LENGTH        BYTE;
    END HEX;
HEX$WORD:
    PROCEDURE (BINARY, OUT$P)                       EXTERNAL;
    DECLARE BINARY           WORD,
            OUT$P            ADDRESS;
    END HEX$WORD;
EQUAL$ADDRESS:
    PROCEDURE (NET$ADR1$P, NET$ADR2$P) WORD         EXTERNAL;
    DECLARE NET$ADR1$P       ADDRESS,
            NET$ADR2$P       ADDRESS;
    END EQUAL$ADDRESS;
COM$ERROR:
    PROCEDURE (MNEMONIC, EDL$STATUS)                EXTERNAL;
    DECLARE MNEMONIC         WORD,
            EDL$STATUS       WORD;
    END COM$ERROR;

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

               /*  EDL communication areas.  */

DECLARE XMIT$HDR STRUCTURE
         (RSVRD (14)       BYTE,
          COMMAND          BYTE,
          RESULT           BYTE,
          RESPONSE$SOCKET  WORD,
          HDR$LENGTH       WORD,
          EXT$P            ADDRESS,
          EXT$IDS          BYTE,
          EXT$SEGMENT      BYTE,
          EXT$LENGTH       WORD,
          DST$ADDRESS (6)  BYTE,
          SRC$ADDRESS (6)  BYTE,
          TYPE             WORD,
          PRINT$COMMAND    WORD,
          PRINT$LENGTH     WORD );

DECLARE PRINT$START  LITERALLY '0001H',   /* Commands. */
        PRINT$DATA   LITERALLY '0002H',
        PRINT$END    LITERALLY '0003H';

DECLARE CHANGE$ADDRESS (6) BYTE AT (.XMIT$HDR.HDR$LENGTH),
        CHANGE$TYPE        WORD AT (.XMIT$HDR.HDR$LENGTH),
        DL$READ   STRUCTURE
         (ID          WORD,
          VALUE       WORD,
          FILLER (2)  WORD)      AT (.XMIT$HDR.HDR$LENGTH);
```

```
DECLARE RECEIVE$BUF STRUCTURE
         (RSVRD (14)        BYTE,
          COMMAND           BYTE,
          RESULT            BYTE,
          RESPONSE$SOCKET   WORD,
          BUF$LENGTH        WORD,
          DST$ADDRESS (6)   BYTE,
          SRC$ADDRESS (6)   BYTE,
          TYPE              WORD,
          USER$DATA (1500)  BYTE );

DECLARE PRINT$RESPONSE  WORD AT (.RECEIVE$BUF.USER$DATA);

DECLARE PRINT$OK      LITERALLY '8000H',   /* Responses. */
        PRINT$QUIT    LITERALLY '8001H';

DECLARE PRINSE$ADDRESS (6)    BYTE

              /* Be sure to update this address when
                 PRINSE is to run at another station. */

              INITIAL (00H, 0AAH, 00H, 0FFH, 0FFH, 0FAH),

              /* The following type code is assigned to
                 Intel Corporation.  Refer to Chapter 1
                 for information regarding its use. */

        PRINT$TYPE            LITERALLY '0950H',

        CONF$TEST             BYTE,
        RETURN$P              ADDRESS,
        ETHER$SOCKET          LITERALLY '0001H',
        THIS$SOCKET           LITERALLY '0100H';

DECLARE EDL$CONNECT    LITERALLY '01H',
        EDL$ADDMCID    LITERALLY '03H',
        EDL$TRANSMIT   LITERALLY '05H',
        EDL$SUPPLYBUF  LITERALLY '06H',
        EDL$READC      LITERALLY '09H';

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

EDL$SEND:  PROCEDURE (COMMAND, BUFFER$P,
                      BUFFER$LN, STATUS$P);

DECLARE COMMAND    BYTE,                  /* Input. */
        BUFFER$P   ADDRESS,
        BUFFER$LN  WORD;

DECLARE STATUS$P   ADDRESS,               /* Output. */
        EDL$STATUS BASED STATUS$P WORD;

DECLARE REQUEST BASED BUFFER$P STRUCTURE  /* Local. */
         (RSRVD (14)      BYTE,
          COMMAND         BYTE,
          RESULT          BYTE,
          RESPONSE$SOCKET WORD);
```

```
DECLARE XMX$STATUS        WORD,
        BAD$STATUS        LITERALLY '0001H',
        EMPTY             LITERALLY '0FFH';

 REQUEST.COMMAND = COMMAND;
 REQUEST.RESPONSE$SOCKET = THIS$SOCKET;
 CALL XMX$SEND (BUFFER$P, BUFFER$LN,
                ETHER$SOCKET, .XMX$STATUS);
 IF (XMX$STATUS AND BAD$STATUS) = 0
  THEN DO;
   XMX$STATUS = EMPTY;
   DO WHILE XMX$STATUS = EMPTY;
    RETURN$P = XMX$RECEIVE (.XMX$STATUS);
   END /* DO */;
  END /* THEN */;
 IF (XMX$STATUS AND BAD$STATUS) = 0
  THEN EDL$STATUS = REQUEST.RESULT;
  ELSE EDL$STATUS = XMX$STATUS;

END EDL$SEND;
/*********************************************************/
DECLARE OBJECT (11) STRUCTURE
         (NAME (20) BYTE)
          INITIAL ('TOTAL SENT          ',
                   'PRIMARY COLLISIONS  ',
                   'SECONDARY COLLISIONS',
                   'EXCEEDED COLLISIONS ',
                   'TRANSMIT TIMEOUTS   ',
                   'TOTAL RECEIVED      ',
                   'CRC ERRORS          ',
                   'FRAME ERRORS        ',
                   'RESOURCE ERRORS     ',
                   'HOST ADDRESS        ',
                   'LOADING             ');

SUMMARY:  PROCEDURE;

     /*  Displays a summary of data link errors.  */

DECLARE I          WORD;

    SHOW$OBJECT: PROCEDURE;

     CALL MOVE (20, .OBJECT(I).NAME, .BUFFER);
     DL$READ.ID = I;
     CALL EDL$SEND (EDL$READC, .XMIT$HDR, 26, .STATUS);
     IF STATUS > 0 THEN CALL COM$ERROR ('RC', STATUS);
     CALL DECIMAL (DL$READ.VALUE, .BUFFER(20), 6);
     BUFFER(26) = 0DH;
     BUFFER(27) = 0AH;
     CALL WRITE (0, .BUFFER, 28, .STATUS);

    END SHOW$OBJECT;

 DO I = 1 TO 4;
  CALL SHOW$OBJECT;
 END /* DO */;
 DO I = 6 TO 8;
  CALL SHOW$OBJECT;
 END /* DO */;

END SUMMARY;
```

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

PRINSE$SEND:  PROCEDURE;

DECLARE BAD$STATUS          LITERALLY '0001H',
        EMPTY               LITERALLY '0FFH';

 /* Supply a buffer in anticipation of PRINSE's answer. */

 RECEIVE$BUF.COMMAND = EDL$SUPPLYBUF;
 RECEIVE$BUF.RESPONSE$SOCKET = THIS$SOCKET;
 CALL XMX$SEND (.RECEIVE$BUF, 1532, ETHER$SOCKET, .STATUS);
 IF (STATUS AND BAD$STATUS) <> 0
  THEN CALL COM$ERROR ('SB', STATUS);

        /* Send a print command to PRINSE. */

 CALL EDL$SEND (EDL$TRANSMIT, .XMIT$HDR,
               XMIT$HDR.HDR$LENGTH + 26, .STATUS);
 IF STATUS > 0 THEN CALL COM$ERROR ('TR', STATUS);

           /* Now wait for PRINSE's answer. */

 STATUS = EMPTY;
 DO WHILE STATUS = EMPTY;
  RETURN$P = XMX$RECEIVE (.STATUS);
 END /* DO */;
 IF (STATUS AND BAD$STATUS) <> 0
  THEN CALL COM$ERROR ('CR', STATUS);
 IF RECEIVE$BUF.RESULT > 0
  THEN CALL COM$ERROR ('CR', RECEIVE$BUF.RESULT);

END PRINSE$SEND;

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

                  /*  File Parameters.  */

DECLARE INPUT          LITERALLY '1',
        STATUS         WORD,
        ACTUAL$COUNT   WORD,
        BUFFER (1496)  BYTE,
        DISK           CONNECTION;

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

         /*  Read console to get path name.  */
         /*  Then open input file.           */

 CALL READ (1, .BUFFER, 128, .ACTUAL$COUNT, .STATUS);
 CALL OPEN (.DISK, .BUFFER, INPUT, 0, .STATUS);
 IF STATUS > 0
  THEN DO;
   CALL ERROR (STATUS);
   CALL EXIT;
  END /* THEN */;

         /*  Start up the Ethernet Controller.  */

 CALL ETHER$INIT (.PRINSE$ADDRESS, .CONF$TEST, .STATUS);
 IF STATUS > 1 THEN CALL COM$ERROR ('ST', STATUS);
 IF CONF$TEST > 0 THEN CALL COM$ERROR ('CT', CONF$TEST);
```

```
                    /*  Set up type code.  */

CHANGE$TYPE = PRINT$TYPE;
CALL EDL$SEND (EDL$CONNECT, .XMIT$HDR, 20, .STATUS);
IF STATUS > 0 THEN CALL COM$ERROR ('CN', STATUS);

          /*  Initialize transmit header.  */

XMIT$HDR.HDR$LENGTH = 18;
XMIT$HDR.EXT$P = .BUFFER;
XMIT$HDR.EXT$IDS = 0;
XMIT$HDR.EXT$SEGMENT = 0;
CALL MOVE (6, .PRINSE$ADDRESS, .XMIT$HDR.DST$ADDRESS);
XMIT$HDR.TYPE = PRINT$TYPE;

        /*  Connect with the PRINSE program.  */

XMIT$HDR.PRINT$COMMAND = PRINT$START;
XMIT$HDR.EXT$LENGTH = 42;   /* Padding. */
XMIT$HDR.PRINT$LENGTH = 0;
CALL PRINSE$SEND;
IF PRINT$RESPONSE <> PRINT$OK
 THEN DO;
  CALL WRITE (0, .('REMOTE PRINT SERVER IS BUSY.',
                         0DH, 0AH), 30, .STATUS);
  CALL SUMMARY;
  CALL EXIT;
 END /* THEN */;

           /*  Send the whole disk file.  */

XMIT$HDR.PRINT$COMMAND = PRINT$DATA;
ACTUAL$COUNT = 1;

DO WHILE ACTUAL$COUNT <> 0;
 CALL READ (DISK, .BUFFER, 1494, .ACTUAL$COUNT, .STATUS);
 IF STATUS > 0 THEN CALL ERROR (STATUS);
 XMIT$HDR.PRINT$LENGTH = ACTUAL$COUNT;
 IF ACTUAL$COUNT > 42
   /*  Total data length must be >= 46.     */
   /*  Four bytes of data are in XMIT$HDR.  */
  THEN XMIT$HDR.EXT$LENGTH = ACTUAL$COUNT;
  ELSE XMIT$HDR.EXT$LENGTH = 42;
 CALL PRINSE$SEND;
 IF PRINT$RESPONSE <> PRINT$OK
  THEN DO;
   CALL WRITE (0, .('TRANSMISSION INTERRUPTED.',
                         0DH, 0AH), 27, .STATUS);
   CALL SUMMARY;
   CALL EXIT;
  END /* THEN */;
END /* DO */;

                   /*  Termination.  */

XMIT$HDR.PRINT$COMMAND = PRINT$END;
XMIT$HDR.EXT$LENGTH = 42;  /* Padding. */
CALL PRINSE$SEND;
CALL WRITE (0, .('FILE TRANSMITTED.',
                    0DH, 0AH), 19, .STATUS);
```

```
    CALL SUMMARY;
    CALL CLOSE (DISK, .STATUS);
    IF STATUS > 0 THEN CALL ERROR (STATUS);
    CALL EXIT;

END REMOTE$PRINT;
```

Print Server Program

```
PRINT$SERVER:  DO;

DECLARE WORD       LITERALLY 'ADDRESS',
        CONNECTION LITERALLY 'WORD',
        TRUE       LITERALLY '0FFFFH',
        FALSE      LITERALLY '0000H',
        FOREVER    LITERALLY 'WHILE TRUE';

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

                  /*  ISIS System Calls.  */

OPEN:
    PROCEDURE (CONN$P, PATH$P, ACCESS,
                    ECHO, STATUS$P) EXTERNAL;
    DECLARE (CONN$P, PATH$P, STATUS$P) ADDRESS,
            ACCESS   WORD,
            ECHO     CONNECTION;
    END OPEN;

READ:
    PROCEDURE (CONN, BUF$P, COUNT,
                ACTUAL$P, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            COUNT    WORD,
            (BUF$P, ACTUAL$P, STATUS$P) ADDRESS;
    END READ;
WRITE:
    PROCEDURE (CONN, BUF$P, COUNT, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            COUNT    WORD,
            (BUF$P, STATUS$P) ADDRESS;
    END WRITE;
CLOSE:
    PROCEDURE (CONN, STATUS$P) EXTERNAL;
    DECLARE CONN     CONNECTION,
            STATUS$P ADDRESS;
    END CLOSE;
EXIT:
    PROCEDURE EXTERNAL;
    END EXIT;
ERROR:
    PROCEDURE (ERRNUM) EXTERNAL;
    DECLARE ERRNUM WORD;
    END ERROR;
```

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

                         /*  XMX calls.  */

XMX$SEND:
    PROCEDURE  (BUFFER$PTR, BUFFER$LENGTH,
                SOCKET, STATUS$P)                  EXTERNAL;
    DECLARE BUFFER$PTR          ADDRESS,
            BUFFER$LENGTH       WORD,
            SOCKET              WORD;
    DECLARE STATUS$P            ADDRESS;
    END XMX$SEND;
XMX$RECEIVE:
    PROCEDURE (STATUS$P) ADDRESS                   EXTERNAL;
    DECLARE STATUS$P            ADDRESS;
    END XMX$RECEIVE;
ETHER$INIT: PROCEDURE
        (ECHO$ADDRESS$P, TEST$RESULT$P, STATUS$P) EXTERNAL;
    DECLARE ECHO$ADDRESS$P      ADDRESS;
    DECLARE TEST$RESULT$P       ADDRESS,
            STATUS$P            ADDRESS;
    END ETHER$INIT;

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

           /*  Remote Print Library Calls  */

HEX:
    PROCEDURE (IN$P, IN$LENGTH, OUT$P)             EXTERNAL;
    DECLARE IN$P                 ADDRESS,
            IN$LENGTH            BYTE,
            OUT$P                ADDRESS;
    END HEX;
HEX$WORD:
    PROCEDURE (BINARY, OUT$P)                      EXTERNAL;
    DECLARE BINARY               WORD,
            OUT$P                ADDRESS;
    END HEX$WORD;
EQUAL$ADDRESS:
    PROCEDURE (NET$ADR1$P, NET$ADR2$P) WORD        EXTERNAL;
    DECLARE NET$ADR1$P           ADDRESS,
            NET$ADR2$P           ADDRESS;
    END EQUAL$ADDRESS;
COM$ERROR:
    PROCEDURE (MNEMONIC, EDL$STATUS)               EXTERNAL;
    DECLARE MNEMONIC             WORD,
            EDL$STATUS           WORD;
    END COM$ERROR;
```

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

             /*  EDL communication areas.  */

DECLARE XMIT$BUF STRUCTURE
         (RSVRD (14)      BYTE,
          COMMAND         BYTE,
          RESULT          BYTE,
          RESPONSE$SOCKET WORD,
          BUF$LENGTH      WORD,
          EXT$P           ADDRESS,
          EXT$SEGMENT     WORD,
          EXT$LENGTH      WORD,
          DST$ADDRESS (6) BYTE,
          SRC$ADDRESS (6) BYTE,
          TYPE            WORD,
          PRINT$RESPONSE  WORD,
          PADDING (44)    BYTE );

DECLARE PRINT$OK     LITERALLY '8000H',   /* Responses. */
        PRINT$QUIT   LITERALLY '8001H';

DECLARE CHANGE$ADDRESS (6) BYTE AT (.XMIT$BUF.BUF$LENGTH),
        CHANGE$TYPE        WORD AT (.XMIT$BUF.BUF$LENGTH);

DECLARE RECEIVE$BUF STRUCTURE
         (RSVRD (14)        BYTE,
          COMMAND           BYTE,
          RESULT            BYTE,
          RESPONSE$SOCKET   WORD,
          BUF$LENGTH        WORD,
          DST$ADDRESS (6)   BYTE,
          SRC$ADDRESS (6)   BYTE,
          TYPE              WORD,
          PRINT$COMMAND     WORD,
          PRINT$LENGTH      WORD,
          PRINT$TEXT (1496) BYTE );

DECLARE USER$DATA (1500)  BYTE
                    AT (.RECEIVE$BUF.PRINT$COMMAND);

DECLARE PRINT$START  LITERALLY '0001H',   /* Commands. */
        PRINT$DATA   LITERALLY '0002H',
        PRINT$END    LITERALLY '0003H';

DECLARE ATTACHED$ADDRESS (6)  BYTE,
        ATTACHED              WORD INITIAL (FALSE),

        /* The following type code is assigned to
           Intel Corporation.  Refer to Chapter 1
           for information regarding its use. */

        PRINT$TYPE            LITERALLY '0950H',

        CONF$TEST             BYTE,
        RETURN$P              ADDRESS,
        ETHER$SOCKET          LITERALLY '0001H',
        THIS$SOCKET           LITERALLY '0100H';

DECLARE EDL$CONNECT   LITERALLY '01H',
        EDL$ADDMCID   LITERALLY '03H',
        EDL$TRANSMIT  LITERALLY '05H',
        EDL$SUPPLYBUF LITERALLY '06H';
```

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

EDL$SEND:  PROCEDURE (COMMAND, BUFFER$P,
                      BUFFER$LN, STATUS$P);

DECLARE COMMAND    BYTE,                    /* Input. */
        BUFFER$P   ADDRESS,
        BUFFER$LN  WORD;

DECLARE STATUS$P   ADDRESS,                 /* Output. */
        EDL$STATUS BASED STATUS$P WORD;

DECLARE REQUEST BASED BUFFER$P STRUCTURE  /* Local. */
         (RSRVD (14)      BYTE,
          COMMAND         BYTE,
          RESULT          BYTE,
          RESPONSE$SOCKET WORD),
        XMX$STATUS        WORD,
        BAD$STATUS        LITERALLY '0001H',
        EMPTY             LITERALLY '0FFH';

 REQUEST.COMMAND = COMMAND;
 REQUEST.RESPONSE$SOCKET = THIS$SOCKET;
 CALL XMX$SEND (BUFFER$P, BUFFER$LN,
                ETHER$SOCKET, .XMX$STATUS);
 IF (XMX$STATUS AND BAD$STATUS) = 0
  THEN DO;
   XMX$STATUS = EMPTY;
   DO WHILE XMX$STATUS = EMPTY;
    RETURN$P =  XMX$RECEIVE (.XMX$STATUS);
   END /* DO */;
  END /* THEN */;
 IF (XMX$STATUS AND BAD$STATUS) = 0
  THEN EDL$STATUS = REQUEST.RESULT;
  ELSE EDL$STATUS = XMX$STATUS;

END EDL$SEND;

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

REPRIN$SUPPLY:  PROCEDURE;

DECLARE BAD$STATUS        LITERALLY '0001H';

 RECEIVE$BUF.COMMAND = EDL$SUPPLYBUF;
 RECEIVE$BUF.RESPONSE$SOCKET = THIS$SOCKET;
 CALL XMX$SEND (.RECEIVE$BUF, 1532, ETHER$SOCKET,
                .STATUS);
 IF (STATUS AND BAD$STATUS) <> 0
  THEN CALL COM$ERROR ('SB', STATUS);

