intel®

iNA 960 PROGRAMMER'S REFERENCE MANUAL

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

Order Number: 122193-001

iNA 960 PROGRAMMER'S REFERENCE MANUAL

Order Number: 122193-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 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

Intel retains the right to make changes to these specifications at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may only be used to identify Intel products:

BITBUS	i_m	iRMX	Plug-A-Bubble
COMMputer	iMMX	iSBC	PROMPT
CREDIT	Insite	iSBX	Promware
Data Pipeline	int_el	iSDM	QueX
Genius	int_elBOS	iSXM	QUEST
i	Intelevision	Library Manager	Ripplemode
$\overset{\Delta}{1}$	$int_eligent$ Identifier	MCS	RMX/80
I^2ICE	$int_eligent$ Programming	Megachassis	RUPI
ICE	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SOLO
iDBP	iOSP	MULTICHANNEL	SYSTEM 2000
iDIS	iPDS	MULTIMODULE	UPI
iLBX			

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	3/84	R.T.

PREFACE

This manual provides all of the details necessary to use iNA 960 Release 1, and is intended for use by system designers and application programmers.

This manual contains nine chapters and five appendixes:

- Chapter 1, "Introduction," presents an overview, describes the ISO model, and summarizes the portion of the ISO network implemented by iNA 960.
- Chapter 2, "User Interface to iNA 960," describes the target environments of iNA 960 and the user interface to iNA 960.
- Chapter 3, "Data Link Layer," contains the data link services, data link commands, and the data link configuration macros.
- Chapter 4, "Network Layer," contains the description of the internet address, which must be used when the transport services are used.
- Chapter 5, "Transport Control Layer," describes virtual circuit service, datagram service, buffers, and the user interface to the services.
- Chapter 6, "Network Management Facility," describes layer management, down-line loading, dumping, echo testing, and the configuration macros (boot server and NMF).
- Chapter 7, "ROM-Based NMF," describes the boot consumer facility that can be burned into ROM (this facility is used for booting a station from the network).
- Chapter 8, "iRMX 86 System Generation," describes the configuration of the communication software for iRMX 86-based systems.
- Chapter 9, "Component Support Interface," describes the message delivery mechanisms available, the hardware environment required, and the steps needed to configure and initialize the communications software for non-iRMX-based systems.
- Appendix A, "iNA 960 Files," describes all the files included on the iNA 960 delivery diskettes.
- Appendix B, "Network Management Facility Objects," describes all of the objects for the network management facility in iNA 960.
- Appendix C, "Sample User Routines for Component Support," describes how iNA can be configured to run on an 8086/82586-based system.
- Appendix D, "CONFIGURE Command Parameters," gives the format from the CONFIGURE command argument field and describes the command parameters.
- Appendix E, "MULTIBUS Interprocessor Protocol (MIP)," is a specification of the MIP interface.

Related Publications

For further information on the iNA 960, refer to the following publications:

- *iNA 960 Architecture Reference Manual*, order number 122194
- *iMMX 800 Software Reference Manual*, order number 144912
- *82586 Reference Manual*, order number 210891
- *iAPX 186 High Integration 16-bit Microprocessor Data Sheet*, order number 210451

- *iRMX 86 Configuration Guide*, order number 172765
- *iAPX 86, 88 Family Utilities User's Guide*, order number 121616
- *iSBC 550 Kit Ethernet Controller Kit Programmer's Manual*, order number 122113

References

1. *ISO/TC97/SC16 N 1433 Second DP8073.* Information Processing Systems – Open Systems Interconnection – Connection Oriented Transport Protocol Specification. April, 1983.
2. *IEEE 802.2 Local Area Network Standards.* Logical Link Control Sublayer.
3. *IEEE 802.3 Local Area Network Standards.* CSMA/CD Access Method and Physical Layer Specifications.

Notational Conventions

UPPERCASE	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.
italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
directory-name	Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename*.
filename	Is a valid name for the part of a *pathname* that names a file.
pathname	Is a valid designation for a file; in its entirety, it consists of a *directory* and a *filename*.
pathname1, pathname2, ...	Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.

punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered:

```
SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')
```

CONTENTS

FIGURES

TABLES

CHAPTER 1
INTRODUCTION

1.1 iNA 960

iNA 960 is a general purpose local area network software package implementing the class 4 services of the ISO transport specification and network management functions. iNA 960 is designed to operate in environments consisting of the 8086, 8088, and 80186 microprocessors and the 82586 communications co-processors. iNA 960 also supports Intel's board-level Local Area Network products, the iSBC 550 KIT and the iSBC 186/51. Examples for using iNA 960 include network design stations, manufacturing process control, communicating word processors, and financial services workstations.

1.2 Functional Overview

The iNA 960 design is an 8073 standard implementation of the class 4 transport protocol defined by the International Standard Organization. The transport layer provides a reliable full-duplex message delivery service on top of the *IEEE 802.3* standard packet delivery service implemented by the 82586 (or equivalent) physical and data link protocols.

The software (which consists of linkable modules) can be configured to implement a range of capabilities and interfaces. In addition to reliable process-to-process message delivery, the capabilities include a datagram service, a boot server, a direct user access to the data link layer, and a comprehensive network management facility. The interface options accommodate the various system environments.

iNA 960 can be configured to run either under the iRMX 86 operating system along with user software, or on top of a dedicated 8086, 8088, or 80186 processor coupled with an 82586 to provide a communications front-end processor.

The software also includes a network management service. This facility enables the user to monitor and adjust the network's operation in order to maintain it and optimize its performance.

The current release of iNA 960 includes a *null* network layer supporting the data link layer and transport layer interfaces without providing internetwork routing service. This capability will be implemented in later releases of iNA 960.

1.3 The ISO Model Summary

The following paragraphs summarize the seven layers of the ISO model. Figure 1-1 illustrates the seven-layer ISO model.

1.3.1 Application Layer

The application layer supports public files and file consumers. File consumers are stations (nodes) that access files on other stations (nodes). Transparent access for file consumers is supported, although a file consumer need not take advantage of it. File consumers may or may not have local mass storage devices. If local mass storage is

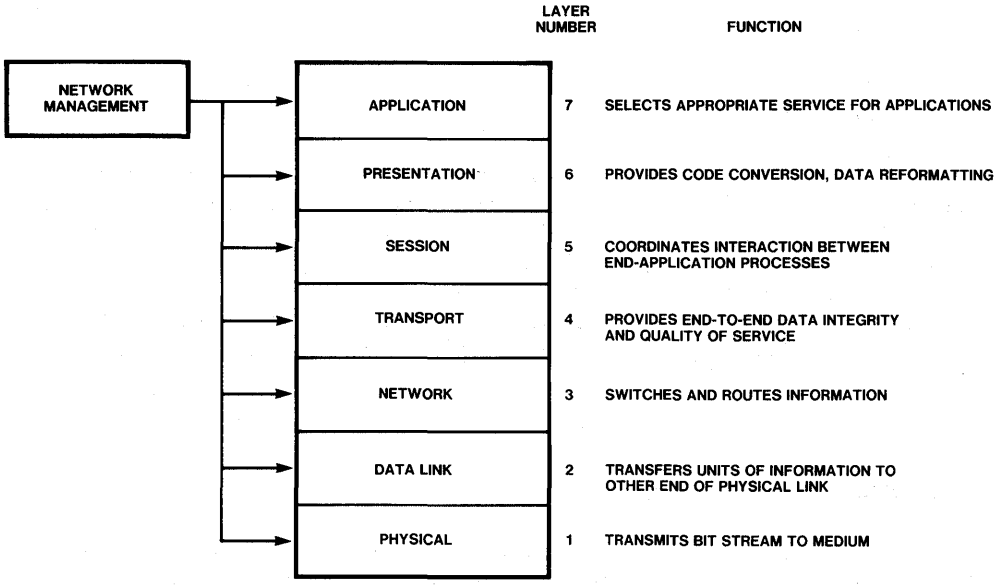

Figure 1-1. ISO Open Systems Interconnection Model 122193-1

not available, the file consumers must rely on other systems for initialization and all mass storage capability.

Public file stations are systems that permit file consumers to access files on their local disks and to queue printer requests for their local printers. The application layer only supports a limited number of transactions going on simultaneously. For example, a workstation with local mass storage could be both a file consumer and a public file station.

This layer is not implemented in iNA 960.

1.3.2 Presentation Layer

The presentation layer presents information to communicating application entities in a format that preserves meaning while resolving syntax differences. An example of a presentation layer function would be the conversion of character codes from EBCIDIC to ASCII.

A presentation entity can access another presentation entity in only two ways: either by initiating a session connection or by accepting a session connection. This session connection takes place in the session layer.

This layer is not implemented in iNA 960.

1.3.3 Session Layer

The session layer provides the necessary methodology for cooperating presentation entities to organize and synchronize the dialogue and to manage the exchange of data. This layer also provides for dynamically binding process names to transport

addresses. This facility is available for both virtual circuit and datagram transport services. The session layer translates process names to transport addresses and uses the transport service to provide virtual circuit service between any two processes that have names bound to network addresses. The datagram service works in much the same manner. Thus, in order to communicate with a remote process, a process need only know the name, not the transport address of the remote process. The session layer uses the name server of the network management facility to maintain and translate the process name to transport address bindings.

A process name is a character string that consists of an individual name or a member of a group. Process names are the endpoints of session virtual circuits or datagrams. If an endpoint is fully specified, it matches a particular individual or a particular member of a group. If an endpoint is partially specified, it matches any member of a particular group.

When a virtual circuit is used, a perfect channel is provided. No errors occur in transmission, and all packets are delivered in the same order that they were sent.

When a datagram is used, an attempt is made to deliver each packet as an isolated unit. The packets may arrive out of order, or not at all.

This layer is not implemented in iNA 960.

1.3.4 Transport Layer

The transport layer provides transparent transfer of data between session entities. Therefore, the transport layer relieves the transport users from any concern with the detailed way in which reliable and cost-effective data transfer is achieved.

The transport layer is designed primarily for user software that has moderate to large amounts of data to be transferred over the network. Some examples might be file transfers, block data moves, or file sharing applications.

The transport layer is designed to provide a widely-used, error-free service that does not depend on the particular characteristics of any specific lower layer.

1.3.5 Network Layer

The network layer provides the means to establish, maintain, and terminate network connections between systems containing communication application entities and the functional and procedural means to exchange network service data units between transport entities over network connections.

This layer also provides independence to transport entities from routing and relay considerations associated with the establishment and operation of a network connection. The network layer of iNA 960 offers a datagram service to higher layer users.

1.3.6 Data Link Layer

The data link layer provides functional and procedural methodology to establish, maintain, and release data link connections among network entities. The data link layer is responsible for framing, addressing, error detection, and link management.

1.3.7 Physical Layer

The physical layer provides functional and procedural characteristics to activate, maintain, and deactivate physical connections for bit transmission between data link entities. This layer is directly concerned with the actual transmission medium (wire, fiber optics, radio), the signalling means (voltage or current levels), data rate, and mechanical specifications.

1.4 iNA 960 Software

iNA 960 implements the transport layer, the network layer, and the data link interface portion of the data link layer from the ISO model, as well as a network management facility.

1.4.1 Transport Layer

The Transport Layer provides message delivery services between client processes running on computers (network hosts or nodes) anywhere in the network.

Client processes are identified by a combination of a network address defining the node and a transport service access point defining the interface point through which the client accesses the transport services. The combined parameters, called the transport address, are supplied by the user for both the local and the remote client processes to be connected. The iNA 960 transport layer implements two kinds of message delivery services: virtual circuit and datagram.

The virtual circuit service provides a reliable point-to-point message delivery service ensuring maximum data integrity; it is fully compatible with the *ISO 8073* class 4 protocol. The virtual circuit services entail:

- *Reliable Delivery* – Data is delivered to the destination in the exact order it was sent by the source, with no errors, duplications, or losses, regardless of the quality of service available from the underlying network service.
- *Data Rate Matching (flow control)* – The transport layer attempts to maximize throughput while conserving communication subsystem resources by controlling the rate at which messages are sent. That rate is based on the availability of receive buffers at the destination and its own resources.
- *Multiple Connection Capability (Process Multiplexing)* – Several processes simultaneously use the transport layer with no risk that progress or lack of progress by one process will interfere with others.
- *Variable Length Messages* – The client software can arbitrarily submit short or long messages for transmittal without regard for the minimum or maximum network service data unit (NSDU) lengths supported by the underlying network services.
- *Expedited Delivery* – With this service the client can transmit up to 16 bytes of urgent data bypassing the normal flow control. The expedited data is guaranteed to arrive before any normal data that is submitted afterward.

The datagram service provides a best-effort message delivery between client processes requiring less overhead, therefore allowing higher throughput than virtual circuits. The datagram service entails:

- The datagram service option transfers data between client processes without establishing a virtual circuit. The service is a best-effort capability; data may be lost or misordered. Data can be transferred at one time to a single destination or to several destinations (multicast).

1.4.2 Data Link Layer

The data link option allows the user to access the functionalities of the data link layer directly, instead of going through the network and transport layers. This flexibility is useful when the user needs custom higher layer software or does not need the network layer and transport layer services.

Through the data link the capabilities supporting the lower layers in iNA 960 are made directly available to the user. The data link enables the user to establish and delete data link connections, transmit packets to individual and multiple receivers, and configure the data link software to meet the requirements of the given network environment.

1.4.3 Network Management Facility

Network management supplies a network with planning, operation, and maintenance facilities. The planning capability gathers network usage information such as peak activity, total packets sent, and CRC errors. This information allows the system manager to make adjustments for day-to-day usage of the network. Normal day-to-day operation deals with network functions such as initialization, termination, monitoring, and performance optimization. Maintenance deals with detection, isolation, amputation, and repair of network faults. Many functions can be performed both on local and remote nodes.

iNA 960 includes extensive network management support for initialization, address conversions, operation, and maintenance. Network management is a distributed function that is built into every layer. Its activities are being performed constantly to ensure the proper operation of the network. Enquiries into the state of the network can be made from any system on the network. A central network control station does not exist in iNA 960. Network management is the responsibility of every station.

CHAPTER 2
USER INTERFACE TO iNA 960

2.1 Overview

The iNA 960 software is designed to run either under iRMX 86 or separately from the host on a dedicated front-end processor. Hence, the user has two types of interfaces to iNA 960: the *iRMX 86 interface* and the *component support interface*. In both environments, the interface is based on exchanging memory segments (called request blocks) between iNA 960 and the client. The format and contents of the request blocks are virtually the same in both configurations; only the request block delivery mechanism is different.

This chapter describes these two interfaces. The general form of the request block is given, along with the procedures for implementing each interface.

2.2 iRMX™ 86 Interface

In the iRMX 86 environment, both the user program and iNA 960 run under iRMX 86. The communications software is implemented as a first-level user's job running on the iRMX 86 nucleus. In addition, if the boot server is configured into the communication system, the Basic I/O System (BIOS) is required. Figure 2-1 shows iNA 960 as an iRMX 86 job.

iNA 960 will run in any iRMX 86 environment, including configurations based on the 80130. Some of the typical hardware implementations include the iSBC 550 KIT combined with an 8086-, 8088-, or 80186-based host and the iSBC 186/51 COMMputer board.

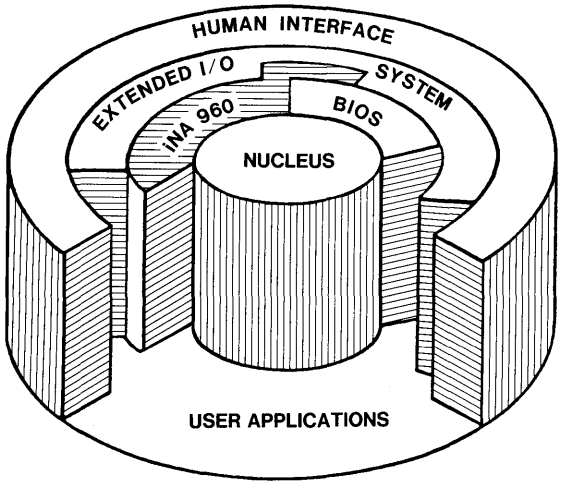

Figure 2-1. iNA 960 as an iRMX™ Job 122193-2

With the iRMX 86 interface, the user sends communications commands to iNA 960 via request blocks. Either the user allocates and constructs the request blocks directly (called the request block interface), or available PL/M style procedures construct and deliver the request blocks for the user (called the procedure interface).

2.2.1 iRMX™ 86 User Id

Before issuing a communications request, the user must acquire a *user id.* This is done by making the following system call:

```
COMM$USER$TOKEN = CQ$CREATE$COMM$USER (Exception_ptr)
```

This system call is performed only once for each user job. The returned value is a token that identifies the COMM system user. All subsequent communications requests should use this value in the User field of the request block or the User parameter of the procedure interface, as described in the following sections.

2.2.2 Request Blocks

For the request block interface, the iRMX 86 user allocates memory segments called *request blocks.* The components of a request block are called *fields.* Each request block consists of *fixed format* fields and *variable format* fields. The fixed format fields make up the first 16 bytes of the request block, and their field definitions remain the same for all request blocks. The definitions of the variable format fields, however, depend on the command being issued.

The general form of a request block is as follows:

```
DECLARE Rb STRUCTURE (
      Reserved              (2) WORD,
      Length                BYTE,
      User                  WORD,
      Response_port         BYTE,
      Return_mailbox        WORD,
      Segment_token         WORD,
      Subsystem             BYTE,
      Opcode                BYTE,
      Response              WORD,
      Arguments             (*) BYTE );
```

where

Reserved	is reserved for a system pointer.
Length	is the length (in bytes) of the request block.
User	is an identifier specifying the iRMX 86 user issuing the command. This identifier is obtained by making a call to the function CQ$CREATE$COMM$USER, as described in the previous section.
Response_port	is 0FFH for the iRMX 86 user.
Return_mailbox	is a token for the iRMX 86 mailbox where the request block is returned.
Segment_token	is an iRMX 86 token for the request block segment.

Subsystem	has the following interpretation: 20H–Data Link Layer 40H–Transport Layer Virtual Circuit Services 41H–Transport Layer Datagram Services 80H–Network Management Facility
Opcode	is a code identifying the specific command to be performed. See each command for the appropriate value.
Response	is set by iNA 960 after performing (or attempting to perform) the command. This indicates the result of executing the command.
Arguments	are specific to each command.

2.2.3 Request Block Interface

To issue a particular command, the user fills in the fixed format argument fields as required by the command. The user then calls the following iRMX 86 extension procedure:

```
CALL CQ$COMM$RB (Rb_token, Exception_ptr)
```

where

Rb_token	is an iRMX 86 token for the request block segment.
Exception_ptr	is a pointer to a word field that is to contain the returned exception code. For descriptions of iRMX 86 exception codes, see the Intel publication *Getting Started with the Release 5 iRMX 86 System.*

This procedure delivers the request block to iNA 960 for processing. At some future time, iNA 960 returns the request block to the mailbox specified in the Return_mailbox field of the request block. The iRMX 86 user then retrieves the request block from the mailbox to determine the result of the request and obtain any data received as a result of the request.

After processing a command, iNA 960 returns a response code in the Response field of the request block. This code indicates whether the command was executed successfully; if the command was not executed successfully, the code specifies the reason the command was terminated abnormally.

After extracting the returned request block from the return mailbox, the iRMX 86 user can determine from the Subsystem and Opcode fields which command was returned. By examining the Response field, the user can take appropriate action for that command.

2.2.4 Procedure Interface

The procedure interface provides the iRMX 86 user with a friendlier processing environment than the request block interface. Here, the allocation and formatting of the request blocks are performed by the iNA 960 software. The user simply calls the procedure form of the command with parameters that specify the user's command options. See each command description for the format of the procedure call and for descriptions of the command parameters.

As an example, the ECHO command of the network management facility has the following procedure call format:

```
CALL CQ$NMF$ECHO (Transmit_data_count, Datalink_addr_ptr,
      User, Return_mailbox, Exception_ptr)
```

The response interface is non-procedural. That is, the end of command processing is the same as for the request block interface: the request block created from the procedure call is returned to the mailbox specified by the Return_mailbox parameter. The user must inspect the Subsystem, Opcode, and Response fields of the returned request block to determine the appropriate course of action, then delete the request block segment.

Each procedure also returns an exception code that usually indicates the status of parsing the parameter list for syntax.

Although the procedure interface is easier to use, it is more expensive than the request block interface in terms of processing overhead. The user must decide if the convenience of using the procedure interface is worth the reduction in performance.

2.2.5 User Include Files and Libraries

The iNA 960 delivery diskettes contain *include files* and *user interface libraries.* These are described in Appendix A.

The user must include some of the include files in the PL/M 86 procedures, as determined by the system calls used in the program. The include files are as follows:

CQRB.EXT – Request block interface.
CQTL.EXT – Transport procedure interface.
CQNMF.EXT – NMF procedure interface.
CQDL.EXT – Data link procedure interface.

During the link process the appropriate iNA 960 user interface library must be linked. There are two libraries, one for COMPACT, and one for LARGE and MEDIUM modes. These contain the routines that satisfy external references to system calls made in the application code. The libraries are as follows:

CQC.LIB – COMPACT mode.
CQL.LIB – LARGE or MEDIUM mode.

2.3 Component Support Interface

The iNA 960 software can be configured to support implementations where iRMX 86 is not the primary operating system, where communications tasks are separated from the host to increase performance, or where an existing front-end communications processor configuration is being upgraded. iNA 960 supports such implementations by providing network services on an 8086, 8088, or 80186 processor. In these hardware environments, the component support interface is used for submitting commands to iNA 960. This interface uses a request block interface similar to the iRMX 86 interface. The procedure call interface, however, is not available to the component support user.

See Chapter 9 for details on the component support interface.

CHAPTER 3
DATA LINK LAYER

3.1 Overview

The data link layer is responsible for transmitting data over the network. The data link layer performs framing, addressing, and collision handling. The iNA 960 data link is a datagram service only, and does not ensure accurate reception of data. Reliable communication over the network is provided by the transport layer virtual circuit service.

In addition to transmitting packets of data, the data link layer receives incoming packets, checks them for CRC errors, and routes them to the proper user process. The data link layer also keeps statistics that monitor the performance of the node and the rate of errors. These objects are available to the network management layer to read and set. Finally, the data link layer is responsible for configuring the 82586 controller.

Figure 3-1 shows the partitioning of the data link and physical layers into functions performed by the 82501, 82586, and iNA 960 data link software. Data link software is responsible for controlling the 82586 and interfacing with the other software modules.

The iNA 960 data link and physical layers conform to the *IEEE 802* specifications. This is also illustrated in Figure 3-1. Data link software implements class I of the *Logical Link Control (LLC)* sublayer as described in the *IEEE 802.2* standard. The 82586 and 82501 together implement the *Media Access Control* sublayer as described in the *IEEE 802.3* standard. The media access method used is Carrier-Sense Multiple-Access with Collision Detection (CSMA/CD).

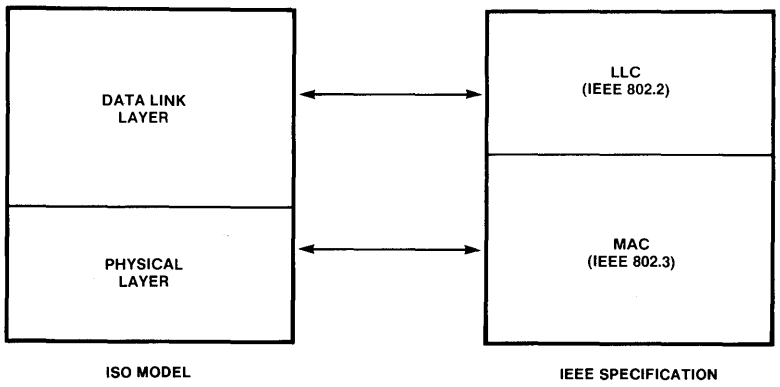

Figure 3-1. Comparison of ISO Model/IEEE Specification 122193-3

The user accesses the services of the data link through the iRMX 86 request block, iRMX 86 procedure call, or component support interface. Data link commands available include Transmit, Post buffers, Data link host address setup, and Configure (the hardware controller). Data link parameters can be read and cleared using the network management facility.

The data link layer is initialized by a configuration module as described in Section 3.5.

3.2 Hardware Environment

Future releases of iNA 960 may support different data link protocols. Currently, however, only a single data link controller class is supported: the 82586. Within this class of controllers, there are four subclasses. This differentiation is necessary because of variations at the board level implementations or simulations of the 82586 interface.

Following are the four subclasses of the 82586 controller class:

iSBC 550FW	A board level simulation of the 82586. It requires slightly different interfaces from those of the 82586.
iSBC 552	Uses memory-mapped signaling and some predefined address locations.
iSBC 186/51	The single board COMMputer. The software resets the loopback mode on the board when data link is initialized.
general_82586	The same as the 186/51, except the loopback mode is not reset.

3.3 Data Link Communication Protocol

As shown in Figure 3-1, the data link layer and the physical layer together correspond to the LLC class I sublayer of the *IEEE 802.2* specification and the MAC sublayer of the *IEEE 802.3* specification. This specification features a CSMA/CD media access protocol and packet framing as described in this section. Data link users are identified by link service access points (LSAPs), also described in this section.

3.3.1 Link Service Access Points (LSAPs)

Data link users communicate via link service access points (LSAPs). An LSAP is a code that identifies a specific user process or another layer (see Figure 3-2). These codes are explicitly defined as follows:

Data Link Layer	00H
Transport Layer	FEH
Network Management Layer	08H
User Process	multiple of 4 in the range $12 \leq \text{LSAP} \leq 255$

Each receiving process is identified by a *destination* LSAP (DLSAP), and each sending process is identified by a *source* LSAP (SLSAP).

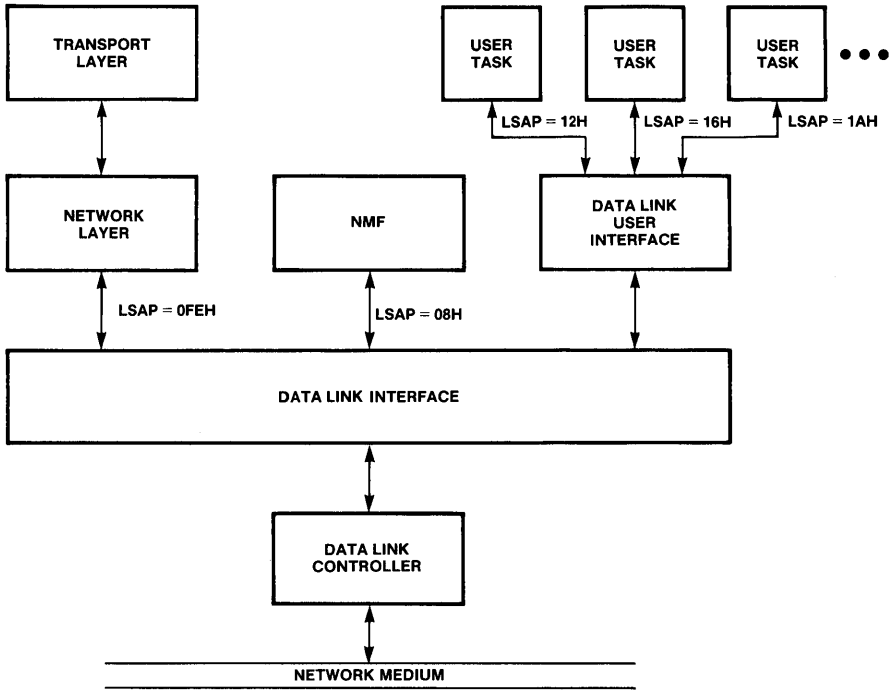

Figure 3-2. Data Link Interface 122193-4

When a packet is sent, the destination process is identified by the DLSAP field in the first data buffer. Before a destination process can receive a packet, however, its DLSAP must be included in a list of the active LSAPs for the destination data link. The data link layer only receives packets targeted for LSAPs on its active list.

To add an LSAP to the active list, use the CONNECT command. When a connection is no longer needed, it should be disconnected. This is done with the DISCONNECT command. A maximum of eight LSAPs can be active at one time.

3.3.2 Logical Link Control Sublayer (IEEE 802.2 class I)

The iNA 960 software incorporates a class I Logical Link Control (LLC) sublayer within the data link layer, as described in the *IEEE 802.2* specification. This is a datagram service only; it does not ensure the reliable reception of data.

3.3.3 Media Access Sublayer (IEEE 802.3)

In addition to the LLC sublayer, the data link layer also contains a Media Access sublayer (MAC) that is based on the *IEEE 802.3* specification. The MAC uses a medium access technology known as Carrier-Sense Multiple-Access with Collision

Detection (CSMA/CD). In addition, the medium used has a bus topology as stated in the *IEEE 802.3* specification. The MAC sublayer of the data link layer performs the following functions:

- *Framing* – frame boundary delimitation and frame synchronization.
- *Addressing* – handling of source and destination addresses.
- *Error detection* – detection of physical medium transmission errors.
- *Medium allocation* – collision avoidance.
- *Contention resolution* – collision handling.

Before transmitting a packet, the MAC tests the carrier-sense signal. If the signal indicates that the bus is not busy, the packet is transmitted. If, however, the bus is busy, the MAC waits for the carrier-sense signal to change. After the channel has been freed, the MAC delays transmission for an additional period of time called the *interframe gap period*, then attempts to transmit the packet again.

One station can access the bus. Before the carrier-sense signal can propagate to all the other stations, a second station may also access the bus. This results in two interfering transmissions, in other words, a *collision.*

When the physical layer detects a collision, it raises the collision detect signal. The MAC then transmits a bit pattern called a *jam.* This ensures that the duration of the collision is long enough to be detected by all the stations involved in the collision. After the jam, each station terminates transmission and waits a random period of time before attempting a retransmission.

As many retransmissions as needed are attempted to successfully transmit the packet (up to a configurable limit). However, since repeated collisions are an indication of a busy medium, the MAC attempts to reduce this load by successively increasing the time interval from which the retry delay period is randomly selected.

3.4 Data Link Commands

This section gives a brief introduction to the data link commands, followed by a detailed description of each command.

Data link identifies users by LSAPs. Therefore, a user must establish an LSAP with data link. Any incoming packet with this LSAP is routed to this user. The connection between a user and an LSAP is established and severed using the following commands:

CONNECT	Establishes a connection between a user process and an LSAP.
DISCONNECT	Severs the connection between the user process and the LSAP.

Data is transmitted over the network with the following command:

TRANSMIT	Transmits a given packet over the network.

A *packet descriptor* is an iNA 960 request block with argument fields that contain locations of buffers attached to the request block. A *buffer* is any user memory segment set aside to hold data. To receive incoming data destined for a particular user (LSAP), packet descriptors and sufficient buffers must be posted. This is done with the following two commands:

POST_RPD	Posts a receive packet descriptor that optionally specifies a set of buffers to hold incoming packets.

POST_RBD	Posts a single buffer for holding incoming data (in addition to any buffers posted by the POST_RPD command).

Finally, certain data link parameters may be set dynamically by the application. These parameters include the host id (except for the iSBC 550 FW) and the multicast address list. In addition, the 82586 class hardware controller can be configured in a limited way (see Appendix D). Commands that change these parameters are the following:

CONFIGURE	Sends configuration information to the data link controller.
IA_SETUP	Sets the host id of the local node.
MC_ADD	Adds a multicast address to the list of active multicast addresses for the node.
MC_REMOVE	Removes a multicast address from the list of active multicast addresses for the node.

3.4.1 CONNECT

This command provides a connection between the COMM system user (specified by the User parameter) and the given DLSAP. If the DLSAP is already active, a new association replaces the old one. Only eight DLSAPs can be active, and each user can have only one connection.

NOTE

A connection to a given LSAP must be made before any buffers or packet descriptors can be posted to that LSAP. An attempt to connect more than eight DLSAPs results in a Connect command error (exception code 10H). Also, each data link COMM system user (each iRMX 86 job using COMM) can have only one LSAP.

Request Block Format

```
DECLARE Connect_rb STRUCTURE (
      Rb_header        (6) WORD,
      Subsystem        BYTE,
      Opcode           BYTE,
      Response         WORD,
      Dlsap            BYTE,
      Reserved         BYTE,
      Port             BYTE );
```

Procedure Call Format

```
CALL CQ$DL$CONNECT (Subsystem, Dlsap, User,
      Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	82H.
Dlsap	The DLSAP for the connection. This must be a multiple of 4 and have a value between 12 and 255.
Port	Must be set to 0FFH.
Subsystem	20H.

3.4.2 DISCONNECT

If the specified connection exists, it is severed. If the connection does not exist, the command is ignored. When a connection is disconnected, all receive buffers and packet descriptors posted with this LSAP (and this LSAP's associated *unique* COMM user id) are returned to the user. Since a packet descriptor can hold up to four buffers, the user must ensure that the number of buffers posted does not exceed four times the number of packet descriptors.

Request Block Format

```
DECLARE Disconnect_rb STRUCTURE (
      Rb_header         (6) WORD,
      Subsystem         BYTE,
      Opcode            BYTE,
      Response          WORD,
      Dlsap             BYTE,
      Reserved          BYTE );
```

Procedure Call Format

```
CALL CQ$DL$DISCONNECT (Subsystem, Dlsap, User,
      Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	83H.
Dlsap	The DLSAP for the connection. This must be a multiple of 4 and have a value between 12 and 255.
Subsystem	20H.

3.4.3 TRANSMIT

This command transmits a packet consisting of from one to four buffers. If the total number of bytes transmitted is less than the minimum packet size, padding is done by data link. This padding is transparent to the user.

Request Block Format

```
DECLARE Transmit_rb STRUCTURE (
      Rb_header       (6) WORD,
      Subsystem       BYTE,
      Opcode          BYTE,
      Response        WORD,
      Reserved        WORD,
      Buf_count       WORD,
      Byte_count      (4) WORD,
      Buf_loc         (4) POINTER,
      Dst_addr_ptr    POINTER );
```

Procedure Call Format

```
CALL CQ$DL$TRANSMIT (Subsystem, Buf_param_token,
      Dst_addr_ptr, User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	84H.
Buf_count	The number of buffers specified by the command. The number of buffers can range from 0 to 4.
Byte_count	An array of 4 words. Each Byte_count (i) is the size in bytes of buffer i.
Buf_loc	An array of 4 pointers. Each Buf_loc (i) is a pointer to the start of buffer i.
Dst_addr_ptr	A pointer to an array of 6 bytes where the destination Host_id is stored.
Subsystem	20H.
Buf_param_token	The token for a segment that has the following form:

```
DECLARE Buf_param STRUCTURE (
      Buf_count       WORD,
      Byte_count      (4) WORD,
      Buf_token       (4) TOKEN );
```

Here, Buf_count and Byte_count are as described above. Buf_token (i) is a token to buffer memory segment i.

User Data Buffers

The first buffer specified by the TRANSMIT command contains ISO control information in addition to user data. The second and all subsequent buffers contain only user data. The first buffer takes the following form:

```
DECLARE First_transmit_buffer STRUCTURE (
      Destination_lsap     BYTE,
      Source_lsap          BYTE,
      Iso_cmd              BYTE,
      Data                 (*) BYTE );
```

The fields of the user transmit buffers have the following interpretation:

Destination_lsap	The LSAP for the destination entity to which the packet is forwarded.
Source_lsap	The LSAP for the source entity that sends the packet.
Iso_cmd	Must be set to 03H for user data, as specified by *IEEE 802.2.*
Data	An array of BYTES that contains the actual data to be transmitted.

All of the remaining buffers have this format:

```
DECLARE Next_transmit_buffer STRUCTURE (
      Data                   (*) BYTE );
```

Note that for the first transmit buffer, the value of the Byte_count parameter includes Destination_lsap, Source_lsap, and Iso_cmd.

3.4.4 POST_RPD

The POST_RPD command posts a single packet descriptor together with up to four user buffers. This user packet descriptor and any associated buffers are kept by data link until they are used to hold information on a receive packet. An LSAP must be established with the CONNECT command before receive packet descriptors and buffers can be posted to that LSAP.

When a packet is received by data link, it is associated with a user packet descriptor and buffers posted at the receiving destination LSAP. The packet descriptor provides information on the return mailbox where the user waits for receive data.

Request Block Format

```
DECLARE Post_rpd_rb STRUCTURE (
      Rb_header        (6) WORD,
      Subsystem        BYTE,
      Opcode           BYTE,
      Response         WORD,
      Dlsap            BYTE,
      Reserved         BYTE,
      Buf_count        WORD,
      Return_count     WORD,
      Byte_count       (4) WORD,
      Buf_loc          (4) POINTER );
```

Procedure Call Format

```
CALL CQ$DL$POST$RPD (Subsystem, Dlsap, Buf_param_token,
      User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	85H.
Dlsap	The DLSAP for the connection. This must be a multiple of 4 and have a value between 12 and 255.
Buf_count	The number of buffers associated with the packet descriptor. The number of buffers can range from 0 to 4. When the request block is given to data link, this field contains the number of buffers posted with this request block. When the request block is returned to the user, this field contains the number of buffers returned with the packet descriptor.
Return_count	Filled in by data link. This is the total number of bytes in the receive packet returned by data link.
Byte_count	An array of 4 words. Each Byte_count (i) is the size in bytes of buffer i.
Buf_loc	An array of 4 pointers. Each Buf_loc (i) is a pointer to the start of buffer i.
Buf_param_token	The token for a segment that has the following form:

```
DECLARE Buf_param STRUCTURE (
      Buf_count        WORD,
      Byte_count       (4) WORD,
      Buf_token        (4) TOKEN );
```

Here, Buf_count and Byte_count are as described above. Buf_token (i) is a token to buffer memory segment i.

User Data Buffers

The first buffer returned with a packet descriptor contains destination and source addresses, ISO control information, and user data. The second and all subsequent buffers contain only user data. The first receive buffer must be at least 17 bytes long; it has the following form:

```
DECLARE First_receive_buffer STRUCTURE (
      Destination_addr      (16) BYTE,
      Source_addr           (16) BYTE,
      Information_len       WORD,
      Destination_lsap      BYTE,
      Source_lsap           BYTE,
      Iso_cmd               BYTE,
      Data                  (*) BYTE );
```

The remaining buffers have this format:

```
DECLARE Next_receive_buffer STRUCTURE (
      Data                  (*) BYTE );
```

The fields of the user receive buffers have the following interpretation:

Destination_addr	The host id of the destination node.
Source_addr	The host id of the source node.
Information_len	Identical to the Return_count field of the request block. Both parameters contain the number of information bytes in a received packet. This value is the number of bytes received subsequent to an Information_len field.
Destination_lsap	The LSAP for the destination entity to which the packet is forwarded.
Source_lsap	The LSAP for the source entity that sends the packet.
Iso_cmd	Set to 03H for user data, as specified by *IEEE 802.2*.
Data	An array of bytes that contains the actual data.

For the first receive buffer, the Byte_count parameter includes the first 17 bytes of control information.

Note that the last returned buffer may contain less information bytes than the full buffer size. To determine the number of data bytes in the last buffer, subtract all the previous buffer sizes (these buffers will be full) from Return_count.

3.4.5 POST_RBD

This command posts a single receive buffer described by the parameters Buf_count and Buf_token. An LSAP must be established with the CONNECT command before any receive buffers can be posted at this LSAP.

After the command to post a buffer is given, the request block is *immediately* returned to the user. Note that this is in contrast to the POST_RPD command. Data link keeps the buffer posted and associates it with a packet descriptor when this buffer is used and returned to the user.

Request Block Format

```
DECLARE Post_rbd_rb STRUCTURE (
      Rb_header       (6) WORD,
      Subsystem       BYTE,
      Opcode          BYTE,
      Response        WORD,
      Dlsap           BYTE,
      Reserved        BYTE,
      Byte_count      WORD,
      Buf_loc         POINTER );
```

Procedure Call Format

```
CALL CQ$DL$POST$RBD (Subsystem, Dlsap, Buf_token,
      Byte_count, User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	86H.
Dlsap	The DLSAP for the connection. This must be a multiple of 4 and have a value between 12 and 255.
Byte_count	The size (in bytes) of the buffer.
Buf_loc	A pointer to the start of the buffer.
Buf_token	A token for the receive data buffer segment.

User Data Buffers

See the POST_RPD command for a description of the format of the user data buffers.

3.4.6 CONFIGURE

This command configures the data link controller. The configuration information is contained in a segment of memory that may be up to 12 bytes long. With the request block format, the actual configuration data is part of the request block. For the procedure format, a pointer to the configuration data is passed as a parameter to the procedure.

For the 82586 class controller, the following restrictions apply to the configuration data:

- The address allocation bit is always reset.
- The save bad packet option is always OFF.
- The host id must be 6 bytes long.
- Data link performs packet padding operations. The user must not alter the minimum packet length parameter.

See Appendix D for a description of the configuration segment.

Request Block Format

```
DECLARE Configure_rb STRUCTURE (
      Rb_header          (6) WORD,
      Subsystem          BYTE,
      Opcode             BYTE,
      Response           WORD,
      Reserved           WORD,
      Count              WORD,
      Configure          (12) BYTE );
```

Procedure Call Format

```
CALL CQ$DL$CONFIGURE (Subsystem, Count, Config_info_ptr,
      User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	88H.
Count	The size in bytes of the configuration information. This can be up to 12 bytes.
Configure	An array of 12 bytes that is the actual configuration information segment.
Config_info_ptr	A pointer to the configuration information segment.

3.4.7 IA_SETUP

This command sets up the individual address (host id) for a node. For the iSBC 550 the command becomes an address read command because the host id on an iSBC 550 FW cannot be changed by the software.

Request Block Format

```
DECLARE Ia_setup_rb STRUCTURE (
      Rb_header       (6) WORD,
      Subsystem       BYTE,
      Opcode          BYTE,
      Response        WORD,
      Reserved        WORD,
      Count           WORD,
      Address         (6) BYTE );
```

Procedure Call Format

```
CALL CQ$DL$IA$SETUP (Subsystem, Count, Ia_addr_ptr,
      User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	89H.
Count	The size in bytes of the host id. This number must be 6.
Address	An array of 6 bytes that comprise the host id.
Ia_addr_ptr	A pointer to the start of a memory segment that contains the host id.

3.4.8 MC_ADD

This command adds a multicast address to the data link multicast address list. Multicast addresses can only be added one at a time. Note that the iNA 960 data link performs perfect multicast filtering, whereas the 82586 controller performs imperfect multicast filtering.

With the request block format, the actual address is part of the request block. With the procedure format, the address is stored as a segment of memory. A pointer to the segment is then passed as a parameter to the procedure.

Request Block Format

```
DECLARE Mc_add_rb STRUCTURE (
      Rb_header     (6) WORD,
      Subsystem     BYTE,
      Opcode        BYTE,
      Response      WORD,
      Reserved      WORD,
      Count         WORD,
      Mc_address    (6) BYTE );
```

Procedure Call Format

```
CALL CQ$DL$ADD$MC (Subsystem, Count, Mc_addr_ptr,
      User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	87H.
Count	The size in bytes of a multicast address. This number must be 6.
Mc_address	An array of 6 bytes that comprise the multicast address.
Mc_addr_ptr	A pointer to the start of a memory segment that contains the multicast address.

3.4.9 MC_REMOVE

This command removes a single multicast address from the list of active multicast addresses for a given data link. With the request block format, the address is part of the request block. For the procedure format, the address is stored in memory, and a pointer to the memory location of the address is passed as a parameter to the procedure.

Request Block Format

```
DECLARE Mc_remove_rb STRUCTURE (
      Rb_header          (6) WORD,
      Subsystem          BYTE,
      Opcode             BYTE,
      Response           WORD,
      Reserved           WORD,
      Count              WORD,
      Mc_address         (6) BYTE );
```

Procedure Call Format

```
CALL CQ$DL$REMOVE$MC (Subsystem, Count, Mc_addr_ptr,
      User, Return_mailbox, Exception_ptr);
```

Command Parameters

Opcode	8AH.
Count	The size in bytes of a multicast address. This number must be 6.
Mc_address	An array of 6 bytes that comprise the multicast address.
Mc_addr_ptr	A pointer to the start of a memory segment that contains the multicast address.

3.5 Configuration

The procedure used to configure and initialize the COMM system is described in Chapters 8 and 9. This procedure includes setting up the configuration file for each layer. These files consist of calls to configuration macros to set the operating parameters of the layers. This section defines the configuration macros for the data link layer.

3.5.1 Data Link Configuration Macros

The data link configuration macros define the data stucture of the data link layer. Calls to these macros are included in the configuration file DLCFG.A86. This file should be updated to reflect the desired data link configuration, and the result assembled to generate the configuration module DLCFG.OBJ. This module is then included in the link list when linking the communication system.

The file DLCFG.MAC contains the data link configuration macros. Therefore, the configuration file must contain the following include statement before any macro calls:

```
$INCLUDE (:SD:COMM/CONFIG/DLCFG.MAC)
```

Following are the data link configuration macros; they are given in the order that they should appear in the configuration program:

DL_CTRL	Specifies the controller subclass of the hardware.
DL_INT	Specifies the interrupt level used by the data link layer.
DL_SIGN	Specifies how the iNA processor signals the data link controller.
DL_SCP_ADDRESS	Specifies the location of the system control pointer (SCP).
DL_ISCP_ADDRESS	Specifies the location of the intermediate system control pointer (ISCP).
DL_HOSTID	Specifies where the default host id can be read.
DL_CONFIG	Specifies the default configuration of the 82586 controller and some channel characteristics.
DL_INTERNAL	Used internally to generate the *IEEE 802* protocol for the 82586 class of controllers.

3.5.2 DL_CTRL

This macro informs the data link software whether the target system is an iSBC 550FW, iSBC 186/51, iSBC 552, or general_82586. This macro takes the following form:

```
% DL_CTRL (Ctlr_subclass)
```

where

Ctrl_subclass is the controller subclass and has the following values:

00H – 550FW
01H – 186/51
02H – 552
03H – general_82586

The general_82586 controller subclass is created to allow for an iSBC 186/51-like environment. The basic assumption is that the environment must provide I/O ports where a default host id can be read during initialization. If these ports are not available, the user can provide one or more "dummy" host id ports that can be read during initialization. Subsequently, an Ia_setup command can be used to establish the desired host id.

Aside from the fact that the iSBC 186/51 provides six port addresses where the default host id can be read, the difference between subclasses 1 and 3 is that for subclass 1 data link resets the loopback mode on the iSBC 186/51 during initialization. No action is taken for a general_82586 subclass controller. For a general_82586 controller, the user must assume the responsibility for initializing the environment where data link functions.

3.5.3 DL_INT

This macro specifies the interrupt level used by data link and has the following form:

```
% DL_INT (Int_level)
```

where

Int_level — is the interrupt level. This parameter has the same format as that defined by the iRMX 86 nucleus and has these bit assignments:

bits 7–15	must be set to 0.
bits 4–6	first digit (0–7) of the interrupt level.
bit 3	set to 0 if a slave level is specified. Here, bits 0–2 specify the second digit of the interrupt level.
	set to 1 if a master level is specified. Here, bits 4–6 represent the entire level.
bits 0–2	second digit (0–7) of the interrupt level (defined only if bit 3 is 0).

3.5.4 DL_SIGN

The macro DL_SIGN is used to specify how the iNA processor signals the data link controller. In addition, the macro specifies the I/O port or memory location used by the processor to signal the controller (if signaling is used.) The macro has the following form:

```
% DL_SIGN (Signal_type, Signal_info)
```

where

Signal_type — specifies how the processor signals the data link controller. The values are as follows:

- 00H – processor never signals the controller. Signal_info is not defined.
- 01H – memory-mapped signaling. The processor signals the controller with a write to the memory location specified by Signal_info.

02H – I/O-mapped signaling. The processor signals the controller with a write to the I/O port location specified by Signal_info.

Signal_info depends on the value of Signal_type, as follows:

If Signal_type is 0, this parameter is not defined.

If Signal_type is 1, this parameter must contain the upper 16 bits of the 20-bit memory location to which the processor writes. In particular, memory-mapped signaling locations are at 16-byte boundaries.

If Signal_type is 2, this parameter must contain the 16-bit number of the I/O port used by the processor to signal the controller.

The Signal_type should be set according to the controller subclass. The following values are used:

iSBC 186/51	– 02H (I/O-mapped)
iSBC 550 FW	– 02H (I/O-mapped)
iSBC 552	– 01H (memory-mapped)
general_82586	– 01H (memory-mapped) or 02H (I/O-mapped)

Signal_info should be set as follows:

iSBC 186/51	– 00C8H (the port that channels to the 82586)
iSBC 550 FW	– One of the I/O ports defined in the *iSBC 550FW Hardware Reference Manual*
iSBC 552	– 4200H (a reserved number)
general_82586	– Any value compatible with the hardware

3.5.5 DL_SCP_ADDRESS

The macro DL_SCP_ADDRESS is used to specify the address of the system control pointer (SCP) of the data link controller. This macro has the following form:

```
% DL_SCP_ADDRESS (Scp_addr_offset, Scp_addr_base)
```

where

Scp_addr_offset is the offset of the SCP address.

Scp_addr_base is the base of the SCP address.

For the iSBC 186/51, iSBC 552, and general_82586 controller subclasses, the address of the SCP should be set to FFFF:6H. The iSBC 550 FW allows several SCP locations. See the *iSBC 550 FW Hardware Reference Manual* for details.

3.5.6 DL_ISCP_ADDRESS

This macro is used to specify the intermediate system control pointer (ISCP) and has the following form:

```
% DL_ISCP_ADDRESS (Iscp_addr_offset, Iscp_addr_base)
```

where

Iscp_addr_offset is the offset of the ISCP address.

Iscp_addr_base is the base of the ISCP address.

The ISCP offset must be set to 0 to be compatible with the 82586 controller. Other than this requiremenet, the ISCP location is completely configurable. However, if the user desires to conform to the iRMX 86 operating system, the ISCP should be located at 00FF:0H.

3.5.7 DL_HOSTID

The macro DL_HOSTID is used to specify the port numbers where the default host id can be read. The format of the macro is as follows:

```
% DL_HOSTID (Hostid_cnt, Hostid_loc0, Hostid_loc1,
     Hostid_loc2, Hostid_loc3, Hostid_loc4, Hostid_loc5)
```

where

Hostid_cnt specifies the number of host id address bytes. This number must be 6.

Hostid_loc0–5 are the I/O port numbers of the ports where the host id address bytes can be read.

The host id read from the port locations specified by this macro constitutes the default host id upon initialization. The host id may subsequently be changed using the Ia_setup command.

For all controller subclasses, the Hostid_cnt must be set to 6. Furthermore, for the iSBC 186/51, Hostid_loc0–5 should be set to

0F0H, 0F2H, 0F4H, 0F6H, 0F8H, 0FAH

For the iSBC 550FW and iSBC 552 subclasses, Hostid_loc0–5 are undefined.

For the general_82586 subclass controller, Hostid_loc0–5 should be set to the port locations where a default host id can be read, if available. Data link always tries to establish a default host id. Therefore, for the general_82586 subclass controller, any Hostid_loc0–5 parameters are treated as valid port locations to establish the default host id.

3.5.8 DL_CONFIG

This macro is used to initialize the communications controller and to provide some additional link characteristics. The format is as follows:

```
% DL_CONFIG (Linespeed0, Linespeed1, Mc_number, Config_cnt,
     Config0, Config1, Config2, Config3, Config4, Config5)
```

where

Linespeed0–1 specifies the physical channel transmission rate in bits per second. For example, if Linespeed0 is 0000H and Linespeed1 is 0010H, the transmission rate is 100000H (approximately 10 Mbps).

Mc_number	specifies the maximum number of multicast address bytes used. This number cannot exceed 60.
Config_cnt	specifies the number of significant configuration bytes in the 6 word parameters (Config0–5). This parameter can be between 0 and 12.
Config0–5	are 6 words that have the same form as the argument field of the configuration command. Only the first Config_cnt bytes are significant. The remaining fields should be set to 0.

The default configuration has the following values:

```
Linespeed0–1  =  (0000H, 0010H)
Mc_number     =  60
Config_cnt    =  0
Config0–5     =  (0, 0, 0, 0, 0, 0)
```

A Config_cnt of 0 leaves the data link controller at its default configuration after power up.

The DL_CONFIG macro allows the user to directly configure the 82586 communications controller. The configuration argument (as determined by Config_cnt and the configuration words Config0–5) is passed straight through to the 82568 during initialization. This forms the argument of the Configure command of the 82586. See Appendix D or the *82586 Reference Manual* and the *iSBC 550 Ethernet Controller Kit Programmer's Reference* for details on the argument fields of the 82586 Configure command. The configurability of the 82586 running under iNA 960 has limitations. See Section 3.4.6.

To better understand the configuration process during initialization, consider the following two cases. The first case is identical to the default and has the same affect as if no configuration were given (Config_cnt = 0). Here, the parameters are as follows:

```
Config_cnt    =  0CH
Config0–5     =  (080CH, 2600H, 6000H, F200H, 0H, 2EH)
```

The second case configures data link to run with No Carrier Sense. The parameters are as follows:

```
Config_cnt    =  0CH
Config0–5     =  (080CH, 2600H, 6000H, F200H, 8H, 2EH)
```

The user should note that when two or more nodes are connected without carrier sensing, such as, when two nodes are connected using an Ethel cable, the Transmit With No Carrier Sense bit of the configuration argument should be set as in the second configuration example.

3.5.9 DL_INTERNAL

This macro configures the internal data link to run with the *IEEE 802* protocol and the 82586 category of controllers. This macro has no arguments. A call to this macro has the following form:

```
% DL_INTERNAL
```

3.6 Exception Codes

For the procedure call interface, the location indicated by the Exception_pointer parameter is set to one of the following:

0 EXP$OK, the procedure parameters are copied into corresponding RBs with no error detected.

8004 EXP$ERR, an error is detected when the procedure parameters are copied to the corresponding request blocks.

For the request block interface, the Response field of the returned request block has the following values in data link:

0 Failure. Reason not specified or unknown.

1 Execution with no errors.

2 Number of configuration information bytes exceeds maximum.

4 Received packet overflow. The received packet requires more than 4 user receive buffers and therefore is truncated.

6 Size of transmit packet exceeds maximum (1500 bytes).

8 Invalid data link opcode.

10 CONNECT/DISCONNECT command error.

12 Subsystem not defined.

14 Number of address bytes in command exceeds maximum.

16 The 82586 reports that command execution is not OK.

CHAPTER 4
NETWORK LAYER

4.1 iNA 960 Network Layer

The network layer in iNA 960 Release 1 provides datagram delivery within a single network environment; no internetwork routing is provided.

The network layer implements a zero-length protocol in which the network layer header consists of one byte with a value of zero.

The network layer on iNA 960 is not accessible to the user. However, the network address defined below must be used when the transport services are used.

4.2 Internet Address

In iNA 960, the internet address consists of a 32-bit subnet identifier, a 48-bit Ethernet host identifier, and a 16-bit Network Service Access Point identifier (NASP ID). The length field is maintained so this internet address format can fit into the future "Variable Length Format" internet address scheme. The internet address described in Figure 4-1 is used at the network service interface point. iNA 960 does not use any internet address within the network protocol.

The internet address must contain the following values:

- Length (one byte) – OCH
- Subnet ID (doubleword) – 01H
- Host ID (6 bytes) – the ETHERNET host address
- NSAP ID (one word) – 01H

A requested Subnet ID other than 1 causes a "cannot reach" error response.

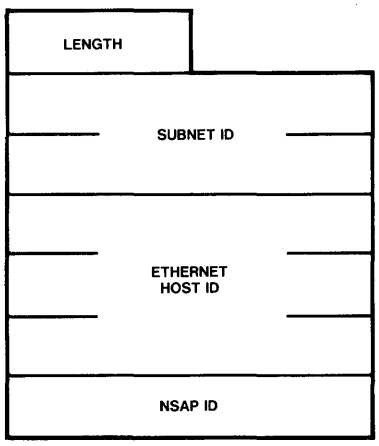

Figure 4-1. iNA 960 Internet Address Implementation 122193-5

CHAPTER 5
THE TRANSPORT CONTROL LAYER

5.1 Transport Services

The iNA Transport Service (iTS) provides message delivery services between user processes running on computers (network hosts or nodes) anywhere in an Intel Network Architecture (iNA)-compatible system.

The user processes are identified by means of transport addresses (TAs). A TA consists of a Network Address (NA) and a Transport Service Access Point (TSAP-ID). The TSAP-ID identifies the access point between the user process and the iTS.

The iNA Transport Service provides the following two types of services to the user:

1. A reliable connection-oriented Virtual Circuit (VC) message delivery service between two transport addresses.
2. A non-guaranteed connectionless datagram message delivery service between one transport address and one or several other transport addresses (multicast).

5.1.1 Virtual Circuit Service

The virtual circuit iTS uses the standard *ISO 8073 Class 4* transport protocol, which provides the following services:

- *Reliable Delivery*–Data is delivered to the destination in the exact order it was sent by the source, with no errors, duplicates, or losses, regardless of the quality of service available from the underlying network service.
- *Data Rate Matching (flow control)*–The iTS attempts to maximize throughput while conserving communication subsystem resources by controlling the rate at which messages are sent. This is based on the availability of receive buffers at the destination and its own resources.
- *Process Multiplexing*–Several processes can use the iTS simultaneously with no risk that progress or lack of progress by one process will interfere with other processes.
- *Variable Length Messages*–Short or long messages can be arbitrarily submitted for transmittal without regard for the minimum or maximum Network Service Data Unit (NSDU) lengths supported by the underlying network services.
- *Expedited Data Service*–Short, urgent messages can be transmitted ahead of the normal messages by bypassing the normal flow control mechanisms.

The iTS provides these services by means of a connection or virtual circuit. Pairs of users set up the connection (connection establishment phase), transfer data (data transfer phase) and terminate or disconnect (connection disestablishment phase) the connection between themselves.

5.1.2 Datagram Services

The datagram iTS uses proprietary mechanisms to transfer data between user processes without setting up a connection. This service gives no guarantees of delivery. Data can be lost or misordered. Data can be transferred at one time to a single destination or to several destinations (multicast).

5.2 Buffers

Use of the iTS requires the passing of address information and data back and forth between the user and iTS. The address information and data transferred by the user to iTS must first be loaded into a user buffer memory area. Pointers, segment tokens, or buffer descriptors are then used to specify the location of the buffers.

The three types of buffers used by iTS are the following:

1. The Transport Address Buffer
2. Contiguous Buffers
3. Non Contiguous Buffers

5.2.1 Pointers and Tokens

In iRMX, a buffer can be located by either a PL/M long pointer or by a segment token. If a segment token is used, the buffer must start at an 8086 paragraph boundary, and the token represents the base address (i.e., paragraph number). The offset is 0 if a token is used. The iTS interface allows the specification of buffers by either pointers or tokens.

5.2.2 Transport Address Buffer

The user must tell the iTS the local and remote TAs between which a virtual circuit is set up or a datagram is transferred. Here the word *local* means point of reference. *Remote* refers to a node or end of a connection that is on the other end of the local node.

A transport address consists of a Network Address (NA) defining the node and a TSAP-ID defining the point of access to a user process using iTS. At the local end, the NA is maintained by the network layer and is not needed by the transport layer. Thus, the iTS requires only the local TSAP-ID. However, iTS requires the complete remote TA (remote NA and remote TSAP-ID).

The formats of NAs have not yet been standardized. The length of an NA depends on the underlying network service provided to the transport layer. The contents of the NA are transparent to the transport layer, but the length must be known to allocate memory to save this address. Similarly, the lengths and format of TSAP-IDs have not been standardized and depend on transport user conventions. However, the lengths of TSAP-IDs must be known to store them in memory.

Thus, fundamentally, the iTS interface must allow for variable length NAs and TSAP-IDs in order to be flexible in the future. The method provided by iTS is to store these addresses in a user-defined buffer area, with a pointer to this area specified by the user to iTS. A single remote/local transport address pair is stored in a single contiguous variable length buffer (the transport address buffer) as illustrated by the PL/M declaration in Figure 5-1. The first byte of the transport address buffer must be 0 (for compatibility with future extensions of iNA).

For any implementation, it is assumed that the user is aware of the lengths and formats of the network addresses used to specify the remote end nodes of a connection or a datagram data transfer. The length of the NA is loaded into the remote NA length field of the TA buffer. Enough space is allocated in the buffer to load the address.

Similarly, user processes communicating via the transport service must agree on format and lengths of TSAP-IDs. The lengths are loaded into the appropriate TA buffer length fields, and enough buffer space is allocated to accommodate those TSAP-IDs.

```
DECLARE     Transport$address$buffer            STRUCTURE (
            Reserved                            BYTE, /* Must be set to 0*/
            Loc$tsap$id$len                     BYTE,
            Loc$tsap$id (loc$tsap$id$len)       BYTE
            Rem$net$addr$len                    BYTE
            Rem$net$addr (rem$net$addr$len)     BYTE,
            Rem$tsap$id$len                     BYTE,
            Rem$tsap$id (rem$tsap$id$len)       BYTE )
```

Figure 5-1. Transport Address Buffer Structure

Memory resource constraints impose limits on the lengths of the NA and TSAP-IDs. For any implementation, these limits can be specified by the user at system generation time via the transport configuration parameters.

With one exception, the NA and TSAP-IDs are transparent to the iTS. When all bytes of the NA = 0 or all bytes of the TSAP-ID = 0, the NA or TSAP-ID is called *unspecified*. Otherwise, the NA or TSAP-ID is called *fully specified*.

A complete TA (NA plus TSAP-ID) is designated as follows:

1. *Unspecified* if both the NA and TSAP-ID are unspecified.
2. *Partially specified* if the NA is unspecified but the TSAP-ID is fully specified.
3. *Fully specified* if both the NA and TSAP-ID are fully specified.

Although the network addresses are transparent to the transport, the specific underlying network layer used by iNA 960 imposes restrictions on NA formats.

5.2.3 Contiguous Buffers

With the virtual circuit service, the transport protocol allows the optional transfer of a small amount (32 or 64 bytes) of user data during the connection establishment and disestablishment phases. If the user wishes to transfer data during these phases, the iTS interface requires that a single contiguous buffer block be allocated in user memory to send or receive the data. The lengths of the buffers depend on the command issued. The user refers to the start of the buffer via a pointer or iRMX segment token.

5.2.4 Noncontiguous Buffers

For the virtual circuit data transfer (during the data transfer phase) or datagram data transfer, the user may allocate multiblock noncontiguous buffers to hold the data. The buffers are referenced by a user buffer descriptor block. This will be either a special memory area in the procedural interface or part of a Request Block (RB) in the RB interface. In either case, the descriptor block will have the format specified by the PL/M declaration of Figure 5-2.

The user may specify buffer locations via pointers or segment tokens. If tokens are used, the buffer must start on a paragraph boundary. For tokens, the user sets all the offset fields of the descriptor block to 0 and loads all base fields with the token for each contiguous memory segment block.

```
DECLARE     buffer$descriptor     STRUCTURE (
            num$blks              BYTE,
            blocks(num$blks)      STRUCTURE(
              offset              WORD,
              base                WORD,
              length              WORD ));
```

Figure 5-2. Buffer Descriptor Structure

5.2.5 Post Receive Buffer Policies

For Virtual Circuits

The iTS relies on the user to post all receive buffers for data received from a remote TS via the virtual circuit service. iTS permits posting a buffer available only for a specific connection. Data received under another connection cannot use the buffer.

The virtual circuit service supports both normal and expedited data services. Receive buffers for normal data are posted and maintained separately from receive buffers for expedited data.

Normal (nonexpedited) data is presented to iTS for transmission as arbitrarily long messages called TSDUs. When the data is received by the remote Transport Service (TS), it is passed to the receive buffers posted by the user (if available). If a buffer is filled before the end of the TSDU, it is returned to the user. When the end of the TSDU is buffered, the buffer is returned even if it is not filled. Such a return buffer is marked EOM (End of Message) to indicate the end of the TSDU to the user. Thus, iTS guarantees that no more than one TSDU is returned in a user's buffer.

For Datagrams

iTS relies on the user to post all receive buffers for data received from a remote TS via the datagram service. iTS permits posting a buffer available only to a specific TSAP. Only datagrams addressed to that TSAP can use that buffer for passing data.

The data in each datagram sent by the iTS is a self-contained entity. If the total datagram buffer space available for a TSAP is less than the length of the received datagram data, the datagram is discarded with no data buffered. Otherwise, the data is buffered. Data from one datagram can be buffered in one or more receive buffers posted. The buffer containing the last byte of data in the received datagram is marked "eom" to the user. The EOM buffer is returned when the last data of the datagram is buffered, even if space remains in the buffer. Thus, a returned buffer can contain data from at most one received datagram.

Datagram receive buffers are posted and maintained separately from virtual circuit receive buffers.