END REPRIN$SUPPLY;
```

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

REPRIN$RECEIVE:  PROCEDURE;

DECLARE BAD$STATUS          LITERALLY '0001H',
        EMPTY               LITERALLY '0FFH';

 /* Check whether previously supplied
    buffer has been filled.           */

 STATUS = EMPTY;
 DO WHILE STATUS = EMPTY;
  RETURN$P = XMX$RECEIVE (.STATUS);
 END /* DO */;
 IF (STATUS AND BAD$STATUS) <> 0
  THEN CALL COM$ERROR ('CR', STATUS);
 IF RECEIVE$BUF.RESULT > 0
  THEN CALL COM$ERROR ('CR', RECEIVE$BUF.RESULT);

END REPRIN$RECEIVE;

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

REPRIN$SEND:  PROCEDURE;

 CALL EDL$SEND (EDL$TRANSMIT, .XMIT$BUF,
               XMIT$BUF.BUF$LENGTH + 26, .STATUS);
 IF STATUS > 0 THEN CALL COM$ERROR ('TR', STATUS);

END REPRIN$SEND;

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

                  /*  File Parameters.  */

DECLARE OUTPUT          LITERALLY '2',
        STATUS          WORD,
        ACTUAL$COUNT    WORD,
        PRINT           CONNECTION,
        PRINT$PATH (20) BYTE;

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

      /*  Read console to get pathname.  */
      /*  Then open output file.         */

 CALL READ (1, .PRINT$PATH, 20, .ACTUAL$COUNT, .STATUS);
 CALL WRITE (0, .('ETHERNET PRINT SERVER.',
                   0DH, 0AH), 24, .STATUS);
 CALL OPEN (.PRINT, .PRINT$PATH, OUTPUT, 0, .STATUS);
 IF STATUS > 0
  THEN DO;
   CALL ERROR (STATUS);
   CALL EXIT;
  END /* THEN */;

 CALL WRITE (PRINT, .(0CH), 1, .STATUS); /* Form Feed */
 IF STATUS > 0
  THEN DO;
   CALL ERROR (STATUS);
   CALL EXIT;
  END /* THEN */;
```

```
        /*  Start up the Ethernet Controller.  */

CALL ETHER$INIT (0, .CONF$TEST, .STATUS);
IF STATUS > 1    THEN CALL COM$ERROR ('ST', STATUS);
IF CONF$TEST > 0 THEN CALL COM$ERROR ('CT', CONF$TEST);

              /*  Set up type code.  */

CHANGE$TYPE = PRINT$TYPE;
CALL EDL$SEND (EDL$CONNECT, .XMIT$BUF, 20, .STATUS);
IF STATUS > 0 THEN CALL COM$ERROR ('CN', STATUS);

          /*  Initialize transmit header.  */

XMIT$BUF.BUF$LENGTH = 60;
XMIT$BUF.EXT$LENGTH = 0;
XMIT$BUF.TYPE = PRINT$TYPE;

               /*  Supply a buffer.  */

CALL REPRIN$SUPPLY;

                 /*  Process loop.  */

DO FOREVER;
 CALL REPRIN$RECEIVE;
 IF RECEIVE$BUF.PRINT$COMMAND = PRINT$DATA
  THEN DO;
   IF ATTACHED AND EQUAL$ADDRESS (.RECEIVE$BUF.SRC$ADDRESS,
                                  .ATTACHED$ADDRESS)
    THEN DO;
     CALL WRITE (PRINT, .RECEIVE$BUF.PRINT$TEXT,
                        RECEIVE$BUF.PRINT$LENGTH, .STATUS);
     XMIT$BUF.PRINT$RESPONSE = PRINT$OK;
    END /* THEN */;
    ELSE XMIT$BUF.PRINT$RESPONSE = PRINT$QUIT;
  END /* THEN */;

  ELSE DO;
   IF RECEIVE$BUF.PRINT$COMMAND = PRINT$START
    THEN DO;
     IF NOT ATTACHED
      THEN DO;
       ATTACHED = TRUE;
       CALL MOVE (6, .RECEIVE$BUF.SRC$ADDRESS,
                     .ATTACHED$ADDRESS);
       XMIT$BUF.PRINT$RESPONSE = PRINT$OK;
      END /* THEN */;
      ELSE XMIT$BUF.PRINT$RESPONSE = PRINT$QUIT;
    END /* THEN */;
```

```
        ELSE DO;
         IF RECEIVE$BUF.PRINT$COMMAND = PRINT$END
          THEN DO;
           CALL WRITE (PRINT, .(0CH), 1, .STATUS);
                        /* Form Feed. */
           ATTACHED = FALSE;
           XMIT$BUF.PRINT$RESPONSE = PRINT$OK;
          END /* THEN */;
          ELSE XMIT$BUF.PRINT$RESPONSE = PRINT$QUIT;
        END /* ELSE */;

      END /* ELSE */;

     /* Get ready for the next print command. */
     CALL REPRIN$SUPPLY;
     /* Send the print response. */
     CALL MOVE (6, .RECEIVE$BUF.SRC$ADDRESS,
                  .XMIT$BUF.DST$ADDRESS);
     CALL REPRIN$SEND;

    END /* FOREVER */;

   END PRINT$SERVER;
```

APPENDIX A
CONFIDENCE TEST RESULTS

Results of the confidence tests are returned by the Ethernet Communications Controller during initialization. (Refer to Chapter 2 for details of the interface.) The result code identifies the test that failed, as indicated below:

Processor Board

01H —DRAM data ripple.
02H —DRAM memory march.
03H —SRAM data ripple.
04H —SRAM memory march.
05H —Lower PROM CRC.
06H —Upper PROM CRC.
07H —8255A read-after-write.
08H —8237 read-after-write.
09H —8259 read-after-write.
0AH—8253 counter 0.
0BH—8253 counter 1.
0CH—8253 counter 2.
0DH—DMA channel 1.
0EH—DMA channel 2.
0FH —DMA channel 3.

SerDes Board

10H —Ethernet address CRC.
11H —Broadcast packet loopback.
12H —Receive incorrect CRC.
13H —Address recognition: accept.
14H —Address recognition: reject.
15H —Transmit loopback failure: tests 11 through 14.

APPENDIX B
MULTIBUS INTERPROCESSOR PROTOCOL (MIP)

What is MIP?

The Multibus Interprocessor Protocol (MIP) is a specification of a set of mechanisms and protocols that enable reliable and efficient exchange of data among tasks executing on various single-board computers connected to a common Multibus system bus. Since MIP is a specification, it only becomes useful to you when it is implemented. This implementation is known as a MIP facility. The MIP specification ensures compatibility among MIP facilities. For an example of how MIP facilities are used in a Multibus configuration of single-board computers, see figure B-1.

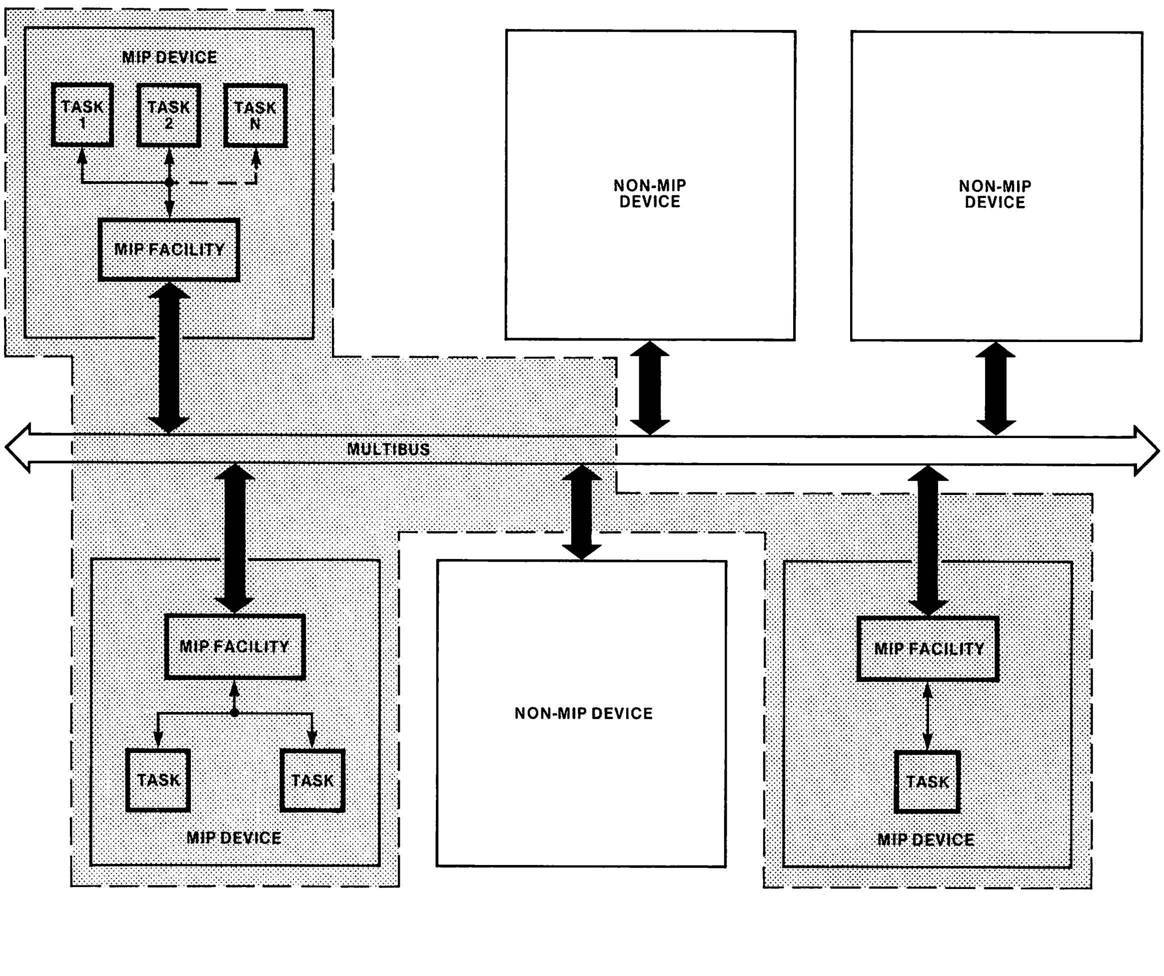

Figure B-1. A MIP System. 769-21

MIP facilities isolate user tasks from the complexities of communicating across the Multibus system bus. Without MIP facilities, tasks trying to communicate across the bus would have to solve one or more of the following problems:

- The tasks may be running on different kinds of processors.
- The tasks may be running under different kinds of operating systems.
- Different boards have different Multibus signalling mechanisms.
- Not all boards share the same memory space.
- Boards sometimes share memory but reference it by different addresses.
- Tasks sharing areas of memory may interfere with one another if not correctly coordinated.

MIP facilities hide these details from user tasks, thereby making it easier to develop programs for Multibus configurations that include several intelligent boards.

MIP supports communication among intelligent devices such as single-board computers and intelligent device controllers. MIP can be used by any device on which a MIP facility can be programmed. The design of MIP does not limit the kinds of processors or operating systems that can execute MIP facilities. MIP can be used by the MCS-85 or the iAPX-86 families of processors. MIP facilities can run under the ISIS-II, iRMX-80, iRMX-86, or iRMX-88 operating systems. In addition, you can implement MIP facilities to run on other processors or under other operating systems.

Implementing MIP

When using this specification as a guide for implementing MIP, be aware that it deals only with global concerns; implementational details (for example, initialization or memory management) are not addressed. You may add features that enable your implementation to better interface with its local environment (*e.g.*, the processor, the operating system, or application tasks). Be aware also that the specification assumes a general processing environment. For example, the algorithms in the specification are designed to work in a multitasking environment. If your environment is simpler, you may streamline your implementation, *as long as you retain the basic protocol needed to communicate with other versions of MIP*.

When implementing MIP using the MIP model, follow these guidelines:

- If an element or structure is never shared with another MIP facility, then its function in the model is merely descriptive.
- If an algorithm requires the cooperation of another communicating MIP facility, then the algorithm is required.

The MIP Model

Basic Components

A software application consists of several functional units called *tasks*. A task may be a program, a part of a program, or a system of related programs.

MIP facilities support communication among tasks that are executing on different processor boards attached to a common Multibus system bus. A MIP facility is a functioning implementation of MIP. The set of intercommunicating tasks, along with associated processor boards, operating systems, and MIP facilities, is called a *MIP system*. Each processor board in a MIP system runs a MIP facility. Each MIP facility may be a different implementation of MIP, but adherence to this specification ensures compatibility among them.

The term *device* is used for each processor board in a MIP system. Each device has a *device-ID*, a number ranging from zero to the number of devices communicating in one MIP system (less 1).

Any two tasks can communicate with each other by passing data in an area of memory that is accessible by both of the devices on which the tasks execute. A contiguous block of memory through which data is passed under control of MIP facilities is called a *buffer*. The content of buffers is not interpreted by MIP facilities.

Communications are delivered to tasks at *ports*. A port is a logical delivery mechanism that enables delivery in "first-in, first-out" (FIFO) order. In the MIP model, a port is represented as a queue. In some operating systems, ports are called "mailboxes" or "exchanges". The ports at a given device are identified by a *port-ID*, a number that ranges from zero to the number of ports (less 1) at the device. To provide system-wide addressability, a port is also identified by a *socket*, a pair of items in the form (d,p), where "d" is the device-ID and "p" is the port-ID.

Refer now to figure B-2. Task B on device 0 is receiving communications at port 1, also known as socket (0,1). Task C is active at socket (1,0). Socket (1,1) is not active (no task is receiving messages). Socket (2,1) is not defined.

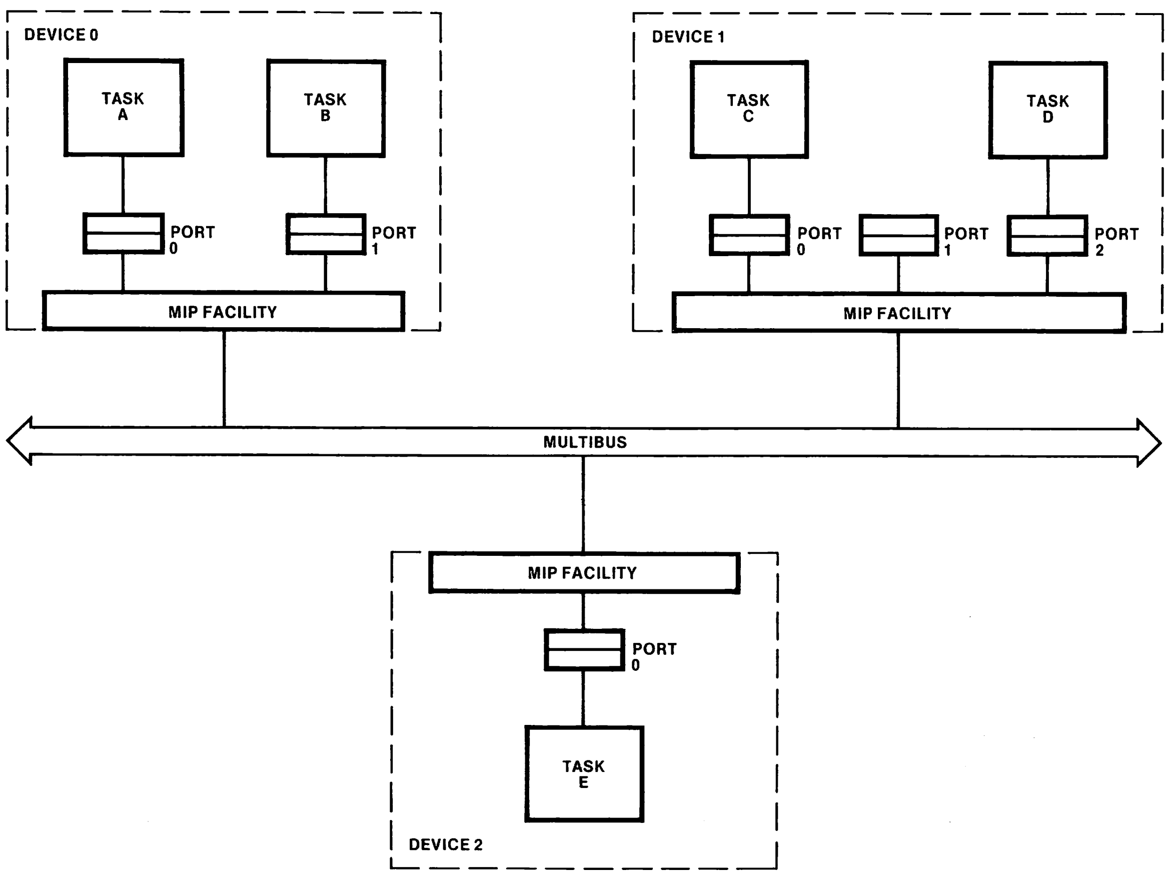

Figure B-2. A Configuration of Ports. 769-22

Each port is also known by a *function-name*. Function-names are symbolic means of identifying ports, making tasks that identify ports by their function-names independent of changes in configuration.

Three-Level Structure

The MIP model is composed of three levels of interface:

1. The *virtual* level, by which user tasks interact with the MIP facility
2. The *physical* level, by which MIP facilities on different devices interact with each other
3. The *logical* level, that translates between the virtual level and the physical level

An implementation of MIP must rigidly adhere to the functions, structures, and constants specified here for the physical level. Any implementation that deviates from this requirement is not compatible with the MIP architecture and may not be able to communicate with other MIP facilities.

At the logical level, however, the algorithms and data structures specified here merely impose a logical framework. Implementations need only satisfy the relationships between events and actions, but do not need to duplicate either the algorithms or data structures as defined.

The virtual level of the model simply suggests one way for tasks to view the MIP system. Any other viewpoint will work as well, so long as the information passed thru the virtual level interface is sufficient to accomplish the desired results. You may wish to create an interface that is more consistent with the interfaces to the operating system you are using.

Figure B-3 illustrates the three-level structure. Refer to this figure during the following discussion.

Physical Level

The physical communication mechanism between devices is a fixed size, unidirectional, FIFO queue called a *Request Queue*. An element in a request queue is known as a *Request Queue Entry* (RQE). An RQE is added to a Request Queue at the "*give*" end of the queue and removed from the "*take*" end. Each Request Queue is managed by a *Request Queue Descriptor* (RQD). An RQD and associated RQE's forming one queue occupy a contiguous block of memory, as illustrated in figure B-4. The RQD keeps track of the give and take locations as well as other information about the queue.

Each Request Queue contains at least two RQEs, and each queue is accessed at the give end by only one device and at the take end by only one device. This helps to avoid memory contention between devices using the same queue.

Two-way communication between two devices is implemented by a pair of Request Queues, known collectively as a *channel*. The device that uses the give end of a request queue is the *owner* of the queue. The owner is responsible for initializing the queue. See figure B-5 for a conceptual diagram of a channel.