5.3 Request Block Interface Commands

This section gives a detailed description of all iTS commands and responses using the Request Block (RB) interface. As detailed previously in Section 2.2.3, the user issues an RB command by first allocating an RB memory segment. Then the user fills in the fixed format fields, fills in the opcode for the command, and formats the argument field according to the format prescribed for the command. Section 5.3.2 defines the formats of the argument fields for each iTS command. Four classes of iTS RB argument fields exist. These classes are referenced in Section 5.3.2 and are illustrated by Figures 5-3 to 5-6.

```
DECLARE   open$rb$args   STRUCTURE (
          reference      WORD );
```

Figure 5-3. Open RB Argument Fields

```
DECLARE   connection$request$rb$args   STRUCTURE(
          iso$readon$code              BYTE,
          reserved (4)                 BYTE,
          ack$delay$estimate           WORD,
          ta$buffer$offset             WORD,
          ta$buffer$base               WORD,
          persistence$count            WORD,
          abort$timeout                WORD,
          reference                    WORD,
          conn$class                   BYTE,
          negot$options                WORD,
          user$data$buffer$offset      WORD,
          user$data$buffer$base        WORD,
          user$data$len                BYTE );
```

Figure 5-4. Connection Request RB Argument Fields

```
DECLARE   standard$vc$rb$args   STRUCTURE(
          iso$reason$code       BYTE,
          reserved (15)         BYTE,
          reference             WORD,
          conn$class            BYTE,
          buf$len               WORD,
          num$blks              BYTE,
          block(num$blks)       STRUCTURE (
                                    offset    WORD,
                                    base      WORD,
                                    length    WORD ));
```

Figure 5-5. Standard VC RB Argument Fields

```
DECLARE   datagram$rb$args      STRUCTURE(
          reserved (4)          BYTE,
          ta$buffer$offset      WORD,
          ta$buffer$base        WORD,
          tsap$class            BYTE,
          buf$len               WORD,
          num$blks              BYTE,
          block(num$blks)       STRUCTURE (
                                    offset    WORD,
                                    base      WORD,
                                    length    WORD ));
```

Figure 5-6. Standard VC RB Argument Fields

After iTS processes the RB, it is returned to the response mailbox specified in a fixed format field of the RB. iTS fills the fixed format response code field of the RB with a code indicating the result of the command. Section 5.3.2 specifies in detail for each command the response codes and their meanings.

5.3.1 Command Description Conventions

The remainder of this section is divided into subsections, one for each iTS command. Each command is described as follows:

Command:	A short descriptive name for the command.
Subsystem:	The code the user must fill into the RB fixed format subsystem field.
Opcode:	The symbolic name for the operation code that the user must fill into the RB fixed format opcode field. The numerical equivalent is defined in a PL/M INCLUDE file.
RB Class:	One of the four classes of argument fields that applies to this command.
Input Arguments:	A list with description of arguments that the user must input to the RB before sending the RB to iTS. The argument names coincide with the ones in Figures 5-3, 5-4, 5-5, or 5-6 depending on the RB argument field class.
Output Arguments:	A list with descriptions of RB arguments that iTS sends back to the user. This includes a description of any returned buffers.
Function:	A description of the operation performed by the command.
Response Codes:	A list of symbolic response codes returned by iTS to the user in the fixed format response code field of the RB. The meaning of each response code for that command is described. Numeric equivalents for the response codes are defined in a PL/M INCLUDE file.

Several parameter default values or limits are implementation dependent. These implementation dependencies are resolved by the iTS system configuration procedure (see Section 5.5).

5.3.2 OPEN

Command

OPEN

Subsystem

40H (Virtual Circuit)

RB Class

Open (Figure 5-3)

Input Arguments

None

Output Arguments

Reference–(WORD) a value identifying the connection data base(CDB) allocated by this command.

Function

This is the first command that must be issued whenever a new virtual circuit (or connection) is opened. The use of an iTS virtual circuit or connection requires the allocation of a memory area called a Connection Data Base (CDB).

All CDBs reside in memory on the same board that contains the communications software. The maximum number, max_cdbs, of CDBs allowed is specified as a configuration parameter.

A CDB maintains the state of the connection. Via entries in the CDB, the iTS can keep track of the sequencing of send and receive data, maintain flow control status, and recover from unacknowledged data packets.

The iTS user uses the connection by referencing the CDB using a 16-bit number called a *connection endpoint identifier* or *reference*. The reference is returned to the user when the CDB is allocated by this command. The iTS returns the reference to the user in the reference field of the open RB. The user then refers to the connection in other iTS commands as an input argument by using the reference value obtained here.

The very first reference returned by the transport after system initialization is selected via a 16-bit random number generation scheme. Thereafter, new references returned are incremented by 1. When the 16-bit reference numbers overflow, the reference of zero is skipped.

Response Codes

ok$resp–The CDB was allocated and the reference returned.

no$resource–Could not allocate any more CDBs. The reference is returned as 0.

5.3.3 SEND CONNECT REQUEST

Command

SEND CONNECT REQUEST

Subsystem

40H (Virtual Circuit)

Opcode

send$conn$req

RB Class

Connection Request (Figure 5-4)

Input Arguments

ta$buffer$offset–(WORD) The offset of a pointer to a TA buffer. The TA buffer must be loaded with addressing information specifying end TAs of the connection. The TAs must be fully specified. The offset is set to 0 if an iRMX segment token points to the TA buffer. The length of the remote net address and local or remote TSAP-IDs must not exceed the limits specified in the system configuration or an addressing error occurs. Multiple connections can be requested from the same local TSAP, to the same remote TA, or between the same local TSAP/remote TA pairs.

ta$buffer$base–(WORD) The base of a pointer to or of a segment token for a TA buffer.

persistence count–(WORD) The number of times to retry an active connection attempt upon connection refusal before giving up.

If the value equals 0, use default (configuration dependent)
If the value equals 1 to 0FFFEH (number of retries)
If the value equals 0FFFFH, (do not give up)

abort$timeout– (WORD) Specifies the retransmission timeout given in units of 51 ms.

If the value equals 0 use default (configuration dependent)
If the value equals 1 to 0FFFEH, (timeout value)
If the value equals 0FFFFH, (do not timeout)

reference–(WORD) Identifies the CDB this request applies to.

conn$class–(BYTE) Set to 0.

negot$options–(WORD) Specifies class of service and options requested for negotiation on this connection (see Reference 1 listed in the Preface):

If the value set is 0, use the default options specified by the configuration parameter def_negot_options.

If the value set is not 0, the user can specify the following options:

Break word up into 3 nibbles (1 = least significant)

- Nibble 1
  If nibble if equals 0, use 7 bit sequential numbers
  If nibble 1 equals 2, use 31 bit sequential numbers
- Nibble 2
  If nibble 2 equals 4, for class four service
- Nibble 3:
  If nibble 3 equals 0, no expedited service but checksums are to be used.
  If nibble 3 equals 1, expedited service and checksums are to be used.
  If nibble 3 equals 2, no expedited service and no checksums are to be used.
  If nibble 3 equals 3, expedited service but no checksum are to be used during data transfer.

The most significant bit of the word must be set for user-specified (non-default) negotiation options.

user$data$buffer$offset–(WORD) The offset of a pointer to a contiguous 64-byte buffer.

user$data$buffer$base–(WORD) The base of a pointer to or of a segment token for a contiguous 64-byte buffer.

If both base and offset equal 0, no buffer is allocated and no transparent user data is sent with the request. If either base or offset is not 0, the buffer is assumed to be allocated. It must be loaded with 0 to 32 bytes of user data to be sent with the request.

user$data$len–(BYTE) Specifies the length of the user data.

- If the value specified is 0 to 32,
- If the value specified is greater than 32,an error will occur.

Output Arguments

ack$delay$estimate–(WORD) will always be returned with 0.

iso$reason$code–(BYTE)

- The iso$reason$code equals 82H if the connection negotiation failed. That is, the request was accepted by the remote TS, but the local TS aborted the connection because the options the remote TS negotiated were unacceptable.
- The iso$reason$code disconnects reason code if the connection was rejected by the remote TS.
- The iso$reason$code equals 0 otherwise.

user$data$buffer–(CONTIGUOUS BUFFER) If the connection attempt was successful and a user buffer was allocated, the RB will return in its user buffer any data contained in the connection confirmation received from the remote TS. If the connection attempt was rejected by the remote TS and the local TS gives up, the RB will return in its user buffer up to 64 bytes of any data contained in the disconnect request from the remote TS. The received data overwrites any data that was in the buffer used for the original connection request.

user$data$len–(BYTE) Set to the length of any data received in response to the connection request.

Function

This command actively requests a connection to a fully specified remote TA using specified ISO connection negotiation options. It is assumed that a local CDB was allocated and a reference was returned to the user as a result of a previous OPEN command. The reference returned previously is specified in the current command to request the connection using the corresponding allocated CDB.

The user can request a connection with or without transparent data.

The user can ask that transport request the connection a specified number of times in spite of a rejection by the remote transport service. This retry count is the persistence count that the user specifies in the RB. When the number of retries exceeds the count, the local TS gives up and indicates connection rejection to the user. However, persistence is not invoked regardless of the count if the ISO reason code returned in the remote TS's rejection TPDU is either 0 or 88H. Persistence is also not applied when the local user decides to close the connection while the transport is requesting the connection.

This RB is returned to the user either upon detection of an error, upon connection establishment, or upon rejection and the local TS giving up. Thus, the receipt of this RB by the user serves as a connection confirmation or failure indication to the user.

Response Codes

ok$resp–The request was accepted by the remote TS, and the connection is now established in the data transfer phase.

ok$closed$resp–The local user aborted the connection while the connection request was outstanding.

conn$reject–The connection attempt was rejected by the remote TS, and the local TS gave up after the persistence count expired.

negot$failed–The request was accepted by the remote TS, but the local TS aborted the connection because of a negotiation failure at the local end.

loc$timeout–The request was unanswered and the retransmission timer timed out, aborting the connection attempt.

illegal$req–Invalid negotiation options were specified, and the connection attempt was aborted.

buffertoolong–user$data$len was greater than 32, and the connection attempt was aborted.

illegal$address–Invalid local or remote TAs were specified, or the network address length exceeds max_net_addr_len, or the local or remote TSAP-ID exceeds max_tsap id_len.

dup$req–A request was already in progress for this reference or the connection was already established.

network$error–A network layer error exists at the transport network interface.

unknown$reference–The user-specified reference does not correspond to an allocated CDB.

5.3.4 AWAIT CONNECT REQUEST/TRAN

Command

AWAIT CONNECT REQUEST/TRAN

Subsystem

40H (Virtual Circuit)

Opcode

await$conn$req$tran

RB Class

Connection Request (Figure 5-4)

Input Arguments

ta$buffer$offset–(WORD) The offset of a pointer to a TA buffer. The TA buffer must be loaded with addressing information specifying the end TAs of the connection. The remote TA may be fully specified, partially specified, or unspecified. If the remote TA is partially specified or unspecified, the TSAP-ID or network address length in the buffer must still be fully specified to those lengths used in the network. The contents field is filled with zeros up to the length for unspecified TSAP-IDs or network addresses. This offset is set to 0 if the TA buffer is pointed to by an iRMX segment token. The lengths of the remote net address and local or remote TSAP-IDs must not exceed the limits specified in the system configuration or an addressing error occurs. Multiple connection requests can be accepted at the same local TSAP, from the same remote TA, or between the same local TSAP/remote TA pairs.

ta$buffer$base–(WORD) The base of a pointer to or of a segment token for a TA buffer.

abort$timeout–(WORD) Same as for SEND CONNECT REQUEST command.

reference–(WORD) Same as for SEND CONNECT REQUEST command.

conn$class–(BYTE) Set to 0.

negot$options–(WORD) Same as for SEND CONNECT REQUEST command.

user$data$buffer$offset–(WORD) The offset of a pointer to a contiguous 64-byte buffer.

user$data$buffer$base–(WORD) The base of a pointer to or of a segment token for a contiguous 64-byte buffer.

If both the base and the offset equal 0, then no buffer is allocated and no transparent user data is expected with incoming connection requests. For this CDB, the local TS will respond to incoming connection requests that are without user data.

If either the base or the offset does not equal 0, then the buffer is assumed to be allocated. The local iTS can accept user data with an incoming connection request.

If the iTS receives a connection request with user data and the request has compatible addressing and negotiation parameters, then the data will be passed to the user in this buffer when this RB is returned to the user.

Output Arguments

ack$delay$estimate–(WORD) Always 0.

iso$reason$code–(BYTE)

The iso$reason$code equals

- The ISO disconnect reason code if the connection was aborted by the remote TS during the connection establishment phase.

The iso$reason$code equals

- 0 otherwise.

ta$buffer–(ADDRESS BUFFER) Contains within the remote TA fields the address of the remote TS user with which the connection was established.

negot$options–(WORD) The agreed-upon negotiation options using the encoding defined for the SEND CONN REQ command.

user$data$buffer–(CONTIGUOUS BUFFER) If data is to be returned from an incoming connection request, the data is returned to the user in this buffer.

user$data$len–(BYTE) Set to the length of the user data returned.

Function

This command indicates that the iTS user is willing to consider incoming connection requests from a remote transport service. iTS itself will decide to accept or reject the request based only on addressing, negotiation option, and user data buffer availability information. The connection request is not passed to the iTS user for further consideration. The user thus pre-establishes its connection acceptance criteria with this command prior to any connection request received from a remote TS. This mechanism of pre-establishing the connection criteria is called *passive open.*

It is assumed that a local CDB was allocated and a reference was returned to the user as a result of a previous OPEN command. The reference returned previously is specified in the current command to await connection requests using the corresponding allocated CDB.

For this command, a remote TA is specified in either the fully specified, partially specified, or unspecified mode. The local TSAP-ID must be fully specified (not 0).

By a series of these commands, a user can await connection requests. For an incoming connection request, the CDBs listening for requests via these commands are scanned. A request is *matched* to a CDB if the request passes the following tests described below:

1. Address match tests.
2. Negotiation option tests.
3. User data buffer availability test.

For the address match test, an await connection request may be fully specified, partially specified, or unspecified.

- Fully specified (fully specified remote TA) means that only incoming connection requests from the exact remote TA specified will be considered.
- Partially specified (partially specified remote TA) means that a connection request from any remote NA, but only one specific TSAP at that NA, will be considered.
- Unspecified (unspecified remote TA) means that a connection request from any remote TA will be considered.

A connection request passes the address match test only if all of the following conditions exist:

- The await connection request command is issued prior to receipt of the connection request, whose TA satisfies the remaining four requirements.
- The lengths of the NA and TSAP-ID fields of the remote source TA in the incoming request equal the corresponding lengths specified in this command's TA buffer.
- The remote source TA in the incoming request matches the local user remote TA specified in this command's TA buffer.
- The length of the destination TSAP-ID in the incoming request equals the length of the local TSAP-ID defined in this command's TA buffer.
- The destination TSAP-ID in the incoming request matches the local TSAP-ID defined in this command's TA buffer.

The address match test is performed first for a received connection request. For multiple CDBs listening for connection requests, the matching attempt is done in the following order of decreasing precedence:

- For a CDB with a fully specified remote TA in the TA buffer of this command.
- For a CDB with a partially specified remote TA in the TA buffer of this command.
- For a CDB with an unspecified remote TA in the TA buffer of this command.

If the address match test fails for one CDB, then that CDB is skipped and the incoming connection request is checked against other CDBs.

If an address match is found, a check is next made for compatible negotiation options as defined by the ISO standard. For incompatible options, the connection request is not matched to the CDB specified in this command and is checked against other CDBs awaiting requests.

For compatible addresses and negotiation options, a check is made to see if the incoming request contains user data. If so, then the CDB must be expecting data as defined by this command (a nonzero pointer to the data buffer). If the CDB is not expecting data, the connection request with data is not matched to the CDB specified in this command and the incoming connection request with data is checked against other CDBs. If the CDB is expecting data, the incoming request is matched to it.

If the incoming request has no data, then it is matched to the CDB if it passes address match and negotiation option tests.

If the incoming request matches a CDB, then for passive open await connection requests the connection is immediately accepted for the request by the matched CDB. In this case, the RB is not returned until completion of the three-way handshake (see Reference 1 listed in the Preface) that establishes the connection. Thus, the return of this RB serves as a confirmation of connection establishment. Also, any user data received with the request is passed to the user on return of the RB.

However, if no awaiting CDB is found that matches the incoming connection request, then iTS will reject the connection request. In this case, the RB is not returned to the user. This permits iTS to await further connection requests that may be valid.

Figure 5-7 presents a flow-chart summarizing the connection request consideration policy of iTS. For the passive open AWAIT CONNECT REQUEST command in which the request is not passed to the user, the "No" path is taken at the decision point, "Pass Request to User?"

The user may rescind their willingness to listen for connection requests by issuing a CLOSE command with reference corresponding to the connection rescinded. This will delete the CDB and terminate use of the reference.

Response Codes

ok$resp–The request was accepted. The connection is now established in the data transfer phase.

ok$closed$resp–The local user withdrew its willingness to listen for remote connection requests.

loc$timeout–The request was accepted but transport timed out before completion of the three-way handshake. The connection is aborted.

rem$abort–The connection request was accepted by transport but the remote TS aborted the connection during the connection establishment phase.

illegal$req–The user specified invalid negotiation options. The connection attempt was aborted.

illegal$address–The user specified invalid TA address options or the local TSAP-ID or remote TA length exceeds the configuration limits.

dup$request–This is a duplicate connection request. That is the transport is already awaiting a remote request or the connection is already established.

network$error–A network layer error is reported at the transport/network interface.

unknown$reference–The CDB corresponding to this reference is not allocated.

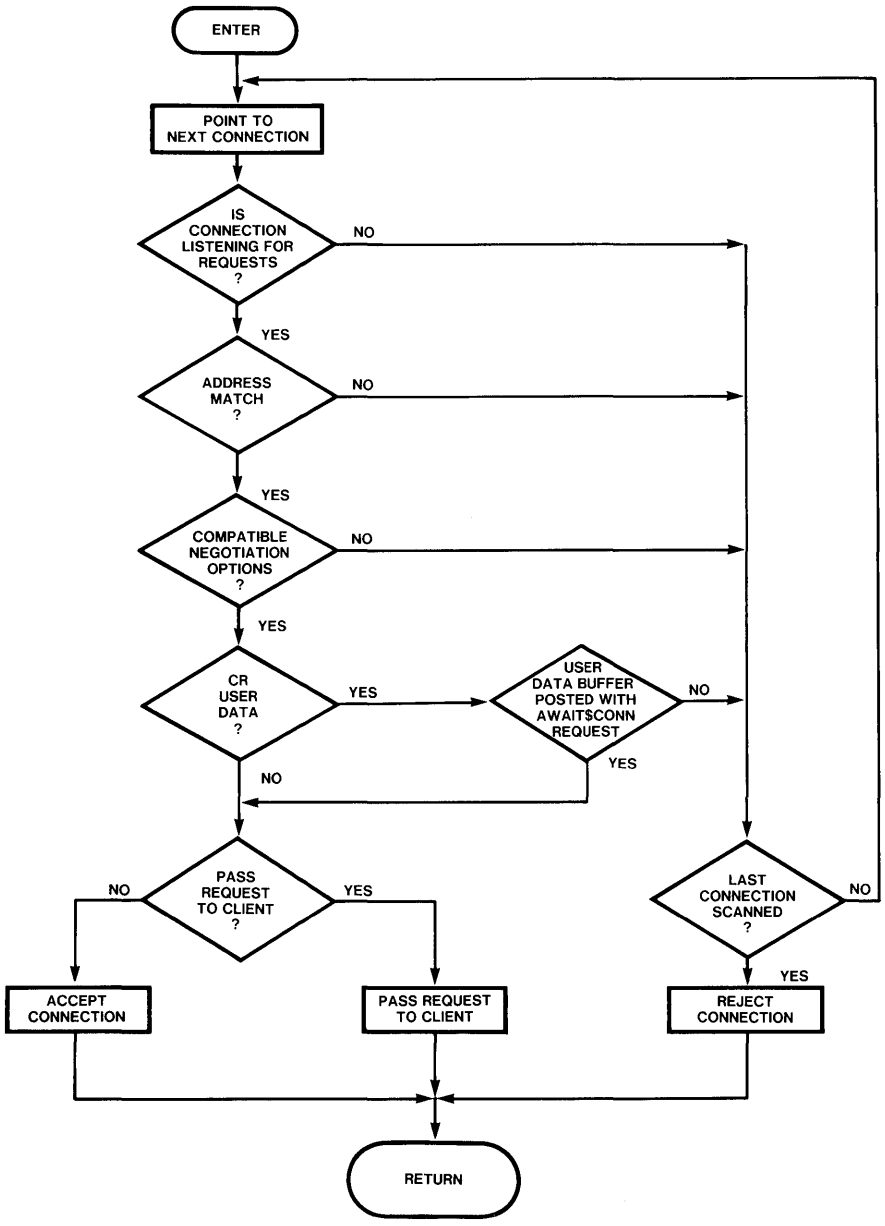

Figure 5-7. Connection Request Consideration Policy 122193-6

5.3.5 AWAIT CONNECT REQUEST/USER

Command

AWAIT CONNECT REQUEST/USER

Subsystem

40H (Virtual Circuit)

Opcode

await$conn$req$user

RB Class

Connection Request (Figure 5-4)

Input Arguments

ta$buffer$offset–(WORD) The offset of a pointer to a TA buffer. The TA buffer must be loaded with addressing information specifying the end TAs of the connection. The remote TA may be fully specified, partially specified, or unspecified. If the remote TA is either partially specified or unspecified, the TSAP-ID or network address length in the buffer must still be fully specified to those lengths used in the network. The contents field is filled with zeros up to the length for unspecified TSAP-IDs or network addresses. This offset is set to 0 if the TA buffer is pointed to by an iRMX segment token. The length of the remote net address and local or remote TSAP-IDs must not exceed the limits specified in the system configuration, or an addressing error occurs. Multiple connection requests can be accepted to the same local TSAP, from the same remote TA, or between the same local TSAP/remote TA pairs.

ta$buffer$base–(WORD) The base of a pointer to or of a segment token for a TA buffer.

abort$timeout–(WORD) Same as for SEND CONNECT REQUEST command.

reference–(WORD) Same as for SEND CONNECT REQUEST command.

conn$class–(BYTE) Set to 0.

negot$options–(WORD) Same as for SEND CONNECT REQUEST command.

user$data$buffer$offset–(WORD) The offset of a pointer to a contiguous 64-byte buffer.

user$data$buffer$base–(WORD) The base of a pointer to or of a segment token for a contiguous 64-byte buffer.

If both base and offset equal 0, then no buffer is allocated and no transparent user data is expected with incoming connection requests. The local iTS will respond for this CDB only to incoming connection requests without user data. If either the base or offset does not equal 0, then the buffer is assumed to be allocated. The local iTS can accept user data with an incoming connection request. If the iTS receives a connection request with user data and the request has compatible addressing and negotiation parameters, then the data will be passed to the user in this buffer when this RB is returned to the user.

Output Arguments

ack$delay$estimate–(WORD)

- = 0, always

iso$reason$code–(BYTE)

- = 0, always

ta$buffer–(ADDRESS BUFFER) Contains within the remote TA fields the address of the remote TS user with which the connection is being established. The local user can use this address information to partially base its decision to accept or reject the connection.

negot$options–(WORD) The agreed upon negotiation options using the encoding defined for the SEND CONNECT REQUEST command. This permits the user to base its decision to accept or reject the connection partly on the new options negotiated.

user$data$buffer–(CONTIGUOUS BUFFER) If data is to be returned from an incoming connection request, the data is returned in this buffer. The user can use this data to partially base its decision to accept or reject the connection.

user$data$len–(BYTE) Set to the length of the returned user data.

Function

This command indicates that the iTS user is willing to consider incoming connection requests from a remote transport service. If the request has compatible addressing and negotiation option information, the request is passed to the user for further consideration. iTS will wait for the user's reply as to whether the connection was accepted or rejected.

It is assumed that a local CDB was allocated and a reference was returned to the user as a result of a previous OPEN command. The reference returned previously is specified in the current command to await connection requests using the corresponding allocated CDB.

For this command, a remote TA is specified in either the fully specified, partially specified, or unspecified mode. The local TSAP-ID must be fully specified (not 0).

By a series of these commands, along with passive opens, a user can await connection request. For an incoming connection request, the CDBs listening for requests via this command or via passive opens are scanned. A request is considered matched to a CDB if the request passes the following tests:

1. Address match tests.
2. Negotiation option tests.
3. User data buffer availability test.

These tests are identical to those used for a passive open. However, for this command, a matched request is not accepted immediately by iTS. Instead, the connection request is passed to the user for further consideration by returning the RB with the addressing, negotiation option, and any user data information. iTS waits for the user's reply, which is one of the following:

- ACCEPT CONNECT REQUEST command to accept the connection.
- CLOSE command to reject the connection.

If no awaiting matches the incoming connection request, then iTS itself will immediately reject the connection request. In this case, the RB is not returned to the user. This permits iTS to await further connection requests that may be valid.

Figure 5-7 presents a flow chart summarizing the connection request consideration policy of iTS. For the user consideration AWAIT CONNECT REQUEST command in which the request is passed to the user, the "Yes" path is taken at the decision point "Pass Request to User?"

Response Codes

ok$decide$req$resp–The request is acceptable based on addressing, negotiation options and data buffer availability. The RB is being returned so that the user can decide whether to accept the connection. Transport awaits the user's response.

ok$closed$resp–The local user withdrew their willingness to listen for remote connection requests.

illegal$req–The user specified invalid negotiation options. The connection attempt was aborted.

illegal$address–The user specified an invalid TA, or the local TSAP-ID or remote TA length exceeds the configuration limits.

dup$request–This is a duplicate connection request. That is, transport is already awaiting a remote request or a local response from the user for this reference.

network$error–A network layer error is reported at the transport/network interface.

unknown$reference–The CDB corresponding to this reference is not allocated.

5.3.6 ACCEPT CONNECT REQUEST

Command

ACCEPT CONNECT REQUEST

Subsystem

40H (Virtual Circuit)

Opcode

accept$conn$req

RB Class

Connection Request (Figure 5-4)

Input Arguments

reference–(WORD) Same as for SEND CONNECT REQUEST command.

user$data$buffer$offset–(WORD) The offset of a pointer to a contiguous 64-byte buffer.

user$data$buffer$base–(WORD) The base of a pointer to or of a segment token for a contiguous 64-byte buffer. This buffer (if allocated) should be loaded with any user data that will be sent with the connection confirmation signalling the remote TS that its connection request has been accepted. Only a maximum of 32 bytes can be transmitted.

- If the offset or base equal 0, the buffer is not allocated. No data will be sent.
- If the offset or base do not equal 0, the buffer is allocated.

user$data$len–(BYTE) The length of the user data.

- If the length of the data is 0 to 32 bytes, no error occurs.
- If the length of the data is greater than 32 bytes, an error occurs.

Output Arguments

ack$delay$estimate–(WORD) Always 0.

iso$reason$code–(BYTE)

- = disconnect reason code if the connection was aborted by the remote TS during the connection establishment phase.
- = 0, otherwise.

ta$buffer–(ADDRESS BUFFER) Contains within the remote TA fields the address of the remote TS user with which the connection was accepted.

negot$options–(WORD) The final agreed upon negotiation options, using the encodings defined for the SEND CONNECT REQUEST command.

user$data$buffer–(CONTIGUOUS BUFFER) Up to 64 bytes of data received from the remote TS if the connection was aborted remotely during the connection establishment phase.

user$data$len–(BYTE) The length of any returned data.

Function

With this command, the user indicates their acceptance of a transport connection specified by the reference. This command is the positive response to a previously returned AWAIT CONNECT REQUEST/USER RB. The user returns in the current command an optional buffer of user data to be sent with the connection confirmation.

Response Codes

ok$resp–The connection just became established on completion of the three-way handshake.

ok$closed$resp–The local user aborted the connection before completion of the three-way handshake.

loc$timeout–The local iTS timed out before completion of the three-way handshake. The connection was aborted.

rem$abort–The remote TS aborted the connection before completion of the three-way handshake.

buffertoolong–More than 32 bytes of user data exist. The connection maintains its current state awaiting another local user response.

dup$request–This is a duplicate connection response. That is, the transport already accepted the connection or the connection is already established. This error can occur if this call is made for a connection for which a connection request was not previously returned to the user.

unknown$reference–The CDB corresponding to this reference is not allocated.

5.3.7 SEND DATA or SEND EOM DATA

Command

SEND DATA or SEND EOM DATA

Subsystem

40H (Virtual Circuit)

Opcode

send$data or send$eom$data

RB Class

Standard VC (Figure 5-5)

Input Arguments

reference–(WORD) Same as for SEND CONNECT REQUEST.

num$blks–(BYTE) The number of separate buffer blocks each block a contiguous memory area. This can be 0.

block(i).offset–(WORD) The offset of a pointer to the start of the i^{th} block.

block(i).base–(WORD) The base of a pointer to or of a segment token for the i^{th} block.

block(i).length–(WORD) The length of the data in the th block.

Output Arguments

iso$reason$code–(BYTE)

- = ISO disconnect reason code if the RB was returned due to a remote abort.
- = 0 otherwise.

Function

With this command, the user requests the transmission of the data in the buffers using the normal delivery service of the specified connection. The normal delivery service uses the regular ISO flow control mechanisms.

The SEND EOM DATA command signals that the end of the data marks the end of the Transport Service Data Unit (TSDU).

Any number of the blocks may have zero length; there may also be zero blocks. A send request with zero bytes of data is allowed. If it is SEND EOM DATA, then an EOM (in ISO called EOT) signal will be sent. If it is SEND DATA, the message is null, so no data will be sent. The RB will be returned an indeterminate amount of time later, but always after the previous send request is returned and before any subsequent send requests are returned.

The sum total length of all buffers pointed to by a single SEND$DATA or SEND$EOM$DATA RB is limited to a maximum value specified in Table 5-1. The 65K (1K = 1000 bytes) limits are imposed by the WORD SIZE of the buf$len field in the RB. The buffer size limitations for normal (7-bit) sequence number format ensure that the 7-bit (module 128) sequence number range is not exceeded for smaller max_tpdu-size_negotitated.

The SEND DATA request can be made any time after issuing the initial OPEN command. Thus, iTS accepts SEND DATA requests even if the connection has not yet entered the established state. When a connection enters the established state, iTS transmits the corresponding transmit buffers in the order in which they are queued. Since iTS always attempts to send full Transport Protocol Data Units (TPDUs), it copies information from these transmit buffers into the TPDU without concern for the buffer or block boundaries. However, iTS guarantees that an EOM not only indicates the end of a message, but also the end of a TPDU. In other words, iTS never copies information from more than one message into the same TPDU. The remote receive buffers sizes need not match the transmit buffer sizes; progress in delivering data will be made as long as there is any receive buffer space at the other node.

Response Codes

ok$resp–All the buffers in the request have been successfully transmitted and acknowledged by the remote TS.

ok$closed$resp–The local user aborted the connection and the queued RB is returned without transmitting its data.

loc$timeout–The local iTS timed out without receiving an acknowledgment for some of the data in the request.

rem$abort–The remote TS aborted the connection.

illegal$req–The connection was closing or was already closed.

no$resources–The CDB send queue is full. No more RBs can be queued until some already queued SEND RBs are returned.

unknown$reference–The CDB corresponding to this reference is not allocated.

Table 5-1. Maximum Total Buffer Lengths

max_tpdu_size negotiated	Sequence Number Format	
	7-Bit	31-Bit
7 (128 bytes)	8K*	65K*
8 (256 bytes)	16K	65K
9 (512 bytes)	32K	65K
10 (1024 bytes)	64K	65K
11 (2048 bytes)	65K	65K

* 1K = 1000 bytes

5.3.8 RECEIVE DATA

Command

RECEIVE DATA

Subsystem

40H (Virtual Circuit)

Opcode

receive$data

RB Class

Standard VC (Figure 5-5)

Input Arguments

reference–(WORD) Identifies the CDB for which the receive buffer is being posted.

conn$class–(BYTE) Set to 0.

num$blks–(BYTE) The number of separate buffer blocks, each block a contiguous memory area to buffer the data.

block(i).offset–(WORD) The offset of a pointer to the start of the i^{th} block.

block(i).base–(WORD) The base of a pointer to or of a segment token for the i^{th} block.

block(i).length–(WORD) The length of the i^{th} block.

Output Arguments

reference–(WORD) The reference is returned for the connection that used the buffer.

iso$reason$code–(BYTE) Same as for SEND DATA command.

buf$len–(WORD) The length of the data received in the buffers posted by this command.

block$buffer–(NONCONTIGUOUS BUFFER) Contains the data received from the remote TS.

Function

By using this command, the user posts receive buffers for a specific connection. The buffers are used to store data received using the transport normal delivery service. This service is governed by the regular transport flow control mechanisms.

The sum total length of all receive buffers pointed to by a single RECEIVE DATA RB must not exceed 65 K bytes.

Buffers may be posted prior to establishment of the connection.

Response Codes

ok$resp–All the buffers pointed to by the RB are filled with data and no EOM was signalled.

okeomresp–Transport signalled an EOM. The data in the buffers constitute the end of a TSDU.

ok$closed$resp–The local user aborted the connection or the connection was closing on a user request when the buffer was posted.

loc$timeout–The buffer was returned due to a connection timeout abort.

rem$abort–The remote TS aborted the connection.

no$resources–The CDB normal receive queue is full. No more normal receive buffers can be posted until some normal receive buffers already posted are returned.

unknown$reference–The reference does not correspond to any allocated CDB.