Logical Level

The logical level of the MIP model uses Request Queues to transfer *requests* between source and destination MIP facilities. A request is either a *command* or a *response*. A command is an order sent from a source MIP facility to a destination facility. A

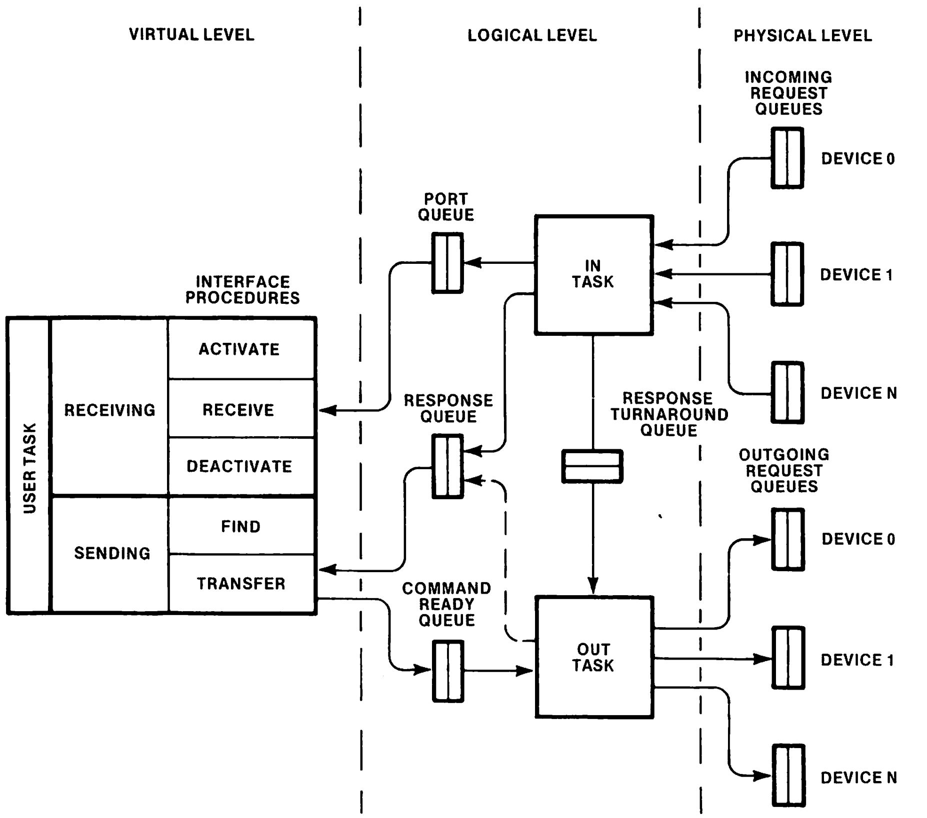

Figure B-3. Data-Flow Structure of the MIP Model. 769-23

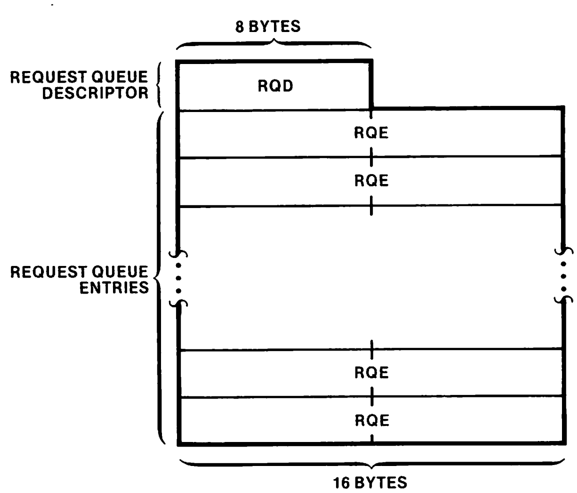

Figure B-4. Format of a Request Queue. 769-4

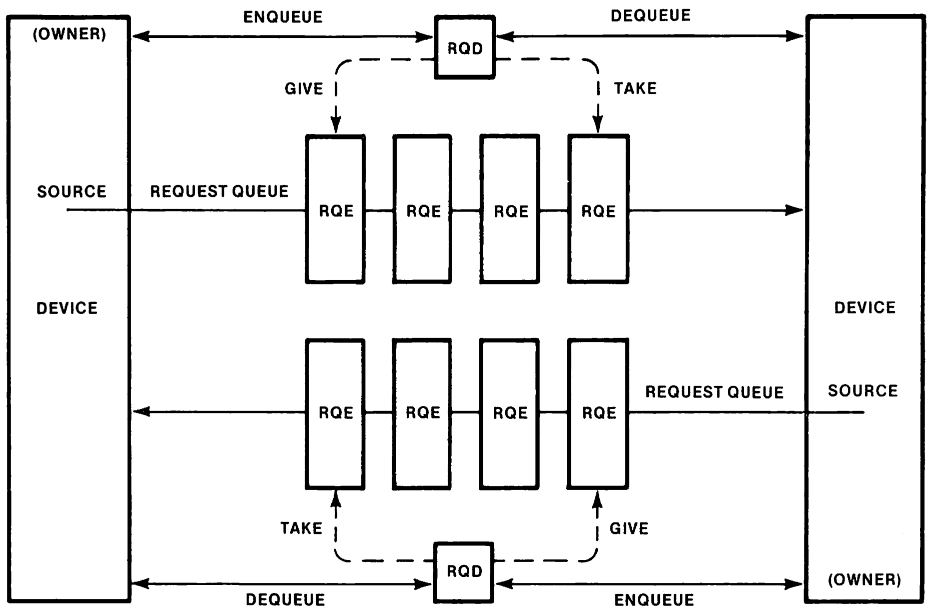

Figure B-5. Conceptual Structure of a Channel. 769-24

response is returned from the destination facility to the source facility and indicates the results of an attempt to deliver a command. The Request Queues carry these requests and their associated parameters between MIP facilities.

The primary procedures of the logical level are IN$TASK and OUT$TASK. In the MIP model these are viewed as asynchronous tasks, thereby giving the flexibility needed to service several user tasks simultaneously in a multi-tasking environment. Since they are asynchronous, all communication with IN$TASK and OUT$TASK is through queues. There is one Port Queue for each destination task and one Response Queue for each source task. For each channel there is one Command Ready Queue, one Response Turnaround Queue, and one incoming and one outgoing Request Queue. (See figure B-3.)

In the MIP model, the Port Queue may contain entire buffers for reasons discussed below under "Buffer Movement." The other queues contain only buffer descriptors, thereby minimizing movement of data in memory.

IN$TASK is driven by its incoming Request Queues. Requests in these queues may be either commands or responses. Commands are routed to the Port Queue of the destination port; a response is generated and queued in the Response Turnaround Queue to be sent back to the source MIP facility by OUT$TASK. Responses from the incoming Request Queues are routed to the Response Queue of the originating task.

OUT$TASK is driven by the Command Ready Queues and Response Turnaround Queues. When OUT$TASK finds a command in one of its Command Ready Queues, it routes it to the destination device's Request Queue. (When a destination device is not functioning, OUT$TASK sends a response directly back to the sending task's Response Queue.) When OUT$TASK finds a response in one of the Response Turnaround Queues, it routes it to the Request Queue of the source task's device.

Virtual Level

User tasks interact with the MIP facility by use of five procedures:

- For sending buffers:
  1. FIND—locates a port, given its function-name
  2. TRANSFER—initiates transfer of a buffer to a given port by placing a command in the destination device's Command Ready Queue. TRANSFER then waits for a response before allowing the sending task to continue.
- For receiving buffers:
  3. ACTIVATE—attaches a task to a port and enables reception of messages at that port
  4. RECEIVE—completes transfer of a buffer by taking a command from the task's Port Queue
  5. DEACTIVATE—disconnects a task from its port and terminates reception of commands at that port

Memory Management

Devices in a MIP system communicate via shared memory. The abilities of the devices to access the memory available on the Multibus system bus can be used to define a partition of that memory. The MIP model partitions all of memory into non-overlapping segments such that, for any segment and any device, either

- The segment is continuously addressable within the address space of the device, or
- The device cannot address any of the segment.

Each segment that can be shared among devices is called an *inter-device segment* (*IDS*) and is identified by an *IDS-ID* (a number ranging from zero to the number of IDS's (less 1) in the MIP system).

Figure B-6 presents a hypothetical memory configuration and shows how the address space is partitioned. Processor A and processor C can communicate through IDS 1. Processor B and processor C can communicate through IDS's 0, 1, and 3. IDS 3, however, is a segment of dual-ported memory and is accessed by processor B using a different range of addresses than processor C uses. Memory segments A,B, and C cannot be used for inter-device communication.

Table B-1 summarizes the memory configuration shown in figure B-6. The table shows the lowest address (the *base address*) by which each device can access each IDS.

Table B-1. System Inter-Device Segment Table.

IDS	Length	Base Addresses		
		Device 0	Device 1	Device 2
0	8000H		18000H	18000H
1	8000H	10000H	10000H	10000H
2	8000H		8000H	20000H

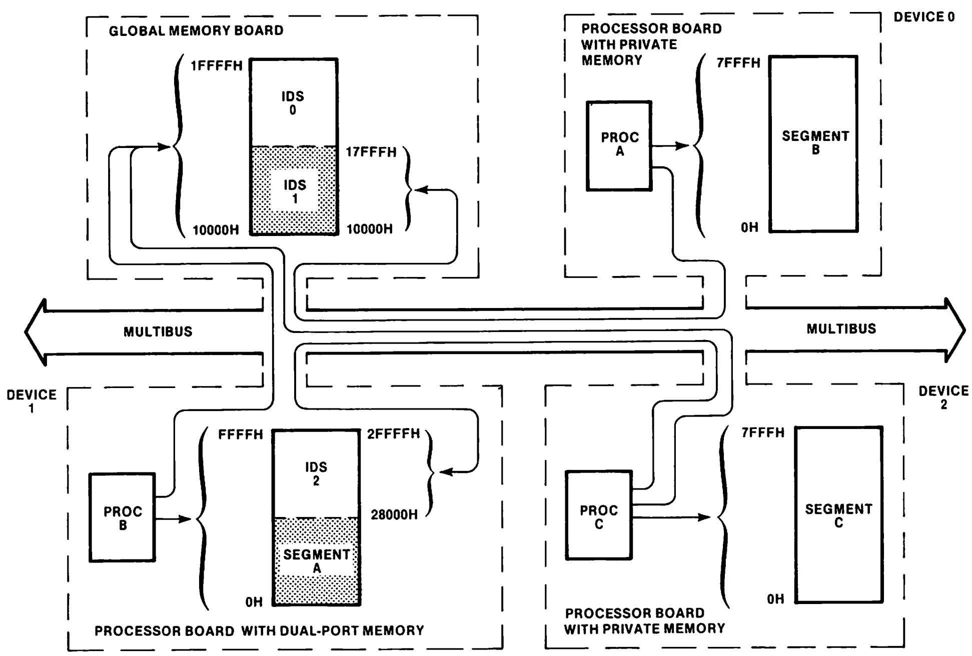

Figure B-6. Example of Inter-Device Memory Segments. 769-6

The MIP model contains special features for handling the "alias" problem posed by dual-port memory. Dual-port memory may be addressed differently from the Multibus system bus than from its local processor. The only case of a shared memory address in a MIP system is the buffer pointer in the RQE. This pointer is stored in a special format, called an *IDS pointer*, that is independent of the addressing peculiarities of the different devices in a MIP system. The MIP pointer is 32 bits wide, permitting an addressing range of 4 gigabytes. The high-order word (16 bits) of the pointer stores the low-order word of the address, and the low-order word of the pointer stores the high-order word of the address. Within each word, the low-order byte is stored before the high-order byte.

When a buffer is transferred, the sending MIP facility converts the local buffer pointer to the MIP pointer format and *normalizes* it by subtracting the IDS base address of the sending device. Upon receiving the RQE, the receiving MIP facility adds the IDS base address of the receiving device and converts to the format required by the receiving device's processor. In this way, user tasks are not concerned with these addressing problems.

Buffer Movement

Generally, buffers are not physically moved from one memory location to another any more often than necessary. Instead, buffers are referenced by descriptors in the RQEs. However, the MIP model provides for operating systems whose memory management policies forbid introduction of new objects (buffers) into their memory spaces. When delivering a buffer, the MIP model copies the buffer from the space managed by the sending operating system into the space managed by the receiving operating system. In such a case, a special status code is returned, so that the sender can know when the buffer is available for reuse.

Signalling

MIP uses a signalling mechanism for efficient utilization of the inter-device request queues. The mechanism is a software handshake using flags in the signal bytes of the RQDs. This mechanism permits MIP facilities to decrease their activity when queue activity decreases.

IN$TASK does not examine incoming request queues that are known to be empty. When the OUT$TASK of a sending facility puts a request in an outgoing queue that was previously empty, it also sets a flag to signal the IN$TASK of the receiving facility that the queue is no longer empty.

Similarly, OUT$TASK does not examine outgoing request queues that are known to be full. When the IN$TASK of a receiving facility removes a request from an incoming queue that was previously full, it also sets a flag to signal the OUT$TASK of the sending facility that the queue is no longer full.

When a MIP facility sets a signal flag it may generate an interrupt for the destination processor. A MIP facility designed to respond to interrupts does not need to examine its signal flags until it receives an interrupt. Reception of an interrupt signifies either that a previously empty input queue now has at least one entry or that a previously full output queue now has at least one empty space. By scanning the signal flags of all devices, the MIP facility can determine which device generated the interrupt.

There are several techniques available for generating interrupts. Which of the following methods you use depends both on the capabilities of the devices involved and on the requirements of the processing environment.

- No interrupt; the device polls the RQD. This technique is suitable if a processor is running only one task or if there is some way of guaranteeing that the RQDs are examined requlary.
- I-O mapped. Some devices (such as the iSBC 550 Ethernet Communications Controller) recognize a write to a specific I-O port address as an interrupt. This is a highly reliable technique; it should be used when available.
- Memory mapped. Some devices (such as the iSBC 544 Intelligent Communications Controller) recognize a write to a specific memory address as an interrupt. This is also a reliable technique.
- Edge level. The sending device raises one of the Multibus interrupt lines after lowering it briefly. The rising edge triggers a processor interrupt. This technique is available on most current Intel processor boards, such as the 80/30, 80/24, and 86/12A.
- Pure level. The sending device asserts one of the Multibus interrupt lines. (If the interrupt line is shared by several devices, the sending device must drop the line after a limited time to avoid continually re-interrupting all the devices.) If the receiving processor has interrupts enabled and is not busy processing other interrupts during this time, an interrupt is triggered. You must implement some kind of signal (such as another interrupt) that enables the receiving device to cause the sending device to drop the interrupt line before the receiving device services the interrupt. To guard against missed interrupts, the receiving MIP facility should periodically poll the signal flags in its incoming request queues.

Error Handling

The MIP architecture provides for device failure. A device is assumed to have failed if it does not return a response to a command within a certain time. The timeout period is implementation-dependent.

When a MIP facility determines that a destination device has failed, it takes three actions:

1. It sets flags to prevent any further activity on the channel.
2. It discards any responses destined for the dead device.
3. It returns all commands for the dead device to the tasks that invoked them (along with an appropriate error indication).

Any further recovery actions are application dependent.

Procedural Specification

Data Types

The following data types are used in the algorithmic specification of MIP:

BYTE: Standard 8-bit variable

WORD: Two-BYTE variable

IDENTIFIER: BYTE variable generally used as an index into an array

STATE: BYTE variable restricted to state constants

POINTER: Device-dependent address reference

IDS$PTR: Two-WORD, device-independent address reference

Processor-Dependent Subroutines

All machine-dependent logic in the algorithmic specification is isolated in the following procedures. In addition to these procedures, the value NULL$PTR is used for some unique pointer value that can serve to indicate a null value. For example:

```
DECLARE  NULL$PTR  LITERALLY  '0000H';
```

Ptr$add

Any implementation of MIP must handle pointer arithmetic according to the requirements of the processor that executes that implementation. Pointer arithmetic is used to calculate the addresses of request queue elements.

```
PTR$ADD:  PROCEDURE  (PTR,
                      SCALAR)  POINTER;

DECLARE PTR        POINTER,           /* Input. */
        SCALAR     BYTE;

DECLARE NEW$PTR    POINTER;           /* Local. */

 /*
     Using knowledge of processor-dependent POINTER
     implementation, add PTR to SCALAR giving NEW$PTR.
 */

 RETURN NEW$PTR;

END PTR$ADD;
```

Convert$local$adr

This routine converts from an address pointer in the local address space to an IDS-relative pointer in the IDS$PTR format. Details of this conversion depend on the pointer format dictated by the local processor.

```
CONVERT$LOCAL$ADR: PROCEDURE (IDS$ID,
                              BUFFER$PTR,
                              MIP$PTR);

DECLARE IDS$ID       IDENTIFIER,    /* Input. */
        BUFFER$PTR   POINTER;

DECLARE MIP$PTR      IDS$PTR;       /* Output. */

 /*
     Get base address for IDS$ID from IDST.
     Subtract from BUFFER$PTR.
 */ ;

END CONVERT$LOCAL$ADR;
```

Convert$system$adr

This routine converts from an IDS-relative pointer in the IDS$PTR format to an address pointer in the local address space. Details of this conversion depend on the pointer format dictated by the local processor.

```
CONVERT$SYSTEM$ADR: PROCEDURE (IDS$ID,
                               MIP$PTR,
                               BUFFER$PTR);

DECLARE IDS$ID       IDENTIFIER,        /* Input. */
        MIP$PTR      IDS$PTR;

DECLARE BUFFER$PTR   POINTER;           /* Output. */

 /*
     Get base address for IDS$ID from IDST.
     Add to BUFFER$PTR.
 */ ;

END CONVERT$SYSTEM$ADR;
```

Time$wait

A destination device is assumed to be dead if it does not respond to a command within a reasonable period of time. Just how you detect a timeout, however, depends on the timing features of the local processor.

```
TIME$WAIT:  PROCEDURE (TIME$OUT, RQL$ID);

DECLARE TIME$OUT WORD,                        /* Input. */
        RQL$ID   IDENTIFIER;

 /*
      Wait for TIME$OUT period or until something is
      placed in the response queue identified by RQL$ID.
 */ ;

END TIME$WAIT;
```

Generate$Interrupt

This routine generates an interrupt to signal another device of a change in queue status (from full to not full, of from empty to not empty).

```
GENERATE$INTERRUPT:  PROCEDURE  (DEVICE$INDEX);

DECLARE DEVICE$INDEX   IDENTIFIER;  /* Input */

 /*
      Using interrupt information in the DCM, generate an
      interrupt for the device specified by DEVICE$INDEX.
 */ ;

END GENERATE$INTERRUPT;
```

Clear$Interrupt

This routine is used by IN$TASK and OUT$TASK to clear the interrupt that invokes them.

```
CLEAR$INTERRUPT:  PROCEDURE;

   /* Acknowledge and clear interrupt, if necessary. */ ;

END CLEAR$INTERRUPT;
```

Physical Level

Request Queue Descriptor

A Request Queue Descriptor controls a request queue. It is physically located before and adjacent to the associated request queue entries.

```
DECLARE    RQD$STRUCTURE  LITERALLY 'STRUCTURE
            (EMPTY$SIGNAL  STATE,
             FULL$SIGNAL   STATE,
             RQ$SIZE       BYTE,
             RQE$LENGTH    BYTE,
             GIVE$INDEX    BYTE,
             GIVE$STATE    STATE,
             TAKE$INDEX    BYTE,
             TAKE$STATE    STATE)';
```

EMPTY$SIGNAL and FULL$SIGNAL are used by the two devices sharing a channel to signal each other when there has been some activity on the channel. Signals are written in the RQD of the outgoing queue and read from the RQD of the incoming queue. The signal values are defined below. Unused bits are reserved for future expansion.

```
DECLARE FULL$NO$LONGER               LITERALLY     '80H',
        EMPTY$NO$LONGER              LITERALLY     '01H',
        NO$CHANGE                    LITERALLY     '00H';
```

RQ$SIZE defines the number of elements in the request queue. RQ$SIZE must be a power of 2 and must have a value of 2 or greater.

RQE$LENGTH defines the number of bytes in a request queue element (RQE). The number of elements is 2 to the power of RQE$LENGTH. For all queues shared between MIP facilities, RQE$LENGTH is 4 (*i.e.*, each entry is 16 bytes long).

GIVE$INDEX identifies the request queue element available for enqueuing data.

TAKE$INDEX identifies the request queue element available for dequeuing data.

GIVE$STATE and TAKE$STATE contain the booleans defined below. Unused bits are reserved for future expansion.

```
DECLARE GIVE$HALT                   LITERALLY    '40H',
        GIVE$FACTOR                 LITERALLY    '80H';

DECLARE TAKE$HALT                   LITERALLY    '40H',
        TAKE$FACTOR                 LITERALLY    '80H';
```

GIVE$FACTOR and TAKE$FACTOR together distinguish between the full state and the empty state when GIVE$INDEX and TAKE$INDEX are equal.

GIVE$HALT and TAKE$HALT prevent further activity in the queue when a device failure is detected.

Request Queue Entry

A Request Queue Entry is an element of a request queue.

```
DECLARE RQE$STRUCTURE  LITERALLY 'STRUCTURE
          (REQUEST       STATE,
           SRC$REQ$ID    IDENTIFIER,
           DEST$DEV$ID   IDENTIFIER,
           DEST$PORT$ID  IDENTIFIER,
           SRC$DEV$ID    IDENTIFIER,
           DATA$PTR      IDS$PTR,
           DATA$LENGTH   WORD,
           IDS$ID        IDENTIFIER,
           OWNER$DEV$ID  IDENTIFIER,
           RSRVD  (3)    BYTE)';
```

REQUEST identifies the RQE as a command or a response, using one of the following values:

```
DECLARE SEND$COMMAND                 LITERALLY    '70H',
        MSG$DELIVERED$NO$COPY        LITERALLY    '80H',
        MSG$DELIVERED$COPY           LITERALLY    '82H',
        SYSTEM$MEMORY$NAK            LITERALLY    '85H',
        DEAD$DEVICE                  LITERALLY    '89H';
```

SRCREQID identifies the sending task so that responses can be returned. The meaning of this identifier is defined by the local MIP implementation.

DESTDEVID is the device identifier part of the destination socket.

DEST$PORT$ID is the port identifier part of the destination socket.

SRCDEVID identifies the device from which a request is issued.

DATA$PTR contains the IDS-relative address of a buffer to be delivered or returned by a MIP facility.

DATA$LENGTH specifies the number of bytes in a buffer.

IDS$ID tells which inter-device segment contains the buffer.

OWNER$DEVICE$ID identifies the device that manages or "owns" the buffer.

RSVRD is undefined space reserved for future expansion.

Queue Procedure Returns

The following constants are used to return the results of procedures associated with the request queues.

```
DECLARE READY                          LITERALLY     '00H',
        FULL                           LITERALLY    '0FFH',
        EMPTY                          LITERALLY    '0FFH',
        FIRST$GIVE                     LITERALLY     '20H',
        FIRST$TAKE                     LITERALLY     '20H',
        HALTED                         LITERALLY     '40H';
```

Init$request$queue

This procedure enters a request queue descriptor in memory, thereby initializing a request queue.

```
INIT$REQUEST$QUEUE:  PROCEDURE  (RQD$PTR,
                                 RQ$LEN);

DECLARE RQ$LEN                  BYTE,              /* Input. */
        RQD$PTR                 POINTER,
        RQD  BASED  RQD$PTR     RQD$STRUCTURE;

 RQD.EMPTY$SIGNAL = NO$CHANGE;
 RQD.FULL$SIGNAL  = NO$CHANGE;
 RQD.RQ$SIZE      = RQ$LEN;
 RQD.RQE$LENGTH   = 4;
 RQD.GIVE$INDEX   = 0;
 RQD.TAKE$INDEX   = 0;
 RQD.GIVE$STATE   = 0;
 RQD.TAKE$STATE   = 0;

END INIT$REQUEST$QUEUE;
```

Term$request$queue

This procedure sets the request queue flags to prevent subsequent activity on a channel.

```
TERM$REQUEST$QUEUE:  PROCEDURE  (RQD$IN$PTR,
                                 RQD$OUT$PTR);

DECLARE RQD$IN$PTR                    POINTER,      /* Input */
        RQD$OUT$PTR                   POINTER,
        IN$RQD  BASED RQD$IN$PTR      RQD$STRUCTURE,
        OUT$RQD BASED RQD$OUT$PTR     RQD$STRUCTURE;

 IN$RQD.TAKE$STATE  = IN$RQD.TAKE$STATE OR TAKE$HALT;
 OUT$RQD.GIVE$STATE = OUT$RQD.GIVE$STATE OR GIVE$HALT;

END TERM$REQUEST$QUEUE;
```

Queue$give$status

This procedure returns the status of a request queue without affecting the queue.

```
QUEUE$GIVE$STATUS: PROCEDURE (RQD$PTR,
                              STATUS);

DECLARE RQD$PTR             POINTER,           /* Input. */
        RQD BASED RQD$PTR   RQD$STRUCTURE;

DECLARE STATUS              BYTE;              /* Output. */

 IF (RQD.TAKE$STATE AND TAKE$HALT) = TAKE$HALT
  THEN DO;
   RQD.GIVE$STATE = RQD.GIVE$STATE OR GIVE$DISABLED;
   STATUS = HALTED;
  END /* THEN */;
  ELSE IF (RQD.GIVE$INDEX = RQD.TAKE$INDEX) AND
          ((RQD.GIVE$STATE AND GIVE$FACTOR) <>
           (RQD.TAKE$STATE AND TAKE$FACTOR))
   THEN STATUS = FULL;
   ELSE STATUS = READY;
 RETURN;

END QUEUE$GIVE$STATUS;
```

Request$give$pointer

This algorithm returns the address of a request queue element (if one is not in use) from the "send" or "give" side of the queue.

```
REQUEST$GIVE$POINTER:  PROCEDURE  (RQD$PTR,
                                   RQE$PTR,
                                   STATUS);

DECLARE RQD$PTR            POINTER,          /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;

DECLARE RQE$PTR            POINTER,          /* Output. */
        STATUS             BYTE;

 IF (RQD.TAKE$STATE AND TAKE$HALT)  = TAKE$HALT
  THEN DO;
   RQD.GIVE$STATE = GIVE$DISABLED;
   STATUS = HALTED;
   RETURN;
  END /* THEN */;
 IF (RQD.GIVE$INDEX = RQD.TAKE$INDEX) AND
     ((RQD.GIVE$STATE AND GIVE$FACTOR) <>
      (RQD.TAKE$STATE AND TAKE$FACTOR))
  THEN DO;
   STATUS = FULL;
   RETURN;
  END /* THEN */;
 STATUS = READY;
 RQE$PTR = PTR$ADD(RQD$PTR,
             SHL(RQD.GIVE$INDEX, RQD.RQE$LENGTH) + 8);
 RETURN;

END REQUEST$GIVE$POINTER;
```

Release$give$pointer

This algorithm is always executed after a successful REQUEST$GIVE$POINTER. It actually enters the element in the request queue, making it available for taking.

```
RELEASE$GIVE$POINTER:  PROCEDURE  (RQD$PTR,
                                   STATUS);

DECLARE RQD$PTR            POINTER,          /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;

DECLARE STATUS             BYTE;             /* Output. */

 IF (RQD.TAKE$INDEX = ((RQD.GIVE$INDEX + 1)
                      AND (RQD.RQ$SIZE - 1)) )
  THEN /* GIVE$FACTOR bit = NOT TAKE$FACTOR bit. */
       RQD.GIVE$STATE = (RQD.GIVE$STATE OR GIVE$FACTOR)
                    AND (RQD.TAKE$STATE AND TAKE$FACTOR);

 RQD.GIVE$INDEX =
  ((RQD.GIVE$INDEX + 1) AND (RQD.RQ$SIZE - 1));

 IF RQD.GIVE$INDEX =
   ((RQD.TAKE$INDEX + 1) AND (RQD.RQ$SIZE - 1))
  THEN STATUS = FIRST$GIVE;  /* Gave to an empty queue. */
  ELSE STATUS = READY;
 RETURN;

END RELEASE$GIVE$POINTER;
```

Request$take$pointer

This algorithm returns the address of a request queue element (if one is available) from the "receive" or "take" side of a request queue.

```
REQUEST$TAKE$POINTER:  PROCEDURE  (RQD$PTR,
                                   RQE$PTR,
                                   STATUS);

DECLARE RQD$PTR            POINTER,         /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;

DECLARE RQE$PTR            POINTER,         /* Output. */
        STATUS             BYTE;

 IF (RQD.GIVE$STATE AND GIVE$HALT) = GIVE$HALT
  THEN DO;
   RQD.TAKE$STATE = TAKE$DISABLED;
   STATUS = HALTED;
   RETURN;
  END /* THEN */;
 IF (RQD.GIVE$INDEX = RQD.TAKE$INDEX) AND
     ((RQD.GIVE$STATE AND GIVE$FACTOR) =
      (RQD.TAKE$STATE AND TAKE$FACTOR))
  THEN DO;
   STATUS = EMPTY;
   RETURN;
  END /* THEN */;
 STATUS = READY;
 RQE$PTR = PTR$ADD(RQD$PTR,
              SHL(RQD.TAKE$INDEX, RQD.RQE$LENGTH) + 8);
 RETURN;

END REQUEST$TAKE$POINTER;
```

Release$take$pointer

This algorithm is always executed after a successful REQUEST$TAKE$POINTER. It actually purges the element from the request queue, making the space available for a subsequent "give" operation.

```
RELEASE$TAKE$POINTER:  PROCEDURE  (RQD$PTR,
                                   STATUS);

DECLARE RQD$PTR            POINTER,          /* Input. */
        RQD BASED RQD$PTR  RQD$STRUCTURE;

DECLARE STATUS             BYTE;             /* Output. */

 IF (RQD.GIVE$INDEX = ((RQD.TAKE$INDEX + 1) AND
                       (RQD.RQ$SIZE - 1)) )
  THEN /* TAKE$FACTOR bit = GIVE$FACTOR bit. */
    RQD.TAKE$STATE = (RQD.TAKE$STATE AND NOT TAKE$FACTOR)
                  OR (RQD.GIVE$STATE AND GIVE$FACTOR);

 RQD.TAKE$INDEX =
  ((RQD.TAKE$INDEX + 1) AND (RQD.RQ$SIZE - 1));

 IF RQD.TAKE$INDEX =
   ((RQD.GIVE$INDEX + 1) AND (RQD.RQ$SIZE - 1))
  THEN STATUS = FIRST$TAKE;  /* Took from a full queue. */
  ELSE STATUS = READY;
 RETURN;

END RELEASE$TAKE$POINTER;
```

Logical Level Database

Configuration Constants

The following constants define the system configuration. In place of the descriptions printed in lower case, substitute the numbers that apply to your configuration.

```
DECLARE DEVICES        LITERALLY 'the number of devices in the MIP
                                  system',

        SOCKETS        LITERALLY 'the number of destination ports',

        PORTS          LITERALLY 'the number of local ports',

        HOME$DEVICE    LITERALLY 'the identifier of this device',

        TIME$DELAY     LITERALLY 'maximum time to wait for a response
                                  before a destination device is
                                  considered dead',

        IDS$S          LITERALLY 'the number of entries in the IDS
                                  table',

        RQL$S          LITERALLY 'the number of local response
                                  queues';
```

Destination Socket Descriptor Table (DSDT)

The DSDT contains information for locating sockets in a MIP system. Each entry associates a socket with a unique function-name. The MIP facility on each device has a DSDT containing entries for all sockets to which tasks on that device send messages.

```
DECLARE DSDT (SOCKETS) STRUCTURE
                (FUNCTION$NAME          WORD,
                 DEST$DEV$ID            IDENTIFIER,
                 DEST$PORT$ID           IDENTIFIER);
```

FUNCTION$NAME is a system-wide name for identifying the socket.

DESTDEVID is the device identifier of the device on which the socket resides.

DEST$PORT$ID is the local port identifier for the socket on the destination device. For the purposes of this algorithmic specification, DEST$PORT$ID is the index of the port in the Local Port Table on the destination device.

Local Port Table (LPT)

The Local Port Table is the list of ports and their parameters that are managed by a device. For the purposes of this algorithmic specification, the index of a port in the LPT is the port's identifier.

```
DECLARE LPT (PORTS) STRUCTURE
                (FUNCTION$NAME          WORD,
                 PORT$QUEUE$PTR         POINTER,
                 PORT$STATE             STATE);
```

FUNCTION$NAME is the system-wide name for identifying the port.

PORT$QUEUE$PTR is the address of the queue in which messages addressed to this port are delivered.

PORT$STATE tells whether a task is receiving messages at this port. Messages sent to the port are accepted if the port is active, rejected (returned) if the port is inactive. Values associated with this item are:

```
DECLARE INACTIVE                        LITERALLY     '00H',
        ACTIVE                          LITERALLY     '01H';
```

Device to Channel Map (DCM)

The DCM table is used to route messages among inter-task and inter-device request queues and to manage the flow of messages into and out of the queues. Each MIP facility has one entry in its DCM for every device in the MIP system, including the device on which the MIP facility resides. The device identifier of a device is its index into the DCM. Each entry in a DCM represents a possible link between the home device and the device associated with that entry. If no such link exists, CHANNEL$STATE contains IDLE.

```
DECLARE DCM (DEVICES) STRUCTURE
                (CHANNEL$STATE              STATE,
                 RQD$OUT$PTR                POINTER,
                 RQD$OUT$SIZE               BYTE,
                 RQD$IN$PTR                 POINTER,
                 RQD$IN$SIZE                BYTE,
                 COM$RDY$QUEUE$PTR          POINTER,
                 RSP$TRNRND$QUEUE$PTR       POINTER,
                 INTERRUPT$TYPE             BYTE,
                 INTERRUPT$ADDRESS          WORD);
```

CHANNEL$STATE is a local management variable in which the run-time state of a channel is maintained. This variable contains the booleans defined below.

```
DECLARE SEND$ACTIVE                 LITERALLY     '80H',
        SEND$FULL                   LITERALLY     '7FH',
        RECEIVE$ACTIVE              LITERALLY     '01H',
        RECEIVE$EMPTY               LITERALLY    '0FEH',
        DYING                       LITERALLY     '04H',
        IDLE                        LITERALLY     '08H';
```

RQDOUTPTR is the local address of the RQD of the interprocessor queue through which commands and responses are sent to the associated device.

RQDOUTSIZE is the number of entries in this queue.

RQDINPTR is the local address of the RQD of the interprocessor request queue through which commands and responses are received from the associated device.

COMRDYQUEUE$PTR is the address of the local queue of commands waiting to be sent to the associated device.

RSP$TRNRND$QUEUE$PTR is the address of the local queue of responses waiting to be sent to the associated device.

INTERRUPT$TYPE tells which kind of interrupt the device recognizes as indication of a change of queue state.

INTERRUPT$ADDRESS may contain an I-O port address, a memory address, or an interrupt level, depending on INTERRUPT$TYPE.

Inter-Device Segment Table (IDST)

The IDST defines the attributes of Inter-Device Segments (IDS's). There is one entry for each IDS in the MIP system. The entries are indexed by the IDS identifier.

```
DECLARE IDST (IDS$S) STRUCTURE
                (LO$PART                    WORD,
                 HI$PART                    WORD);
```

Note that the low-order portion of the IDS base address is stored first, followed by the high-order portion.

Response Queue List (RQL)

The RQL is a table of pointers to the request queues used to return the results of a buffer delivery attempt. Each entry is assigned to a task for use with the TRANSFER function. The entries are indexed by RQL$ID.

```
DECLARE RQL (RQL$S) STRUCTURE
                (RSP$QUEUE$PTR          POINTER);
```

Logical Level Algorithms

Dying$channel

OUT$TASK invokes this subroutine when a device failure is detected. The routine disposes of any commands that may be waiting to be sent to the dead device.

```
DYING$CHANNEL:  PROCEDURE (DEVICE$INDEX);

DECLARE DEVICE$INDEX                BYTE;        /* Input. */

DECLARE STATUS                      BYTE,        /* Local. */
        RQE$COM$PTR                 POINTER,
        COM$RQE BASED RQE$COM$PTR RQE$STRUCTURE,
        RQE$RSP$PTR                 POINTER,
        RSP$RQE BASED RQE$RSP$PTR RQE$STRUCTURE;

 CALL REQUEST$TAKE$POINTER
       (DCM(DEVICE$INDEX).COM$RDY$QUEUE$PTR,
        RQE$COM$PTR,
        STATUS);
 IF STATUS <> EMPTY
  THEN DO; /* Send back DEAD$DEVICE response. */
   CALL REQUEST$GIVE$POINTER
         (RQL(COM$RQE.SRC$REQ$ID).RSP$QUEUE$PTR,
          RQE$RSP$PTR,
          STATUS);
   CALL MOVE (16, RQE$COM$PTR, RQE$RSP$PTR);
   RSP$RQE.REQUEST = DEAD$DEVICE;
   CALL RELEASE$GIVE$POINTER
         (RQL(COM$RQE.SRC$REQ$ID).RSP$QUEUE$PTR,
          STATUS);
   CALL RELEASE$TAKE$POINTER
         (DCM(DEVICE$INDEX).COM$RDY$QUEUE$PTR,
          STATUS);
  END /* THEN */;
  ELSE /* No more outstanding commands. */ DO;
   DCM(DEVICE$INDEX).CHANNEL$STATE = IDLE;
   CALL TERM$REQUEST$QUEUE
         (DCM(DEVICE$INDEX).RQD$IN$PTR,
          DCM(DEVICE$INDEX).RQD$OUT$PTR);
  END /* ELSE */;
 RETURN;

END DYING$CHANNEL;
```

Serve$turnaround$queue

This subroutine of OUT$TASK transfers a response from the Response Turnaround Queue to the output queue of the sending device.

```
SERVE$TURNAROUND$QUEUE:  PROCEDURE (DEVICE$INDEX,
                                    STATUS);

DECLARE DEVICE$INDEX                BYTE;       /* Input. */

DECLARE STATUS                      BYTE;       /* Output. */
```

```
DECLARE RQD$PTR                       POINTER,   /* Local. */
        RQD     BASED RQD$PTR         RQD$STRUCTURE,
        RQE$TRN$PTR                   POINTER,
        TRN$RQE BASED RQE$TRN$PTR RQE$STRUCTURE,
        RQE$OUT$PTR                   POINTER,
        OUT$RQE BASED RQE$OUT$PTR RQE$STRUCTURE;

 CALL REQUEST$TAKE$POINTER
       (DCM(DEVICE$INDEX).RSP$TRNRND$QUEUE$PTR,
        RQE$TRN$PTR,
        STATUS);
 IF STATUS = READY
  THEN DO;
   RQD$PTR = DCM(DEVICE$INDEX).RQD$OUT$PTR;
   CALL REQUEST$GIVE$POINTER (RQD$PTR,
                              RQE$OUT$PTR,
                              STATUS);
   CALL MOVE (16, RQE$TRN$PTR, RQE$OUT$PTR);
   CALL RELEASE$GIVE$POINTER (RQD$PTR,
                              STATUS);
   IF STATUS = FIRST$GIVE
    THEN DO; /* Gave to an empty queue, so... */
     RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER;
     CALL GENERATE$INTERRUPT (DEVICE$INDEX);
    END /* THEN */;
   CALL RELEASE$TAKE$POINTER
         (DCM(DEVICE$INDEX).RSP$TRNRND$QUEUE$PTR,
          STATUS);
  END /* THEN */;
 RETURN;

END SERVE$TURNAROUND$QUEUE;
```

Serve$command$queue

This subroutine of OUT$TASK transfers a command from the Command Wait Queue to the output queue of the destination device.

```
SERVE$COMMAND$QUEUE:  PROCEDURE (DEVICE$INDEX,
                                 STATUS);

DECLARE DEVICE$INDEX                  BYTE;      /* Input. */

DECLARE STATUS                        BYTE;      /* Output. */

DECLARE RQD$PTR                       POINTER,  /* Local. */
        RQD     BASED RQD$PTR         RQD$STRUCTURE,
        RQE$COM$PTR                   POINTER,
        COM$RQE BASED RQE$COM$PTR RQE$STRUCTURE,
        RQE$OUT$PTR                   POINTER,
        OUT$RQE BASED RQE$OUT$PTR RQE$STRUCTURE;

 CALL REQUEST$TAKE$POINTER
       (DCM(DEVICE$INDEX).COM$RDY$QUEUE$PTR,
        RQE$COM$PTR,
        STATUS);
```

```
 IF STATUS = READY
  THEN DO;
   RQD$PTR = DCM(DEVICE$INDEX).RQD$OUT$PTR;
   CALL REQUEST$GIVE$POINTER (RQD$PTR,
                              RQE$OUT$PTR,
                              STATUS);
   CALL MOVE (16, RQE$COM$PTR, RQE$OUT$PTR);
   CALL RELEASE$GIVE$POINTER (RQD$PTR,
                              STATUS);
   IF STATUS = FIRST$GIVE
    THEN DO; /* Gave to an empty queue, so... */
     RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER;
     CALL GENERATE$INTERRUPT (DEVICE$INDEX);
    END /* THEN */;
   CALL RELEASE$GIVE$POINTER
         (DCM(DEVICE$INDEX).COM$RDY$QUEUE$PTR,
          STATUS);
  END /* THEN */;
 RETURN;

END SERVE$COMMAND$QUEUE;
```

Out$task

This algorithm manages activity in the output request queues.

```
OUT$TASK: PROCEDURE;

DECLARE DEVICE$INDEX                BYTE,        /* Local. */
        STATUS                      BYTE,
        RQD$PTR                     POINTER,
        RQD     BASED RQD$PTR       RQD$STRUCTURE;

/*  Initialization.  */

 DO  DEVICE$INDEX = 0 TO DEVICES - 1;
  IF DCM(DEVICE$INDEX).CHANNEL$STATE <> IDLE
   THEN DO;
    CALL INIT$REQUEST$QUEUE(DCM(DEVICE$INDEX).RQD$OUT$PTR,
                    DCM(DEVICE$INDEX).RQD$OUT$SIZE);
    DCM(DEVICE$INDEX).CHANNEL$STATE =
                    SEND$ACTIVE;
   END /* THEN */;
 END /* DO */;

/*  Transfer request loop.  */

 DO FOREVER;
  DO  DEVICE$INDEX = 0 TO DEVICES - 1;
   RQD$PTR = DCM(DEVICE$INDEX).RQD$IN$PTR;
   /* Read signal from in-RQD. */
   IF RQD.FULL$SIGNAL = FULL$NO$LONGER
    THEN DO;
     DCM(DEVICE$INDEX).CHANNEL$STATE =
      DCM(DEVICE$INDEX).CHANNEL$STATE OR RQD.FULL$SIGNAL;
     CALL CLEAR$INTERRUPT;
     RQD.FULL$SIGNAL = NO$CHANGE;
    END /* THEN */;
   IF (DCM(DEVICE$INDEX).CHANNEL$STATE AND DYING) <> 0
    THEN CALL DYING$CHANNEL (DEVICE$INDEX);
```

```
    ELSE DO;
     IF DCM(DEVICE$INDEX).CHANNEL$STATE
        AND SEND$ACTIVE <> 0
       THEN DO; /* Look more closely at this channel. */
        RQD$PTR = DCM(DEVICE$INDEX).RQD$OUT$PTR;
        CALL QUEUE$GIVE$STATUS(RQD$PTR,
                               STATUS);
        IF STATUS = HALTED
         THEN DCM(DEVICE$INDEX).CHANNEL$STATE = DYING;
        IF STATUS = FULL
         THEN DCM(DEVICE$INDEX).CHANNEL$STATE =
              DCM(DEVICE$INDEX).CHANNEL$STATE AND SEND$FULL
              /* Don't bother with trying to send on this
                 channel until it is no longer full. */;

        IF STATUS = READY
         THEN DO;
               CALL  SERVE$TURNAROUND$QUEUE  (DEVICE$INDEX,
STATUS);
          IF STATUS = EMPTY
           THEN CALL SERVE$COMMAND$QUEUE
                      (DEVICE$INDEX, STATUS);
         END /* THEN */;
       END /* THEN */;
    END /* ELSE */;
  END /* DO */;
 END /* FOREVER */;

END OUT$TASK;
```