5.3.9 SEND EXPEDITED DATA

Command

SEND EXPEDITED DATA

Subsystem

40H (Virtual Circuit)

Opcode

sendexpdata

RB Class

Standard VC (Figure 5-5)

Input Arguments

reference–(WORD) Same as for SENDCONNREQ.

num$blks–(BYTE) This must equal 1, because 16 bytes is the largest buffer permitted to be sent.

block(0).offset–(WORD) The offset of a pointer to the start of the block.

block(0).base–(WORD) The base of a pointer to or of a segment token for the block.

block(0).length–(WORD) The length of the data in the block. The block length must be greater than 0 but cannot exceed 16.

Output Arguments

iso$reason$code–(BYTE)

- The iso$response$code equals the ISO disconnect reason code if the RB was returned due to a remote abort. (See Reference 1 listed in the Preface.)
- The iso$reason$code equals 0, otherwise.

Function

With this command, the user requests the transmission of up to 16 bytes of data in the buffer using the expedited delivery service of the specified connection.

With this service, the expedited data is guaranteed to arrive before any normal data submitted afterward.

Response Codes

ok$resp–The expedited data in the buffer was acknowledged.

ok$closed$resp–The local user aborted the connection.

loc$timeout–Transport timed out without receiving expedited acknowledgement of the data.

rem$abort–The remote TS aborted the connection.

illegal req–Either expedited data requested to be transmitted but the service is not available on this connection or the connection was closing or was already closed.

buffertoolong–A user data length greater than 16 was specified or num$blks greater than 1, aborting the transmission.

no$resources–The CDB expedited send queue is full. No more expedited send RBs can be queued at this time until some already queued expedited send RBs are returned.

buffertooshort–The buffer is empty. Either num$blks equals 0 or the block length equals 0.

unknown$reference–The specified reference does not correspond to an allocated CDB.

5.3.10 RECEIVE EXPEDITED DATA

Command

RECEIVE EXPEDITED DATA

Subsystem

40H (Virtual Circuit)

Opcode

receiveexpdata

RB Class

Standard VC (Figure 5-5)

Input Arguments

reference–(WORD) Identifies the connection data base for which the expedited receive buffer is being posted.

conn$class–(BYTE) Set to 0.

num$blks–(BYTE) Should be set to 1, because the data from the ED TPDU will be sent into the first block only.

block(0).offset–(WORD) The offset of a pointer to the start of the first block.

block(0).base–(WORD) The base of a pointer to or of a segment token for the first block.

block(0).length–(WORD) The length of the first block. The length of the first block must be at least 16 bytes, because the buffer must be able to hold data from the longest ED TPDU having a maximum of 16 bytes.

Output Arguments

reference–(WORD) Same as for RECEIVE DATA command.

conn$class–(BYTE) Same as for RECEIVE DATA command.

iso$reason$code–(BYTE) Same as for SEND DATA command.

buf$len–(WORD) The length of the expedited data received in the buffers posted by this command.

block$buffer–(NONCONTIGUOUS BUFFER) Contains the expedited data received from the remote TS.

Function

By using this command, the user posts expedited receive buffers for a specific connection. The buffers are used to store expedited data received using the transport expedited data delivery service. Expedited data bypasses the normal transport flow control mechanisms. (See Reference 1 listed in the Preface.)

Each receive buffer can buffer data from only one ED TPDU received. Data from two or more ED TPDUs are not combined into one buffer even if data were to fit.

The buffers for each request must be at least 16 bytes long to accommodate data in the largest ED TPDU that can be received.

Expedited receive buffers may be posted prior to establishment of the connection.

The queues of expedited receive buffers are maintained separately from the queues of normal receive buffers.

Response Codes

okeomresp–The buffer is returned with expedited data from a single received ED TPDU.

ok$closed$resp–The local user aborted the connection.

loc$timeout–The connection terminated on a timeout.

rem$abort–The remote TS aborted the connection.

illegal$req–Expedited service not available.

buffertooshort–The length of the first buffer block posted with the request less than 16.

no$resources–The CDB expedited receive queue is full. No more expedited receiver buffers can be posted until some that are posted are returned.

unknown$reference–The reference does not correspond to an allocated CDB.

5.3.11 CLOSE

Command

CLOSE

Subsystem

40H (Virtual Circuit)

Opcode

close$req

RB Class

Standard VC (Figure 5-5)

Input Arguments

reference–(WORD) Same as for SEND CONNECT REQUEST

iso$reason$code–(BYTE) The reason for the close using the ISO transport encodings given in the standard. (See Section 5.8.)

num$blks–(BYTE) The number of separate buffer blocks with each block being a contiguous memory area. The blocks contain optional data that can be sent with the disconnect request to close the connection. Set to 0 if no data is to be transmitted.

block(i).offset–(WORD) The offset of a pointer to the start of the i^{th} block.

block(i).base–(WORD) The base of a pointer to or of a segment token for the i^{th} block.

block(i).length–(WORD) The length of the data of the i^{th} block. The total length of data in all blocks cannot exceed 64.

Output Arguments

None

Function

With this command, the user requests the termination of an existing connection or the rejection of an incoming connection request.

If the connection is already established, then this call initiates the ISO transport connection disestablishment procedure. Any normal or expedited data queued up to be sent will not be sent. However, the user may request up to 64 bytes of data to be sent with the disconnect request. This data will be processed by the receiver if a receive buffer is posted with an AWAIT CLOSE command. If the receiver had previously issued an AWAIT CLOSE command, then any data received with the disconnect request will be passed to the buffer allocated with that command along with the ISO reason code. Otherwise, disconnect request data will be discarded.

If a connection request was previously passed to the user to decide whether to accept the connection, this call signals transport that the connection request should be rejected. Data passed with this command can be sent to the remote TS to explain the reason for the disconnection. The iso$reason$code should be set to 88H to indicate that the connection request was refused.

A CLOSE issued in response to a connection request or issued to abort an already established connection will delete the CDB. Any posted receive buffers (normal or expedited) or queued send requests (normal or expedited) will be returned to the user. An AWAIT CLOSE command will also be returned. The CLOSE RB will always be the final RB returned.

If the connection is aborted by a remote TS, then any posted receive buffers, queued send requests, or AWAIT CLOSE RBs will be returned to the local user; and the CDB will be deleted. If there are no queued RBs to report the remote abort, the CDB will not be deleted, but will be marked "closed." The user can send one more RB command (SEND, RECEIVE, STATUS, DEFERRED STATUS, or AWAIT$CLOSE) to determine the final status of the aborted connection. That RB will be returned with a rem$abort response code and the CDB will then be deleted. Any further requests on that connection will generate an unknown$reference error.

Response Codes

ok$closed$resp–For confirmed disconnection, disconnect collision, or if already closing or closed.

ok$reject$conn$resp–For rejection of a connection requested.

loc$timeout–If transport timed out without receiving a confirmation to its disconnect request.

buffertoolong–If a user data length greater than 64 was specified.

unknown$reference–If the reference does not correspond to an allocated CDB.

5.3.12 AWAIT CLOSE

Command

AWAIT CLOSE

Subsystem

40H (Virtual Circuit)

Opcode

await$close

RB Class

Standard VC (Figure 5-5)

Input Arguments

reference–(WORD) Same as for SEND CONNECT REQUEST.

num$blks–(BYTE) The number of separate buffer blocks, each block a contiguous memory area, to receive disconnect request user data. Normally, this is set to one because a maximum of 64 bytes of data can be received from disconnect request; 0 if no buffer is to be allocated. In this case, disconnect request user data will be ignored.

block(i).offset–(WORD) The offset of a pointer to the start of the i^{th} block.

block(i).base–(WORD) The base of a pointer to or of a segment token for the i^{th} block.

block(i).length–(WORD) The length of the i^{th} block.

Output Arguments

reference–(WORD) The reference for the connection that was deleted.

iso$reason$code–(BYTE)

- = ISO reason code received in the disconnect request. (See Section 5.8.)

buf$len–(WORD) The length of any user data received with the disconnect request which closed the connection.

block$buffer–(NONCONTIGUOUS BUFFER) Contains the disconnect request user data received from the remote aborting TS.

Function

With this command, the user requests to be notified that a specified connection has terminated.

A buffer (if allocated) can be returned to the user filled with data from a received disconnect request that may explain the cause of the disconnection. The user needs only to allocate a maximum of 64 bytes of buffer space to accommodate the longest

disconnect message permitted by the ISO standard. If the buffer length allocated is less than the received data length, then only the data that fits in the buffer is returned. The rest is lost. In particular, if no buffer is allocated, any disconnect data received is discarded.

An ISO reason code is also returned to indicate the cause of the disconnection.

Response Codes

ok$closed$resp–The local user had aborted the connection or connection already closed.

loc$timeout–The local transport service had timed out.

rem$abort–A disconnect request was received from the remote TS on the specified connection.

dup$request–Another AWAIT CLOSE command was posted previously on this connection.

unknown$reference–The reference does not correspond to an allocated CDB.

5.3.13 SEND DATAGRAM

Command

SEND DATAGRAM

Subsystem

41H (Datagram)

Opcode

send$datagram

RB Class

datagram (Figure 5-6)

Input Arguments

ta$buffer$offset–(WORD) The offset of a pointer to a TA buffer. The TA buffer must be loaded with addressing information specifying the source (local).

TSAP-ID and destination (remote) TA of the datagram. The TAs must be fully specified. The remote NA can be multicase address. The lengths of the remote net address and local or remote TSAP-IDs must not exceed the limits specified in the system configuration or an addressing error occurs.

ta$buffer$base–(WORD) The base of a pointer to or of a segment token for a TA buffer.

num$blks–(BYTE) The number of separate buffer blocks, each block a contiguous memory area containing datagram data to be sent.

block(i).offset–(WORD) The offset of a pointer to the start of the i^{th} block.

block(i).base–(WORD) The base of a pointer to or of a segment token for the i^{th} block.

block(i).length–(WORD) The length of the data in the i^{th} block. The total length of all data over all blocks must be less than or equal to the maximum NSDU length provided by the network layer (minus a small overhead for the transport datagram header). For an *IEEE 802.3* network layer and a 2-byte TSAP, the maximum length is 1486 bytes.

Output Arguments

None

Function

With this command, the user requests transmission of the data in the buffers using the transport datagram service. This service is connectionless and gives no assurance of delivery of the data. Data can be lost or misordered.

Transport datagram service does not provide a fragmentation/reassembly capability. Therefore, the length of the data cannot exceed the maximum NSDU size provided by the network layer. For an *IEEE 802.3* network layer and a 2-byte TSAP, the maximum length is 1486 bytes.

The destination transport address can be either a single station, multicast, or broadcast network address. The multicast or broadcast network address conventions are transparent to the transport layer. They are dependent on the underlying network service used.

Response Codes

ok$resp–The data has been queued for transmission by the network layer.

buffertoolong–The data length exceeds the maximum NSDU size.

illegal$address–The local TSAP-ID or remote TA exceeds the configuration limits.

5.3.14 RECEIVE DATAGRAM

Command

RECEIVE DATAGRAM

Subsystem

41H (datagram)

Opcode

receive$datagram

RB Class

datagram (Figure 5-6)

Input Arguments

ta$buffer$offset–(WORD) The offset of a pointer to a TA buffer. The remote NA and TSAP-ID fields are not input parameters. However, the fields must be reserved to the proper length to buffer the source TA of a received datagram.

The local TSAP-ID must be loaded into the buffer. The length of the local TSAP-ID must not exceed the limit specified in the system configuration, else an addressing error occurs. The local TSAP-ID which must be nonzero specifies the TSAP that posts the buffer. This posts the buffer on a queue reserved only for that TSAP-ID. Any received datagram with remote TSAP-ID matching the TSAP-ID of this queue can pass its data to the buffer. Datagrams with non-matching remote TSAP-IDs cannot use this buffer.

ta$buffer$base–(WORD) The base of a pointer to or of a segment token for a TA buffer.

tsap$class–(BYTE) Set to 0.

num$blks–(BYTE) The number of separate buffer blocks, each block a contiguous memory area to receive datagram.

data.block(i).offset–(WORD) The offset of a pointer to the start of the i^{th} block.

block(i).base–(WORD) The base of a pointer to or of a segment token for the i^{th} block.

block(i).length–(WORD) The length of the i^{th} block.

Output Arguments

ta$buffer–(ADDRESS BUFFER) Contains in the remote NA and TSAP-ID fields the remote source address of the received datagram.

buf$len–(WORD) The length of the data received in the buffers posted by the command.

block$buffer–(NONCONTIGUOUS BUFFER) Contains the datagram data received from the remote TS.

Function

By using this command, the user posts a receive buffer on behalf of a TSAP to receive data from a transport datagram. The datagram buffer queues are maintained separately from the virtual circuit buffer queues.

Response Codes

ok$resp–The buffers pointed to by the RB are completely filled with data.

okeomresp–Buffer contains data to the end of the datagram. An RB can return data from at most one transport datagram.

illegal$address–An addressing error detected.

no$resources–No more resources to manage the buffers posted for the TSAP.

5.4 Procedure Interface Commands

The Transport Service procedural interface allows the user to invoke the transport commands via a set of procedure calls with parameters. The procedure calls are iRMX OS extensions. They provide a friendlier command interface to the iRMX user than the RB command interface.

These procedure calls will create the request blocks defined from the user's procedure parameter list. As a result, iTS will return the RB to the user's response mailbox when iTS has processed the command. Thus the *response* interface is the same whether the procedural or RB *command* interface is used.

5.4.1 Procedural Call Description Conventions

Previously, Section 5.3.2 gives a detailed description of the procedure name and call arguments corresponding to each of the iTS interface commands defined in Section 5.3. Because the procedural interface does not add any new functionality and the response interface is the same as the RB interface, no command function and response descriptions appear here. These are found under the headings "Output Arguments," "Function," and "Response Codes" for each command.

The remainder of this section is divided into subsections, one for each iTS command. Each command is described as follows:

COMMAND:	Same name given for the RB interface.
PROCEDURE:	The name and parameter list of the interface procedure.
PARAMETERS:	Definition of the parameters of the procedure. Those parameters that are identical in name with the input argument in the RB interface are used the same way as in the RB interface. Only those parameters that are different are described here.

Each procedure also returns an exception code that usually indicates the status of parsing the parameter list for syntax. See *Getting Started with the Release 5 iRMX™ 86 System* for a description of the exception codes.

5.4.2 OPEN

Command

OPEN

Procedure

```
CQ$TLVC$OPEN (user$id, response$mbx, except$ptr)
```

Parameters

user$id*–(WORD) user ID.

response$mbx*–(WORD) iRMX response mailbox token

except$ptr*–pointer to a word containing exception codes.

These parameters are found in every procedure call. They are not described further.

5.4.3 SEND CONNECT REQUEST

Command

SEND CONNECT REQUEST

Procedure

```
,CQ$TLVC$SEND$CONN$REQ(ta$buffer$ptr, persistence$count,
abort$timeout, reference, conn$class, negot$options,
user$data$buffer$token, user$data$len, user$id,
response$mbx, except$ptr)
```

Parameters

ta$buffer$ptr–(POINTER) A pointer to a TA buffer.

user$data$buffer$token–(WORD) An iRMX segment token for the connection request user data buffer. Set to 0 only if no user data buffer is allocated. The user data buffer must be 64 bytes long if allocated.

5.4.4 AWAIT CONNECT REQUEST TRAN

Command

AWAIT CONNECT REQUEST TRAN

Procedure

```
CQ$TLVC$AWAIT$CONN$REQ$TRAN(ta$buffer$ptr,
persistence$count, abort$timeout, reference, conn$class,
negot$options, user$data$buffer$token, user$data$len,
user$id, response$mbx, except $ptr)
```

Parameters

ta$buffer$ptr–(POINTER) A pointer to the TA buffer.

persistence$count–(WORD) Set to 0.

user$data$buffer$token–(WORD) Same as for SEND CONNECT REQUEST.

user$data$len–(BYTE) Set to 0.

5.4.5 AWAIT CONNECT REQUEST USER

Command

AWAIT CONNECT REQUEST USER

Procedure

```
CQ$TLVC$AWAIT$CONN$REQ$USER(ta$buffer$ptr,
persistence$count, abort$timeout, reference, conn$class,
negot$options, user$data$buffer$token, user$data$len,
user$id, response$mbx, except$ptr)
```

Parameters

ta$buffer$ptr–(POINTER) A pointer to a TA buffer.

persistence$count–(WORD) Set to 0.

user$data$buffer$token–(WORD) Same as for SEND CONNECT REQUEST.

user$data$len–(BYTE) Set to 0.

5.4.6 ACCEPT CONNECT REQUEST

Command

ACCEPT CONNECT REQUEST

Procedure

```
CQ$TLVC$ACCEPT$CONN$REQ(reference, negot$options,
user$data$buffer$token, user$data$len, user$id,
response$mbx, except$ptr)
```

Parameters

user$data$buffer$token–(WORD) Same as for SEND CONNECT REQUEST.

5.4.7 SEND DATA or SEND EOM DATA

Command

SEND DATA or SEND EOM DATA

Procedure

```
CQ$TLVC$SEND$DATA(reference, buffer$descr$token, user$id,
response$mbx, except$ptr)

CQ$TLVC$SEND$EOM$DATA(reference, buffer$descr$token,
user$id, response$mbx, except$ptr)
```

Parameters

buffer$descr$token–(WORD) An iRMX segment token for a buffer descriptor memory area formatted as shown in Figure 5-2. Prior to the call, the descriptor blocks must be filled in and the buffers pointed to must contain the data to be transmitted.

5.4.8 RECEIVE DATA

Command

RECEIVE DATA

Procedure

```
CQ$TLVC$RECEIVE$DATA(reference, conn$class,
buffer$descr$token, user$id, response$mbx, except$ptr)
```

Parameters

buffer$descr$token–(WORD) An iRMX segment token for a buffer descriptor memory area formatted as shown in Figure 5-2. Prior to the call, the descriptor block must be filled in.

5.4.9 SEND EXPEDITED DATA

Command

SEND EXPEDITED DATA

Procedure

```
CQ$TLVC$SEND$EXP$DATA(reference, user$data$buffer$token,
user$data$len, user$id, response$mbx, except$ptr)
```

Parameters

user$data$buffer$token–(WORD) An iRMX segment token for the expedited data buffer. The procedural interface requires one contiguous buffer (max of 16 bytes). The buffer should be filled with the data to be transmitted prior to the call.

user$data$len–(BYTE) The length of the data in the buffer (must be less than or equal to 16).

5.4.10 RECEIVE EXPEDITED DATA

Command

RECEIVE EXPEDITED DATA

Procedure

```
CQ$TLVC$RECEIVE$EXP$DATA(reference, conn$class,
user$data$buffer$token, user$data$len, user$id,
response$mbx, except$ptr)
```

Parameters

user$data$buffer$token–(WORD) An iRMX segment token for the expedited data receive buffer. The procedural interface requires one contiguous buffer of length exactly 16 bytes.

user$data$len–(BYTE) Set to 0.

5.4.11 CLOSE

Command

CLOSE

Procedure

```
CQ$TLVC$CLOSE(reference, iso$reason$code,
user$data$buffer$token, user$data$len, user$id,
response$mbx, except$ptr)
```

Parameters

user$data$buffer$token–(WORD) An iRMX segment token for an optional disconnect user data buffer. This must be a contiguous buffer of length less than or equal to 64. Set to 0 only if no buffer is allocated for no data with the disconnect request.

user$data$len–(BYTE) The length of the data in the buffer to be sent (must be less than or equal to 64).

5.4.12 AWAIT CLOSE

Command

AWAIT CLOSE

Procedure

```
CQ$TLVC$AWAIT$CLOSE(reference, user$data$buffer$token,
user$data$len,user$id, response$mbx, except$ptr)
```

Parameters

user$data$buffer$token–(WORD) An iRMX segment token for an optional disconnect user data buffer. This must be a contiguous buffer of length less than or equal to 64. Set to 0 only if no buffer is to be allocated for no data reception on receiving a disconnect request.

user$data$len–(BYTE) Set to 0.

5.4.13 SEND DATAGRAM

Command

SEND DATAGRAM

Procedure

```
CQ$TLDG$SEND$DATAGRAM(ta$buffer$ptr, buffer$descr$token,
user$id, response$mbx, except$ptr)
```

Parameters

ta$buffer$ptr–(POINTER) A pointer to a TA buffer.

buffer$descr$token–(WORD) Same as for SEND$DATA. The length of the data transmitted cannot exceed the maximum NSDU length provided by the underlying network service.

5.4.14 RECEIVE DATAGRAM

Command

RECEIVE DATAGRAM

Procedure

```
CQ$TLDG$RECEIVE$DATAGRAM(ta$buffer$ptr, tsap$class,
buffer$descr$token, user$id, response$mbx,
except$ptr)
```

Parameters

ta$buffer$ptr–(POINTER) A pointer to a TA buffer.

buffer$descr$token–(WORD) Same as for RECEIVE $DATA.

5.5 Configuration

To configure the Transport Layer configuration file, perform the following steps:

1. Customize the Transport Layer to a particular implementation in the configuration file.
2. Decide on which transport layer services are to be provided by an implementation and link the appropriate transport service modules at system generation time.
3. For iRMX users decide whether to use the iRMX Transport Layer procedural interface and link the transport layer procedural library with the application job.

5.5.1 Customizing the Transport Layer

The iTS has several configuration parameters that customize the transport layer to a particular implementation. The seven classes of configuration parameters are as follows:

1. Transport Address Limits
2. Network Layer Interface Parameters
3. Transport Data Base Parameters
4. Client Request Default Parameters
5. Internal Negotiation Option Parameters
6. Retransmission Timer Parameters
7. Flow Control Parameters

The parameters in each class are defined in the following subsections. An assembly language configuration template is also specified. The configuration template models a typical iRMX subsystem configuration file that will be provided with the iTS software package. The configuration file contains default values that can be modified by a system implementor to customize the transport layer. The file is assembled and the resulting code is used to initialize the appropriate internal transport software variables.

Transport Address Limits

These parameters define the maximum lengths of the network address and TSAP-ID components of the transport address.

Specifically, the parameters are as follows:

- max_net_addr_len = Maximum network address length. Must be set to 0CH.
- max_tsap_id_len = Maximum length of local or remote transport service access point identifiers (TSAP-IDs).
- max_nsap_id_len = Leave at default value of 2

Network Layer Interface Parameters

The only parameter in this class is the following:

- loc_nsap id = The local network service access point (NSAP) - Id required to interface to the underlying network layer. Leave at default value of 1.

Transport Data Base Parameters

iTS requires a certain amount of RAM memory to maintain certain data structures used for its operation. As such, limits exist on the size or number of these data structures. These limits are specified by the following parameters:

- max_cdbs = Maximum number of CDBs used to manage virtual circuits. Increase this parameter value to accommodate more concurrent connections. Each new connection requires about 200 bytes additional memory.
- max_conn_class = Maximum connection classes used to share receive buffers among connections. Leave at 0.
- max_datagram_tsaps = Maximum number of TSAPs that can have datagram receive buffers posted at one time. Set to 0 if datagram service is not provided.
- max_tsap class = Maximum number of datagram TSAP classes used to share receive buffers among datagram TSAPs. Leave at 0.
- dg xsum_opt = Checksum Option Flag = 0 if checksums not used. 0FFH if checksums are used.
- max_irbs = Maximum number of internal request blocks used to queue connection related TPDUs for transmission. Leave at default value of 5.
- max_eirbs = Maximum number of internal request blocks used to queue connection related expedited TPDUs for transmission. Leave at default value of 2.
- max_lirbs = Maximum number of long internal request blocks used to queue non-connection-related TPDUs for transmission. Leave at default value of 3.
- max_dirbs = Maximum number of internal request blocks used to queue datagram TPDUs for transmission. Leave at default value of 2.

Client Request Default Parameters

In the RB interface the user can specify that iTS use certain default values. The default values are the following configuration parameters:

- def_persis = Default persistence count to specify the default number of retries to request a connection that is being rejected before giving up.
- def_abort_to = The default abort timeout to specify the default length of time (in 51-millisecond units) to retransmit before timing out and aborting a connection due to a failure to acknowledge from the remote TS.
- def_negot_options = The default connection negotiation options specified via the encoding. The default makes a choice for each of the following options.
  - Sequence number format (7- or 31-bit).
  - Class of Service (only value of 4 supported).
  - Expedited Service/Checksum Options (Yes/No Combination).

Internal Negotiation Options

These concern options negotiated by the transport protocol during connection establishment, but are not specified by the iTS user. The parameters are the following:

- max_tpdu_size = Maximum TPDU size.
- no_addit_opt_field = Additional option field used as an assumed additional option parameter that a remote TS requested, when, in fact, the request provided no such option parameter.
- no_max_tpdu_size_field = Maximum TPDU size field used as an assumed maximum TPDU size that a remote TS requested, when, in fact, the request provided no size.

Retransmission Timer Parameters

These parameters specify the characteristics of the iTS retransmission timer algorithm. These are used to fine tune the performance of the algorithm. The parameters are:

- def_retran_to = Default retransmission timeout. This is the initial retransmission timeout value used to start the retransmission algorithm for each connection. This retransmission timeout sends Connection Requests (CR) or conn.confirm (CC) TPDUs. If CRs are to be passed to the user, the timeout should allow for the user to process requests. Specified in 100 microsecond units.
- min_retran_time = Minimum retransmission time. This is the lower bound on the retransmission time computed by the algorithm to prevent excessive retransmissions. Specified in 100 microsecond units. An inappropriate value can adversely affect performance. Too small a value will cause excessive retransmissions, creating excessive network traffic and excessive CPU overhead to send and receive retransmitted TPDUs. Too large a value will waste time waiting to retransmit (if a retransmission is required). The default value provided gives adequate performance for several Intel data-communications processor boards and for *IEEE 802.3* local area networks. This parameter need only be retuned if performance degradation is observed in other hardware or network configurations.
- closing_abort_to = Closing abort timeout. This is the total amount of time to keep trying to send a disconnect request in order to receive a disconnect confirmation before closing the connection. Specified in 51 millisecond units. This parameter normally does not have to be changed. The value of this parameter could result in a delay (given by this time out in time units) to close a connection only if the remote TS has already disconnected. Otherwise, this parameter has no effect.
- inactivity ak_retran_to = Inactivity AK retransmission timeout. This is the interval between flow control window AK TPDU transmissions when there is no other activity on a connection. Specified in 100 microsecond units. Leave this parameter at the default value.
- max_inactivity count = Maximum inactivity count. This is the number of consecutive flow control window AK TPDUs that are sent without receiving any responses from the remote TS on a connection before considering the remote TS as disconnected and aborting the connection. The product max_inactivity_count and inactivity_ak retran_to parameters (converted to time) is the total length of time allowed by the transport without receiving TPDUs from the remote TS before assuming the connection is inactive and disconnected. Increase this count to increase effective inactivity timeout. Decrease this count to decrease the timeout.

Flow Control Parameters

These parameters specify the characteristics of the transport protocol flow control algorithm. The parameters are as follows:

- max_window_size_normal = Maximum window size for normal (7-bit) sequence number format. The largest receive buffer credit that can be reported on a connection by the iTS to a remote TS for normal sequence number format. The maximum value for this parameter is 0FH, limited by the ISO 8073 Transport protocol. Do not use a larger value. This number actually affects transport layer performance. The value must be tuned as a function of the network layer buffering capacity allocated in the BUFCFG.A86 macro.

  In BUFCFG.A86, the internal network buffering resources are specified. This limits the number of transport packets (called (TPDUs) that can be buffered at one time. For reasonable performance, the parameter max_window_size_normal value should be less than or equal to the maximum number of TPDUs that can be internally buffered, plus 3. For example, if 12 TPDUs can be simultaneously buffered by the network layer, then a reasonable value for this parameter is 15 (decimal).

- max window_size_extended = Maximum window size for extended (31-bit) sequence number format. The largest receive buffer credit that can be reported on a connection by the Transport service to a remote TS for extended sequence number format.

  The value can be from 0 to 65K. The same performance considerations are applied to this parameter as for the parameter max_window_size_normal. On any one connection either max_window_size_normal is used (if 7-bit sequence numbers are negotiated) or max_window_size_extended is used (for 31-bit sequence number negotiation).

- min_credit = Minimum credit. The smallest receive buffer credit that can be reported on a connection by the iTS to a remote TS. Set to 0 if the window can close. Set to 1 if the window never closes. If this parameter is set to 0 to close the window, all the normal ISO 8073 flow control mechanisms will be invoked to open the credit window to prevent deadlock. If the value of min_credit equals 1, and the window is never reported as closed by the receiver (even if no buffers are available), these flow control window mechanisms never come into effect. The default (min credit = 1) actually enhances performance, because the *ISO 8073* open-window mechanisms incur substantial CPU overhead. The overhead can outweigh the performance improvements that might occur by a closed window preventing unnecessary retransmissions (to a receiver with no buffer).

- open_window_to = Open window timeout. This is the interval between successive acknowledgements (AK TPDUs) that announce the opening of a previously closed credit window used to avoid flow control deadlock. Specified in 100 microsecond units. This parameter is not used if min credit equals 1.

- max_open_window_count = Maximum open window count. This is the maximum number of open window AKs that are transmitted before sender assumes that the remote TS has received the open window credit information. When this count is reached, the iTS stops transmitting the open window AKs. This parameter is not used if min_credit equals 1.

5.5.2 Configuration Template

The assembly MACROS required the parameters defined in the above sections. The INTEL supplied defaults are specified here.

```
NAME TCONF

$INCLUDE (TL/TCONF.MAC)
;
; Transport Address Limits
;
%Transport_Address_Limits(max net_addren,max tsap_id_len,max_nsap_id_len)
%'max_net_add_len=12
%'max_tsap id_len=2
%'max_nsap_id_len=2
;
; Network Layer Interface
;
%Network_Layer_Interface(loc_nsap id)
%'loc_nsap_id=1
;
; Transport Data Bases
;
```

```
%Connection_limits(max_cdbs,max_conn_class)
%'max cdbs=21
%'max_conn_class=2
%Datagram_structure(max datagram_tsaps,max tsap class, dg xsum_opt)
%'max_datagram_tsaps=9
%'maqx tsap class=2
%'dg xsum_opt=0
%internal_request_blocks(max irbs,max_eirbs, maxi_lirbs,max dirbs)
%'max_irbs=5
%'max eirbs=2
%'max_lirbs=3
%'max dirbs=2
;
; Client Request Defaults
;
%client_request_defaults(def_persist, def_abort_to, def_negot_options)
%'def persist=1
%'def_abort_to=1800H (about 5 min)
%'def_negot_options=8242H
%'           31 bit seq.no
%'           Class 4
%'           No expedited service/No checksums
;
; Internal Negotiation Options
;
%internal_negot_options (max_tpdu size, no_addit_opt_field,
              no_max_tpdu_size_field)
%'max_tpdu_size=0BH (2048 bytes)
%'no_addit_opt_field=3 (expedited services and no checksum)
%'no_max_tpdu field=7 (128 bytes)
;
; Retransmission Timer Parameters
;
%retran_timer(def_retran_to,min_retran_time,)
%'def_retran_to=10,000 (1 second)
%'min_retran_time=1000 (100 milliseconds)
%closing_abort(closing_abort_to)
%'closing_abort_to=80H (7 seconds)
%inactivity_timer(inactivity_ak_retran_to,max_inactivity count)
%'inactivity_ak_retran_to=300,000 (30 seconds)
%'max_inactivity count=8

;
; Flow Control
;
%window_size (max_window_size_normal, max_window_size_extended, min_credit)
%'max_window_size_normal=0FH
%'max_window_size_extended=0FH
%'min credit=1
%open_window_timer(open_window_to,max_open_window_count)
%'open_window_to=10,000 (1 second)
%'max_open_window_count=8
```

5.5.3 Configuring Transport Services

The following sets of transport services can be configured at system generation time:

- No transport services. The transport configuration file is not used.
- Only the normal virtual circuit service. No expedited service and no datagram service.
- Both the normal and expedited virtual circuit service. No datagram service. The expedited service is meaningful only if normal service is also provided.
- Only the datagram service. No normal or expedited virtual circuit service.
- All transport services including normal virtual circuit, expedited virtual circuit, and datagram services.

5.5.4 iRMX™ Procedural Interface Configuration

For iRMX applications using the iRMX Transport Layer procedural interface, one of the following library modules must be linked into the application job:

- CQTLC.LIB: if the user program is compiled in COMPACT mode.
- CQTLL.LIB: if the usr program is compiled in MEDIUM or LARGE mode.

The user's application programs can make use of the following INCLUDE file to specify the external references to the iRMX Transport interface procedures:

- CQTL.EXT

5.6 Op Code/Response Code Include File

```
Declare                                 /*Opcodes*/

        open$req                          LITERALLY       '0',
        send$conn$req                     LITERALLY       '1',
        await$conn$req$tran               LITERALLY       '2',
        await$conn$req$user               LITERALLY       '3',
        accept$conn$req                   LITERALLY       '4',
        send$data                         LITERALLY       '5',
        send$eom$data                     LITERALLY       '6',
        receive$data                      LITERALLY       '7',
        withdraw$rcv$buf                  LITERALLY       '8',
        send$exp$data                     LITERALLY       '9',
        receive$exp$data                  LITERALLY       '10',
        withdraw$exp$buf                  LITERALLY       '11',
        close$req                         LITERALLY       '12',
        await$close                       LITERALLY       '13',
        status$req                        LITERALLY       '14',
        def$status                        LITERALLY       '15',
        reserved                          LITERALLY       '16',
        send$datagram                     LITERALLY       '17',
        receive$datagram                  LITERALLY       '18',
        withdraw$dg$buf                   LITERALLY       '19';

Declare                               /*Max opcode limits*/

        vc$req$max                        LITERALLY       '15',
        first$dg$cmd                      LITERALLY       '17',
        req$max                           LITERALLY       '19';
```

```
Declare                          /*Non-error Response Codes*/

        ok$resp                          LITERALLY        '1',
        ok$eom$resp                      LITERALLY        '3',
        ok$decide$req$resp               LITERALLY        '5',
        ok$closed$resp                   LITERALLY        '7',
        ok$reject$conn$resp              LITERALLY        '11';

Declare                        /*Error Response Codes*/

        invalid$req                      LITERALLY        '2',
        no$resources                     LITERALLY        '4',
        unknown$reference                LITERALLY        '6',
        buffer$too$short                 LITERALLY        '8',
        buffer$too$long                  LITERALLY        '10',
        illegal$req                      LITERALLY        '12',
        rem$abort                        LITERALLY        '14',
        loc$timeout                      LITERALLY        '16',
        unknown$conn$class               LITERALLY        '18',
        dup$req                          LITERALLY        '20',
        conn$reject                      LITERALLY        '22',
        negot$failed                     LITERALLY        '24',
        illegal$address                  LITERALLY        '26',
        network$error                    LITERALLY        '28',
        protocol$err                     LITERALLY        '30',
        illegal$rb$length                LITERALLY        '32';
```

5.7 ISO Reason Codes

Code	Reason
0	Reason not specified.
1	Congestion at TSAP.
2	Client entity not attached to TSAP.
3	Address unknown.
80H	Normal disconnect initiated by client.
81H	Remote TS congestion at connect request time.
82H	Connection negotiation failed (proposed classes not supported).
83H	Duplicate connection detected.
84H	Mismatched references.
85H	Protocol error.
86H	Not used.
87H	Reference overflow.
88H	Connection request refused on this network connection.
89H	Not used.
8AH	Header or parameter length invalid.

CHAPTER 6
NETWORK MANAGEMENT FACILITY

6.1 Overview

The network management facility (NMF) supplies the network with *planning*, *operation*, and *maintenance* functions. The planning capability gathers network usage information to help determine future expansion of the network. Operation deals with the normal, day-to-day network functions such as initialization, termination, monitoring and performance optimization. Maintenance performs the detection, isolation, amputation, and repair of network faults.

The functions needed to implement all three goals of the NMF overlap considerably. For instance, both planning and maintenance need to access the layer databases. Similarly, the operation and maintenance functions must be able to bootstrap a remote node. For this reason, the NMF functions are grouped not by purpose, but by function.

The functions afforded by the network management facility are layer management, debugging operations, down-line loading, dumping, and echo testing.

Layer management provides the ability to examine and modify the internal databases of layers at both local and remote nodes. These databases include counters that indicate how the network is performing as well as network parameters affecting the throughput of the network.

Debugging operations provided by the NMF are limited to the Read_memory and Set_memory commands. With these commands, the user can read or alter the memory of any host on the system.

Down-line loading provides the capability of loading a set of remote nodes from a single host. This is done by a boot server module of the NMF. Besides offering bootstrap services, the boot server can be used to down-line load databases.

Dumping is similar to down-line loading. A dump is initiated by a remote node issuing a dump command to the target node. The target then responds by transmitting a dump response packet that contains an image of the portion of memory requested.

Echo testing allows a host to determine if a particular node is present on the network, and provides a test of the communication path to that node. Here, the NMF transmits a packet to the remote node and then listens for the node to echo it back.

The NMF performs its operations on a remote node by communicating over the network with the NMF at the remote node. In general, this is done by using the transport control layers of each node to create a virtual circuit connection. For down-line loading and dumping, however, the approach is different. Down-line loading is used for booting systems. In particular, the system being loaded might be the communication system itself. Similarly, dumping is used during debugging or when a fault is detected. Here, the operation of the transport control layer may be suspect. For these reasons, communication between nodes for these two commands is restricted to using just the raw data link facilities.

The network management facility is configured using a configuration module as decribed in Chapters 8 and 9. This module consists of calls to NMF configuration macros detailed in this chapter. Most of these macros are used to configure a boot server into the local node. Stations that do not require the boot server can leave out these particular macro calls.

6.2 Addressing Conventions

For most NMF commands, the user must specify the address of the node on which the command is to be performed. This is done by including a pointer to a buffer in memory. If the target node is a local node, this pointer should be 0. For remote nodes, however, there are two possibilities. In some cases, the buffer contains the standard 6-byte Ethernet host address of the target. Otherwise, the buffer is in the format of the transport address buffer as shown below:

```
Declare Address_buffer STRUCTURE (
      Reserved                BYTE,
      Local_TSAP_id_len       BYTE,
      Local_TSAP_id           (Local_TSAP_id_len) BYTE,
      Remote_net_addr_len     BYTE,
      Remote_net_addr         (Remote_net_addr) BYTE,
      Remote_TSAP_id_len      BYTE,
      Remote_TSAP_id          (Remote_TSAP_id_len) BYTE );
```

where

Reserved	must be set to 0.
Local_TSAP_id_len	is the length of the local TSAP id
Local_TSAP_id	is a valid value for the local TSAP id.
Remote_net_addr_len	is the length of the remote network address.
Remote_net_addr	is a valid network address.
Remote_TSAP_id_len	must be set to 2.
Remote_TSAP_id	must be set to 3.

Table 6-1 lists the NMF commands with the associated addressing convention used for each. Refer to Chapter 4 for further information regarding the transport address buffer.

6.3 NMF Objects

During the operation of the network, the network management facility keeps track of various parameters, called *network management facility objects*, for the local node. Any node on the system can read, clear, or change the values of the objects at the

Table 6-1. The NMF Commands and Their Addressing Conventions

Command	Address Convention
Read_object	Transport address
Set_object	Transport address
Read_and_clear_object	Transport address
Read_memory	Transport address
Set_memory	Transport address
Forced_load	Ethernet address
Dump	Ethernet address
Echo	Ethernet address
Supply_buffer	—
Takeback_buffer	—

local node or at any of the remote nodes, by using the NMF commands Read_object, Read_and_clear_object, and Set_object. See Appendix B for detailed descriptions of the NMF objects.

Objects are identified within a particular node by a two-byte *id code*. This code takes the following form:

*wxyz*H

where

w is a digit that identifies the layer that the object belongs to. The values are as follows:

1 Physical layer	5 Session layer
2 Data link layer	6 Presentation layer
3 Network layer	7 Application layer
4 Transport control layer	8 Network management facility

x specifies the entity within the layer. For example, the transport layer hs two entities: the transport virtual circuit and the transport datagram services sublayers. These have a value of *x* that is 0 and 1 respectively.

yz identifies the particular object.

Such a coding arrangement is designed for future expansion of the capabilities of the communication software. For the present release, however, only the following object categories apply:

20*yz*H	Data link layer objects.
40*yz*H	Transport layer virtual circuit objects.
41*yz*H	Transport layer datagram objects.
80*yz*H	NMF/Boot server objects.

Associated with each object is a *modifier* used in conjunction with the object to locate a particular object value. The meaning of the modifier can be different for different objects. In the current release of iNA 960, the modifier is ignored for data link and NMF objects. For the transport layer, however, a modifier of 0 is used for connection independent objects. Connection dependent objects use the connection reference as the modifier.

NMF objects fall into one of these classifications:

Parameter adjusts the actual operation of the layer.

Counter records the number of times a particular event occurs. A counter is an unsigned integer and can be either of the following types:

wrap-around	clears to 0 on overflow.
sticky	sticks at "infinity" on overflow.

Statistic is a time-averaged measure of some aspect of system operation.

Value is none of the above.

6.3.1 NMF/Data Link Objects

Data link objects are provided that track the raw communication activity of the station. Included, are counters that monitor the total number of packets sent and received by the node. Collision activity is gauged by counters for primary and secondary collisions and for packets dropped due to excessive collisions. Finally, the rates of a number of different types of reception errors are tallied. These are CRC errors, where packets are dropped because the CRC code does not check; alignment errors, where a data boundary is not aligned at a byte boundary; and resource errors, where packets are dropped because no buffers are available for them.

The following is a list of the data link objects along with their object id codes. For a complete description of each object, refer to Appendix B.

2000H Data Link Type
2001H Line Speed
2002H Host Id
2003H Total Sent
2004H Primary Collisions
2005H Secondary Collisions
2006H Exceeded Collisions
2007H Total Received
2008H CRC Errors
2009H Alignment Errors
200AH Resource Errors

6.3.2 NMF/Transport Layer Objects

The transport layer objects included in iNA comprise objects of the virtual circuit subsystem and the datagram subsystem of the transport layer. The objects of the virtual circuit subsystem are further classified as either connection independent or connection dependent objects. For the connection dependent objects, the modifier of the object is used to specify the connection identifier.

The transport objects include error counters, statistics, configuration parameters, and timeouts. The following is a list of the transport objects. See Appendix B for a detailed description of these objects.

NMF/Transport Virtual Circuit Connection Independent Objects

4000H Virtual Circuit Type
4001H Connection id Vector
4002H ISO Transport Number
4003H Maximum Connections
4004H Current Maximum Connections
4005H Maximum On-Board CDBs
4006H ActiveCDBs
4007H CDB Size
4008H Default Persistence Count
4009H Default Abort Timeout
400AH Default Retransmit Timeout
400BH Minimum Retransmit Timeout
400CH Closing Abort Timeout
400DH Flow Control Window Timeout
400EH Inactivity Maximum Count
400FH Total Duplicate Segments Rejected
4010H Total Checksum Errors

4011H Total Retransmissions
4012H Total Resource Errors
4013H Maximum Network Address Length
4014H Maximum TSAP-id Length
4015H Local NSAP-id
4016H Reserved
4017H Reserved
4018H Default Connection Negotiation
4019H Maximum TPDU Size
401AH No Additional Option Field
401BH No Maximum TPDU Size Field
401CH Maximum Normal Window Size
401DH Maximum Extended Window Size
401EH Minimum Credit
401FH Open Window Timeout
4020H Maximum Open Window Count

NMF/Transport Virtual Circuit Connection Dependent Objects

4081H Local TSAP-id
4082H Remote Network Address
4083H Remote TSAP-id
4084H Connection State
4085H Remote Connection id
4086H Persistence Count
4087H Abort Timeout
4088H Retransmit Timeout
4089H Next Transmit Sequence Number
408AH Duplicate Segments Rejected
408BH Segments Retransmitted
408CH Resource Errors
408EH Client Options
408FH Class Options
408FH Additional Options
4090H Maximum TPDU Size
4091H Maximum TPDU Data Length
4092H Inactivity Count
4093H Reserved

NMF/Transport Datagram Objects

4100H Datagram Type
4101H Datagram Receive Queue Size
4102H Reserved
4103H Total Datagrams Transmitted
4104H Total Datagrams Received
4105H Total Datagram Resource Errors
4106H Total Datagram Checksum Errors
4107H Total Datagram Address Errors

6.3.3 NMF/Boot Server Objects

The objects of the network management facility are restricted to those that are used to configure the boot server. Most of these objects are determined at configuration time and may not be altered. The single exception is the boot table that may be changed dynamically during run time. This, in effect, changes the list of nodes that are recognized by the boot server.

The following is a list of the NMF/Boot server objects. See Appendix B for a detailed description of these objects.

8000H NMF Type
8001H Multicast Address
8002H Maximum Number of Nodes
8003H Maximum Number of Addresses
8004H The Boot Table
8005H Number of Class Codes
8006H List of Class Codes
80FFH Number of NMFs

6.4 NMF Commands

This section gives a brief introduction to the network management facility commands, followed by a detailed description of each command.

As explained above, the network management facility manages the NMF objects that control the operation of the data link layer, transport layer, and the boot server. The NMF objects are viewed and modified with the following NMF commands:

Read_object	Returns the value of the given object.
Set_object	Sets the given object with the given value, if possible.
Read_and_clear_object	Returns the value of the given object and sets the object to 0, if possible.

As a debugging aid, the NMF allows the user to read or set the memory of any node on the network using the following commands:

Read_memory	Downloads a given portion of memory from the remote node to the local node.
Set_memory	Loads the memory of the remote node with a given portion of memory from the local node.

A particular node can force another node to initiate a down-line loading sequence with the following command:

Forced_load	Forces a given node to request a down-line load from the boot server.

The following commands are used as debugging and maintenance aids:

Dump	Downloads a given portion of memory from the remote node to the local node using the raw data link facilities.
Echo	Sends a block of random data to a remote node that echoes the data back to the local node.

The user may add new commands to the existing NMF commands. Commands that are not recognized by the NMF are forwarded to the user for dispensation. This is done using the following commands:

Supply_buffer	Supplies a user buffer to the NMF for storing an incoming packet with an unrecognized command field.
Takeback_buffer	Retrieves all outstanding Supply_buffer request blocks from the NMF.

6.4.1 Read/Set/Read_and_clear_object

There are three commands available to interface with the objects of a layer: Read_object, Set_object, and Read_and_clear_object. In addition, each command may operate on several objects during a single invocation.

To use one of these commands, the user sets aside two buffers in memory. A *command buffer* is used to specify the list of objects that are the subject of the command, and (for Set_object) to store the new object values. A *response buffer* is used to return the values read by the NMF. The user creates and fills in the command buffer before invoking the command. The response buffer is filled by the NMF after executing the command. The formats for these two buffers are specified later in this section.

Command failure (such as no response from the remote station) is indicated by a code in the response field of the request block. In this case, nothing is written to the response buffer. However, if an attempt is made to clear an object that cannot be cleared or a nonexistent object is named, then the command is ignored only for that particular object. Thus, the number of object values in the response buffer may be less than the number specified in the command buffer.

The procedure is different for the Set_object command. First, an attempt is made to set the object with the specified value. Then the value of the object is read and the result returned in the response buffer, even if the object cannot be set.

Request Block Interface

```
DECLARE Rb STRUCTURE (
      Rb_header            (6) WORD,
      Layer                BYTE,
      Opcode               BYTE,
      Response             WORD,
      Trans_addr_ptr       POINTER,
      Filled_length        WORD,
      Resp_buf_ptr         POINTER,
      Resp_buf_length      WORD,
      Cmd_buf_ptr          POINTER,
      Cmd_buf_length       WORD );
```

Procedure Interface

```
CQ$NML$READ$OBJECT (CMD_buf_ptr, Cmd_buf_length,
      Resp_buf_ptr, Resp_buf_length, Trans_addr_ptr, User,
      Return_mailbox, Exception_ptr)

CQ$NML$READC$OBJECT (CMD_buf_ptr, Cmd_buf_length,
      Resp_buf_ptr, Resp_buf_length, Trans_addr_ptr, User,
      Return_mailbox, Exception_ptr)

CQ$NML$SET$OBJECT (CMD_buf_ptr, Cmd_buf_length,
      Resp_buf_ptr, Resp_buf_length, Trans_addr_ptr, User,
      Return_mailbox, Exception_ptr)
```

Command Parameters

Op_code	One of the following: 0 – Read_object 1 – Read_and_clear_object 2 – Set_object
Response	Set by the NMF after executing the command. Values are as follows: 1 – OK response 2 – no response from the remote node 8 – error while trying to connect with remote NMF A – command not configured C – illegal request FFFE – command not configured
Trans_addr_ptr	A pointer to a buffer containing the transport address of the target node.
Filled_length	The size, in bytes, of the buffer filled in by the NMF. This field is set by the NMF after executing the command.
Resp_buf_ptr	Points to the response buffer.
Resp_buf_length	The length of the response buffer.
Cmd_buf_ptr	Points to the command buffer.
Cmd_buf_length	The length of the command buffer. For commands on remote nodes, this must be less than 90 bytes.

Command Buffer Format

The format for the command buffer is the following:

```
DECLARE Command_buffer STRUCTURE (
      Number            BYTE,
      Obj_info          (Number) STRUCTURE (
            Object            WORD,
            Modifier          WORD,
            Length            WORD,
            Value             (Length) BYTE ));
```

where

Number	is the number of objects included in the buffer.
Object	is the id code for the given object.
Modifier	is a code used with some objects to select a particular value from within the given object.
Length	is the length, in bytes, of the Value field. Set to 0 for the Read and the Read_and_clear commands.
Value	contains the new value of the object for the Set command. Ignored for the Read and Read_and_clear commands.

Response Buffer Format

The format for the response buffer is the following:

```
DECLARE Response_buffer STRUCTURE (
      Number            BYTE,
      Obj_info          (Number) STRUCTURE (
            Object            WORD,
            Modifier          WORD,
            Length            WORD,
            Value             (Length) BYTE ));
```

where

Number	is the number of objects included in the buffer.
Object	is the id code for the object. Only valid objects are returned by the NMF. In addition, objects that cannot be cleared are not returned during the Read_and_clear command.
Modifier	is a code used with some objects to select a particular value from within the given object.
Length	is the length, in bytes, of the Value field.
Value	contains one of the following: • the value of the given object for a successful Read or Read_and_clear command. • the value read back from the given object after an attempt has been made to set it (for a successful or an unsuccessful Set command).

6.4.2 Read/Set_memory

As an aid to debugging, the network management facility provides the host with commands to read or set the memory of any node on the network. The Read/Set_memory commands to remote nodes use the services of the transport control layers at the two nodes.

To use one of these commands, the user sets aside a buffer in memory. For the Read_memory command, this buffer is used to store the memory image retrieved from the target node. For the Set_memory command, this buffer contains the memory image that is loaded into the memory at the target node.

Request Block Interface

```
DECLARE Rb STRUCTURE (
      Rb_header            (6) WORD,
      Layer                BYTE,
      Opcode               BYTE,
      Response             WORD,
      Trans_addr_ptr       POINTER,
      Filled_length        WORD,
      Buffer_ptr           POINTER,
      Buffer_length        WORD,
      Start_addr           POINTER );
```

Procedure Interface

```
CQ$NML$READ$MEM (Start_addr, Buffer_ptr, Buffer_length,
      Trans_addr_ptr, User, Return_mailbox, Exception_ptr)

CQ$NML$SET$MEM (Start_addr, Buffer_ptr, Buffer_length,
      Trans_addr_ptr, User, Return_mailbox, Exception_ptr)
```

Command Parameters

Op_code	One of the following: 3 – Read_memory 4 – Set_memory
Response	Set by the NMF after executing the command. Values are as follows: 1 – OK response 2 – no response from the remote node 8 – error while trying to connect with remote NMF A – command not configured C – illegal request FFFE – command not configured
Trans_addr_ptr	Pointer to a buffer containing the transport address of the target node.
Filled_length	Size, in bytes, of the memory area filled in by the NMF. This field is set by the NMF after executing the command.

Buffer_ptr	Points to a buffer in memory that does the following: • contains the memory image read from the target node (Read_memory). • contains the memory image to be loaded into the target node (Set_memory).
Buffer_length	Length, in bytes, of the buffer selected by Buffer_ptr. If the target node is a remote node, this value must be less than 90.
Start_addr	Pointer containing the starting address in the memory of the target node where the operation is to be performed.

6.4.3 Forced_load

Any node on the network may make a down-line load request from the boot server using the handshake procedure described in the *iNA 960 Architecture Reference Manual*. In addition, a node may force another node to initiate a down-line load sequence. This is accomplished with the Forced_load command.

When the local NMF receives the Forced_load command, it transmits a packet containing the Forced_load command and a class code for the boot request. This is done using only the raw data link services. The NMF then waits for the remote node to return a Forced_load response packet. If there is no response, the NMF tries two additional times before assuming that the node is not responding.

Request Block Interface

```
DECLARE Rb_forced_load STRUCTURE (
      Rb_header              (6) WORD,
      Layer                  BYTE,
      Opcode                 BYTE,
      Response               WORD,
      Datalink_addr_ptr      POINTER,
      Class_code             WORD );
```

Procedure Interface

```
CQ$NML$FORCE$LOAD (Class_code, Datalink_addr_ptr, User,
      Return_mailbox, Exception_ptr)
```

Command Parameters

Op_code	7
Response	Set by the NMF after executing the command. Values are: 1 – OK response 2 – no response from the remote node A – command not configured C – illegal request FFFE – command not configured
Datalink_addr_ptr	Pointer to a buffer containing the data link address of the node that should request a down-line load.
Class_code	Class code used in the down-line load request.

6.4.4 Dump

The NMF provides the facility to enable a node to get a dump of the memory of a remote node. This is done with the Dump command. Upon receiving this command, the NMF transmits a packet containing the dump command and waits for the remote node to return a dump response packet. If the remote node does not respond, the NMF tries an additional two times before assuming that the remote node is not responding.

To use the Dump command, the user sets aside a buffer in local memory. In addition, the user specifies a starting address in the memory of the remote node. The memory image returned begins at this starting address and is no larger than the buffer specified by the command. The maximum size of the memory image that can be returned is 1489 bytes.

The Read_memory and Dump commands are similar in their operation. The difference, however, is that Read_memory uses the services of the transport control layer to perform the communication between the nodes. Dump uses only the raw data link services.

Request Block Interface

```
DECLARE Rb_dump STRUCTURE (
      Rb_header              (6) WORD,
      Layer                  BYTE,
      Opcode                 BYTE,
      Response               WORD,
      Datalink_addr_ptr      POINTER,
      Filled_length          WORD,
      Buffer_ptr             POINTER,
      Buffer_length          WORD,
      Start_addr             WORD );
```

Procedure Interface

```
CQ$NML$DUMP (Start_addr, Buffer_ptr, Buffer_length,
      Datalink_addr_ptr, User, Return_mailbox, Exception_ptr)
```

Command Parameters

Op_code	5
Response	Set by the NMF after executing the command. Values are as follows: 1 – OK response 2 – no response from the remote node 4 – received packet has wrong packet length field, indicating a breach of the *IEEE 802* data link specifications by the remote node A – command not configured C – illegal request FFFE – command not configured
Datalink_addr_ptr	Pointer to a buffer containing the data link address of the target node.
Filled_length	Size, in bytes, of the memory area filled in by the NMF. This field is set by the NMF after executing the command.

Buffer_ptr	Pointer to a buffer in memory where the NMF is to store the requested memory image.
Buffer_length	Length, in bytes, of the buffer selected by Buffer_ptr.
Start_addr	Pointer containing the starting address in the memory of the target node of the memory image that is requested.

6.4.5 Echo

Echo service is used to determine if a given node is present on the network, to test the communication path to the remote node, and to ascertain the viability and functionality of the remote station.

To use the command, the user specifies the address of the remote node along with a count value. The NMF then transmits a packet containing the echo command with a block of random data. The size of this block is that specified by the count value. The local NMF then waits for the remote node to return the response packet. If the remote node does not respond, the NMF tries an additional two times before assuming that the remote node is not responding.

Upon receipt of the response packet, the local NMF determines whether the returned block of data is the same as the transmitted block. Thus, the echo command tests both the existence of the remote node on the network and the viability of the node.

Request Block Interface

```
DECLARE Rb_echo STRUCTURE (
      Rb_header              (6) WORD,
      Layer                  BYTE,
      Opcode                 BYTE,
      Response               WORD,
      Datalink_addr_ptr      POINTER,
      Transmit_data_count    WORD,
      Received_data_count    WORD );
```

Procedure Interface

```
CQ$NML$ECHO (Transmit_data_count, Datalink_addr_ptr, User,
      Return_mailbox, Exception_ptr)
```

Command Parameters

Op_code	6
Response	Set by the NMF after executing the command. Values are as follows: 1 – OK response 2 – no response from the remote node 6 – transmitted data and received data do not match A – command not configured C – illegal request FFFE – command not configured
Datalink_addr_ptr	Pointer to a buffer containing the data link address of the target node.
Transmit_data_count	Number of bytes to be transmitted in the echo command packet.
Received_data_count	Number of bytes present in the echo response packet.

6.4.6 Supply/Takeback_buffer

When a packet is received with the DLSAP of the NMF (the DLSAP of the NMF is 8), the command field is first examined to determine if the NMF recognizes the command. If it does, the NMF executes the command. If not, the packet is forwarded to the user for dispensation. In this way, the user may add NMF commands to those that already exist.

This procedure can be employed by users who wish to write their own boot server modules. Here, the vendor-supplied boot server is not configured into the system. Then, the NMF no longer recognizes the boot request and boot data request commands. These are instead forwarded to the user to process. The user can then generate the appropriate response packet, using the services of the iNA 960 data link layer.

When processing a packet that contains an NMF command to be forwarded, the NMF needs a buffer to store the packet. This is supplied to the NMF by the user via the Supply_buffer command. If a packet is received with no buffer available, the packet is dropped. If the size of the buffer is smaller than the length of the received packet, only the initial part of the packet is copied into the buffer.

Buffers given to the NMF to store incoming packets remain with the NMF until a packet with an unrecognized command field is received. Thus, unlike other commands, there is no time limit for the Supply_buffer request block to be returned to the user.

Occasionally, the user needs all outstanding buffers to be returned. This is accomplished by issuing the Takeback_buffer command. Upon receipt of this command, all Supply_buffer request blocks are returned to the user (with a response code of 3, OK takeback response) followed by the Takeback_buffer request block.

Supply_buffer RB Interface

```
DECLARE Rb_supply_buffer STRUCTURE (
      Rb_header              (6) WORD,
      Layer                  BYTE,
      Opcode                 BYTE,
      Response               WORD,
      Filled_length          WORD,
      Buffer_ptr             POINTER,
      Buffer_length          WORD )
```

Takeback_buffer RB Interface

```
DECLARE Rb_takeback_buffer STRUCTURE (
      Rb_header              (6) WORD,
      Layer                  BYTE,
      Opcode                 BYTE,
      Response               WORD )
```

Procedure Interface

```
CQ$NML$SUPPLY$BUF (Buffer_ptr, Buffer_length, User,
      Return_mailbox, Exception_ptr)

CQ$NML$TAKEBACK$BUFFER (User, Return_mailbox, Exception_ptr)
```

Command Parameters

Op code	One of the following: 8 – Supply_buffer 9 – Takeback_buffer
Response	Set by the NMF after executing the command. The values are as follows: SUPPLY_BUFFER 1 – OK response; buffer contains a packet 3 – OK response; buffer is returned in response to a takeback command 4 – received packet has wrong packet length field, indicating a breach of the *IEEE 802* data link specifications by the remote node C – illegal request E – buffer too small; buffer must be at least 14 bytes TAKEBACK_BUFFER 1 – OK response C – illegal request
Filled_length	Size, in bytes, of the memory area filled in by the NMF. This field is set by the NMF after executing the command.
Buffer_ptr	Pointer to the buffer in local memory offered by the Supply_buffer command.
Buffer_length	Length, in bytes, of the buffer selected by Buffer_ptr.

6.5 Down-Line Loading

The network management facility provides the means for downloading remote systems. Usually, this feature is implemented to boot various stations. Individual stations without mass storage can use it to boot themselves, or a station can force other remote stations to boot or reboot. In particular, the down-line loading facility can be used to boot a set of nodes with the same version of software. Implementations of this facility are not restricted to booting operations, however. Down-line loading can also be used to load a database from (or to) a remote node.

A down-line loading operation requires the cooperation of two stations. The node that is to be loaded is called the *target station.* The node that supplies the required data is called the *executor station.* At the time of a down-line load request, the target station may be in a state that can use only minimal data link facilities. For example, it may be the COMM system itself that is to be loaded. For this reason, the two stations communicate by a primitive protocol that uses only the raw data link facilities.

The process that runs on the target station is called the *boot consumer*, and the process running on the executor station is termed the *boot server.* Because the two processes together provide a service that is general enough to be used for purposes other that booting, this terminology can be misleading.

Down-line loading is conducted in the following way. The boot consumer transmits requests to the boot server and then waits for the boot server to respond. The response typically consists of some data and some control information. The control information informs the boot consumer how to interpret the data and whether more data is to follow. A typical boot sequence would consist of the boot consumer issuing a request for the first block of data; receiving a packet from the boot server; processing the packet; and then issuing a request for the next block of data. This sequence of events continues until the loading operation is complete.

In addition to a remote node initiating its own down-line load, a third node may force the remote node to request down-line loading service. This is accomplished by the Forced_load command.

6.5.1 Class Codes

Any request for a down-line load is accompanied by a *class code* for the request. Each class code determines the sequence of operations to be performed by the executor station. Each operation can be either of the following:

- A given named file is transmitted to the target station.
- A user-written subroutine is executed by the executor station.

Those files that are transmitted in response to a down-line load request must reside on the same device at the executor node. This device is specified by the Dev_info_block macro in the NMF configuration file. In addition, any modules transmitted have the following *boot module format*:

```
DECLARE Boot_module STRUCTURE (
      Command                BYTE;
      Load_addr              DWORD;
      Length                 WORD;
      Execution_addr         DWORD;
      Memory_image           (*) BYTE )
```

where

Command	indicates if the module is to be executed and if there are more modules to be loaded. Only the two low-order bits are meaningful and have the following values: bit 0 = 1 another module is to be loaded (i.e., the current module is not the last one). bit 1 = 1 the execution address is to be called. bits 2 – 7 must be set to 0.
Load_addr	specifies the first address where the data is to be placed.
Length	specifies the length of the memory image data.
Execution_addr	specifies the execution address of the loaded memory image.
Memory_image	is the memory image to be transferred. This can be between 0 and $2^{16}-1$ bytes long.

To include one or more subroutines in a down-line load request, a pointer for each subroutine is included in the class code definition. Each pointer indicates a location in the memory of the executor station that contains the entry address of the selected subroutine.

These routines can transmit any additional data to the requesting station. The declaration of each routine is in the form:

```
User_boot_subroutine: PROCEDURE
      (Host_id_pointer, Class_code, Block_number) WORD;

DECLARE
      Host_id_pointer POINTER, /* Points to a buffer
                                  containing the host id
                                  of the remote node */
      Class_code       WORD,   /* Class code of the remote
                                  station */
      Block_number     WORD,   /* Block number expected by
                                  the remote station */
End User_boot_subroutine;
```

The WORD returned by the procedure specifies the block number expected by the remote node upon return from the procedure.

In addition to being in the above format, each user-provided subroutine must follow these conventions:

- The routine must follow the LARGE PL/M-86 model of computation.
- The routine should not block for any reason. That is, it should not wait indefinitely at a mailbox, semaphore, etc. This is because execution of the entire communication system will not proceed until control returns from the subroutine.
- The stack size available is 20 bytes.
- The routine runs under the COMM job user id, that is, the same user id as the boot server and the rest of the communication system.
- The routine must be loaded into memory by the user before the communication system starts execution.

As an example, a class code might specify the following sequence of events:

1. Transmit file 'ABC'.
2. Call subroutine 'boot_A'.
3. Transmit file 'XYZ'.

The boot server sends the file 'ABC' as a series of blocks with associated block numbers, say, 0 – 5. The boot server then calls the user subroutine 'boot_A'. The input parameter, Block_number, in this case is 6. If 'boot_A' does not send data to the boot consumer, the value returned from the procedure is 6. On the other hand, suppose 'boot_A' sends three blocks (6, 7, and 8) of data. Here, the value returned from the procedure is 9. Once the boot server has returned from procedure 'boot_A', the file 'XYZ' is split into blocks and transmitted to the boot consumer. The details of the protocol used between the boot consumer and the boot server are given in the *iNA 960 Architecture Reference Manual.*

6.5.2 The Boot Table

Upon initialization, the boot server boots any node that requests its service. During run time, the boot server can be reconfigured to service only a specified set of nodes. The list of node addresses for this set of serviceable nodes forms a dynamic object called the *boot table.* This is NMF object number 8004, and has the following structure:

```
DECLARE Boot_table STRUCTURE (
      Num_nodes        WORD,
      Node_address     (Num_nodes) STRUCTURE (
            Address_byte    (6) BYTE ));
```

Here, Num_nodes specifies the number of addresses in the boot table, each address being 6 bytes long. At initialization, the boot table has the following form:

```
(1, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH)
```

The sole address in the boot table is the broadcast address. In this case, the boot server boots any node that requests its service.