Receive$command

This subroutine of IN$TASK transfers a command from an incoming request queue to the port queue associated with the socket specified in the command, first checking to make sure that the port is active. The routine then generates an appropriate response and enters it in the Response Turnaround Queue associated with the sending device.

```
RECEIVE$COMMAND: PROCEDURE (RQE$IN$PTR);

DECLARE RQE$IN$PTR                    POINTER,    /* Input. */
        IN$RQE  BASED RQE$IN$PTR      RQE$STRUCTURE;

DECLARE RQE$MSG$PTR                   POINTER,    /* Local. */
        MSG$RQE BASED RQE$MSG$PTR RQE$STRUCTURE,
        LOCAL$DATA$PTR                POINTER,
        STATUS                        BYTE;

 IF LPT (IN$RQE.DEST$PORT$ID).PORT$STATE <> ACTIVE
  THEN IN$RQE.REQUEST = SYSTEM$PORT$INACTIVE;
  ELSE DO; /* Deliver command. */
   CALL REQUEST$GIVE$POINTER
         (LPT(IN$RQE.DEST$PORT$ID).PORT$QUEUE$PTR,
          RQE$MSG$PTR,
          STATUS);
```

```
   IF STATUS = FULL
    THEN IN$RQE.REQUEST = SYSTEM$MEMORY$NAK;
    ELSE DO;
     CALL CONVERT$SYSTEM$ADR (IN$RQE.IDS$ID,
                              IN$RQE.DATA$PTR,
                              LOCAL$DATA$PTR);
     CALL MOVE (IN$RQE.DATA$LENGTH, /* Copies whole */
                RQE$MSG$PTR,        /* buffer into  */
                LOCAL$DATA$PTR);    /* port queue.  */
     CALL RELEASE$GIVE$POINTER
          (LPT(IN$RQE.DEST$PORT$ID).PORT$QUEUE$PTR,
           STATUS);
     IN$RQE.REQUEST = MSG$DELIVERED$COPY;

     /*                           NOTE

     Instead of copying the whole buffer, you may copy
     only IN$RQE.DATA$PTR, IN$RQE.DATA$LENGTH,
     IN$RQE.IDS$ID, and IN$RQE.OWNER$DEV$ID.  In this
     case, IN$RQE.REQUEST is set to MSG$DELIVERED$NO$COPY.
     */
    END /* ELSE */;
  END /* ELSE */;

 /* Create response. */
 CALL REQUEST$GIVE$POINTER
       (DCM(IN$RQE.SRC$DEV$ID).RSP$TRNRND$QUEUE$PTR,
        RQE$MSG$PTR,
        STATUS);
 CALL MOVE (16, RQE$IN$PTR, RQE$MSG$PTR);

 /*                           NOTE

    If IN$RQE.REQUEST is set to MSG$DELIVERED$NO$COPY,
    the only fields that must be returned are
    IN$RQD.REQUEST and IN$RQD.SRC$REQ$ID.

 */

 MSG$RQE.DEST$DEV$ID = IN$RQE.SRC$DEV$ID;
 MSG$RQE.SRC$DEV$ID  = IN$RQE.DEST$DEV$ID;
 CALL RELEASE$GIVE$POINTER
       (DCM(IN$RQE.SRC$DEV$ID).RSP$TRNRND$QUEUE$PTR,
        STATUS);
 RETURN;

END RECEIVE$COMMAND;
```

Receive$response

This subroutine of IN$TASK transfers a response from an incoming request queue to the response queue of the initiating task.

```
RECEIVE$RESPONSE:  PROCEDURE (RQE$IN$PTR);

DECLARE RQE$IN$PTR                  POINTER,    /* Input. */
        IN$RQE  BASED RQE$IN$PTR    RQE$STRUCTURE;

DECLARE RQE$RSP$PTR                 POINTER,    /* Local. */
        STATUS                      BYTE;
```

```
 CALL REQUEST$GIVE$POINTER
      (RQL(IN$RQE.SRC$REQ$ID).RSP$QUEUE$PTR,
           RQE$RSP$PTR,
           STATUS);
 CALL MOVE (16, RQE$IN$PTR, RQE$RSP$PTR);
 CALL RELEASE$GIVE$POINTER
      (RQL(IN$RQE.SRC$REQ$ID).RSP$QUEUE$PTR,
       STATUS);
 RETURN;

END RECEIVE$RESPONSE;
```

In$task

This algorithm manages activity in the incoming request queues.

```
IN$TASK:  PROCEDURE;

DECLARE DEVICE$INDEX            BYTE,          /* Local. */
        RQD$PTR                 POINTER,
        RQD    BASED RQD$PTR    RQD$STRUCTURE,
        RQE$IN$PTR              POINTER,
        IN$RQE BASED RQE$IN$PTR RQE$STRUCTURE,
        STATUS                  BYTE;

 DO FOREVER;
  DO  DEVICE$INDEX = 0 TO DEVICES - 1;
   RQD$PTR = DCM(DEVICE$INDEX).RQD$IN$PTR;
   IF RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER
    THEN DO;
     DCM(DEVICE$INDEX).CHANNEL$STATE =
      DCM(DEVICE$INDEX).CHANNEL$STATE OR RQD.EMPTY$SIGNAL;
     CALL CLEAR$INTERRUPT;
     RQD.EMPTY$SIGNAL = NO$CHANGE;
    END /* THEN */;
   IF (DCM(DEVICE$INDEX).CHANNEL$STATE AND
                                 (DYING OR IDLE) = 0)
     AND (DCM(DEVICE$INDEX).CHANNEL$STATE AND
                                      RECEIVE$ACTIVE <> 0)
    THEN DO; /* serve the input request queue. */
     CALL REQUEST$TAKE$POINTER
           (DCM(DEVICE$INDEX).RQD$IN$PTR,
            RQE$IN$PTR,
            STATUS);
     IF STATUS = HALTED
      THEN DCM(DEVICE$INDEX).CHANNEL$STATE = DYING;
     IF STATUS = EMPTY
      THEN DCM(DEVICE$INDEX).CHANNEL$STATE =
        DCM(DEVICE$INDEX).CHANNEL$STATE AND RECEIVE$EMPTY
           /* Don't bother with looking for input on this
             channel until it becomes active again. */;

     IF STATUS = READY
      THEN DO;
       IF IN$RQE.REQUEST = SEND$COMMAND
        THEN CALL RECEIVE$COMMAND  (RQE$IN$PTR);
        ELSE CALL RECEIVE$RESPONSE (RQE$IN$PTR);
```

```
        CALL RELEASE$TAKE$POINTER
             (DCM(DEVICE$INDEX).RQD$IN$PTR,
              STATUS);
        IF STATUS = FIRST$TAKE
         THEN /* Took from a full queue, so... */ DO;
          RQD$PTR = DCM(DEVICE$INDEX).RQD$OUT$PTR;
          /* Post signal in out-RQD. */
          RQD.FULL$SIGNAL = FULL$NO$LONGER;
         END /* THEN */;
       END /* THEN */;
     END /* THEN */;
   END /* DO */;
  END /* FOREVER */;

 END IN$TASK;
```

Virtual Level

Status Constants

The following values, along with values associated with RQE$REQUEST, are returned by the virtual level procedures to indicate the results of the procedures.

```
DECLARE SYSTEM$PORT$AVAILABLE        LITERALLY    '84H',
        SYSTEM$PORT$UNKNOWN          LITERALLY    '81H',
        SYSTEM$PORT$ACTIVE           LITERALLY    '83H',
        SYSTEM$PORT$INACTIVE         LITERALLY    '87H';
```

Find$system$port

This function provides you with the means to locate a socket by its function-name.

```
FIND$SYSTEM$PORT:  PROCEDURE (FUNCTION$NAME,
                              SOCKET$DEVICE,
                              SOCKET$PORT,
                              STATUS);

DECLARE FUNCTION$NAME  WORD;            /* Input. */

DECLARE SOCKET$DEVICE  IDENTIFIER,      /* Output. */
        SOCKET$PORT    IDENTIFIER,
        STATUS         BYTE;

DECLARE SOCKET$INDEX   BYTE;            /* Local. */

 DO  SOCKET$INDEX = 0 TO SOCKETS - 1;
  IF (FUNCTION$NAME = DSDT(SOCKET$INDEX).FUNCTION$NAME)
  THEN DO;
   STATUS = SYSTEM$PORT$AVAILABLE;
   SOCKET$DEVICE = DSDT(SOCKET$INDEX).DEST$DEV$ID;
   SOCKET$PORT   = DSDT(SOCKET$INDEX).DEST$PORT$ID;
   RETURN;
  END /* THEN */;
 END /* DO */;
 STATUS = SYSTEM$PORT$UNKNOWN;
 RETURN;

END FIND$SYSTEM$PORT;
```

Transfer$buffer

This function causes generation of a command to transfer a buffer to a destination device and port. The command is queued in the Command Wait Queue of the destination device. The procedure waits for a reply before relinquishing control.

```
TRANSFER$BUFFER: PROCEDURE  (BUFFER$PTR,
                             BUFFER$LENGTH,
                             IDS$ID,
                             SOCKET$DEVICE,
                             SOCKET$PORT,
                             RQL$ID,
                             STATUS);

DECLARE BUFFER$PTR          POINTER,        /* Input. */
        BUFFER$LENGTH       WORD,
        IDS$ID              IDENTIFIER,
        SOCKET$DEVICE       IDENTIFIER,
        SOCKET$PORT         IDENTIFIER,
        RQL$ID              IDENTIFIER;

DECLARE STATUS              BYTE;           /* Output. */

DECLARE RQE$PTR             POINTER,        /* Local. */
        RQE BASED RQE$PTR   RQE$STRUCTURE,
        CALL$STATUS         BYTE;

 CALL REQUEST$GIVE$POINTER
       (DCM(SOCKET$DEVICE).COM$RDY$QUEUE$PTR,
        RQE$PTR,
        CALL$STATUS);
 RQE.REQUEST          = SEND$COMMAND;
 RQE.SRC$REQ$ID       = RQL$ID;
 RQE.DEST$DEV$ID      = SOCKET$DEVICE;
 RQE.DEST$PORT$ID     = SOCKET$PORT;
 RQE.SRC$DEV$ID       = HOME$DEVICE;
 RQE.IDS$ID           = IDS$ID;
 RQE.OWNER$DEV$ID     = HOME$DEVICE;
 CALL CONVERT$LOCAL$ADR (IDS$ID,
                         BUFFER$PTR,
                         RQE.DATA$PTR);
 RQE.DATA$LENGTH      = BUFFER$LENGTH;
 CALL RELEASE$GIVE$POINTER
       (DCM(SOCKET$DEVICE).COM$RDY$QUEUE$PTR,
        CALL$STATUS);

 CALL TIME$WAIT (TIME$DELAY, RQL$ID);

 CALL REQUEST$TAKE$POINTER (RQL(RQL$ID).RSP$QUEUE$PTR,
                            RQE$PTR,
                            CALL$STATUS);
 IF CALL$STATUS = EMPTY
   /* No response came back within TIME$DELAY period. */
   THEN DO;
   DCM(SOCKET$DEVICE).CHANNEL$STATE = DYING;
   STATUS = DEAD$DEVICE;
  END /* THEN */;
```

```
  ELSE DO;
   STATUS = RQE.REQUEST;
   CALL RELEASE$TAKE$POINTER (RQL(RQL$ID).RSP$QUEUE$PTR,
                              CALL$STATUS);
  END /* ELSE */;
 RETURN;

END TRANSFER$BUFFER;
```

Activate$system$port

This function enables receipt of messages at a local port. If the port is not currently active, the address of the port queue is returned.

```
ACTIVATE$SYSTEM$PORT: PROCEDURE (FUNCTION$NAME,
                                 PORT$QUEUE$PTR,
                                 STATUS);

DECLARE FUNCTION$NAME   WORD,       /* Input. */
        PORT$QUEUE$PTR  POINTER;

DECLARE STATUS          BYTE;       /* Output. */

DECLARE PORT$INDEX      BYTE;       /* Local. */

 DO  PORT$INDEX = 0 TO PORTS - 1;
  IF FUNCTION$NAME = LPT(PORT$INDEX).FUNCTION$NAME
   THEN IF LPT(PORT$INDEX).PORT$STATE = ACTIVE
         THEN DO;
          STATUS = SYSTEM$PORT$ACTIVE;
          RETURN;
         END /* THEN */;
         ELSE DO;
          STATUS = SYSTEM$PORT$AVAILABLE;
          PORT$QUEUE$PTR = LPT(PORT$INDEX).PORT$QUEUE$PTR;
          LPT(PORT$INDEX).PORT$STATE = ACTIVE;
          RETURN;
         END /* ELSE */;
 END /* DO */;
 STATUS = SYSTEM$PORT$UNKNOWN;
 RETURN;

END ACTIVATE$SYSTEM$PORT;
```

Deactivate$system$port

This function terminates reception of messages at a port.

```
DEACTIVATE$SYSTEM$PORT: PROCEDURE (FUNCTION$NAME,
                                   STATUS);

DECLARE FUNCTION$NAME WORD;     /* Input. */

DECLARE STATUS        BYTE;     /* Output. */

DECLARE PORT$INDEX    BYTE;
```

```
 DO  PORT$INDEX = 0 TO PORTS - 1;
  IF FUNCTION$NAME = LPT(PORT$INDEX).FUNCTION$NAME
   THEN IF LPT(PORT$INDEX).PORT$STATE = INACTIVE
         THEN DO;
          STATUS = SYSTEM$PORT$INACTIVE;
          RETURN;
         END /* THEN */;
         ELSE DO;
          STATUS = SYSTEM$PORT$AVAILABLE;
          LPT(PORT$INDEX).PORT$STATE = INACTIVE;
          RETURN;
         END /* ELSE */;
 END /* DO */;
 STATUS = SYSTEM$PORT$UNKNOWN;
 RETURN;

END DEACTIVATE$SYSTEM$PORT;
```

Receive$buffer

This function retrieves a buffer from a port queue if there is a buffer in the queue.

```
RECEIVE$BUFFER:  PROCEDURE  (PORT$QUEUE$PTR,
                             USER$BUFFER$PTR,
                             STATUS);

DECLARE PORT$QUEUE$PTR             POINTER,    /* Input. */
        RQD BASED PORT$QUEUE$PTR RQD$STRUCTURE;

DECLARE USER$BUFFER$PTR            POINTER,    /* Output. */
        STATUS                     BYTE;

DECLARE RQE$PTR                    POINTER;    /* Local. */

 CALL REQUEST$TAKE$POINTER  (PORT$QUEUE$PTR,
                             RQE$PTR,
                             STATUS);
 IF STATUS = READY
  THEN DO;
   CALL MOVE (RQD.RQE$LENGTH,
              RQE$PTR,
              USER$BUFFER$PTR);
   CALL RELEASE$TAKE$POINTER (PORT$QUEUE$PTR,
                              STATUS);
  END /* THEN */;

 RETURN;

END RECEIVE$BUFFER;
```

APPENDIX C
EXAMPLE MIP FACILITIES

Two implementations of MIP are presented here: the first one written in PL/M-80, and the second in ASM-86.

PL/M Example

The first example, called XMX, is intended for a two-device system consisting of an 8085 host processor and an iSBC 550 Ethernet Communications Controller installed in an Intellec Series II or III Microcomputer Development System running under the ISIS-II operating system. The following assumptions govern the implementation:

- All memory is contained in one IDS.
- The configuration is static, so there is no need for function names or parameterized initialization procedures.
- Only one task executes on the host processor; therefore, there is no need for port addressing or response ID's.
- The entire XMX module is linked to the main host module and executes synchronously with it; therefore, there is no need for inter-task queues. Interface to XMX is through the procedures XMX$SEND and XMX$RECEIVE.
- The host task always waits for a response before issuing another command.
- The Ethernet Controller issues a command only in response to a command from the host task. The host never receives an unsolicited command.
- The Ethernet Controller must be interrupted for it to poll its request queues. An interrupt is caused by writing the value 02H to I-O port 0A4H.
- XMX does not respond to interrupts. Instead, it polls the signal bytes of its request queues.