The boot table can be changed any time after initialization by performing the following steps:

1. Create the structure Boot_table in the format shown above. This contains a list of the node addresses for the nodes to be included in the boot table. Only these nodes will be recognized by the boot server.
2. Allocate a command buffer and a response buffer large enough to execute the Set_object command on NMF/boot server object 8004.
3. Fill in the command buffer with the appropriate format, and include a copy of Boot_table.
4. Execute the Set_object command on object 8004 using the command buffer.
5. If the command was successful, check the response buffer to confirm that the boot table was set.

The NMF does not check the contents of Boot_table. Thus, if

Boot_table.Num_nodes = 6

then there must be 6 addresses in the boot table, and the size of Boot_table must be 38 bytes.

6.6 Configuring the Boot Server and the NMF

Configuration of the COMM system is detailed in Chapters 7 and 8. As part of the configuration process, the user creates a network management facility configuration file, NMFCFG.A86. This file consists of calls to several configuration macros via user-supplied parameters. NMFCFG.A86 is then assembled and linked to the COMM system.

The NMF configuration macros and their associated parameters are described in this section. Most of these macros are used to configure the boot server supplied with the system. If this boot server is not used, the boot server configuration macros are not included in the NMF configuration file.

6.6.1 Boot_server_multicast_address

This macro takes the following form:

```
%Boot_server_multicast_address (Multicast_address)
```

where

Multicast_address is the multicast address of the boot server.

Remote nodes requesting the services of the boot server issue packets containing a boot request and the multicast address of the boot server. This address is also used by a local boot consumer to locate a remote boot server.

6.6.2 Max_nodes

The boot server is initialized to boot any node that requests its services. Here, the boot table has the following value:

```
(1, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH)
```

During run time, the boot table (NMF/boot server object 8004) may be changed using the Set_object command. This restricts the boot server to recognizing boot requests from nodes specified in the boot table. The Max_nodes macro allocates space to hold the boot table by specifying the maximum number of node addresses that can appear in the boot table at one time.

This macro takes the following form:

```
%Max_nodes (Max_value)
```

where

Max_value — is the maximum number of addresses in the boot table. The space reserved for this table is:

(6 × Max_value) WORDS

Max_value must be greater than or equal to 1.

6.6.3 Max_simultaneous_boots

Max_simultaneous_boots sets the upper bound for the number of boot consumers serviced at a single time by the boot server. This value should be as small as necessary to conserve resources.

This macro has the following form:

```
%Max_simultaneous_boots (Max_value)
```

where

Max_value	is the maximum number of nodes that the boot server can boot at any one time.

If the boot server is booting Max_value nodes and a new node requests service, the request is ignored.

6.6.4 Class_code_info

This macro has the following form:

```
%Class_code_info (File_name, File_size)
```

where

File_name	is the pathname of a file containing the definitions of the class codes recognized by the boot server.
File_size	is the maximum size of File_name.

The file specified in the Class_code_info macro is read by the boot server during initialization. This specifies the set of class codes the boot server recognizes and the associated list of files that are transmitted to the requesting station. The boot server only reads this file when it initializes. It does not modify this file in any way or read it at any other time. The user can, however, modify this file and then reboot the system if the file is changed.

It is the user's responsibility to ensure that the actual size of File_name does not exceed the parameter File_size. In addition, it is important that the file specified in Class_code_info is in the format indicated by the following PL/M-style structure:

```
DECLARE Cc_info STRUCTURE (
      Num_class_codes      BYTE,
      Class_code_spec      (Num_class_codes) STRUCTURE (
            Class_code     WORD,
            Num_entries    BYTE,
            Entry          (Num_entries) STRING ));
```

Here, a STRING is a sequence of bytes, with the first byte specifying the number of bytes in the string. For example, the file iNA 960 would be represented as follows:

```
7, 'iNA 960'
```

If the first byte is 0, it would normally indicate a null string. In this case, however, the entry is interpreted as a pointer, and the next 4 bytes are used as the value of the pointer.

When the boot server receives a boot request with a given class code, it runs through the list of entries for that class code in the order that they appear in the above file. For each entry that is a filename, the boot server transmits the specified file to the requesting station. When the entry is a pointer, the boot server uses the pointer as an entry to a subroutine and calls that subroutine.

The class code information file (CC.INFO in the configuration file), must be in the format shown above. The following is an example of this file, described as a PL/M-style data type.

```
Class_code_info (*) DATA (

0002,    /* The boot server recognizes 2 class codes.                    */

1234H,   /* The first class code is '1234H'.                             */
3,       /* The boot server performs the following 3 steps               */
12, '/user/ina960',     /* 1. Transmits the file '/user/ina960'         */
0, 2, 2F40,             /* 2. Calls a subroutine at location 2F40:2     */
16, '/user/ina960.pat',/* 3. Transmits the file '/user/ina960.pat'      */

4321H,   /* The second class code is '4321H'.                            */
1,       /* The boot server performs the following step                  */
12, '/user/ina960')     /* 1. Transmits the file '/user/ina960'         */
```

Note that this file is not an ASCII file. For instance, the first entry in the file is a word with the value 2. The hex representation of the first two bytes in the file is 02,00. The order is reversed because in a word field, the least significant byte gets placed before the most significant byte. The only ASCII entries in the file are the filenames, enclosed in apostrophes. (The apostrophes in the above example should not be included in the file. They are present to increase readability.)

6.6.5 Device_info_block

The Device_info_block macro specifies the physical device that contains all the files used by the boot server. When this macro is called, the boot server physically attaches the named device. This macro has the following form:

```
%Device_info_block (Logical_name, Device_name, File_driver)
```

where

Logical_name	is the name for the connection that appears in the directory of the ROOT job.
Device_name	is the device unit that is used by the boot server.
File_driver	is NAMED. This is the only type of file the boot server expects to find.

It is imperative that the class code definition file and all other files used by the boot server reside on the device specified in this macro. In addition, if a call to this macro is included in the NMF configuration, the associated call to the same macro should be removed from the configuration of the extended I/O system (EIOS), if used.

6.6.6 Nmf_cnfg

This macro sets the configuration state of the network management facility. This can be used to restrict the services provided by the NMF. Commands such as Read_and_clear_object, Set_object, and Set_memory can be catastrophic if provided to normal users. By configuring only those services required by each user, system security is maintained and considerable memory is saved.

The format for this macro is the following:

```
%Nmf_cnfg (Cnfg_state, Rem_access)
```

where

Cnfg_state — can be any of the following:

- 0 – users cannot give commands to the NMF. The NMF only processes packets with the NMF DSAP Id.
- 1 – in addition to the services provided by '0', NMF can operate on local objects, read/set memory of the local node, and perform the commands Echo, Dump, and Forced_load.
- 2 – in addition to the services provided by '1', the user can read objects and memory of remote stations.
- 3 – in addition to the services provided by '2', the user can modify objects and memory of remote stations.

Rem_access — can be either of the following:

- 0 – Remote stations do not have access to this station.
- 1 – Remote stations do have access to this station. In this case, Cnfg_state must be greater than 0.

The macro call NMF_cnfg (0, 1) is not allowed.

Whenever a user issues a boot server request that is not configured into the system, the response code returned by the NMF is 0AH or 0FFFE – Exception, command not configured.

6.6.7 Sample Configuration

The file NMFCFG.A86 should contain the macro calls to configure the network management facility. If the boot server is to be left out or is written by the user, then the following macro calls must not be made:

- Max_nodes
- Max_simultaneous_boots
- Class_code_info
- Device_info_block

The following example of an NMF configuration file, NMFCFG.A86, configures the boot server as well as the NMF.

```
NAME NMF_CNFG_MACROS

$ INCLUDE (:SD:COMM/CONFIG/NMFCFG.MAC)

%Boot_server_multicast_address
      (01, 0AAH, 0, 0FFH, 0FFH, 0FFH)
      %' The multicast address of the boot server.

%Max_nodes (20)
      %' The maximum number of nodes in the boot
      %' table (if used).

%Max_simultaneous_boots (10)
      %' The maximum number of nodes that can be booted
      %' at one time.

%Class_code_info (/USER/CC.INFO, 200H)
      %' The class code information file is called
      %' /USER/CC.INFO.
      %' /USER/CC.INFO is at most 200H bytes.

%Device_info_block (WD0, IW0, NAMED)
      %' The physical device IW0 is connected and
      %' the file connection is cataloged as WD0 in the
      %' ROOT job directory. All files used by the boot
      %' server are on this device.

%Nmf_cnfg (4, 1)
      %' The full capability of the NMF is configured
      %' into the system.
END
```

The NMF can be left out of the COMM system by omitting the files NMFCFG.OBJ and NMF.LNK during the linking process.

CHAPTER 7
ROM-BASED NMF

7.1 Overview

A primitive NMF can be burned into ROM for system start-up and reset. This *ROM-based NMF* or *NMF ROM* is primarily used for downloading. Since at system initialization the communication system is not present, the functions of the NMF that use the transport layer are not ROM-based. The NMF functions that use the service of only the data link layer are ROM-based.

NMF ROM consists primarily of the boot consumer, which, in conjunction with the boot server, can download the operating system. Nodes are expected to have NMF ROM burnt into ROM's. After a hardware reset, the NMF ROM begins execution. It invokes the boot consumer which, in turn, downloads the operating system into local memory. After the operating system is loaded, NMF ROM is not used again. The similarity between NMF ROM and a bootstrap loader is evident; the only difference is that the bootstrap loader downloads the system from the local mass storage controller, whereas NMF ROM downloads the system from the boot server.

The NMF ROM-based functions fall into two categories:

- *Remote invocations* – Upon receipt of a packet, the ROM-based NMF only responds if the packet has the NMF DLSAP. Furthermore, only the following commands are recognized: Echo, Forced_load, Dump.

- *Local invocations* – The only user-initiated function recognized by the ROM-based NMF is the down-line loading facility. This facility uses a simple proctocol in conjunction with the boot server to receive data and store it in the appropriate memory locations. The ROM-based NMF can be configured to initiate a boot at system reset.

The ROM-based NMF consists of two files, ROMA.OBJ and ROMB.OBJ. In addition, the user must include a User_code module that performs any functions required to be placed in ROM. For example, power-on diagnostics would be placed in the User_code module. To configure the ROM-based NMF, the NMF ROM object code, the user code, and a configuration file (CNFG.OBJ) are linked and located and the resulting file is burned into ROM.

7.2 Boot_consumer

In order to force the NMF ROM to attempt a remote boot, include a call to the procedure Boot_consumer in the User_code module. Boot_consumer is a typed procedure included in the ROM-based NMF. It takes a class code as an input parameter and returns a status byte for the result of the operation. A call to this procedure initiates an attempt to contact the boot server with a downline load request with the given class code. The procedure returns a value of OK or NOTOK depending on whether the boot server responded to the request.

This procedure is defined below:

```
Boot_consumer: PROCEDURE (Class_code) EXTERNAL BYTE;
      DECLARE Class_code        WORD;

      /* Responses are  TRUE  - Ok response
                        FALSE - No reponse from boot server  */

END Boot_consumer;
```

7.3 User_code

User code such as power-on diagnostics can be included in the ROM-based NMF. To do this, create a User_code module in the format shown below. If the communications CPU is an 80186, include code that initializes the chip-select lines.

```
User_code: DO;

        /* For the 80186, initialize the chip-select lines */

        /* User code goes here                              */

      DISABLE;
      CALL Init_controller;
        /*This procedure initializes the communications
          controller and enables interrupts. Now, the
          NMF ROM responds to Echo, Dump, IEEE XID and
          Test commands.                                   */

      Status = Boot_consumer (Class_code);
        /* This section of code is reached only when the
           remote boot is unsuccessful.                    */

      IF Status <> 0 THEN DO;
        /* No response from boot server. Take corrective
           action such as calling the boot server again.   */
      END;

END User_code;
```

The file AUTBOT.P86 is included in the delivery diskette. The user should examine this file and use it as the user code module if it meets his requirements.

AUTBOT.P86 is the file generated by the following source code:

```
Autoboot: DO;

      Boot_consumer: PROCEDURE (Class_code) EXTERNAL BYTE;
            DECLARE Class_code  WORD;
      END Boot_consumer;

      Init_controller: PROCEDURE EXTERNAL;
      END Init_controller;

      DECLARE Status WORD;

      /* If the processor is an 80186, then code to
         initialize the chip-select lines goes here.      */
```

```
        DISABLE;
        CALL Init_controller;

        DO WHILE (0FFH);
              Status = Boot_consumer (Class_code);
              /* If the boot process is successful, then program
                 control never reaches here; hence, the infinite
                 loop.                                          */
        END;

END Autoboot;
```

7.4 Configuring the ROM-Based NMF

The file CNFG.OBJ is linked with the rest of the ROM-based NMF to form a file that is located and burned into ROM. CNFG.OBJ is the object file generated from the configuration file CNFG.A86, described in this section. CNFG.A86 contains the macro calls used to configure the ROM-based NMF.

7.4.1 Boot_server_multicast_address

This macro sets the multicast address of the boot server. The NMF uses this address to locate the boot server. This macro has the following form:

```
%Boot_server_multicast_address (Multicast_address)
```

where

Multicast_address is the multicast address of the boot server.

7.4.2 Data_link_type

This macro specifies the type of the data link layer. Such a macro is needed for future releases that support different data link types. Currently, the only valid value is IEEE802. This macro takes the following form:

```
%Data_link_type (Dl_type)
```

where

Dl_type is IEEE802.

7.4.3 Host_id

The Host_id macro sets up the host id of the Ethernet controller. There are two possibilities:

- The 550a Ethernet controller cannot have its host id changed. In this case, the macro must still be invoked; however, a host id of 0 should be used.
- The 82586 Ethernet controller can have its host id changed using this macro. If the default host id of the 186/51 is desired, the host id should be set to a broadcast address.

The Host_id macro has the following form:

```
%Host_id (H_id)
```

where

H_id	is the host id.

7.4.4 Board_type

This macro specifies the type of programmable interrupt controller (PIC) used. iNA currently supports two PICs: the 8259a and the PIC in the 80130. This macro specifies the master PIC in the system. This macro has the following form:

The macro has the following form:

```
%Board_type (Type, Wake_up_port, Reserved,
     Scp_offset, Scp_base)
```

where

Type	is one of the following: 186/51 – for the iSBC 186/51. 550A – for the iSBC 550A.
Wake_up_port	is 0 for the iSBC 186/51. Otherwise, it is the wake-up port of the iSBC 550A.
Reserved	must be set to 0.
Scp_offset	is 0 for the iSBC 186/51. Otherwise, it is the offset value of the SCP address of the iSBC 550A.
Scp_base	is 0 for the iSBC 186/51. Otherwise, it is the base value of the SCP address of the iSBC 550A.

7.4.5 Master_PIC

This macro specifies the type of programmable interrupt controller (PIC) used. iNA currently supports two PICs: the 8259a and the PIC in the 80130. This macro specifies the master PIC in the system. This macro has the following form:

```
%Master_PIC (Type, Base_port, Increment, Buffered)
```

where

Type	is the type of programmable interrupt controller, 80130 or 8259a.
Base_port	is the port address of the PIC.
Increment	is a value used to specify the other ports of the PIC. These ports will be at I/O addresses having the following form: Base_port + (i * Increment) where i is a non-negative integer. Increment will usually have the value 1 or 2.
Buffered	has the following values: Y – if the PIC supports vectored (cascaded) interrupts. N – if the PIC does not support vectored interrupts.

7.4.6 Data_link_interrupt

This macro is used to configure the type of data link interrupts used in the system. Although the form of the macro includes the specifications for either a master or a slave interrupt, the interrupt must be configured as a master. The macro has the following form:

```
%Data_link_interrupt (Int_type, Int_value, Int_level)
```

where

Int_type — specifies the way the data link controller interrupts the CPU:

0 – level mode interrupts
1 – memory-mapped interrupts
2 – I/O-mapped interrupts

This value must be 0 if the 186/51 is used.

Int_value — depends on the value of Int_type, as follows:

Int_type = 0 – Int_value must be 0.

Int_type = 1 – Int_value is the upper 16 bits of the memory address for the interrupt. The lower 4 bits are taken to be 0.

Int_type = 2 – Int_value is the port address.

Int_level — specifies the interrupt level according to the following code:

bits 15–7 — are 0.

bits 6–4 — specify the master level (0–7) of the interrupt.

bit 3 — has the following values:

0 – The interrupt has a slave level, bits 6–4 specify the master level, and bits 2–0 specify the slave level.

1 – The interrupt has a master level, and bits 6–4 specify the entire level number.

bits 2–0 — specify the slave level of the interrupt if bit 3 is 0.

7.4.7 System_configuration_pointer

This is the only optional configuration macro. It is used only with the iSBC 186/51. This macro specifies the size of the system data bus and the location of the intermediate system configuration pointer (ISCP) of the 82586 controller. The format for this macro is as follows:

```
%System_configuration_pointer (Sys_bus, Iscp_low, Iscp_high)
```

where

Sys_bus — is one of the following:

0 – 8-bit system data bus
1 – 16-bit system data bus

Iscp_low — specifies the lower 16 bits of the ISCP.

Iscp_high — specifies the higher 8 bits of the ISCP.

7.4.8 Sample Configuration

The ROM-based NMF configuration file, CNFG.A86, contains macro calls that configure the NMF, data link, and board parameters. Following is an example of this configuration file:

```
NAME NMF_ROM_CNFG

$INCLUDE (:Fn:CNFG.A86)
%Master_PIC (80130, 0E0H, 2, Y)
%Boot_server_multicast_address
  (1, 0AAH, 0, 0FFH, 0FFH, 0FFH)
%Host_id (0FFH, 0FFH, 0FFH, 0FFH, 0FFH, 0FFH)
%Data_link_type (IEEE802)
%Board_type (ISBC 186/51, 0, 0, 0, 0)
%Data_link_interrupt (0, 0, 38H)

END
```

7.5 Linking and Locating the ROM-Based NMF

To prepare the ROM-based NMF for burning into EPROM's, follow this procedure:

1. Copy the diskette containing the ROM-based NMF into the directory NMFROM on the iRMX 86 system Winchester.
2. Write the user code.
3. If you use the file AUTBOT.P86 and have the iSBC 186/51, include the code to initialize the 80186 chip select lines in the file AUTBOT.P86.
4. Compile the user code with the PL/M-86 compiler controls COMPACT, ROM, and OPTIMIZE(3).
5. Modify and assemble the configuration file CNFG.A86.
6. The submit file ROM.CSD, which links and locates the ROM-based NMF, may need modification. If user code is included, the user code object file should replace AUTBOT.OBJ in the linker invocation. In addition, the BOOTSTRAP control may be included in the locator invocation.
7. Link and locate the ROM-based NMF by typing the following:

   ```
   SUBMIT NMFROM/ROM.CSD (Location_of_data,
         Location_of_code)
   ```

   where

   Location_of_data is the address where the data is to be located.

   Location_of_code is the address where the code is to be located.

   The LINK86 warning message "Group Enlarged" is acceptable. LOC86 warning messages 38 and 65 are also acceptable.
8. Burn the file ROM into the EPROMs.

CHAPTER 8
iRMX™ 86 SYSTEM GENERATION

8.1 Overview

To run on the iRMX 86 operating system, iNA 960 requires an environment that includes the iRMX 86 nucleus, an 8086 or 80186 CPU, an 82586-like Ethernet controller, and a dedicated timer. For example, iNA 960 can be configured to run on an iSBC 186/51 or an iSBC 86/330 with a 550FW conversion kit as the data link controller. In this configuration, the iNA 960 communications software runs as a first-level job on the iRMX 86 nucleus.

The iNA 960 software must be configured according to the underlying hardware. For an iSBC 186/51 environment, iNA 960 has a set of default configuration files that can be used directly. However, for the iSBC 86/330 with the 550 firmware kit, and for any other hardware setups, the configuration files must be modified.

In addition to configuring iNA 960 for the hardware, the data link layer, the transport layer, and network management must be configured. This procedure includes provisions for selecting several optional functions. These optional functions are the user data link, the transport normal virtual circuit services, the transport expedited services, the transport datagram services, the network management functions, and the bootstrap loader-server.

This chapter describes the procedure for configuring the hardware and software. After configuring iNA 960, the COMM job can be used in the same manner as other first-level user's jobs. (See the *iRMX 86 Configuration Guide* for a description of the configuration steps).

The end of the chapter contains a section describing the procedure for initializing the iAPX 186 microprocessor in iRMX 86 compatible mode.

8.1.1 iNA 960 System Generation Procedure

The iNA 960 software is configured and the communication system is generated by the following steps:

1. Prepare the hardware environment.
2. Load the iNA 960 software modules onto the development system.
3. Configure iNA 960 by modifying the following configuration files:

RMXCFG.A86	Configures the iNA 960 base system.
BUFCFG.A86	Configures the iNA 960 buffer usage.
DLCFG.A86	Configures the data link layer.
NMLCFG.A86	Configures network management functions.
TLCFG.A86	Configures the transport control layer.

4. Select those optional functions required by the system. The optional functions are the user data link interface, the transport datagram services, the transport normal virtual circuit services, the transport expedited services, the network management functions, and the bootstrap loader-server.
5. Link the configuration files with COMM.LNK and any other necessary iRMX 86 library files. Locate the resulting COMM job.
6. Use ICU to configure the COMM job as one of the first-level user jobs.

8.1.2 Preparing the Hardware

The iNA 960 software modules include default configuration files. These files configure the software for an iSBC 186/51 hardware environment. In addition, the interrupts are assumed to be configured as follows:

iAPX 186 Timer 1 OUT ⟷ 80130 IR4 Input (E78–E43)
82586 INT ⟷ 80130 IR0 Input (E39–E47)

For an 86/330 with 550FW environment or for any other hardware configuration, the configuration files need certain modifications. These changes are described in the following sections.

8.1.3 Transferring the iNA 960 Files

On an iRMX 86-based development system such as the 86/330, the iNA 960 software should be transferred to the development system Winchester disk. To do this, insert the iNA 960 Object - COMM diskette into disk drive 0 and type the following:

```
SUBMIT :FD0:CSD/INSTALL.CSD (:logical_device:)
```

where

logical_device is the logical device name of the destination file system. It is best to use the system drive :SD: for the logical device name.

The submit file then creates the directory :*logical_device*:COMM and copies the iNA 960 Communication system directories and files to the COMM directory. If a directory named COMM already exists on the destination file system, either rename it or change the name of the target directory in the submit file.

WARNING

The default configuration and submit files described in this chapter assume that the iNA 960 Communication system resides in :SD:COMM. If the system is installed on a different directory or if you are an ISIS user, the configuration and submit files must be changed to reflect the actual locations of the files they reference. For the configuration files this means the INCLUDE statements must be changed. For submit files, the pathname of each file in the link list must be changed.

8.2 Configuration

Configuring the iNA 960 software consists of configuring the base hardware and the buffer usage, and also configuring the operating parameters of the data link layer, transport layer, and the NMF. Default configuration files are included; they may be used directly or may be edited to reflect the desired configuration. Also included is a submit file that assembles, links, and locates the communication system.

8.2.1 Configuring the Base System

For internal timing, the iNA 960 software requires a hardware timer such as the iAPX 186 on-chip timer, an 8253, or something compatible. The default configuration file RMXCFG.A86 configures the system for the iAPX 186 on-chip timer. If

the 8253 is used or if timer parameters other than the default are needed, a different hardware timer configuration file must be created. This file consists of a single call of the following macro:

```
%COMM_TIMER (Type, Base_port, Timer, Level, Rate)
```

where

Type	is the type of hardware timer used: 8253 or 80186.
Base_port	is the port address of the programmable interrupt timer. This address depends on the timer, as follows: 80186 – The 80186 control block base address (0FF00H on reset). 8253 – The port address of timer 0.
Timer	is the timer number (0–2) of the timer used by iNA 960.
Level	is an encoded value for the interrupt level of the interrupt controller to which the timer is connected. This value takes the following form: x8H – For master interrupt levels M0–M7 (where x can be from 0 to 7). yzH – For slave interrupt levels 00–77 (where y and z can be from 0 to 7)
Rate	is the frequency in KHz of the input clock to the selected timer. This value must be between 10 and 2500 (10 KHz–2.5 MHz).

For the iSBC 186/51 user the default configuration file RMXCFG.A86 provides proper timer configuration. This file has the following form:

```
NAME CONF.A86
$INCLUDE (:SD:COMM/CONFIG/RMXCFG.MAC)
%COMM_TIMER(80186, 0FF00H, 1, 48H, 1500)
END
```

Here RMXCFG.MAC contains the definitions of the macros used for configuring the system.

8.2.2 Configuring the Buffer Usage

The iNA 960 software allows the user to select the number of buffers used by the internal transmitter and receiver. Thus, the software can be configured to make optimum use of memory. The following entities are available for configuration:

RXBD	*receiver buffer descriptor* – Each RXBD takes a 142-byte block of memory, including 128 bytes for the buffer.
TXBD	*transmitter buffer descriptor* – Each TXBD takes a 1520 byte block of memory, including 1500 bytes for the buffer.
RPD	*receive packet descriptor* – Each receive packet, regardless of length, uses one RPD. Each RPD takes a 36-byte block of memory.
ICB	*internal command block* – The ICBs are resources internal to iNA 960. They are used for passing requests internal to the iNA 960 software. Each ICB takes 36 bytes of memory.

The following macro is used to allocate memory for the above entities:

```
%COMM_BUFF (N_rxbd, N_txbd, N_rpd, N_icb)
```

where

N_rxbd	is the number of RXBDs. The minimum number of RXBDs required for receiving a full length *IEEE 802* packet is 12. However, the more RXBDs specified, the more back-to-back receiving can be achieved (providing enough RPDs are specified).
N_txbd	is the number of TXBDs. Each TXBD can send a full length *IEEE 802* packet. Setting N_txbd to 2 provides full transmit throughput. A larger value does not increase the transmit throughput.
N_rpd	is the number of RPDs. N_rpd, together with N_rxbd, determines the number of back-to-back receiving of packets.
N_icb	is the number of ICBs. For the current release of the iNA 960 software, this number must equal N_txbd.

For iSBC 186/51 users the default configuration file BUFCFG.A86 provides adequate initialization of the buffers. This file has the following form:

```
NAME BUFFCONF
$INCLUDE (:SD:COMM/CONFIG/BUFCFG.MAC)
%COMM_BUFF (144, 2, 20, 2)
END
```

Here, BUFCFG.MAC contains the macro definition of COMM_BUFF. COMM_BUFF is the buffer configuration macro described above.

The allocation of the number of internal receive buffers via the RXBD and RPD parameters has a major impact on transport layer data throughput performance. The critical dependency is the number of 1500-byte transport packets (TPDUs) that can be buffered at one time without running out of internal receiver buffer resources.

For example, for the default 186/51 configuration (144 128-byte BDs and 20 RPDs), a maximum of twelve 1500-byte TPDUs can be buffered at one time. For optimum performance, the number of TPDUs specified here determines the proper maximum window size parameters that must be used when configuring the transport layer.

8.2.3 Configuring the Data Link Layer

The data link software must be configured for the particular hardware environment. For instance, all of the following are specified during data link configuration:

- Number of data links
- Protocol, controller class, and controller subclass
- Interrupt parameters
- Addressing information
- Channel transmission rate
- Maximum number of multicast addresses
- The configuration array used to initialize the 82586

Note that some operating parameters are fixed, such as number of data links, protocol, and controller class. These parameters are included for future expansion.

The default configuration file DLCFG.A86 has the following form:

```
NAME DLCFG
$INCLUDE (:SD:COMM/CONFIG/DLCFG.MAC)
%Dl_ctrl (1)
%Dl_int (08H)
%Dl_signal (2, 0C8H)
%Dl_scp_addr (6H, 0FFFFH)
%Dl_iscp_addr (0, 0FFH)
%Dl_host_id (6, 0F0H, 0F2H, 0F4H, 0F6H, 0F8H, 0FAH)
%Dl_internal
END
```

See Chapter 3 for a description of the data link definition macros and their associated parameters.

8.2.4 Configuring the Transport Control Layer

The transport control layer has a large number of configuration parameters that customize the transport layer to a particular implementation. These parameters can be split into the following parameter groups:

- Transport address limits
- Network layer interface parameters
- Transport data base parameters
- Client request default parameters
- Internal negotiation default parameters
- Retransmission timer parameters
- Flow control parameters

Chapter 5 contains descriptions of all of these parameter groups and shows the format for the transport control layer configuration file. Following is the default transport configuration file, TLCFG.A86:

```
NAME TLCFG
$INCLUDE (:SD:COMM/CONFIG/TLCFG.MAC)
%Transport_address_limits (12, 2, 2)
%Network_layer_interface (1)
%Connection_limits (21, 2)
%Datagram_structures (9, 0, 0)
%Internal_request_blocks (5, 2, 3, 2)
%Client_request_defaults (1, 1800H, 8242H)
%Internal_negot_options (0BH, 3, 7)
%Retran_timer (10000, 1000)
%Closing_abort (80H)
%Inactivity_timer (300000, 8)
%Window_size (0FH, 0FH, 1)
%Open_window_timer (10000, 8)
END
```

8.2.5 Configuring the Network Management Facility

The user has many options when configuring network management. In particular, configuration may include a boot server that can down-load remote systems. Chapter 6 describes the macros, parameters, and options available to the user when creating the network management configuration file.

The completed network management configuration file resembles the following default configuration file, NMLCFG.A86:

```
NAME NML_CNFG_MACROS
$INCLUDE (:SD:COMM/CONFIG/NMLCFG.MAC)
%BOOT_SERVER_MULTICAST_ADDRESS
      (1, 0AAH, 0, 0FFH, 0FFH, 0FFH)
%NML_CNFG (3, 1)
; MAX_NODES (10)
; MAX_SIMULTANEOUS_BOOTS (10)
; CLASS_CODE_INFO (:SD:COMM/CONFIG/CC_INFO, 200)
; DEV_INFO_BLOCK (WD0, IW0, NAMED)
END
```

Note that in this default NMF configuration, the boot server is not selected.

8.3 Selecting Optional Functions

Before linking and locating the configuration files, the optional functions to be included in the COMM system must be selected. Table 8-1 lists the files that provide these options.

The default submit file COMM.CSD (see next section) includes TL.LNK and TLVC.LNK. To include any other configuration files, edit COMM.CSD to add the desired files to the link list.

8.3.1 Data Link Options

At link time, the COMM job can be configured to include user data link services. To do this, include the file LNK/EDL.LNK in the COMM.CSD submit file link list.

Table 8-1. Optional Modules

Filename	Code Size	Option
LNK/EDL.LNK	4K	User data link interface.
LNK/TL.LNK	3.3K	Transport layer core.
LNK/TLDG.LNK	2.9K	Transport datagram functions.
LNK/TLVC.LNK	15.4K	Transport normal virtual circuit service.
LNK/TLEX.LNK	1.9K	Transport virtual circuit expedited services.
LNK/BTSRV.LNK	4.7K	Bootstrap loader server.
LNK/NMF.LNK	6K	Network management functions. Subsets of the NMF can be configured.

NOTE

Always include the data link configuration file, DLCFG.OBJ, in the COMM.CSD submit file whether or not the user data link is required.

8.3.2 Transport Layer Options

At link time, the COMM job can be configured using several options for the transport layer. To configure the transport options, select from the following modules:

LNK/TL.LNK	– Transport layer core
LNK/TLVC.LNK	– Transport normal virtual circuit services
LNK/TLEX.LNK	– Transport virtual circuit expedited services
LNK/TLDG.LNK	– Transport datagram services
CONFIG/TLCFG.OBJ	– Transport configuration file

The user has the following options:

1. No transport layer services are configured into the COMM job. For example, the COMM job may support only external data link services. In this configuration, do not link in any of the above modules.
2. Only the normal virtual circuit transport service is configured into the COMM job. This is the default system provided with the delivery diskette. In this configuration, link in the modules TL.LNK, TLVC.LNK, and TLCFG.OBJ.
3. Both the normal and expedited virtual circuit services are configured into the COMM job. For this configuration, link in the modules TL.LNK, TLVC.LNK, TLEX.LNK, and TLCFG.OBJ.
4. Only the transport datagram services, without any virtual circuit services, are configured into the COMM job. In this configuration, link in the modules TL.LNK, TLDG.LNK, and TLCFG.OBJ.
5. All transport services, including normal, expedited, and datagram services, are configured into the COMM job. For this configuration, link in all of the above transport modules.

8.3.3 NMF Options

The COMM job can be configured to include a subset of the network management services and to include the boot server functions. Following are the NMF modules:

LNK/NMF.LNK	– NMF services
LNK/BTSRV.LNK	– Boot server services
CONFIG/NMFCFG.OBJ	– NMF configuration file

The user has the following options:

1. No NMF services are configured into the COMM job. In this configuration, do not link in any of the above modules.
2. Only the NMF services are configured into the COMM system. For this configuration, link in the modules NMF.LNK and NMFCFG.OBJ.
3. In addition to the NMF services, the boot server is configured into the COMM job. For this configuration, link in all of the above modules.

8.4 Linking and Locating the Configuration Files

Once the configuration files are customized, the corresponding object files are linked with the COMM job and with necessary library files. The result is then located to match the user's implementation. If any of the optional functions are used, first add the necessary files to the link list in COMM.CSD, then perform the following:

```
>SUBMIT :SD:COMM/CSD/COMM.CSD (start_address)
```

where

start_address is the starting address of the COMM job.

COMM.CSD assembles the configuration files, links the selected modules, and locates the COMM job at the specified address. The default COMM.CSD takes the following form:

```
ASM86 :SD:COMM/CONFIG/RMXCFG.A86, &
      :SD:COMM/CONFIG/BUFCFG.A86, &
      :SD:COMM/CONFIG/DLCFG.A86, &
      :SD:COMM/CONFIG/TLCFG.A86, &
      :SD:COMM/CONFIG/NMLCFG.A86, &

LINK86 :SD:COMM/LNK/COMM.LNK, &
       :SD:COMM/LNK/TL.LNK, &
       :SD:COMM/LNK/TLVC.LNK, &
       :SD:COMM/CONFIG/RMXCFG.OBJ, &
       :SD:COMM/CONFIG/BUFCGF.OBJ, &
       :SD:COMM/CONFIG/DLCFG.OBJ, &
       :SD:COMM/CONFIG/TLCFG.OBJ, &
       :SD:COMM/LNK/AIMRMX.LIB &
       :SD:COMM/LNK/COMM.LIB &
TO :SD:COMM/LNK/COMMCF.LNK &
     NOCM NOLI NOMAP NOSB &
     NOPUBLICS EXCEPT (COMMENTRY, CQERRORCODE)

LOC86 :SD:COMM/LNK/COMMCF.LNK &
TO :SD:COMM/LNK/COMM &
       ORDER (CLASSES (CODE, ENTRYPTS, DATA, STACK)) &
       SEGSIZE (STACK (0)) &
       ADDRESSES (CLASSES (CODE (%0))) &
       NOINITCODE, NOLINES, NOCOMMENTS, NOSYMBOLS, &
       OBJECTCONTROLS (NOLINES, NOCOMMENTS, NOPUBLICS, &
       NOSYMBOLS)
```

See the *MCS-86 Utilities User's Guide* for descriptions of the LINK86 and LOC86 commands and the associated command options.

8.5 Configuring the COMM Job into iRMX™ 86

After linking and locating the COMM job, the iRMX 86 ICU can be used to include the COMM job into the iRMX 86 system, just as with the first-level user's job. Following are the suggested parameters:

```
User Job
(ODS) Object Directory Size [0 - 0FF0H]                    0003H
(PMI) Pool Minimum [20H - 0FFFFH]                          0600H
```

```
(PMA) Pool Maximum [20H - 0FFFFH]                          FFFFH
(MOB) Maximum Objects [1 - 0FFFFH]                         FFFFH
(MTK) Maximum Tasks [1 - 0FFFH]                            FFFFH
(MPR) Maximum Priority [0 - 0FFH]                          0000H
(AEH) Address of Exception Handler [CS:IP]           0000H:0000H
(EM)  Exception Mode [Never/Prog/Environ/All]              Never
(PV)  Parameter Validation [Yes/No]                           No
(TP)  Task Priority [0 - 0FFH]                              0130
(TSA) Task Start Address [CS:IP]                     start_address
(DSB) Data Segment Base [0 - 0FFFFH]                       0000H
(SSA) Stack Segment Address [SS:SP]                  0000H:0000H
(SS)  Stack Size [0 - 0FFFFH]                              0400H
(NDX) Numeric Processor Extension Used [Yes/No]               No
```

where

start_address is the address of the label "CommEntry" from the locate map (COMM.MP2).

Refer to the *iRMX 86 Configuration Guide* for a description of the procedures to configure the first-level user job.

NOTE

User tasks that use the COMM interfaces must have lower priorities (higher numeric values) than the init task priority parameter of the JOB macro.

8.6 COMM Job Requirements

In order to run the COMM job, the Nucleus must be configured with the following system calls used by the COMM job:

```
RQ$CATALOG$OBJECT
RQ$CREATE$COMPOSITE
RQ$CREATE$EXTENSION
RQ$CREATE$MAILBOX
RQ$CREATE$SEGMENT
RQ$CREATE$SEMAPHORE
RQ$CREATE$TASK
RQ$DELETE$COMPOSITE
RQ$DELETE$SEGMENT
RQ$END$INITTASK
RQ$ENTER$INTERRUPT
RQ$EXIT$INTERRUPT
RQ$GET$TASK$TOKENS
RQ$RECEIVE$MESSAGE
RQ$RECEIVE$UNITS
RQ$RESUME$TASK
RQ$SEND$MESSAGE
RQ$SEND$UNITS
RQ$SET$INTERRUPT
RQ$SET$OS$EXTENSION
RQ$SIGNAL$INTERRUPT
RQ$WAIT$INTERRUPT
```

If the boot server is configured into the COMM job, both the Nucleus and BIOS must be configured into the system. In addition to the above system calls, the following system calls must be included in the Nucleus configuration:

```
RQ$DELETE$MAILBOX
RQ$GET$PRIORITY
RQ$GET$TYPE
RQ$LOOKUP$OBJECT
```

If the boot server is configured into the COMM job, the following system calls must be included in the BIOS configuration:

```
RQ$A$ATTACH$FILE
RQ$A$CLOSE
RQ$A$DELETE$CONNECTION
RQ$A$GET$CONNECTION$STATUS
RQ$A$GET$FILE$STATUS
RQ$A$OPEN
RQ$A$PHYSICAL$ATTACH$DEVICE
RQ$A$READ
RQ$A$SEEK
RQ$CREATE$USER
RQ$SET$DEFAULT$PREFIX
RQ$SET$DEFAULT$USER
```

8.7 Initializing the iAPX 186

The iNA 960 includes code to initialize the iSBC 186/51 in iRMX 86 compatible mode. This initialization is required only for iRMX 86 release 5. iRMX 86 release 6 and subsequent releases perform their own initialization.

If the SDM86 is used, the initialization code is not needed. Following is a description of the initialization procedure. See the *iAPX 186 Data Sheet* for a description of the registers used in the initialization code.

The initialization code performs the following actions:

- Sets the relocation register to 0FF00H. This, in particular, sets bit 14, selecting the iRMX 86 compatible mode. The internal interrupt controller then operates as a slave.
- Loads the memory and peripheral chip-select registers UMCS, LMCS, MPCS, MMCS, and PACS. This loading establishes the sizes and address boundaries of the chip-selectable memory and peripheral blocks. In addition, this loading selects the WAIT state timing and READY generation logic for each of the chip-select lines.
- Branches to the specified entry address (for example, the entry address of the ROOT job.)

iAPX 186 initialization is activated by a call of the following macro:

```
%INIT_186_CF (UMCS, LMCS, MPCS, MMCS, PACS,
      Branch_offset, Branch_base)
```

where

UMCS	is a word that initializes the UMCS (upper memory chip-select) register. To leave the register alone, specify 0.
LMCS	is a word that initializes the LMCS (lower memory chip-select) register. To leave the register alone, specify 0.
MPCS	is a word that initializes the MPCS register. This register selects the mid-range memory block size and controls the operation of the peripheral chip-select. To leave this register alone, specify 0.
MMCS	is a word that initializes the MMCS register. This register sets the base address of the mid-range memory block. To leave this register alone, specify 0.
PACS	is a word that initializes the PACS register. This register sets the base address of the peripheral chip-select block. To leave this register alone, specify 0.
Branch_offset	is a word that specifies the offset of the address to be branched to after initialization.
Branch_base	is a word that specifies the base of the address to be branched to after initialization.

The macro INIT_186 sets the relocation register, loads the memory and peripheral chip-select registers with the parameter values, then vectors to the location specified by the Branch parameters. Normally, this address is the entry address of the ROOT job. However, it can also be the entry address for a user module. This module can be used, for example, to perform a diagnostics function or to run an initialization routine specific to the user's system.

NOTE

Only a specific set of values for the chip-select registers is valid. See the *iAPX 186 Data Sheet* for information on the bit fields of these registers.

The iAPX 186 initialization routine consists of a single call of the INIT_186_CF macro. The iNA 960 software package includes a default iAPX 186 initialization file, IN186.A86. This file takes the following form:

```
NAME INITIALIZE_186
$INCLUDE (:SD:COMM/INIT/IN186.MAC)
%INIT_186_CF (0, 0, 80BBH, 0, 003H, Offset, Base)
END
```

To modify IN186.A86, assemble it and generate the object code file, IN186.OBJ. The initialization code is then linked and located by typing:

```
>SUBMIT :SD:COMM/INIT/INIT.CSD (start_address)
```

Here, *start_address* is the address where the initialization code is to be located. Notice that INIT.CSD has a BOOTSTRAP LOC86 control. Here, a long jump to *start_address* is placed at 0FFFF0H when the module is loaded.

CHAPTER 9
COMPONENT SUPPORT INTERFACE

9.1 Overview

The iNA 960 communications software can run under iRMX 86 as described in the previous chapter. In addition, iNA 960 can run as a standalone package. When operating as a standalone system, iNA uses the *component support interface* as detailed in this chapter.

iNA runs independently of the host system, on a processor that is different from the host processor. The host passes commands to iNA by formatting request blocks and delivering them to iNA to execute. Following execution, iNA passes the request block back to the host.

iNA is a hardware-independent software package. The hardware-dependent features are contained in a separate and configurable *hardware-dependent module* (HDM). iNA is also independent of the mechanism that is used to pass request blocks to and from the iNA software. The user has three choices for this *message delivery mechanism* (MDM): the *MULTIBUS interprocessor protocol* (MIP), the *base control block* (BCB) interface, and the *user-supplied* interface. For the MIP and the BCB, the user needs to write only some initialization routines. The user-supplied interface must be written entirely by the user.

This chapter describes the three types of message delivery mechanisms. In addition, the hardware environment required to run the component support interface is detailed, including the steps needed to configure the hardware-dependent module. The chapter concludes with a description of the procedure for generating the COMM system from the component modules.

9.2 Model of Operation

Figure 9-1 illustrates the component support interface model of operation. The component support interface consists of the following modules:

MDM – The message delivery mechanism running on the host processor.

iMDM – The message delivery mechanism running on the iNA processor.

HDM – The hardware-dependent module.

Commands are passed from user tasks to iNA in the form of request blocks, via the two message-delivery mechanisms. Each command is executed by iNA, the request block is modified to reflect the result of the execution, and then the request block is dispatched back to the user task. The following is the sequence of events:

1. The host formats a request block in system memory.
2. The host requests the MDM running on the host system to deliver the request block to the MDM running on the iNA processor. The host must not modify the request block after this step.

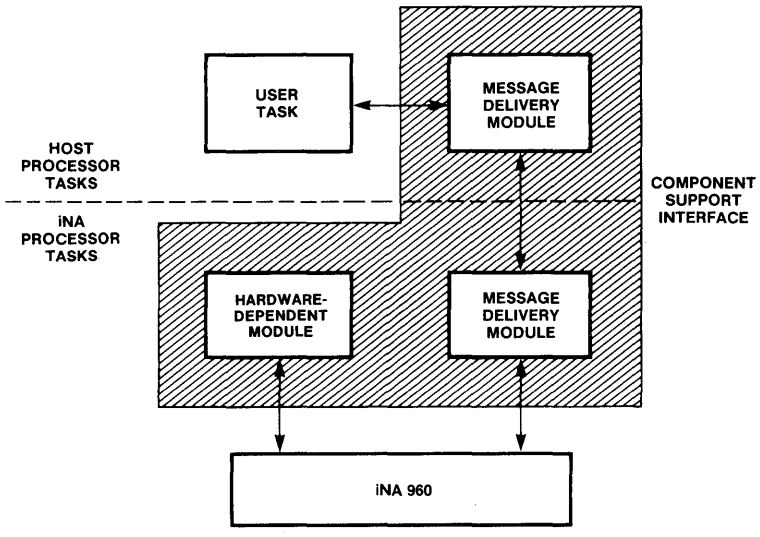

Figure 9-1. The Component Support Interface 122193-7

3. The MDM on the host processor accepts the request block. It uses the return entity field of the request block to store information about the source of the request block. This is used to identify the source task when returning the request block to the same task, after it has been processed by iNA.
4. The MDM on the host generates a channel attention to interrupt the MDM running on the iNA processor.
5. The channel attention interrupt service routine on the iNA processor services the interrupt. Here, the MDM on the host processor and the MDM on the iNA processor communicate following a predefined protocol.
6. The MDM on the iNA processor accepts the request block, if it has the resources to handle it.
7. The MDM on the iNA processor delivers the request block to iNA to process. In order to access the request block and the buffers specified in them, iNA calls a set of address conversion routines.
8. After processing the request block, it is sent back to the original user task in a similar fashion.

9.3 Hardware Environment

Figure 9-2 shows the hardware environment used by the iNA 960 communication system. The 80186 CPU may be replaced by an 8086 CPU, an 8253 PIT and an 8259 Programmable Interrupt Controller (PIC). The 82586 Ethernet controller may be replaced by a 550 FW board set. The system bus does not need to be a MULTIBUS, as user-written subroutines account for variations in buses and hardware.

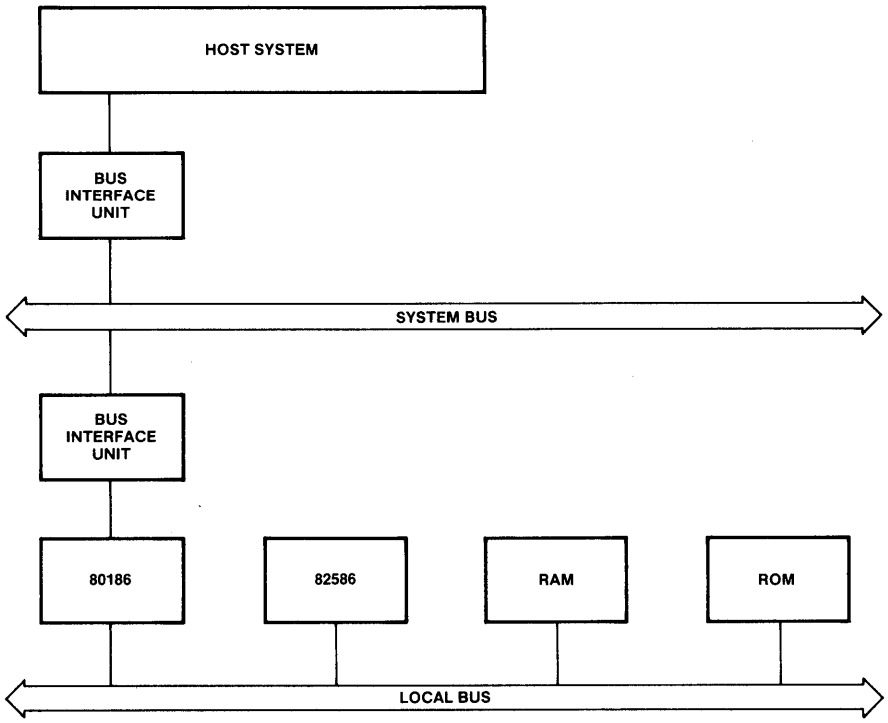

Figure 9-2. iNA 960 Hardware Environment 122193-8

9.4 iNA 960 Address Space

It may not be possible for the iNA processor to access all of system memory at the same time. Different boards have different address spaces. For example, a hypothetical iNA processor may be able to access 256 Kbytes of system memory at any time, whereas system memory can be 1 Mbyte for a 8086-based system, and 16 Mbytes in a 80286-based system.

Request blocks and the buffers specified in them are built by the host in system memory. Due to the different addressing capabilities of the various processors, it may not be possible for iNA to have access to all request blocks and associated buffers at the same time. In any given hardware configuration, there are two possibilities:

- All request blocks and buffers specified in them are accessible by iNA at all times.
- Request blocks and buffers specified in them are not always accessible by iNA.

9.4.1 Restricted Addressing

Restricted addressing is the case where all request blocks and buffers are accessible to the iNA processor at all times. Special routines to manipulate the address space of the iNA processor are not needed. For example, if the iNA processor can access 256 Kbytes of system memory, then all request blocks and the buffers specified in them must reside in this 256-Kbyte block of memory.

9.4.2 Unrestricted Addressing

In the case of unrestricted addressing, request blocks and buffers are scattered all over system memory. The iNA processor must, however, have access to all request blocks passed to the iNA software. Therefore, the MDM on the iNA processor first copies the request block to local memory (called the onboard request block). The MDM then delivers the onboard request block to iNA to process. After the request block has been processed, the MDM copies the onboard request block back to the user's request block.

The iNA MDM maintains a pool of onboard request block buffers. The size and number of these buffers is set during configuration. Whenever a request block is to be delivered to iNA, a buffer from the onboard request block buffer pool is obtained and a copy of the request block is made.

Note that buffers are not copied to local memory. iNA accesses buffers by calling a user-supplied routine (see Section 9.9) that adjusts the address window of the iNA processor to include the desired buffer.

The following overhead is associated with the unrestricted addressing of request blocks and buffers:

- A pool of onboard request block buffers must be kept and maintained.
- Two sets of copies are performed for each request block delivered to iNA.
- The address space of the iNA processor is continuously adjusted.

Although the overhead due to the last point is minimal, the other points require a substantial amount of memory, and considerable processing time. These overheads are unavoidable.

9.5 Message Delivery Mechanism

The message delivery mechanism consists of two tasks running on the host and iNA processors as shown in Figure 9-1. The MDM running on the host is part of the host software and must be written by the user. The MDM running on the iNA processor can be one of two MDMs packaged with the system, or it can also be written by the user. Following are the three types of message delivery mechanisms:

1. *MULTIBUS interprocessor protocol (MIP)* is a set of mechanisms and protocols that provide reliable and efficient exchange of data among tasks executing on various single board computers connected to a common MULTIBUS system bus. MIP solves the problems inherent in a multiprocessor system:
   - Tasks can run on several different processors.
   - Tasks can run on several different operating systems.
   - Boards can use different signaling techniques.
   - Boards can have different memory spaces.
   - Boards can use different addressing schemes for the same shared memory.
   - Tasks can share areas of memory without interfering with one another.
2. *Base control block* (BCB) interface is used in systems that have a single host processor utilizing iNA. It is simpler to use than MIP. However, it is not as efficient.
3. *User-supplied interface* can be used in systems where a user written interface is desired.

9.6 MIP Interface

One of the message delivery mechanisms available to iNA is the MULTIBUS Interprocessor Protocol (MIP). The MIP facility included in the iNA package is an implemetation of the MIP specification. Readers interested in the full specification of this protocol are referred to Appendix E.

The MIP facility isolates user tasks from the complexities of communicating across the MULTIBUS system bus, and is recommended for MULTIBUS systems that have multiple host processors.

MIP facilities support communication among tasks that are executing on different processor boards that are attached to a common MULTIBUS system bus. Each processor board in a MIP system runs a MIP facility. Each MIP facility may be a different implementation of MIP, but adherence to the 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 between 0 and the number of devices in the system (less 1). Figure 9-3 shows an example of a MIP system with a host device and the iNA 960 processor board.

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.

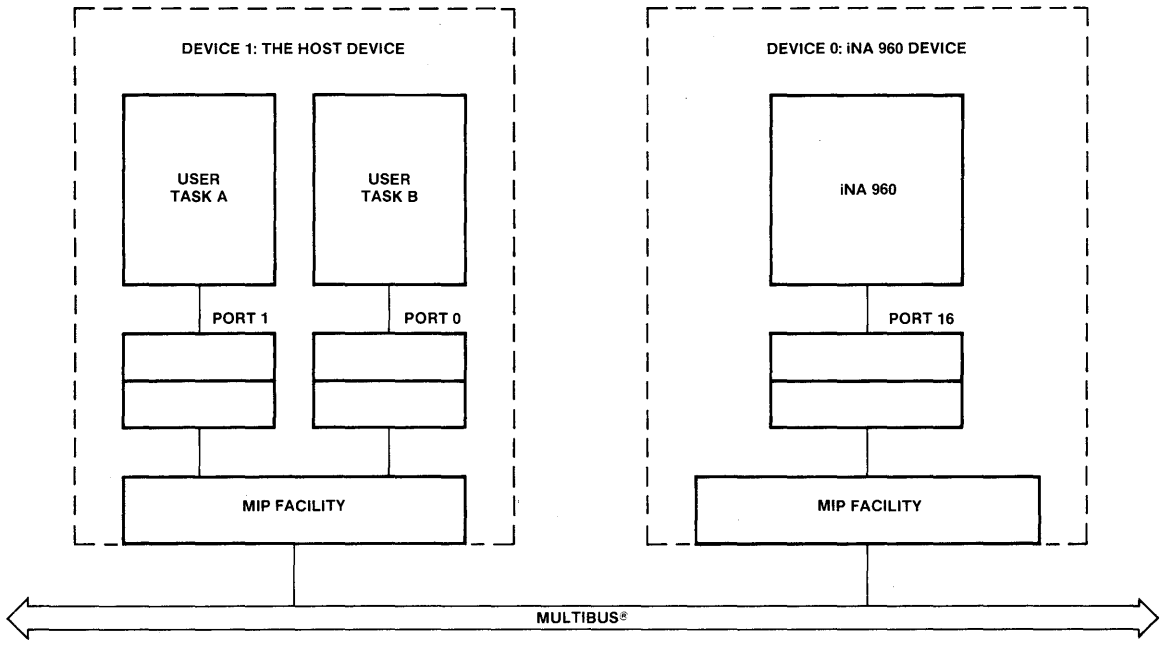

Figure 9-3. MIP Facility with iNA 960 122193-9

Communications are delivered to tasks at *ports*. A port is a logical delivery mechanism that enables delivery in *first-in, first-out* (FIFO) order. The ports at a given device are identified by a *port-id*, a number between 0 and 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. Thus, in the example shown in Figure 9-3, iNA resides on device 0 and receives communications at port 16, that is, socket (00, 16). Similarly, TASK A and TASK B are active at sockets (1,1) and (1,0) respectively.

9.6.1 MIP Initialization Routine

The MIP user must write the initialization code and link it with the rest of the COMM software. This initialization code must do the following:

- It must have a variable called This_device_loc, which is a POINTER that is declared PUBLIC and DATA.
- The memory location addressed by This_device_loc must be initialized with a BYTE value specifying the device id of the iNA processor. It is recommended that the iNA processor be assigned a device id of 0.
- It must have a variable called MIP_device_info_loc, which is a POINTER that is declared PUBLIC and DATA.
- The memory area addressed by MIP_device_info_loc must be initialized with the following:

```
DECLARE MIP_device_info BASED MIP_device_info_loc
      STRUCTURE (
            Device_id         BYTE,
            Status            BYTE,
            RQD_in            DWORD,
            RQD_out           DWORD,
            Reserved          (4) BYTE );
```

where

Device_id	is the device id of the host processor.
Status	must be set to 0FFH.
RQD_in	is the 32-bit absolute address of the request queue descriptor to the iNA processor.
RQD_out	is the 32-bit absolute address of the request queue descriptor from the iNA processor.

- It must initialize both the incoming and outgoing MIP queues.
- It must perform any special hardware initialization required. This may include loading iNA onto the RAM of the iNA processor, if iNA is RAM-based.
- It must follow the PL/M 86 COMPACT model with the ROM option in effect.
- After all of the above, it should jump to the routine Begin_aim. When it does this jump, at least 20 bytes of stack space must be available.

9.6.2 Return_entity Field of a Request Block

The Return_entity field of a request block is 4 bytes long. When sending a request block to iNA, the Return_entity field must be filled by the user. For the MIP message delivery mechanism, the Return_entity field has the following interpretation:

```
DECLARE Return_entity STRUCTURE (
      Port_id          BYTE,
      Device_id        BYTE,
      Reserved         (2) BYTE );
```

where

Port_id	is the port where the request block should be returned after it is processed.
Device_id	is the device id of the host processor. This value must be the same as MIP_device_info.Device_id.
Reserved	is set to 0.

9.6.3 POINTER Fields of a Request Block

All addresses specified in request blocks must be 32-bit absolute addresses. The addresses present in the request queue should also be 32-bit absolute addresses, because the iNA 960 MIP can communicate with just one device. While communicating with this sole device, it assumes an interdevice segment base of 0.

9.6.4 User-Written Routines

In addition to writing the initialization routine, the user has to write the following routines. These routines are described in Section 9.9.

- Sys_to_loc_addr
- Loc_to_sys_addr
- Save_address_space
- Restore_address_space
- Gen_int

9.7 BCB Interface

With the *base control block* interface, commands are given to iNA by placing the address of the request block in a fixed location and then generating a channel attention to the iNA processor. The BCB interface then delivers the request block to iNA to process. Once the request block is processed, the BCB interface returns the request block to the host in a similar fashion.

9.7.1 Base Control Block

The memory location used to store the address of the request block is part of a contiguous block of memory called the *base control block*. In addition to the request block address, the base control block contains fields used by the BCB interface to store

commands and responses used in the implementation of a request block passing protocol. The base control block is 96 bits long and has a format defined by the following PL/M-86 structure:

```
DECLARE BCB_structure STRUCTURE (
            Command                     (16) BITS,
            Command_block_result        (16) BITS,
            Command_block_address       (32) BITS,
            Response_block_address      (32) BITS );
```

Here, Command and Command_block_address are written to by the host, and read from by the BCB interface. These fields are used to issue a command to (or to acknowledge a channel attention from) the BCB interface, as described in the next section. If the command field contains a Start directive, the 32-bit absolute address of the request block to be sent to iNA, must be loaded into the Command_block-_address field before issuing the channel attention.

Similarly, Command_block_result and Response_block_address are written to by the BCB interface, and read by the host. Command_block_result is used by the BCB interface to indicate the status of an issued command. The Response_block_address field is used to pass the 32-bit absolute address of a request block after it has been returned from iNA.

Operation of the two sides of the message delivery mechanism is asynchronous. In particular, the BCB interface will accept new request blocks before it has completed processing old ones, up to a configurable limit. The order that request blocks are returned can be different from the order that they were issued. In addition, if the BCB interface has issued a channel attention to the host, no more request blocks are returned until the host has acknowledged the interrupt.

9.7.2 Command Field

Whenever the host generates a channel attention to the BCB interface, the reason for generating the interrupt is specified in the Command field of the base control block. This field is 16 bits wide and has the following interpretation:

Bit	Mnem	Description
15	Ack	This bit is set by the host to acknowledge a channel attention issued by the BCB interface.
14	Reset	This bit is set by the host to reset iNA. The iNA processor disables interrupts and jumps to location FFFF:0.
13	Start	This bit is set by the host to deliver a request block to the BCB interface. The 32-bit absolute address of the request block must be present in the Command_block_address field.
12-0	-	Unused. Must be set to 0.

The BCB interface interprets the bits in the following manner. If the Ack bit is set, the BCB interface clears the channel attention it had generated. If it had not generated a channel attention, it ignores this bit. The BCB interface then examines the rest of the bits. Not more than one of these bits should be set at any time. If more than one of these bits is set, the BCB interface accepts the command corresponding to the highest bit set (order of priority is Reset, Start). In particular, an Ack can be combined with a Reset or a Start.

9.7.3 Command_block result Field

While the BCB interface is processing the command initiated by the channel attention, it uses the Command_block result field to indicate the status of the command. This field has the following bit assignment:

Bit	Mnem	Description
15	Complete	This bit is set by the BCB interface after it completes processing the channel attention that the host has issued.
14	Busy	This bit is set to 1 by the BCB interface to indicate it is busy processing a channel attention generated by the host. The BCB interface sets it to 1 when it begins processing the channel attention and to 0 upon completion.
13	Status	This bit is valid only if the Start command was issued and is interpreted as follows: 0 – the request block has been delivered to iNA. 1 – the request block has not been delivered to iNA.
12–0	—	Unused.

When bit 13 (the Status bit) is set to 1, it indicates the BCB interface does not have the resources to deliver a request block to iNA. The user should wait for the BCB interface to return some outstanding request blocks before trying to send another request block.

9.7.4 A Protocol Implementation

Apart from conforming to the specifications of the BCB interface, it is up to the user to implement his own BCB interface protocol. A possible implementation to deliver request blocks to iNA is the following:

1. If the Complete bit of the Command_block_response is not 1, wait for the BCB interface to complete processing the previous channel attention.

2. Check the Status bit of the Command_block_response and handle the error if there is one.

3. Load the Command_address field with the address of the next request block to be processed by iNA.

4. Issue a channel attention to the iNA processor.

9.7.5 The BCB Interface Initialization Routine

As part of the initialization process, the BCB interface must be initialized by a user-supplied routine. A typical way to initialize would consist of the host putting a command at a fixed location in memory, and then generating a channel attention. The initialization routine would accept each command and associated parameters

and perform the specified initialization chores. Once initialization is complete, iNA is started. The user's initialization routine is not called again and subsequent channel attentions are processed by the BCB interface.

The user's BCB interface initialization routine should perform the following:

1. It must have a variable named BCB_loc of type DWORD DATA. BCB_loc must be initialized to the 32-bit absolute address of the base control block.
2. Initialize the iNA hardware so that the BCB can be addressed by the iNA software.
3. Call the routine Begin_aim. This will start up the iNA software. The stack size when Begin_aim is called must be at least 20 bytes.

9.7.6 User-Written Routines

In addition to writing the initialization routine, the user has to write the following routines (described in Section 9.9):

- Sys_to_loc_addr
- Loc_to_sys_addr
- Save_address_space
- Restore_address_space
- Gen_int
- Clear_int

9.8 User-Supplied Interface

If the user does not want to use the MIP or BCB interfaces, he can implement his own message delivery mechanism as outlined in this section. To generate the user-supplied interface, the user must write the routines detailed in the next section, as well as the following:

CAINT	This is the channel attention interrupt service routine that is called by iNA upon receiving a channel attention interrupt from the host.
Sendtohost$os	After processing a request block, iNA calls this routine to send the request block back to the host.
Init_msg_delivery_mech	After the iNA software starts running, it calls this routine to initialize the user-written MDM.

In addition, the user must write the initialization code and assemble and link it with the rest of the iNA software.

9.8.1 Initialization

After system reset, iNA and the message delivery mechanism are in an uninitialized state. The user must write, assemble, and link the initialization code to the iNA software. The only requirement for this initialization code is that it ends by jumping to the routine Begin_aim, which starts up the communication system.

9.8.2 CAINT

The routine CAINT is called by iNA upon receiving a channel attention interrupt from the host. This routine is used to prepare the request block for delivery to iNA. CAINT can assume the following:

- All registers have been saved on the stack. They are restored from the stack upon return from this routine.
- The DS register has been initialized with the base of the data segment.
- An end of interrupt has been issued to the interrupt controller.
- Interrupts have been disabled.
- The return_entity field of the request block can be used and interpreted in any way. This field is ignored by iNA.

CAINT must follow these conventions:

- It should not use more than 20 bytes of stack.
- It should follow the PL/M-86 COMPACT model of computation.
- It must be named CAINT. CAINT is declared PUBLIC and is linked with the rest of the COMM system.
- If CAINT changes the address window of the iNA processor, it must restore the window to the original value.
- It must not enable interrupts.
- The request block must be accessible to iNA. If this is not the case, CAINT should copy the request block to an onboard request block. The onboard request block is then delivered to iNA.

To deliver the request block to iNA, CAINT makes a call to the routine Ipsend. This routine has the following format:

```
Ipsend: PROCEDURE (Reserved, Rb_ptr) EXTERNAL;

      DECLARE Reserved     BYTE;      /* Set to 16              */
              Rb_ptr       POINTER;   /* Pointer to request     */
                                      /* block                  */

      /* Ipsend follows the PL/M-86 COMPACT model
         of computation */

END Ipsend;
```

9.8.3 Sendtohost$os

After iNA has completed processing a request block, it calls the routine Sendtohost$os to send the request block back to the host. This routine has the following form:

```
Send$to$host$os: PROCEDURE (Rb_ptr);

      DECLARE Rb_ptr POINTER; /* Pointer to request block */

END Send$to$host$os;
```

This routine must follow these restrictions:

- It must not use more than 20 bytes of stack.
- It must follow the PL/M-86 COMPACT model of computation.
- If it changes the address window of the iNA processor, it must restore the address window to the original value.
- If it changes the state of the interrupt flag, it must restore the original value.

9.8.4 Init_msg_delivery_mech

After iNA starts running, it calls the routine Init_msg_delivery_mech. Any initialization performed by the message delivery mechanism should be present in this routine.