Refer to Chapter 5 for an example of an application that uses XMX.

```
XMX: DO;          /*  Example Message Exchange.   */

DECLARE    WORD           LITERALLY    'ADDRESS',
           IDENTIFIER     LITERALLY    'BYTE',
           STATE          LITERALLY    'BYTE',
           NULL$PTR       LITERALLY    '0000H';

                  /*  Configuration constants.   */

DECLARE TIME$LOOPS        LITERALLY '1000',
        HOME$DEVICE       LITERALLY '01H';

DECLARE    RQD$STRUCTURE  LITERALLY 'STRUCTURE
 (EMPTY$SIGNAL   STATE,
  FULL$SIGNAL    STATE,
  RQ$SIZE        BYTE,
  RQE$LENGTH     BYTE,
  GIVE$INDEX     BYTE,
  GIVE$STATE     STATE,
  TAKE$INDEX     BYTE,
  TAKE$STATE     STATE)';
```

```
                    /*  Signal constants.  */

DECLARE FULL$NO$LONGER                          LITERALLY     '80H',
        EMPTY$NO$LONGER                         LITERALLY     '01H',
        NO$CHANGE                               LITERALLY     '00H';

DECLARE WAKE$UP$PORT                            LITERALLY    '0A4H';

                    /*  RQD state constants.  */

DECLARE GIVE$HALT                               LITERALLY     '40H',
        TAKE$HALT                               LITERALLY     '40H',
        GIVE$FACTOR                             LITERALLY     '80H',
        TAKE$FACTOR                             LITERALLY     '80H';

DECLARE   RQE$FORMAT$1  LITERALLY
 'REQUEST      STATE,
  SRC$REQ$ID   IDENTIFIER,
  DEST$DEV$ID  IDENTIFIER,
  DEST$PORT$ID IDENTIFIER,
  SRC$DEV$ID   IDENTIFIER',
          RQE$FORMAT$2  LITERALLY
 'DATA$PTR$LO  WORD,
  DATA$PTR$HI  WORD,
  DATA$LENGTH  WORD,
  IDS$ID       IDENTIFIER,
  OWNER$DEV$ID IDENTIFIER,
  RSRVD  (3)   BYTE';

DECLARE RQE$FORMAT LITERALLY 'RQE$FORMAT$1, RQE$FORMAT$2';
DECLARE RQE$STRUCTURE LITERALLY 'STRUCTURE (RQE$FORMAT)';

                    /*  Request constants.  */

DECLARE SEND$COMMAND                            LITERALLY     '70H',
        MSG$DELIVERED$NO$COPY                   LITERALLY     '80H',
        MSG$DELIVERED$COPY                      LITERALLY     '82H',
        SYSTEM$MEMORY$NAK                       LITERALLY     '85H',
        DEAD$DEVICE                             LITERALLY     '89H';

                    /*  Actual Request Queues.  */

DECLARE QUEUE$ENTRIES  LITERALLY '2';

DECLARE IN$RQD RQD$STRUCTURE                          PUBLIC
              INITIAL (NO$CHANGE,
                       NO$CHANGE,
                       QUEUE$ENTRIES,
                       4, 0, 0, 0, 0),
        IN$RQE (QUEUE$ENTRIES)  RQE$STRUCTURE;

DECLARE OUT$RQD RQD$STRUCTURE                         PUBLIC
              INITIAL (NO$CHANGE,
                       NO$CHANGE,
                       QUEUE$ENTRIES,
                       4, 0, 0, 0, 0),
        OUT$RQE (QUEUE$ENTRIES) RQE$STRUCTURE;
```

```
                    /*  Request function returns.  */

DECLARE READY                                 LITERALLY     '00H',
        FULL                                  LITERALLY    '0FFH',
        EMPTY                                 LITERALLY    '0FFH',
        FIRST$GIVE                            LITERALLY     '20H',
        FIRST$TAKE                            LITERALLY     '20H',
        HALTED                                LITERALLY     '40H';

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

                    /*  Channel activity.  */
                    /*     One channel.    */

DECLARE ACTIVE                                LITERALLY     '00H',
        IDLE                                  LITERALLY    '0FFH';

DECLARE CHANNEL$STATE STATE INITIAL (ACTIVE),
        RECEIVE$STATE STATE INITIAL (EMPTY);

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

REQUEST$GIVE$POINTER:  PROCEDURE  (RQD$PTR,
                                   STATUS$P) ADDRESS;

DECLARE RQD$PTR            ADDRESS,          /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;

DECLARE STATUS$P           ADDRESS,          /* Output. */
        STATUS BASED STATUS$P WORD;

 IF (RQD.TAKE$STATE AND TAKE$HALT)  = TAKE$HALT
  THEN DO;
   STATUS = HALTED;
   RETURN NULL$PTR;
  END /* THEN */;
 IF (RQD.GIVE$INDEX = RQD.TAKE$INDEX) AND
     ((RQD.GIVE$STATE AND GIVE$FACTOR) <>
      (RQD.TAKE$STATE AND TAKE$FACTOR))
  THEN DO;
   STATUS = FULL;
   RETURN NULL$PTR;
  END /* THEN */;
 STATUS = READY;
 RETURN RQD$PTR + SHL(RQD.GIVE$INDEX, RQD.RQE$LENGTH) +8;

END REQUEST$GIVE$POINTER;

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

RELEASE$GIVE$POINTER:  PROCEDURE  (RQD$PTR,
                                   STATUS$P);

DECLARE RQD$PTR            ADDRESS,            /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;
```

```
DECLARE STATUS$P          ADDRESS,          /* Output. */
        STATUS BASED STATUS$P WORD;

 IF (RQD.TAKE$INDEX = ((RQD.GIVE$INDEX + 1)
                     AND (RQD.RQ$SIZE - 1)) )
  THEN /* GIVE$FACTOR bit = NOT TAKE$FACTOR bit. */
       RQD.GIVE$STATE = (RQD.GIVE$STATE OR GIVE$FACTOR)
               AND (NOT (RQD.TAKE$STATE AND TAKE$FACTOR));

 RQD.GIVE$INDEX =
  ((RQD.GIVE$INDEX + 1) AND (RQD.RQ$SIZE - 1));

 IF RQD.GIVE$INDEX =
   ((RQD.TAKE$INDEX + 1) AND (RQD.RQ$SIZE - 1))
  THEN STATUS = FIRST$GIVE;  /* Gave to an empty queue. */
  ELSE STATUS = READY;
 RETURN;

END RELEASE$GIVE$POINTER;

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

REQUEST$TAKE$POINTER:  PROCEDURE  (RQD$PTR,
                                   STATUS$P) ADDRESS;

DECLARE RQD$PTR           ADDRESS,          /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;

DECLARE STATUS$P          ADDRESS,          /* Output. */
        STATUS BASED STATUS$P WORD;

 IF (RQD.GIVE$STATE AND GIVE$HALT) = GIVE$HALT
  THEN DO;
   STATUS = HALTED;
   RETURN NULL$PTR;
  END /* THEN */;
 IF (RQD.GIVE$INDEX = RQD.TAKE$INDEX) AND
     ((RQD.GIVE$STATE AND GIVE$FACTOR) =
      (RQD.TAKE$STATE AND TAKE$FACTOR))
  THEN DO;
   STATUS = EMPTY;
   RETURN NULL$PTR;
  END /* THEN */;
 STATUS = READY;
 RETURN RQD$PTR + SHL(RQD.TAKE$INDEX, RQD.RQE$LENGTH) +8;

END REQUEST$TAKE$POINTER;

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

RELEASE$TAKE$POINTER:  PROCEDURE  (RQD$PTR,
                                   STATUS$P);

DECLARE RQD$PTR           ADDRESS,          /* Input. */
        RQD BASED RQD$PTR RQD$STRUCTURE;
```

```
DECLARE STATUS$P          ADDRESS,          /* Output. */
        STATUS BASED STATUS$P WORD;

 IF (RQD.GIVE$INDEX = ((RQD.TAKE$INDEX + 1) AND
                       (RQD.RQ$SIZE - 1)) )
  THEN /* TAKE$FACTOR bit = GIVE$FACTOR bit. */
       RQD.TAKE$STATE = (RQD.TAKE$STATE AND NOT TAKE$FACTOR)
                     OR (RQD.GIVE$STATE AND GIVE$FACTOR);

 RQD.TAKE$INDEX =
  ((RQD.TAKE$INDEX + 1) AND (RQD.RQ$SIZE - 1));

 IF RQD.TAKE$INDEX =
   ((RQD.GIVE$INDEX + 1) AND (RQD.RQ$SIZE - 1))
  THEN STATUS = FIRST$TAKE;  /* Took from a full queue. */
  ELSE STATUS = READY;
 RETURN;

END RELEASE$TAKE$POINTER;

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

DYING$CHANNEL:  PROCEDURE ;

 CHANNEL$STATE = IDLE;
 IN$RQD.TAKE$STATE  = IN$RQD.TAKE$STATE OR TAKE$HALT;
 OUT$RQD.GIVE$STATE = OUT$RQD.GIVE$STATE OR GIVE$HALT;

END DYING$CHANNEL;

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

XMX$SEND: PROCEDURE  (BUFFER$PTR,
                      BUFFER$LENGTH,
                      SOCKET,
                      STATUS$P)                   PUBLIC;

DECLARE BUFFER$PTR          ADDRESS,          /* Input. */
        BUFFER$LENGTH       WORD,
        SOCKET              WORD;

DECLARE STATUS$P            ADDRESS,          /* Output. */
        STATUS BASED STATUS$P WORD;

DECLARE RQE$PTR             ADDRESS,          /* Local. */
        RQE BASED RQE$PTR   RQE$STRUCTURE,
        LOC$STATUS          WORD,
        TIMER               WORD;

 IF CHANNEL$STATE = IDLE
  THEN DO;
   STATUS = DEAD$DEVICE;
   RETURN;
  END /* THEN */;

 RQE$PTR = REQUEST$GIVE$POINTER (.OUT$RQD, .LOC$STATUS);
```

```
IF LOC$STATUS = READY
 THEN DO;
  RQE.REQUEST          = SEND$COMMAND;
  RQE.SRC$REQ$ID       = 0;
  RQE.DEST$DEV$ID      = HIGH (SOCKET);
  RQE.DEST$PORT$ID     = LOW  (SOCKET);
  RQE.SRC$DEV$ID       = HOME$DEVICE;
  RQE.IDS$ID           = 0;
  RQE.OWNER$DEV$ID     = HOME$DEVICE;
  RQE.DATA$PTR$LO      = BUFFER$PTR;
  RQE.DATA$PTR$HI      = 0;
  RQE.DATA$LENGTH      = BUFFER$LENGTH;
  CALL RELEASE$GIVE$POINTER (.OUT$RQD, .LOC$STATUS);
  /* Since this program is the only sender, it
     always gives to an empty queue, so ... */
  OUT$RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER;
  OUTPUT (WAKE$UP$PORT) = 2;
 END /* THEN */;

 ELSE /* either FULL or HALTED */ DO;
  /* Since only one command is outstanding at
     one time, the queue should never be full. */
  CALL DYING$CHANNEL;
  STATUS = DEAD$DEVICE;
  RETURN;
 END /* ELSE */;

DO TIMER = 0 TO TIMELOOPS; /* Wait for a response. */
 IF IN$RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER
  THEN DO;
   RECEIVE$STATE = ACTIVE;
   IN$RQD.EMPTY$SIGNAL = NO$CHANGE;
  END /* THEN */;
 IF RECEIVE$STATE = ACTIVE
  THEN DO;
   RQE$PTR = REQUEST$TAKE$POINTER (.IN$RQD, .LOC$STATUS);
   IF LOC$STATUS = READY
    THEN DO;
     STATUS = RQE.REQUEST;
     CALL RELEASE$TAKE$POINTER (.IN$RQD, .LOC$STATUS);
     IF LOC$STATUS = FIRST$TAKE
      THEN DO;
       OUT$RQD.FULL$SIGNAL = FULL$NO$LONGER;
       OUTPUT (WAKE$UP$PORT) = 2;
      END /* THEN */;
     RETURN;
    END /* THEN */;
   IF LOC$STATUS = EMPTY
    THEN RECEIVE$STATE = EMPTY;
    ELSE CALL DYING$CHANNEL;
  END /* THEN */;
  ELSE CALL TIME (250);
END /* DO */;

/* No response came back within a reasonable time. */
CALL DYING$CHANNEL;
STATUS = DEAD$DEVICE;

END XMX$SEND;
```

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

XMX$RECEIVE:  PROCEDURE  (STATUS$P) ADDRESS     PUBLIC;

DECLARE STATUS$P                    ADDRESS,   /* Output. */
        STATUS BASED STATUS$P       WORD,
        USER$BUFFER$PTR             ADDRESS;

DECLARE IN$RQE$PTR                  ADDRESS,   /* Local. */
        IN$RQE BASED IN$RQE$PTR     RQE$STRUCTURE,
        OUT$RQE$PTR                 ADDRESS,
        OUT$RQE BASED OUT$RQE$PTR RQE$STRUCTURE,
        LOC$STATUS                  WORD;

 IF IN$RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER
  THEN DO;
   RECEIVE$STATE = ACTIVE;
   IN$RQD.EMPTY$SIGNAL = NO$CHANGE;
  END /* THEN */;
 IF (CHANNEL$STATE <> IDLE) AND (RECEIVE$STATE = ACTIVE)
  THEN DO;
   IN$RQE$PTR = REQUEST$TAKE$POINTER (.IN$RQD, .LOC$STATUS);
   IF LOC$STATUS = READY
    THEN DO;
     STATUS = IN$RQE.REQUEST;
     USER$BUFFER$PTR = IN$RQE.DATA$PTR$LO;
     /* It can only be a command, so return response. */
     OUT$RQE$PTR =
            REQUEST$GIVE$POINTER (.OUT$RQD, .LOC$STATUS);
     IF LOC$STATUS = READY
      THEN DO;
       CALL MOVE (16, IN$RQE$PTR, OUT$RQE$PTR);
       OUT$RQE.REQUEST = MSG$DELIVERED$NO$COPY;
       OUT$RQE.SRC$DEV$ID = IN$RQE.DEST$DEV$ID;
       OUT$RQE.DEST$DEV$ID = IN$RQE.SRC$DEV$ID;
       CALL RELEASE$GIVE$POINTER (.OUT$RQD, .LOC$STATUS);
       /* The output queue must have been empty,
          so signal.                             */
       OUT$RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER;
       OUTPUT (WAKE$UP$PORT) = 2;
      END /* THEN */;

     CALL RELEASE$TAKE$POINTER (.IN$RQD, .LOC$STATUS);
     IF LOC$STATUS = FIRST$TAKE
      THEN DO;
       OUT$RQD.FULL$SIGNAL = FULL$NO$LONGER;
       OUTPUT (WAKE$UP$PORT) = 2;
      END /* THEN */;
     RETURN USER$BUFFER$PTR;
    END /* THEN */;

   IF LOC$STATUS = EMPTY
    THEN RECEIVE$STATE = EMPTY;
    ELSE CALL DYING$CHANNEL;

  END /* THEN */;
 STATUS = EMPTY;
 RETURN NULL$PTR;

END XMX$RECEIVE;

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

END XMX;
```

Assembler Example

This second example of a MIP facility is intended for use in a multitasking environment. It can support an arbitrary number of devices, but in this example it is configured to communicate with two devices other than the 8086 processor on which it runs. One of these is the Ethernet Communications Controller; the other may be any processor board. The following assumptions govern this implementation of MIP:

- All memory is contained in one IDS.
- Only one request is outstanding at one time.
- The operating system supports mailboxes, semaphores, and a timer.
- The operating system uses a priority task scheduling mechanism. A higher priority task may pre-empt a lower priority one.
- Messages begin at addresses that are evenly divisible by 16. Messages contain length, owner device-ID, and IDS-ID fields as illustrated in figure C-1.

Six modules constitute this MIP facility:

- MIPDEF—defines the data structures
- RQPROC—contains the request queue procedures
- MIPINIT—initializes the operating system interfaces
- MIPCON—called by user tasks to associate a MIP port with an operating system mailbox
- MIPSND—called by user tasks to send a message
- INTASK—services incoming messages

MIPSND is reentrant and may be executed by several tasks at once. It contains a critical region, however, that may be executed by only one task at a time. Access to the critical region is controlled by the semaphore MIPUSEPERMIT.

INTASK is an asynchronous task driven by interrupts communicated to it by the operating system through the semaphore INTERRUPTSEMAPHORE.

MIPSND and INTASK communicate with each other through the shared variables SENDSTATE, SENDRESULT and SENDDEVICE, and also by passing the dummy message SENDMSG through the mailbox MIPSENDWTMBX.

This example makes several calls on operating system functions. These are explained below:

- ALLOCATE—gets space for SENDMSG.
- ENABLEINTERRUPT—tells the operating system to begin posting the interrupt associated with a specific semaphore. When the operating system recogizes the interrupt, it increments the semaphore.
- SENDUNIT—increments the specified semaphore.
- RECEIVEUNIT—decrements the specified semaphore. If the semaphore is zero, the calling task is made to wait until another task calls on SENDUNIT.
- ENQUEUE—places an item in the specified mailbox.
- DEQUEUE—removes an item from a specific mailbox. If no item is in the mailbox, the task waits for the specified number of time units or until some other task calls on ENQUEUE to place an item in the mailbox.

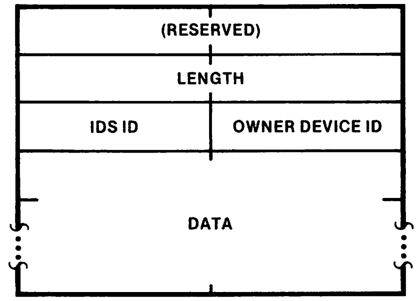

Figure C-1. Example Message Format. 769-25

```
$TITLE('MIP DATA STRUCTURES')

        NAME     MIPDEF


DGROUP  GROUP   DATA
DATA    SEGMENT PUBLIC 'DATA'

                PUBLIC MIPDEVCNT, THISDEVICE

MIPDEVCNT       DB  2  ; NUMBER OF DEVICES KNOWN
                       ; TO THIS DEVICE
THISDEVICE      DB  1  ; DEV ID OF THIS DEVICE

; DEFINE REQUEST QUEUES FOR USE WITH CONTROLLER

FROMCONTROLLER  DB  0,0,2,4,0,0,0,0
                DB  32 DUP (0)
TOCONTROLLER    DB  0,0,2,4,0,0,0,0
                DB  32 DUP (0)

; DEFINE REQUEST QUEUES FOR USE WITH THE PROCESSOR

FROMPROCESSOR   DB  0,0,2,4,0,0,0,0
                DB  32 DUP (0)
TOPROCESSOR     DB  0,0,2,4,0,0,0,0
                DB  32 DUP (0)

; SET DEVICE INFO UP FOR TWO OTHER DEVICES:
;     THE ETHERNET CONTROLLER AND
;     ANOTHER PROCESSOR BOARD.

                PUBLIC MIPDEVICEINFO

;DECLARE MIP$DEVICE$INFO(MAX$NO$DEVICES)
; STRUCTURE (
;  DEV$ID         BYTE,
;  STATUS         BYTE,
;  RQD$IN         POINTER,
;  RQD$OUT        POINTER,
;  INT$TYPE       BYTE,
;  TIME$TO$WAIT   BYTE,
;  INT$ADR        WORD ) PUBLIC

; FIRST FOR THE CONTROLLER
```

```
MIPDEVICEINFO   DB 0,0FFH           ; DEVID, STATUS
                                    ; (INIT READY)
                DD FROMCONTROLLER   ; RQD IN
                DD TOCONTROLLER     ; RQD OUT
                DB 1,0,0A4H,0       ; INT TYPE,
                                    ; TIME TO WAIT,
                                    ; INT ADDR.

; NOW FOR THE PROCESSOR

                DB 4,0H             ; DEVID, STATUS
                                    ; (INIT NOT READY)
                DD FROMPROCESSOR    ; RQD IN
                DD TOPROCESSOR      ; RQD OUT
                DB 0,0,0,0          ; INT TYPE,
                                    ; TIME TO WAIT,
                                    ; INT ADDR.

                PUBLIC  MIPDEVTOENTRY

;  INDEX TABLE INTO MIPDEVICEINFO.
;  USES DEVICE ID AS A KEY.

MIPDEVTOENTRY   DB      0,0,0,0,1,0,0,0

                PUBLIC PORTTOMAILBOX

;  TABLE TO CONVERT PORT ID INTO MAILBOX NUMBER.
;  USER TASKS PLACE ENTRIES IN THIS TABLE BY MEANS
;  OF THE MIPCONNECT PROCEDURE.

PORTTOMAILBOX   DB 16 DUP (0)

                PUBLIC SENDMSG,SENDRESULT
                PUBLIC SENDSTATE,SENDDEVICE

;  COMMUNICATION AREAS BETWEEN INTASK AND MIPSND.

SENDMSG         DW 1 DUP (0) ; ADDRESS TOKEN FOR
                             ; DUMMY MESSAGE.
SENDRESULT      DB 0
SENDSTATE       DB 0
SENDDEVICE      DB 0

DATA            ENDS

                END

$TITLE('MIP REQUEST QUEUE ROUTINES')

        NAME    RQPROC

; DEFINE RQD RESULTS

GERROR  EQU      1H
GBUSY   EQU      4H
FIRSTG  EQU      8H
GDISAB  EQU     10H
GFULL   EQU     20H
```

```
DISABT  EQU     40H
FULLF   EQU     80H
TERROR  EQU      1H
TBUSY   EQU      4H
FIRSTT  EQU      8H
TDISAB  EQU     10H
TEMPTY  EQU     20H
DISABG  EQU     40H
EMPTYF  EQU     80H

; DEFINE MIP COMMANDS AND RESPONSES

CSEND   EQU     70H
SENTOK  EQU     80H
UNKNP   EQU     81H
ACTIVP  EQU     83H
INSUFM  EQU     85H
INACTP  EQU     87H
DEADP   EQU     89H

CGROUP  GROUP   CODE
CODE    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:CGROUP

; REQUEST POINTER ROUTINES

        PUBLIC  REQUESTGIVEPTR,REQUESTTAKEPTR

; BECAUSE OF SYMMETRY OF LOGIC, REQUEST$GIVE
; AND REQUEST$TAKE ROUTINES ARE COMBINED.

REQUESTGIVEPTR:
        MOV     DH,0     ; FLAG AS GIVE PTR
        MOV     BX,5
        JMP     SHORT L1

REQUESTTAKEPTR:
        MOV     DH,80H   ; FLAG AS TAKE PTR
        MOV     BX,1

; NOW LOAD THE REGISTERS

L1:     CALL LOADREG

; CHECK IF DISABLED

        MOV     DL,AH    ; GIVESTATE
        OR      DL,BH    ; TAKESTATE
        AND     DL,DISABG OR GDISAB
        JZ      L4
        MOV     AL,GDISAB OR GERROR ; IS DISABLED
        RET              ; ZERO FLAG IS RESET

; WAS NOT DISABLED, SEE IF FULL/EMPTY
```

```
L4:     CMP     AL,BL   ; COMPARE INDEXES
        JNE     L6
        MOV     DL,AH   ; CHECK FULL/EMPTY FACTOR
        XOR     DL,BH
        AND     DL,FULLF
        CMP     DH,DL
        JE      L6      ; NOT FULL/EMPTY
        MOV     AL,GFULL OR GERROR ; IS FULL/EMPTY
        RET             ; ZERO FLAG IS RESET

; HAVE ENTRY, CALCULATE ADDRESS OF IT

L6:     MOV     CL,CH   ; GET RQESIZE
        MOV     AH,0
        SHL     AX,CL
        MOV     CX,ES   ; MOVE BASE OF RQDPTR
        ADD     AL,8    ; ADD RQD AREA
        ADD     AX,SI   ; ADD ENTRY OFFSET
        XCHG    BX,AX
        JNC     L8
        ADD     CX,1000H ; HAD OVERFLOW IN OFFSET;
                         ; ADD TO BASE.

; ALL DONE, RETURN ENTRY TO USER

L8:     MOV     ES,CX
        XOR     AX,AX   ; ZERO FLAG IS SET
        RET

; RELEASE POINTER ROUTINES

        PUBLIC  RELEASEGIVEPTR,RELEASETAKEPTR

; BECAUSE OF SYMMETRY OF LOGIC, RELEASE$GIVE
; AND RELEASE$TAKE ROUTINES ARE COMBINED.

RELEASEGIVEPTR:
        MOV     DH,0H
        MOV     BX,5
        JMP     SHORT M1

RELEASETAKEPTR:
        MOV     DH,80H
        MOV     BX,1

; NOW LOAD REGISTERS

M1:     CALL    LOADREG
```

```
; BUMP POINTERS AND UPDATE STATUS/INDEX

        DEC     CL; GET MASK FOR MODULO ARITHMETIC
        INC     AL
        AND     AL,CL ; DO MODULO
        CMP     AL,BL
        JNE     M6 ; JMP IF INDEXES ARE NOT EQUAL
        AND     AH,NOT FULLF; SET FULL/EMPTY FACTR
        MOV     CH,BH
        OR      DH,DH
        JNZ     M5
        NOT     CH
M5:     AND     CH,EMPTYF
        OR      AH,CH

; STORE INDEX,STATE

M6:     CALL    STOREREG

; SEE IF FIRST TAKE/GIVE

        INC     BL
        AND     BL,CL
        CMP     BL,AL
        MOV     AL,0
        JNZ     M20   ; NOT FIRST GIVE/TAKE
        OR      DH,DH
        JNZ     SHORT M10
        MOV     BYTE PTR ES:[SI],1H    ; FIRST GIVE
        JMP     SHORT M12
M10:    LES     SI,DWORD PTR [DI+5]
        MOV     BYTE PTR ES:[SI+1],80H; FIRST TAKE

; NOW GENERATE INTERRUPT

M12:    MOV     CL,BYTE PTR[DI+9H] ; INT TYPE
        MOV     DX,[DI+0BH]        ; INT ADDRESS
        MOV     AL,2
        CMP     CL,1   ; I-O MAPPED?
        JNE     M20    ; IF NOT, GENERATE NO SIGNAL
        OUT     DX,AL ; I-O INTERRUPT
M20:    XOR     AX,AX
        RET

; COMMON ROUTINE FOR LOADING REGISTERS

LOADREG:
        LES     SI,DWORD PTR [DI+BX]
        MOV     CX,ES:[SI+2] ; RQ SIZE, RQE LENGTH
        MOV     AX,ES:[SI+4] ; GIVE INDEX, STATE
        MOV     BX,ES:[SI+6] ; TAKE INDEX, STATE
        OR      DH,DH
        JZ      LR1
        XCHG    AX,BX  ; SWITCHED FOR TAKE PTR
LR1:    RET
```

```
; COMMON ROUTINE FOR STORING REGISTERS

STOREREG:
        OR      DH,DH
        JZ      SR1
        MOV     ES:[SI+6],AX
        RET
SR1:    MOV     ES:[SI+4],AX
        RET

CODE ENDS

        END


$TITLE('MIP INITIALIZATION ROUTINE')

        NAME    MIPINIT

DGROUP  GROUP   DATA
DATA    SEGMENT PUBLIC 'DATA'

        EXTRN   MIPUSEPERMIT:NEAR, SENDMSG:NEAR
        EXTRN   INTERRUPTSEMAPHORE:NEAR
I       DW 0

DATA    ENDS

CGROUP  GROUP   CODE
CODE    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:CGROUP,DS:DGROUP
        PUBLIC  MIPINIT
        EXTRN   ENABLEINTERRUPT:NEAR
        EXTRN   SENDUNIT:NEAR
        EXTRN   ALLOCATE:NEAR

MIPINIT      PROC NEAR

; SENDMSG = ALLOCATE(1,@I);
; ASK OS FOR ADDRESS OF A MESSAGE AREA TO BE
; USED FOR COMMUNICATION BETWEEN INTASK AND
; MIPSEND.

        MOV     AL,1
        PUSH    AX
        PUSH    DS
        MOV     AX,OFFSET DGROUP:I
        PUSH    AX
        CALL    ALLOCATE ; GET MSG SPACE FROM OS
        MOV     WORD PTR SENDMSG,AX ; SAVE ADDRESS

; CALL SENDUNIT(MIPUSEPERMIT);

        PUSH    WORD PTR DGROUP:MIPUSEPERMIT
        CALL    SENDUNIT
```

```
; CALL ENABLEINTERRUPT(INTERRUPTSEMAPHORE);
; PERMIT INTERRUPTS TO BE SEEN.

        PUSH    WORD PTR INTERRUPTSEMAPHORE
        CALL    ENABLEINTERRUPT
        RET

MIPINIT ENDP

CODE    ENDS

        END


$TITLE ('CONNECT FUNCTION')

        NAME     MIPCON

DGROUP  GROUP   DATA
DATA    SEGMENT PUBLIC 'DATA'

        EXTRN   PORTTOMAILBOX:NEAR

DATA ENDS

CGROUP  GROUP   CODE
CODE    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:CGROUP,DS:DGROUP

        PUBLIC  MIPCONNECT

; ASSOCIATES A SYSTEM MAILBOX WITH A MIP PORT.

MIPCONNECT      PROC NEAR

        POP     DI      ; RETURN ADDRESS
        POP     AX      ; MAILBOX POINTER
        POP     BX      ; PORT
        PUSH    DI      ; SAVE RETURN ADDRESS

        CMP     BL,10H ; MAX PORT ID IS 15
        JB      OKPORT ; JUMP IF WITHIN LIMIT

        MOV     AL,1H   ; BAD PORT. RETURN ERROR.
        RET

OKPORT: MOV     BH,0    ; ISOLATE PORT BYTE.
                        ; PLACE MAILBOX IN TABLE.
        MOV     BYTE PTR PORTTOMAILBOX[BX],AL
        XOR     AX,AX   ; RETURN ZERO
        RET

MIPCONNECT      ENDP

CODE    ENDS

        END
```

```
$TITLE ('MIP SEND')

   NAME MIPSEND

; DEFINE RQD RESULTS

GERROR  EQU      1H
GBUSY   EQU      4H
FIRSTG  EQU      8H
GDISAB  EQU     10H
GFULL   EQU     20H
DISABT  EQU     40H
FULLF   EQU     80H

TERROR  EQU      1H
TBUSY   EQU      4H
FIRSTT  EQU      8H
TDISAB  EQU     10H
TEMPTY  EQU     20H
DISABG  EQU     40H
EMPTYF  EQU     80H

; DEFINE MIP COMMANDS AND RESPONSES

CSEND   EQU     70H
SENTOK  EQU     80H
UNKNP   EQU     81H
ACTIVP  EQU     83H
INSUFM  EQU     85H
INACTP  EQU     87H
DEADP   EQU     89H

DGROUP  GROUP   DATA
DATA SEGMENT   PUBLIC 'DATA'
        EXTRN   MIPDEVICEINFO:NEAR
        EXTRN   THISDEVICE:BYTE
        EXTRN   MIPSENDWTMBX:NEAR, SENDRESULT:BYTE
        EXTRN   SENDSTATE:BYTE, SENDDEVICE:BYTE
        EXTRN   MIPUSEPERMIT:NEAR
        EXTRN   PORTTOMAILBOX:NEAR
        EXTRN   MIPDEVCNT:BYTE, MIPDEVTOENTRY:NEAR
DATA    ENDS

CGROUP  GROUP   CODE
CODE    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:CGROUP,DS:DGROUP

        PUBLIC  MIPSEND, CALCDEVPTR
        EXTRN   REQUESTGIVEPTR:NEAR
        EXTRN   RELEASEGIVEPTR:NEAR
        EXTRN   DEQUEUE:NEAR, ENQUEUE:NEAR
        EXTRN   SENDUNIT:NEAR, RECEIVEUNIT:NEAR

MIPSEND       PROC NEAR
        POP     DI ; RETURN ADDRESS
        POP     SI ; MSGPTR (ON 16-BYTE BOUNDARY)
        POP     AX ; DESTINATION SOCKET
        PUSH    DI
```

```
; IF DESTINATION IS FOR A LOCAL PORT,
;  CALL THE OS SEND ROUTINE.

        CMP     AH,THISDEVICE
        JNE     REMOTE

        MOV     AH,0   ; ISOLATE LOCAL PORT ID
        MOV     BX,AX  ; GET MAILBOX
        MOV     AL,BYTE PTR PORTTOMAILBOX[BX]
        OR      AL,AL      ; IS IT ZERO?
        JZ      INACTIVE   ; ZERO MEANS INACTIVE.

        PUSH    AX       ; MAILBOX
        PUSH    SI       ; MESSAGE POINTER
        CALL    ENQUEUE ; PUT POINTER IN MAILBOX
        MOV     AL,0H    ; RETURN STATUS = SENTOK
        RET

INACTIVE:
        MOV     AL,7H    ; RETURN STATUS = INACTP
        RET

;   PROCEED IF NOT BUSY, ELSE BLOCK.

REMOTE:
        PUSH    SI    ; SAVE MSGPTR.
        XCHG    AL,AH ; FOR SIMPLICITY LATER.
        PUSH    AX    ; SAVE SOCKET.

;  SECURE PERMISSION TO PROCEED.  ONLY ONE CALLER
;  AT A TIME MAY PROCEED BEYOND THIS POINT.

        PUSH    WORD PTR DGROUP:MIPUSEPERMIT
        CALL    RECEIVEUNIT

;  GET THE DEVICE INFO FOR THE DESTINATION DEVICE.

        POP     BX   ; GET SOCKET
        PUSH    BX
        MOV     BH,0 ; MASK OUT PORT; LEAVE DEVICE
                     ; LOOK UP INDEX INTO DEV INFO
        MOV     AL,BYTE PTR MIPDEVTOENTRY[BX]
        CALL    CALCDEVPTR ; GET PTR TO DEV INFO
        JNE     DEAD ; IF NOT EQUAL, DEVICE DEAD
        INC     BX   ; SET DEV PTR TO STATUS BYTE

;  LOOP UNTIL WE HAVE PUT ITEM INTO THE REQUEST
;  QUEUE, OR UNTIL A FATAL ERROR OCCURS.

TOP:    PUSH    BX    ; SAVE DEV PTR
        CLI     ; DISABLE INTERRUPTS TO PREVENT
                ; INTERFERENCE FROM INTASK.
        MOV     DI,BX ; DEV PTR
        CALL    REQUESTGIVEPTR
        JNZ     NOGIVE ; NOT ZERO MEANS ERROR
```

```
; THERE IS A FREE RQE.  FILL IT IN.
; ES:[BX] POINTS TO THE RQE.

        MOV     BYTE PTR ES:[BX],CSEND ; COMMAND
        POP     CX   ; DEVINFO
        POP     AX   ; SOCKET
        POP     SI   ; MSGPTR
        PUSH    SI
        PUSH    AX
        PUSH    CX
        MOV     WORD PTR ES:[BX+2H],AX  ; SOCKET
        MOV     CL,THISDEVICE
        MOV     BYTE PTR ES:[BX+4H],CL

; CONVERT ADDRESS TOKEN TO IDS POINTER.
; ASSUME NO ALIASING IN THIS SYSTEM.

        MOV     CL,4
        MOV     DI,SI ; MAKE COPY OF MSGPTR
        MOV     DX,SI ; ANOTHER COPY
        SHL     DX,CL ; GIVES LOWER 16 BITS
        AND     SI,0F000H
        ROR     SI,CL ; GIVES UPPER 16 BITS
        MOV     ES:[BX+5],SI
        MOV     ES:[BX+7],DX ; PUT INTO RQE

; PUT LENGTH, IDS-ID, AND OWNDEV INTO RQE FROM MSG

        PUSH    ES                ; SAVE RQE BASE
        MOV     ES,DI
        MOV     AX,ES:[DI+2]      ; LENGTH
        MOV     CX,ES:[DI+4]      ; IDS/OWNDEV
        POP     ES
        MOV     ES:[BX+9],AX      ; PUT LENGTH AWAY
        MOV     ES:[BX+0BH],CX    ; OWNDEV AND IDS

; TELL INTASK WHAT WE ARE WAITING FOR.

        MOV     SENDDEVICE,AL     ; DEST DEVICE
        MOV     SENDRESULT,0
        MOV     SENDSTATE,2H      ; WAITING FOR REPLY

        POP     DI                ; DEV PTR
        PUSH    DI
        CALL    RELEASEGIVEPTR
        JMP     WAITR

NOGIVE:
        TEST    AL,GFULL     ; IS QUEUE FULL?
        STI
        JZ      DEAD         ; IF NOT FULL, DEAD
        MOV     SENDSTATE,1H; WAITING TIL NOT FULL

WAITR:  STI ; ENABLE INTERRUPTS SO INTASK CAN RUN
```

```
;  WAIT UNTIL INTASK GETS A RESPONSE OR A FULL TO
;  NOT FULL TRANSITION, OR UNTIL TIMEOUT.

        PUSH    WORD PTR DGROUP:MIPSENDWTMBX
        MOV     AX,200          ; TIME UNITS
        PUSH    AX
        CALL    DEQUEUE
        CMP     SENDRESULT,0H ; IF ZERO, TIMEOUT
        POP     BX              ; DEVICE POINTER
        JNZ     REPLY

DEAD:   POP     DX                  ; DISCARD SOCKET
        POP     DX                  ; DISCARD MSG PTR
        MOV     BYTE PTR [BX],0H ; DEV STATUS DEAD
        MOV     AL,DEADP            ; RET STATUS DEAD
        JMP     EXIT

; SOMETHING IS IN THE MAILBOX. IF WAITING FOR A
; REPLY THEN IT IS THE REPLY, ELSE IT IS THE
; FULL TO NOT FULL TRANSITION.

REPLY:  CMP     SENDSTATE,2H ; WAITING FOR REPLY?
        JE      DOREPL
        JMP     TOP ; IF NOT, PROCESS TRANSITION.

DOREPL: POP     DX               ; SOCKET
        POP     DX               ; MESSAGE POINTER
        MOV     AL,SENDRESULT ; RETURN VALUE

EXIT:   MOV SENDSTATE,0H ; NOT WAITING
        PUSH    AX         ; SAVE STATUS
        PUSH    WORD PTR DGROUP:MIPUSEPERMIT
                           ; LET OTHER CALLERS GO
        CALL    SENDUNIT ; RETURN PERMIT TO OS
        POP     AX         ; RECALL STATUS
        AND     AL,7FH     ; RESET HIGH BIT
        RET                ; RETURN TO CALLER

MIPSEND      ENDP


; THIS ROUTINE CALCULATES DEVICE POINTER, WHICH
; POINTS TO DEVICE INFO FOR DESTINATION DEVICE.
; IT ASSUMES THE DEVICE ID IS IN AL.
; IT USES AX, BX, AND CX.

CALCDEVPTR:

        MOV     CL,0EH
        MUL     CL
        MOV     BX,AX
        LEA     BX,WORD PTR MIPDEVICEINFO[BX]
        CMP     BYTE PTR [BX+1],0FFH ; ACTIVE?
        RET

CODE    ENDS

        END
```

```
$TITLE('MIP INPUT TASK')
        NAME    INTASK

; DEFINE RQD RESULTS

GERROR  EQU      1H
GBUSY   EQU      4H
FIRSTG  EQU      8H
GDISAB  EQU     10H
GFULL   EQU     20H
DISABT  EQU     40H
FULLF   EQU     80H

TERROR  EQU      1H
TBUSY   EQU      4H
FIRSTT  EQU      8H
TDISAB  EQU     10H
TEMPTY  EQU     20H
DISABG  EQU     40H
EMPTYF  EQU     80H

; DEFINE MIP COMMANDS AND RESPONSES

CSEND   EQU     70H
SENTOK  EQU     80H
UNKNP   EQU     81H
ACTIVP  EQU     83H
INSUFM  EQU     85H
INACTP  EQU     87H
DEADP   EQU     89H

DGROUP  GROUP   DATA
DATA    SEGMENT PUBLIC 'DATA'

        EXTRN   MIPDEVICEINFO:NEAR
        EXTRN   SENDSTATE:BYTE, SENDDEVICE:BYTE
        EXTRN   MIPSENDWTMBX:NEAR
        EXTRN   SENDMSG:NEAR,    SENDRESULT:BYTE
        EXTRN   PORTTOMAILBOX:NEAR, MIPDEVCNT:BYTE
        EXTRN   INTERRUPTSEMAPHORE:NEAR

TRESULT DB 0
SREQID  DB 0            ; MUST FOLLOW TRESULT

DATA    ENDS

CGROUP  GROUP   CODE
CODE    SEGMENT PUBLIC 'CODE'
        ASSUME  CS:CGROUP,DS:DGROUP

        EXTRN   REQUESTGIVEPTR:NEAR
        EXTRN   REQUESTTAKEPTR:NEAR
        EXTRN   RELEASEGIVEPTR:NEAR
        EXTRN   RELEASETAKEPTR:NEAR
        EXTRN   CALCDEVPTR:NEAR
        EXTRN   ENABLEINTERRUPT:NEAR
        EXTRN   ENQUEUE:NEAR
        EXTRN   SENDUNIT:NEAR, RECEIVEUNIT:NEAR

        PUBLIC  MIPINTASK
```

```
MIPINTASK       PROC NEAR

; THIS IS THE BASIC SERVICE ROUTINE. WAIT FOR AN
; INTERRUPT AT THE INTERRUPT SEMAPHORE. THEN LOOK
; INTO THE REQUEST QUEUES.

SLEEP:  PUSH    WORD PTR DGROUP:INTERRUPTSEMAPHORE
        CALL    RECEIVEUNIT

; LOOK AT ALL KNOWN DEVICES.

        MOV     DL,0FFH ; START COUNTER AT -1
        PUSH    DX
NEXT:                   ; LOOK AT NEXT DEVICE
        POP     DX      ; GET DEVICE COUNTER
        INC     DL
        CMP     DL,MIPDEVCNT  ; END OF DEVICES?
        JE      SLEEP   ; IF SO, THEN LEAVE.

; LOOK AT RQ FOR EACH DEVICE.

        PUSH    DX          ; SAVE DEVICE COUNTER
        MOV     AL,DL
        CALL    CALCDEVPTR; GET PTR TO DEV INFO
        JNE     NEXT        ; JUMP IF DEVICE IS DEAD

        INC     BX          ; POINT TO DEVICE STATUS
        LES     SI,DWORD PTR [BX+1H]; PTR TO INRQD

; TEST SIGNALS

        XOR     AX,AX        ; WILL CLEAR SIGNALS
        XCHG    AX,ES:[SI] ; GET FULL & EMPTY SGNL
        OR      AX,AX        ; ARE BOTH ZERO?
        JZ      NEXT         ; JMP IF BOTH ARE ZERO

        PUSH    BX   ; SAVE DEV PTR
        JNS     TAKE ; TEST FULL SIGNAL (SIGN BIT)

        PUSH    AX   ; SAVE SIGNALS

; WE HAVE A FULL TO NOT FULL TRANSITION.
; SEE IF ANYONE WAS WAITING.

        CMP     SENDSTATE,1H ; WAITING FOR CHANGE?
        JNZ     EMPTYT       ; JUMP IF NOT.

        CMP     DL,SENDDEVICE; FOR THIS DEVICE?
        JNZ     EMPTYT       ; JUMP IF NOT.

        MOV     SENDRESULT,1 ; TELL MIPSND
        PUSH    WORD PTR DGROUP:MIPSENDWTMBX
        PUSH    WORD PTR DGROUP:SENDMSG
        CALL    ENQUEUE
```

```
; NOW LOOK FOR AN EMPTY TO NOT-EMPTY TRANSITION.

EMPTYT: POP     AX   ; GET SIGNALS BACK
        TEST    AL,1 ; EMPTY SIGNAL SET?
        JNZ     TAKE ; JMP IF SET

        POP     BX   ; NO SUCH TRANSITION
        JMP     SHORT NEXT ; NEXT DEVICE

; NOW TAKE ALL THINGS FROM THIS RQ UNTIL AN ERROR
; OCCURS.  THE MOST LIKELY ERROR IS THAT THE
; QUEUE IS EMPTY.

TAKE:   POP     DI   ; DEV PTR
        PUSH    DI
        CALL    REQUESTTAKEPTR
        JZ      OKTAKE

; THE TAKE RETURNED WITH AN ERROR. THAT MAY MEAN:
;    (1) IT WAS EMPTY, OR (2) IT WAS DISABLED.

        TEST    AL,10H  ; DISABLED?
        POP     SI      ; DEVICE POINTER
        JZ      NEXT    ; IF ZERO, THEN WAS EMPTY

DISABLED:
        MOV     BYTE PTR [SI],0  ; DEV STATUS DEAD
        JMP     SHORT NEXT       ; NEXT DEVICE

; THERE IS SOMETHING IN THE QUEUE.  TAKE IT.
; ES:[BX] POINTS TO RQE.

OKTAKE: MOV     AX,WORD PTR ES:[BX] ; GET REQUEST
                                    ; AND SRCREQID
        MOV     SREQID,AH  ; SAVE SRCREQID
        CMP     AL,CSEND   ; IS IT A COMMAND?
        JNE     RECRSP     ; NO? MUST BE RESPONSE.

; SEE IF SOCKET IS OPEN.

        MOV     AL,ES:[BX+3H] ; GET PORT
        MOV     AH,0H
        MOV     DI,AX         ; CONVERT TO MAILBOX
        MOV     AL,BYTE PTR PORTTOMAILBOX [DI]
        OR      AL,AL         ; IS MAILBOX ZERO?
        JZ      INACTV

        PUSH    AX  ; SAVE MAILBOX FOR LATER USE

; THE SOCKET IS OPEN.

        MOV     TRESULT,SENTOK
```

```
; GET IDS POINTER FROM RQE AND
; CONVERT TO ADDRESS TOKEN.

        MOV     AX,ES:[BX+5]
        MOV     DX,ES:[BX+7H]
        AND     AX,0FFF0H ; GET RID OF LOW 4 BITS
        AND     DX,0FH
        OR      AX,DX   ; FORM TOKEN
        MOV     CL,4
        ROR     AL,CL   ; REVERSE LOWER 4, UPPER 12
        PUSH    AX      ; MAILBOX ALREADY ON STACK

; NOW SAVE LENGTH, IDS-ID, AND OWNDEV

        MOV     CX,ES:[BX+9]    ; LENGTH
        MOV     DX,ES:[BX+0BH]  ; IDS-ID/OWN-DEV
        MOV     ES,AX           ; TOKEN FOR MSG
        XOR     DI,DI           ; CLEAR
        MOV     ES:[DI+2],CX ; PUT LENGTH INTO MSG
        MOV     ES:[DI+4],DX ; PUT IDS AND OWN DEV

; SEND IT TO USER.

        CALL    ENQUEUE ; PARAMS ALREADY ON STACK
        JMP     SHORT GENRSP

INACTV: MOV     TRESULT,INACTP ; PORT NOT ACTIVE
        JMP     SHORT GENRSP   ; RETURN RESPONSE

; THE RECEIVED ITEM IS A RESPONSE.
; LET MIPSND KNOW ABOUT IT.

RECRSP: MOV     SENDRESULT,AL ; REQUEST CODE
        CMP     SENDSTATE,2H  ; WAITING FOR REPLY?
        JNE     ENDRSP        ; NO? DISCARD IT.

        PUSH    WORD PTR DGROUP:MIPSENDWTMBX
        PUSH    WORD PTR SENDMSG
        CALL    ENQUEUE
ENDRSP: JMP     SHORT RELTAKE ; RESPONSE DONE

;  GENERATE A RESPONSE TO A RECEIVED COMMAND.

GENRSP: XOR     CX,CX   ; ZERO COUNTER

TRYINGTOGIVE:
        POP     DI   ; GET DEVICE PTR
        PUSH    DI
        INC     CX
        JZ      DEAD ; IF ZERO THEN TIMEOUT
        PUSH    CX
        CALL    REQUESTGIVEPTR
        POP     CX
        JNZ     NOGIVE ; NOT ZERO MEANS ERROR.
```

```
; THERE IS SPACE IN THE RQ.

        MOV     AX,WORD PTR TRESULT ; GET BOTH
                             ; TRESULT AND SREQID
        MOV     ES:[BX],AX
        POP     DI           ; RETRIEVE DEV PTR
        PUSH    DI
        CALL    RELEASEGIVEPTR
        JMP     SHORT RELTAKE

NOGIVE:              ; EITHER FULL OR DISABLED
        TEST    AL,GDISAB    ; DISABLED?
        JZ      TRYINGTOGIVE ; NO? KEEP TRYING.

DEAD:   JMP     DISABLED     ; GIVE UP.

RELTAKE:
        POP     DI           ; DEV PTR
        PUSH    DI
        CALL    RELEASETAKEPTR
        JMP     TAKE

MIPINTASK       ENDP

CODE    ENDS

        END
```

APPENDIX D
ETHERNET DATA LINK LIBRARY

Overview

The Ethernet Data Link Library (EDL80.LIB) provides a set of procedures that simplify the interface between the iSBC 550 Ethernet Communications Controller and users of the Ethernet Development System. The routines in the library are designed to run on the 8080 or 8085 processor of a Series II or Series III Microcomputer Development System under the ISIS operating system.

The Ethernet Data Link Library offers an easy way to use an Ethernet network. The library routines embody a MIP facility. They help to communicate with the External Data Link (EDL) of the Ethernet Controller without concern for details such as how to initialize the Ethernet Controller or how to use the MIP facility.

The Ethernet Data Link Library contains these routines:

1. CQSTRT
2. CQCONN
3. CQDISC
4. CQAMID
5. CQDMID
6. CQXMIT
7. CQCKTX
8. CQSBUF
9. CQCKRX
10. CQREAD
11. CQRDCL

CQSTRT configures the MIP facility and starts it and other communications software that run on the Ethernet Controller.

Before using the network, you must tell the Data Link Layer on the Ethernet Communications Controller which type codes and multicast addresses to accept. Type codes are not interpreted by the Data Link Layer; they are used to identify the Client Layer protocols associated with each frame. A multicast address associates one station with a group of other stations that have the same multicast address. The CQCONN routine specifies type codes; the CQAMID routine specifies multicast addresses. CQDISC and CQDMID tell the Data Link Layer to cease accepting certain type codes and multicast addresses.

The Ethernet Controller has no memory that the station host can access; therefore, to receive packets from the network, you must supply buffer space using the CQSBUF routine. When a packet is received, EDL returns the buffer containing that packet. The CQSBUF function effectively implements the ReceiveFrame function of the Ethernet Specifications.

CQXMIT passes a buffer to EDL to send over the network. This function effectively implements the TransmitFrame function of the Ethernet Specifications.

Packet transmission and reception are asynchronous operations. A buffer passed to the Data Link Layer may be held for an arbitrarily long time. For this reason, the

CQSBUF and CQXMIT calls do not wait for the buffers to be returned; instead, you must use CQCKRX and CQCKTX to check whether the system is done with these buffers.

The Library Procedures

This section explains each of the library routines and shows the form with which to declare them in a PL/M-80 program. The data type WORD refers to a two-byte item that is not used as an address. In PL/M-80, WORD may be defined thus:

```
DECLARE WORD LITERALLY 'ADDRESS';
```

The routines of EDL80.LIB are bound to your program by using the LINK command. For example, suppose the name of your program is MYPROG.OBJ. Suppose also that all files are on disk drive :F1:. To link the EDL80.LIB routines to your program, enter:

```
LINK :F1:MYPROG.OBJ, :F1:EDL80.LIB, :F1:PLM80.LIB
  TO :F1:MYPROG.LNK
```

In all of the library routines, STATUS$P is a pointer to a WORD variable that indicates the results of calling the routine. Always be sure to check this field after calling the routine. The values that may be returned in this word are defined in the description of each routine, but one set of values is common to all of the routines (except CQSTRT). This is the set of values in the range 81H through 89H that indicate an error detected by one of the MIP facilities involved in the communication with the Ethernet Controller. These values are defined below:

- 81H — Unknown destination port or device
- 83H — Port on destination device is already active
- 85H — Destination device has insufficient resources to receive message
- 87H — Port on destination device is not active
- 89H — Destination device does not respond

CQSTRT

This procedure initializes the MIP facility on the Ethernet Controller and starts execution of the communications software. CQSTRT must be executed before any other EDL80.LIB routine.

```
CQSTRT: PROCEDURE (STATUS$P) EXTERNAL;

DECLARE STATUS$P ADDRESS;     /* Output. */

END CQSTRT;
```

The possible status returns are:

- 0 — Operation complete
- 1 — No response from the Ethernet Controller

CQCONN

A call on this procedure instructs the Data Link Layer to receive packets containing a specific data link type code. Note that, when there is more than one host at a station, EDL does not distinguish between type codes specified in CQCONN requests from different hosts. Therefore, any host may receive packets containing type codes specified by any other host at the same station. Up to eight types may be active at any time; however, the Ethernet Controller uses two type codes, leaving space for only six types.

```
CQCONN: PROCEDURE (TYPE, STATUS$P) EXTERNAL;

DECLARE TYPE       WORD,          /* Input. */
        STATUS$P   ADDRESS;       /* Output. */

END CQCONN;
```

TYPE is a 16-bit Ethernet data link type code for which the Ethernet Controller should start looking.

The possible status returns are:

0 — Operation complete

1 — Exceeded limit of eight type codes

81H through 89H — MIP facility error (as defined above)

CQDISC

This procedure causes the Data Link Layer to cease forwarding those packets that contain a specific data link type code.

```
CQDISC: PROCEDURE (TYPE, STATUS$P) EXTERNAL;

DECLARE TYPE       WORD,          /* Input. */
        STATUS$P   ADDRESS;       /* Output. */

END CQDISC;
```

TYPE is the 16-bit Ethernet data link type code for which the Ethernet Controller should stop looking.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

CQAMID

This procedure instructs the Data Link Layer to recognize packets containing a specific multicast address. Note that, when a station has more than one host processor, EDL does not distinguish between multicast addresses specified in CQAMID requests from different processors. Therefore, any host may receive packets containing multicast addresses specified by other hosts at the same station. Up to eight multicast addresses may be active at one time.

```
CQAMID: PROCEDURE (MCID$P, STATUS$P) EXTERNAL;

DECLARE MCID$P     ADDRESS,       /* Input. */
        STATUS$P   ADDRESS;       /* Output. */

END CQAMID;
```

MCID$P contains the address of a six-byte multicast address for which the Ethernet Controller should start looking.

The possible status returns are:

0 — Operation complete

1 — Exceeded limit of eight multicast addresses

81H through 89H — MIP facility error

CQDMID

This procedure causes the Data Link Layer to cease recognizing a specific multicast address.

```
CQDMID: PROCEDURE (MCID$P, STATUS$P) EXTERNAL;

DECLARE MCID$P    ADDRESS,      /* Input. */
        STATUS$P  ADDRESS;      /* Output. */

END CQDMID;
```

MCID$P contains the address of the six-byte multicast address for which the Ethernet Controller should stop looking. If this address is not active, the routine has no effect.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

CQXMIT

This procedure queues up a packet to be transmitted. Figure D-1 shows the format of the transmit buffer. The items in the buffer are described below:

- (RESERVED). The first 18 bytes of the buffer are reserved for use by EDL80.LIB and the Ethernet Controller.
- LENGTH. Enter the length (in bytes) of the contiguous portion of the packet, counting from the end of the EXTENSION LENGTH field.
- EXTENSION POINTER. Enter a 24-bit IDS pointer to an extension to the buffer. Note that the high-order eight bits of this address are stored separately from the high-order 16 bits. If EXTENSION LENGTH is zero, this pointer is ignored.
- IDS-ID. Enter the identifier of the inter-device segment in which the extension area is located.
- EXTENSION LENGTH. Enter the length (in bytes) of the extension; enter zero if the buffer lies in one continuous area of memory.
- DESTINATION ADDRESS. Enter the data link address or multicast address of the Ethernet station or stations to which you wish to send this packet.
- SOURCE ADDRESS. The Data Link Layer fills this field with the hardware address of the sending station.
- TYPE. Fill in the data link type code.
- DATA. Enter 46 to 1500 bytes of user data. To meet network minimum packet size requirements, you must pad smaller messages to make them at least 46 bytes long.

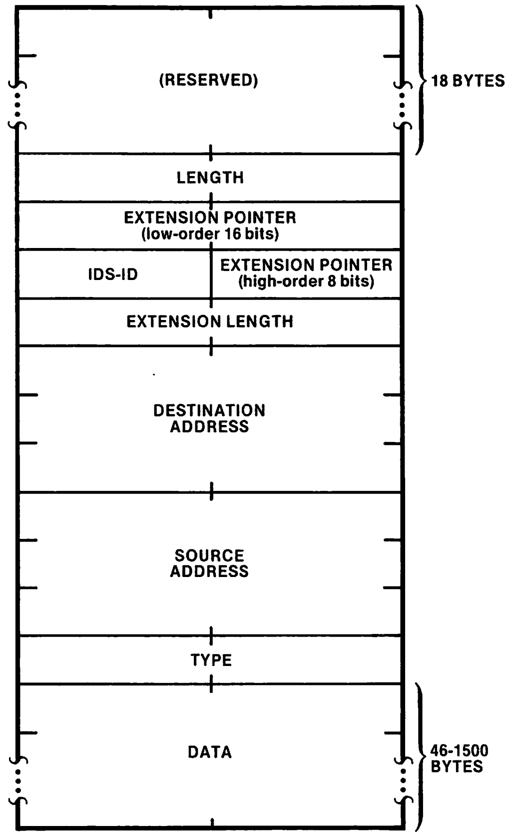

Figure D-1. Transmit Buffer. 769-26

The interface to CQXMIT is described below:

```
CQXMIT: PROCEDURE (BUFFER$P, STATUS$P) EXTERNAL;

DECLARE BUFFER$P  ADDRESS,      /* Input. */
        STATUS$P  ADDRESS;      /* Output. */

END CQXMIT;
```

BUFFER$P contains the address of the buffer area described in figure D-1.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

CQCKTX

This function retrieves transmit buffers that have been passed to EDL *via* the CQXMIT procedure.

```
CQCKTX: PROCEDURE (STATUS$P) ADDRESS EXTERNAL;

DECLARE STATUS$P  ADDRESS;      /* Output. */

END CQCKTX;
```

This is a typed procedure that returns a value of type ADDRESS. If EDL is finished with a previously submitted transmit buffer, CQCKTX returns the address of the buffer and sets the status word to reflect the status of the buffer. If the buffer is still in use, CQCKTX returns a zero address and sets the status to zero.

The possible status returns are:

0 — No errors

1 — The transmit request was rejected because the DATA field was shorter than 46 bytes or longer than 1500 bytes

81H through 89H — MIP facility error

CQSBUF

This procedure provides a buffer in which to place a packet from the network. When EDL receives a packet, it copies it into this buffer and returns the buffer to the CQCKRX procedure. The data area of the buffer should be at least 1500 bytes long to ensure that a maximum-length packet does not overflow the end of the buffer. You may call CQSBUF several times in succession, thereby making several buffers available for receipt of packets. Make sure that the number of buffers supplied is great enough to receive all the packets that might arrive before more buffers can be supplied. If the Ethernet Controller receives a packet but does not have a user buffer in which to place it, the packet is discarded.

Figure D-2 illustrates the format of a receive buffer. The fields are filled by the Ethernet Controller as explained below:

- (RESERVED). First 18 bytes are reserved for use by EDL80.LIB and the Ethernet Controller
- LENGTH. The length in bytes of the received packet, counting from the beginning of the destination address through the end of the data area
- DESTINATION ADDRESS. The physical address of the receiving station or a multicast address
- SOURCE ADDRESS. The data link address of the station from which the packet came
- TYPE. The data link type code. This can only contain one of the types specified in a previous CQCONN call
- DATA. Filled with 46 to 1500 bytes of received data. If this area is not long enough to contain a packet received from the network, data beyond the end of the buffer is overwritten.

The interface to CQSBUF is described below:

```
CQSBUF: PROCEDURE (BUFFER$P, STATUS$P) EXTERNAL;

DECLARE BUFFER$P  ADDRESS,      /* Input. */
        STATUS$P  ADDRESS;      /* Output. */

END CQSBUF;
```

BUFFER$P contains the address of the receive buffer area.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

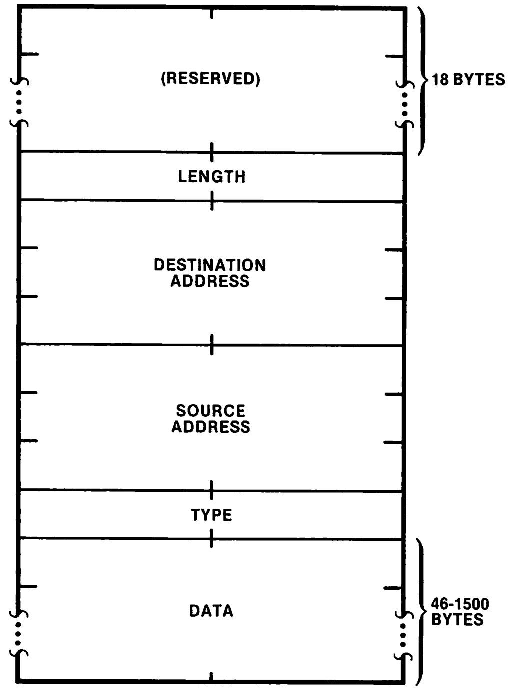

Figure D-2. Receive Buffer. 769-27

CQCKRX

This function determines whether any packets have been received.

```
CQCKRX: PROCEDURE (STATUS$P) ADDRESS EXTERNAL;

DECLARE STATUS$P  ADDRESS;      /* Output. */

END CQCKRX;
```

This is a typed procedure that returns a value of type ADDRESS. If any packets have been received, CQCKRX returns the address of the oldest one. If no packets have been received, CQCKRX returns an address of zero.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

CQREAD

This procedure accesses certain items of information held by the Data Link Layer. Refer to Chapter 4 for a definition of the accessible data link objects.

```
CQREAD: PROCEDURE (OBJECT, RETURN$P, STATUS$P) EXTERNAL;

DECLARE OBJECT    WORD;         /* Input. */

DECLARE RETURN$P  ADDRESS,      /* Output. */
        STATUS$P  ADDRESS;

END CQREAD;
```

OBJECT contains the identifying number of the data link object to be read.

RETURN$P contains the address of a six-byte area in which to place the value of the object. If the object is less than six bytes long, CQREAD fills the lowest-address bytes of the area; the content of the remaining bytes is undefined. If OBJECT is not a valid identifier of a data link object, the content of the return area is undefined.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

CQRDCL

This procedure reads an accessible data link object, and, if the object is a counter, clears it after it has been read. If the object is not a counter, CQRDCL functions the same as CQREAD.

```
CQRDCL: PROCEDURE (OBJECT, RETURN$P, STATUS$P) EXTERNAL;

DECLARE OBJECT    WORD;          /* Input. */

DECLARE RETURN$P  ADDRESS,       /* Output. */
        STATUS$P  ADDRESS;

END CQRDCL;
```

OBJECT contains the identifying number of an accessible data link object.

RETURN$P contains the address of a six-byte area in which to place the value of the object. If the object is less than six bytes long, CQRDCL fills the lowest-address bytes; the content of the remaining bytes is undefined. If OBJECT is not a valid identifier of a data link object, the content of the return area is undefined.

The possible status returns are:

0 — Operation complete

81H through 89H — MIP facility error

Example Calling Sequences

```
CALL CQSTRT (.STATUS);

                   LXI     B,STATUS
                   CALL    CQSTRT

CALL CQCONN (OUR$TYPE, .STATUS);

                   LHLD    OURTYPE
                   MOV     B,H
                   MOV     C,L
                   LXI     D,STATUS
                   CALL    CQCONN

CALL CQDISC (5009H, .STATUS);

                   LXI     D,STATUS
                   LXI     B,5009H
                   CALL    CQDISC
```

```
CALL CQAMID (.BROADCAST, .STATUS);

                LXI     D,STATUS
                LXI     B,BROADCAST
                CALL    CQAMID

CALL CQDMID (.PROJECT$GROUP, .STATUS);

                LXI     D,STATUS
                LXI     B,PROJECTGROUP
                CALL    CQDMID

CALL CQXMIT (.OUT$BUFFER, .STATUS);

                LXI     D,STATUS
                LXI     B,OUTBUFFER
                CALL    CQXMIT

DO WHILE (RETURN$P := CQCKTX (.STATUS)) = 0;

        CT:     LXI     B,STATUS
                CALL    CQCKTX
                SHLD    RETURNPTR
                MOV     A,H
                ORA     L
                JZ      CT
END;

CALL CQSBUF (.IN$BUFFER, .STATUS);

                LXI     D,STATUS
                LXI     B,INBUFFER
                CALL    CQSBUF

DO WHILE (RETURN$PTR := CQCKRX (.STATUS)) = 0;

        CR:     LXI     B,STATUS
                CALL    CQCKRX
                SHLD    RETURNPTR
                MOV     A,H
                ORA     L
                JZ      CR
END;

CALL CQREAD (OBJ$ID, .OBJ$VALUE, .STATUS);

                LHLD    OBJID
                PUSH    H
                LXI     D,STATUS
                LXI     B,OBJVALUE
                CALL    CQREAD

CALL CQRDCL (09H, .OBJ$VALUE, .STATUS);

                LXI     B,9H
                PUSH    B
                LXI     D,STATUS
                LXI     B,OBJVALUE
                CALL    CQRDCL
```

INDEX

Notes