9.9 User-Supplied Routines

Some of the routines that iNA uses depend on the hardware environment. These routines are specified in this section and they must be written by the user. Combine these routines in the file USERRT.LNK and link this file with the COMM system.

The user-supplied routines must be written according to the following conventions:

- They must conform to the PL/M-86 COMPACT model of computation.
- They must have all CONSTANTS in CGROUP. That is, the ROM option must be selected.
- They must not use more than 20 bytes of stack size.
- If a routine changes the state of the interrupt flag, it must restore the original value before returning.

9.9.1 Sys_to_loc_addr

This procedure converts a system address to a local address. If necessary, the address space of the iNA processor is reset so the location specified by the system address is accessible via the local address. This procedure takes the following form:

```
Sys_to_loc_addr: PROCEDURE (Sys_address) POINTER PUBLIC;

DECLARE
      Sys_address     DWORD;

IF Sys_address > 0FFFFFFH
THEN DO; /* larger than 16 Mbytes */
/*
      - mask most significant 8 bits
      - convert least significant 24 bits to a PLM86 POINTER
      - RETURN (PLM86 POINTER) */

ELSE DO; /* smaller than 16 Mbytes */
/*
      - note the parameters pertaining to the current address
        space. If this takes more than 1 assembly instruc-
        tion, then this must be done with interrupts disabled
      - set the address space of the iNA processor so that
        the 24 bit address Sys_address is accessible by iNA
      - RETURN (the PLM86 POINTER that iNA should use to
        access Sys_address) */

END;
END Sys_to_loc_addr;
```

9.9.2 Loc_to_sys_addr

This procedure converts a local address to a system address. The format for this procedure is the following:

```
Loc_to_sys_addr: PROCEDURE (Sys_ptr) DWORD PUBLIC;

DECLARE
      Sys_ptr    POINTER;

/*
      This routine must convert the PL/M 86 pointer Sys_ptr
      to a 32-bit absolute address, and set the most
      significant byte of the double word to 1.
*/

END Loc_to_sys_addr;
```

9.9.3 Save_address_space

This routine saves the current address space of the iNA processor. Here, the address space parameters (see the Sys_to_loc_addr routine) are saved on the stack. When this routine is called, the stack is in a form illustrated by the following PL/M-86 structure:

```
DECLARE Stack STRUCTURE (
      Return_address          WORD,
      Rest_of_stack           (x) WORD );
```

Upon return from this routine, the stack has the following form:

```
DECLARE Stack STRUCTURE (
      Saved_parameters        (y) WORD,
      Rest_of_stack           (x) WORD );
```

The Save_address_space routine has the following form:

```
Save_address_space: PROCEDURE PUBLIC;
/*    save the address space parameters on the stack */
END Save_address_space;
```

9.9.4 Restore_address_space

This routine restores the address space of the iNA processor using parameters that were stored on the stack by the Save_address_space routine. When this routine is called, the stack is in a form illustrated by the following PL/M-86 structure:

```
DECLARE Stack STRUCTURE (
      Return_address          WORD,
      Saved_parameters        (y) WORD,
      Rest_of_stack           (x) WORD );
```

Upon return from this routine, the stack has the following form:

```
DECLARE Stack STRUCTURE (
      Rest_of_stack           (x) WORD );
```

This routine has the following form:

```
Restore_address_space: PROCEDURE PUBLIC;
/*     restore the address space from the parameters on
       the stack */
END Restore_address_space;
```

9.9.5 Gen_int

This routine has the following form:

```
Gen_int: PROCEDURE PUBLIC;
/*    generate a channel attention to the host */
END Gen_int;
```

9.9.6 Clear_int

This routine has the following form:

```
Clear_int: PROCEDURE PUBLIC;
/*    if iNA has generated an interrupt to the host,
      then clear the interrupt */
END Clear_int;
```

9.10 Configuring the Hardware-Dependent Module

The hardware-dependent module must be configured for the timer, interrupt controller, and interrupt controller inputs in the system. This is done with the macros PIT, PIC, and PIC_inputs.

9.10.1 PIT

PIT configures the hardware-dependent module for the type of programmable interval timer (PIT) used. This macro has the following form:

```
PIT(Type, Base_port, Increment, Alc, Frequency)
```

where

Type	is the type of timer used: 8253 or 80186.
Base_port	is a value that depends on the type of timer used. Base_port is interpreted as follows: 8253 – this parameter is the port address of the PIT. 80186 – this parameter is the base of the 80186 control block address. The hardware-dependent module sets the control block in I/O space to this value. The least significant byte of this parameter must be 0.
Increment	is a value used to specify the other ports of the PIT. These ports will be at I/O addresses Base_port + (i × Increment) where i is a non-negative integer. Increment will usually have the value 1 or 2. For the 80186 PIT, this parameter is ignored.
Alc	specifies the timer (0, 1, or 2) used by the hardware-dependent module.
Frequency	is the frequency in Khz, of the PIT.

9.10.2 PIC

This macro configures the hardware-dependent module for the type of programmable interrupt controller (PIC) used. There are currently two supported, the 8259a and the PIC in the 80186. If the 80186 is specified as the PIT, it must also be specified as the PIC. This macro specifies the master PIC in the system. The data link, timer, and host interrupts must all be received at this PIC. This macro has the following form:

```
PIC (Type, Base_port, Increment, Buffered)
```

where

Type — is the type of programmable interrupt controller, 80186, or 8259a.

Base_port — is the port address of the PIC. For the 80186 PIC, this parameter is ignored.

Increment — is a value used to specify the other ports of the PIC. These ports will be at I/O addresses

$$\text{Base\_port} + (i \times \text{Increment})$$

where i is a non-negative integer. Increment will usually have the value 1 or 2. For the 82186 PIC, this parameter is ignored.

Buffered — has the following values:

'Y' if the PIC supports vectored (cascaded) interrupts.

'N' if the PIC does not support vectored interrupts.

9.10.3 PIC_inputs

PIC_inputs is used to specify the inputs to the PIC. This macro has the following form:

```
PIC_inputs (Type, Alc_input, Reserved, Dl_input,
      Dl_edge_vs_level, Ca_input, Ca_edge_vs_level)
```

where

Type — is the type of programmable interrupt controller, 80186, or 8259a.

ALC_input — specifies the pin of the PIC (0-7) that receives interrupts from the timer. If the 80186 timer is used, this parameter is ignored.

Reserved — must be set to 0.

Dl_input — specifies the pin of the PIC (0-7) that receives interrupts from the data link controller.

Dl_edge_vs_level — depends on the interrupt at Dl_input. It has the following value:

0 if the interrupt is edge triggered

1 if the interrupt is level triggered

This parameter is ignored if the 8259 is used.

Ca_input — specifies the pin of the PIC (0-7) that receives interrupts generated by channel attentions from the host.

Ca_edge_vs_level — depends on the interrupt at Ca_input. It has the following value:

0 if the interrupt is edge triggered

1 if the interrupt is level triggered

This parameter is ignored if the 8259 is used.

9.11 Configuring the Message Delivery Mechanism

In addition to configuring the hardware-dependent module, the MIP and BCB interface user must configure the message delivery mechanism. This is done by including a macro call to CBA in the configuration file.

9.11.1 CBA

The CBA macro is used to specify whether or not the request blocks and buffers are accessible by iNA at all times. If not, this macro also specifies the size and number of the onboard request block buffers. This macro has the following form:

```
CBA (Rb_copy, Num_rb_buffers, Rb_buffer_size)
```

where

Rb_copy — specifies whether the request blocks and buffers are accessible at all times. Values are:

- 0 request blocks and buffers are not accessible at all times. Request blocks are copied to onboard request blocks.
- 1 request blocks and buffers are accessible at all times. In this case the remaining parameters are not used. Request blocks are not copied to onboard request blocks.

Num_rb_buffers — specifies the number of onboard request blocks that the message delivery mechanism should maintain. This value will limit the number of commands that the user can give to iNA at the same time.

Rb_buffer_size — is the size of the above buffers. This should be as big as the largest request block to be given to iNA.

If the MIP or BCB interface with the copy option is used for the message delivery mechanism, then Rb_copy must be set to 0. In this case, the maximum number of request blocks that iNA can process at the same time is limited to Num_rb_buffers. Each request block delivered to iNA must be less than Rb_buffer_size bytes in length.

If Rb_copy is 1, then any number of request blocks of any size can be delivered to iNA.

This macro should not be invoked if the user-supplied message delivery mechanism is used.

9.12 Component Support System Generation

The component support interface is configured and the COMM system is generated by the following steps:

1. Choose the message delivery mechanism. The following are the available options:
   - MIP with request blocks copied to iNA memory.
   - BCB interface with request blocks copied to iNA memory.
   - BCB interface with request blocks not copied to iNA memory.
   - A user-written interface.

2. Edit the file :SD:COMM/CONFIG/CMPCFG.A86 to reflect the hardware configuration that supports iNA.
3. The file :SD:COMM/CONFIG/COMM.CSD generates iNA that runs under iRMX 86. Make a copy of this file by giving the following command:

```
>COPY  :SD:COMM/CONFIG/COMM.CSD TO
       :SD:COMM/CONFIG/COMMCMP.CSD
```

WARNING

The default configuration and submit files described in this chapter assume that the iNA 960 Communication system resides in :SD:COMM. If the system is installed on a different directory, or if you are an ISIS user, the configuration and submit files must be changed to reflect the actual locations of the files they reference. For the configuration files, this means the INCLUDE statements must be changed. For submit files, the pathname of each file in the link list must be changed.

The file COMM.CSD has the following form:

```
ASM86  :SD:COMM/CONFIG/RMXCFG.A86,  &
       :SD:COMM/CONFIG/BUFCFG.A86,  &
       :SD:COMM/CONFIG/DLCFG.A86,  &
       :SD:COMM/CONFIG/TLCFG.A86,  &
       :SD:COMM/CONFIG/NMLCFG.A86,  &

LINK86  :SD:COMM/LNK/COMM.LNK,  &
        :SD:COMM/LNK/TL.LNK,  &
        :SD:COMM/LNK/TLVC.LNK,  &
        :SD:COMM/CONFIG/RMXCFG.OBJ,  &
        :SD:COMM/CONFIG/BUFCGF.OBJ,  &
        :SD:COMM/CONFIG/DLCFG.OBJ,  &
        :SD:COMM/CONFIG/TLCFG.OBJ,  &
        :SD:COMM/LIB/AIMRMX.LIB  &
        :SD:COMM/LNK/COMM.LIB  &

TO  :SD:COMM/LNK/COMMCF.LNK  &
        NOCM NOLI NOMAP NOSB  &
        NOPUBLICS EXCEPT (COMMENTRY, CQERRORCODE)

LOC86  :SD:COMM/LNK/COMMCF.LNK  &
TO  :SD:COMM/LNK/COMM  &
        ORDER (CLASSES (CODE, ENTRYPTS, DATA, STACK))  &
        SEGSIZE (STACK (0))  &
        ADDRESSES (CLASSES (CODE (%0)))  &
        NOINITCODE, NOLINES, NOCOMMENTS, NOSYMBOLS  &
        OBJECTCONTROLS (NOLINES, NOCOMMENTS,  &
        NOPUBLICS, NOSYMBOLS)
```

4. Edit the submit file COMMCMP.CSD in the following way. First, replace the line:

```
ASM86  :SD:COMM/CONFIG/RMXCFG.A86
```

by

```
ASM86  :SD:COMM/CONFIG/CMPCFG.A86 MACRO(60)
```

5. The LNK invocation to generate COMMCF.LNK links a number of files including the following:

```
:SD:COMM/LNK/AIMRMX.LIB
:SD:COMM/LNK/RMXCFG.OBJ
```

Do not link these files while generating COMMCF.LNK. Replace them with the following:

- One of the following:

  `:SD:COMM/LNK/MIP.LNK` – the MIP message delivery mechanism.

  `:SD:COMM/LNK/BCBNC.LNK` – the BCB interface message delivery mechanism without the copy option.

  `:SD:COMM/LNK/BCBCPY.LNK` – the BCB interface message delivery mechanism with the copy option.

  `:SD:COMM/LNK/USRMDM.LNK` – the user-supplied message delivery mechanism.

- One of the following:

  `:SD:COMM/LNK/AIM86.OBJ` – if iNA runs on an 8086 CPU.

  `:SD:COMM/LNK/AIM186.OBJ` – if iNA runs on an 80186 CPU.

- `:SD:COMM/LNK/CMPCFG.OBJ` – the object file produced by the configuration of the component support.

- `:SD:COMM/LNK/USRRTN.OBJ` – this module must contain all the user-written routines.

6. Modify the arguments to the LOC86 command to reflect the hardware environment.
7. To generate the iNA system, type the following:

```
>SUBMIT :SD:COMM/CONFIG/COMMCMP.CSD
```

APPENDIX A
iNA 960 FILES

A.1 The Delivery Diskettes

The delivery package contains the following diskettes in both ISIS and iRMX 86 format:

- *iNA 960 Object Code R1.0 (COMM)* Contains the object code for the Transport, Data Link, and NMF functions.
- *iNA 960 Object Code R1.0 (Boot Consumer)* Contains the object code of the boot consumer.

Files on the iRMX 86 formatted diskettes are organized in directories. The same files on the ISIS diskettes are flat. Following are the directories on the iRMX 86 diskettes:

LNK	Contains the iNA 960 core and optional modules.
CSD	Contains the SUBMIT file for installing, linking, and locating iNA 960.
CONFIG	Contains the iNA 960 configuration files.
LIB	Contains the user interface libraries.
INC	Contains the INCLUDE files for using iNA 960.
INIT	Contains the initialization code for iSBC 186/51.

A.2 Directory LNK

COMM.LNK	Core of iNA 960 object code.
COMM.LIB	Library file required by COMM.LNK.
BTSRV.LNK	Boot server functions.
TL.LNK	Transport core.
TLVC.LNK	Transport normal virtual circuit services.
TLEX.LNK	Transport expedited services.
TLDG.LNK	Transport datagram services.
EDL.LNK	External data link services.
NMF.LNK	Network management functions.
AIMRMX.LIB	Library file required by AIMRMX.LNK.
AIM186.OBJ	AIM-186 module.
AIM86.OBJ	AIM-86 module.
MIP.LNK	MIP module.
BCBNC.LNK	BCBI (non-copy) module.
BCBCPY.LNK	BCBI (copy) module.

A.3 Directory CSD

INSTALL.CSD	Submit file to install iNA 960 to hard disk storage.
COMM.CSD	Submit file to assemble the configuration files, link all the optional modules, and locate the iNA 960.

A.4 Directory CONFIG

The following files appear as pairs consisting of a configuration macro definition file (MAC extension) and a default configuration file (A86 extension).

RMXCFG.MAC RMXCFG.A86	Configuration file to configure iNA 960 in the iRMX 86 environment.
CMPCFG.MAC CMPCFG.A86	Configuration file to configure iNA 960 in the component support environment.
BUFCFG.MAC BUFCFG.A86	iNA 960 buffer configuration file.
TLCFG.MAC TLCFG.A86	Transport configuration file.
DLCFG.MAC DLCFG.A86	Data link configuration file.
NMFCFG.MAC NMFCFG.A86	NMF configuration file.

A.5 Directory LIB

CQC.LIB	iNA 960 user interface library for PLM-86 COMPACT mode.
CQL.LIB	iNA 960 user interface library for PLM-86 LARGE and MEDIUM modes.

A.6 Directory INC

CQRB.EXT	External declarations for request block interfaces.
CQTL.EXT	External declarations for transport procedure interfaces.
CQDL.EXT	External declarations for data link procedure interfaces.
CQNMF.EXT	External declarations for NMF procedure interfaces.
CQTL.LIT	Literal declarations for transport interface parameters.
CQNMF.LIT	Literal declarations for NMF interface parameters.

A.7 Directory INIT

ICODE.OBJ	Initialization code for iSBC 186/51.
IN186.MAC IN186.A86	Configuration files for the initialization code.
INIT.CSD	Submit file for linking and locating the init code.

A.8 The Boot Consumer Diskette

CNFG.A86 CNFG.MAC	The configuration file for the boot consumer.
ROMA.OBJ ROMB.OBJ	The boot consumer.
AUTBOT.OBJ AUTBOT.P86 AUTBOT.LST	A sample user-written main module.
ROM.CSD	The submit file to link and locate the boot consumer.

APPENDIX B
NETWORK MANAGEMENT FACILITY OBJECTS

B.1 NMF/Data Link Objects

Id	Type	Access	Size	Description
2000H	value	R	BYTE	*Data Link Type.* Reserved for future expansion. Set to 1.
2001H	value	R	DWORD	*Line Speed.* The physical transmission rate in bits/second.
2002H	value	R	48 BIT	*Host Id.* The network physical address.
2003H	counter (wraparound)	RC	DWORD	*Total Sent.* Total number of packets sent by the station.
2004H	counter (sticky)	RC	WORD	*Primary Collisions.* The number of packets transmitted by the station that had at least 1 collision.
2005H	counter (sticky)	RC	WORD	*Secondary Collisions.* The number of collisions encountered after each primary collision.
2006H	counter (sticky)	RC	WORD	*Exceeded Collisions.* The number of packets discarded because the maximum number of collisions was exceeded.
2007H	counter (wraparound)	RC	DWORD	*Total Received.* The number of packets forwarded from the network to the client.
2008H	counter (sticky)	RC	WORD	*CRC Errors.* The number of packets dropped because of CRC errors.
2009H	counter (sticky)	RC	WORD	*Alignment Errors.* The number of packets dropped due to alignment errors.
200AH	counter (sticky)	RC	WORD	*Resource Errors.* The number of times data link ran out of resources.

B.2 NMF/Transport Virtual Circuit Connection Independent Objects

Id	Type	Access	Size	Description
4000H	value	R	BYTE	*Virtual Circuit Type.* Set to 0.
4001H	value	R	—	*Connection ID Vector.* WORD array where each nonzero element is an allocated connection ID. The size of this object is as follows: (2 × max_connections) BYTES
4002H	value	R	BYTE	*ISO Transport Number.* The version number of the ISO virtual circuit subsystem.
4003H	value	R	WORD	*Maximum Connections.* The maximum number of connections supported by the virtual circuit subsystem.
4004H	value	R	WORD	*Current Maximum Connections.* The maximum number of connections currently available.
4005H	value	R	WORD	*Maximum On-board CDB's.* The number of connection databases that have space already allocated (on-board).

B.2 NMF/Transport Virtual Circuit Connection Independent Objects (Cont'd.)

Id	Type	Access	Size	Description
4006H	value	R	WORD	*Active CDB's.* The number of connection databases currently in use.
4007H	value	R	WORD	*CDB Size.* The size in bytes of a connection database.
4008H	parameter	RS	WORD	*Default Persistence Count.* The number of times the local transport entity attempts to establish a connection when the remote transport entity explicitly rejects the connection attempt. It is assigned to new connections that request default persistence count.
4009H	parameter	RS	WORD	*Default Abort Timeout.* The amount of time (in units of 51 ms.) an unacknowledged segment is transmitted before automatically aborting the connection. It is assigned to new connections that request default abort timeout. A value of 0FFFFH indicates that an automatic abort is never to occur.
400AH	parameter	RS	DWORD	*Default Retransmit Timeout.* The initial amount of time (in units of 100 ms.) the transport layer waits before retransmitting an unacknowledged segment. This value is used on all new connections.
400BH	parameter	RS	DWORD	*Minimum Retransmit Timeout.* The minimum time (in units of 100 ms.) the transport layer waits before transmitting an unacknowledged segment. The initial value is configurable, but may be reset by the user.
400CH	parameter	RS	WORD	*Closing Abort Timeout.* The amount of time (in units of 51 ms.) the transport layer waits after sending a connection close request before aborting the connection.
400DH	parameter	RS	DWORD	*Flow Control Window Timeout.* Once a connection is established, the local transport sends flow control window acknowledgement packets to the remote entity at regular intervals to signal to the remote entity that it is still functioning when no other activity is on the connection. These packets also inform the remote transport of the most current local flow control window status. This object specifies the time interval (in units of 100 ms.) between these packets.
400EH	parameter	RS	WORD	*Inactivity Maximum Count.* The number of times the local transport transmits an unacknowledged flow control window acknowledgement packet before aborting the connection.
400FH	counter (sticky)	RC	WORD	*Total Duplicate Segments Rejected.* The total number (over all connections) of received segments that were rejected due to duplicate sequence numbers.
4010H	counter (sticky)	RC	WORD	*Total Checksum Errors.* The total number (over all connections) of received segments that were rejected because of checksum errors.
4011H	counter	RC	WORD	*Total Retransmission.* The total (sticky) number of times (over all connections) that unacknowledged segments were retransmitted.

B.2 NMF/Transport Virtual Circuit Connection Independent Objects (Cont'd.)

Id	Type	Access	Size	Description
4012H	counter (sticky)	RC	WORD	*Total Resource Errors.* The total number (over all connections) of segments discarded because receive buffers were not available.
4013H	value	R	BYTE	*Maximum Network Address Length.* The maximum length of the network address.
4014H	value	R	BYTE	*Maximum TSAP-ID Length.* The maximum length of local or remote TSAP-ID's.
4015H	value	R	WORD	*Local NSAP-ID.* The local NSAP-ID required to interface to the underlying network layer.
4016H	value	R	BYTE	*Reserved.*
4017H	value	R	BYTE	*Reserved.*
4018H	parameter	RS	WORD	*Default Connection Negotiation Options.* Specifies the default connection negotiation options.
4019H	parameter	RS	BYTE	*Maximum TPDU Size.* The value (specified as a power of 2) used for maximum TPDU size in the negotiation phase of connection establishment by the local transport entity.
401AH	parameter	RS	BYTE	*No Additional Option Field.* An additional option field (as encoded in ISO 8073) used as an assumed additional option parameter requested by a remote entity when, in fact, no such option parameter has been provided in the request.
401BH	parameter	RS	BYTE	*No Maximum TPDU Size Field.* The maximum TPDU size (specified as a power of 2) used as an assumed maximum TPDU size requested by a remote entity when, in fact, no size was specified by the remote entity in its request.
401CH	parameter	RS	WORD	*Maximum Normal Window Size.* The largest receive buffer credit that can be reported on a connection by the local TS to a remote TS for normal sequence number format.
401DH	parameter	RS	WORD	*Maximum Extended Window Size.* The largest receive buffer credit that can be reported on a connection by the local TS to a remote TS for extended sequence number format.
401EH	parameter	RS	WORD	*Minimum Credit.* The smallest receive buffer credit that can be reported on a connection by the local TS to a remote TS. This object has the following values: 0 — window can close 1 — window can never close
401FH	parameter	RS	DWORD	*Open Window Timeout.* The interval (in units of 100 ms.) between successive acknowledgements (AK TPDU's) that announce the opening of a previously closed credit window to avoid flow control deadlock.
4020H	parameter	RS	WORD	*Maximum Open Window Count.* The maximum number of open window AK's transmitted before the sender assumes that the remote TS has received the open window credit information. When this count is reached, the local TS stops transmitting the open window AK's.

B.3 NMF/Transport Virtual Circuit Connection Dependent Objects

Id	Type	Access	Size	Description
4081H	value	R	—	*Local TSAP-ID.* The local TSAP-ID for the specified connection.
4082H	value	R	—	*Remote Network Address.* The network address of the entity at the remote end of the connection. If the user performs a partially specified or unspecified passive open, this object will be 0 until the connection is established.
4083H	value	R	—	*Remote TSAP-ID.* The remote TSAP-ID for the specified connection.
4084H	value	R	BYTE	*Connection State.* The state of the connection. This object has the following values: 0 — Listen 1 — Cr Sent 2 — Ack Wait 3 — Estab 4 — Closing 5 — Closed 6 — Open 7 — Calling 8 — Cr Received
4085H	value	R	WORD	*Remote Connection ID.* The connection ID of the remote end of the specified connection. This object is set after the connection is established.
4086H	parameter	RS	WORD	*Persistence Count.* The number of times a connection request is retransmitted when the remote entity explicitly refuses it.
4087H	parameter	RS	WORD	*Abort Timeout.* This has the same meaning as the default abort timeout, but it is the actual abort timeout used on a specific connection.
4088H	parameter	RS	WORD	*Retransmit Timeout.* This has the same meaning as the default retransmit timeout, but it is the actual retransmit timeout used on a specific connection.
4089H	value	R	WORD	*Next Transmit Sequence Number.* The sequence number to be used with the next segment to be transmitted (not always the highest).
408AH	counter (sticky)	RC	WORD	*Duplicate Segments Rejected.* The total number of duplicate received segments discarded by transport on the specified connection.
408BH	counter (sticky)	RC	WORD	*Segments Retransmitted.* The total number of times that an unacknowledged segment has been retransmitted over the specified connection.
408CH	counter (sticky)	RC	WORD	*Resource Errors.* The total number of times that segments received on the specified connection were rejected because receive buffers were not available.
408DH	value	R	WORD	*Client Options.* The options specified by the client at the time the connection request was made.
408EH	value	R	BYTE	*Class Options.* The ISO class of services and sequence number format actually negotiated on the connection. The only values are as follows: 40 — class 4 and normal (7-bit) format 42 — class 4 and extended (31-bit) format
408FH	value	R	BYTE	*Additional Options.* The additional options actually negotiated on the connection. Only bits 0 and 1 are meaningful. The values are as follows: 0 — no expedited service and checksum 1 — expedited service and checksum 2 — no expedited service and no checksum 3 — expedited service and no checksum

B.3 NMF/Transport Virtual Circuit Connection Dependent Objects (Cont'd.)

Id	Type	Access	Size	Description
4090H	value	R	BYTE	*Maximum TPDU Size.* The negotiated maximum TPDU size (as a power of 2) over the connection.
4091H	value	R	WORD	*Maximum TPDU Data Length.* The maximum length (in bytes) of the data that can be sent in one TPDU; that is, the maximum TPDU size minus the header length.
4092H	value	R	WORD	*Inactivity Count.* The number of times an unacknowledged flow control window acknowledgement packet has been retransmitted over the connection.
4093H	value	R	—	*Reserved.*

B.4 NMF/Transport Datagram Objects

Id	Type	Access	Size	Description
4100H	value	R	BYTE	*Datagram Type.* Set to 1.
4101H	value	R	BYTE	*Datagram Receive Queue Size.* The maximum number of TSAP-ID's for which the client can post buffers.
4102H	value	R	BYTE	*Reserved.*
4103H	counter (sticky)	RC	WORD	*Total Datagrams Transmitted.* The total number of datagrams transmitted.
4104H	counter (sticky)	RC	WORD	*Total Datagrams Received.* The total number of datagrams received.
4105H	counter (sticky)	RC	WORD	*Total Datagram Resource Errors.* The total number of datagrams rejected due to lack of buffers.
4106H	counter (sticky)	RC	WORD	*Total Datagram Checksum Errors.* The total number of datagrams rejected due to checksum errors.
4107H	counter	RC	WORD	*Total Datagram Address Errors.* The total number of datagrams rejected due to illegal address fields in the header.

B.5 NMF/Boot Server Objects

Id	Type	Access	Size	Description
8000H	value	R	WORD	*NMF Type.* Reserved for future expansion. Set to 1.
8001H	value	R	6 BYTES	*Multicast Address* of the boot server. Set at configuration time.
8002H	value	R	WORD	*Maximum Number of Nodes* that the boot server can boot at the same time. Set at configuration time.
8003H	value	R	WORD	*Maximum Number of Addresses* in the boot table.

B.5 NMF/Boot Server Objects (Cont'd.)

Id	Type	Access	Size	Description
8004H	parameter	RS	—	*The Boot Table.* The first WORD specifies the number of addresses in the table.
8005H	value	R	WORD	*Number of Class Codes* recognized by the boot server. Set at configuration time.
8006H	value	R	—	*List of Class Codes* that the boot server recognizes. The size of this object is 2× number of class codes BYTES. Set at configuration time.
80FFH	value	R	WORD	*Number of NMF's* present in the system. Reserved for future expansion. Set to 1.

APPENDIX C
SAMPLE USER ROUTINES FOR COMPONENT SUPPORT

This appendix contains sample user-written routines for the iSBC 552. Use these routines as a guideline when writing routines for your own specific hardware.

The iSBC 552 offers a low-cost Ethernet controller module for MULTIBUS-based systems. The iSBC 552 has the following features:

- 8 MHz 80186/82586 co-processors
- 82501 Ethernet Serial Interface chip
- 16 data bit/24 address bit MULTIBUS master capability
- 256 Kbytes of RAM/ROM

The iSBC 552 is totally memory-mapped. Local address bits 18 and 19 are decoded to divide the 1-Mbyte 80186 memory space into four 256-Kbyte quadrants. The upper quadrant (768K to 1M) and lower quadrant (0 to 256K) are used solely for local memory and will access the same physical memory. Quadrant 2 (256K to 512K) is used for memory-mapped local I/O. Quadrant 3 (512K to 768K) is used as the 256K window to the 16-Mbyte MULTIBUS memory.

The listings consist of 4 modules. The first module, BOOT_START, contains the initialization code for the BCBI interface and the routines GEN_INT and CLEAR_INT.

The second module, INITIALIZE_552, contains the routine that initializes the chip-select lines of the 80186 processor.

The third module, LOC_TO_SYS_ADDR_FOR_552, contains the routine LOC_TO_SYS_ADDR.

The last module, COPY, contains the routines SYS_TO_LOC_ADDR, SAVE_ADDRESS_SPACE, and RESTORE_ADDRESS_SPACE.

```
SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE
BOOT_START NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM86.86 :F1:LLIBOT.A86 PAGEWIDTH(78) NOOJ

LOC  OBJ              LINE     SOURCE

                         1     NAME      BOOT_START
                         2
                         3     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                         4     ;;
                         5     ;; MODULE NAME: BOOT START ENTRY
                         6     ;;
                         7     ;; FUNCTION: THIS IS THE START OF THE
                         8     ;;       MAIN PROGRAM. IT ASSUMES THAT
                         9     ;;       INA HAS ALREADY BEEN LOADED
                        10     ;;       ONTO THE RAM.
                        11     ;;
                        12     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        13
                        14     CGROUP GROUP CODE
                        15     DGROUP GROUP DATA
                        16     ASSUME CS:CGROUP, DS:DGROUP
```

```
                        17
----                    18      DATA SEGMENT PUBLIC 'DATA'
                        19
0000 ??                 20      SAVED_INT_TYPE          DB ?
0001 ????               21      SAVED_INT_ADDR          DW ?
                        22
0003 (40                23      MY_STACK                DW 40 DUP (?)
     ????
     )
0053                    24      MY_STACK_TOP            LABEL WORD
                        25
                        26      PUBLIC INIT_WINDOW      BCB_LOC
0053 (2                 27      INIT_WINDOW             DW 2 DUP (?)
     ????
     )
0057 (2                 28      BCB_LOC                 DW 2 DUP (?)
     ????
     )
                        29
----                    30      DATA ENDS
                        31
                        32      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        33      ;; LITERAL DECLS USED BY THIS ROUTINE
                        34      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        35
  FF22                  36      PIC_EOI_P               EQU 0FF22H
                        37      EXTRN                   CA_INT_VECTOR_TYPE
                                :ABS   ; CNFG PARM
  FFFF                  38      MASK_ALL_INTS           EQU 0FFFFH
                        39
  FF28                  40      PIC_MASK_P              EQU 0FF28H
                        41      EXTRN                   ENABLE_CA:ABS
                                ; CNFG PARM
                        42
  FF24                  43      PIC_POLL_P              EQU 0FF24H
                        44      EXTRN                   INT_FROM_USER:ABS
                                ; CNFG PARM
                        45
  0800                  46      SCP_0_OFF               EQU 0800H
  1000                  47      SCP_1_OFF               EQU 1000H
  1800                  48      SCP_2_OFF               EQU 1800H
  4000                  49      SCP_BASE                EQU 4000H
  4000                  50      BASE_552                EQU 4000H
                        51
  2100                  52      LED_ADDR                EQU 2100H
  2102                  53      MB_WINDOW               EQU 2102H
  2104                  54      MB_IO_ENABLE            EQU 2104H
  2106                  55      MB_IO_DISABLE           EQU 2106H
  2108                  56      MB_INT_DISABLE          EQU 2108H
  210A                  57      MB_INT_ENABLE           EQU 210AH
                        58
----                    59      ISCP_STR STRUC
0000                    60          BUSY                DB ?
0001                    61          STATUS              DB ?
0002                    62          SCB_OFFSET          DW ?
0004                    63          SCB_BASE_1          DW ?
0006                    64          SCB_BASE_2          DB ?
0007                    65          UNUSED_1            DB ?
0008                    66          INT_TYPE            DB ?
```

```
0009                    67          UNUSED_2            DB ?
000A                    68          INT_ADDR            DW ?
----                    69      ISCP_STR ENDS
                        70
----                    71      CODE SEGMENT BYTE PUBLIC 'CODE'
                        72      EXTRN CONFIDENCE_TESTS:NEAR
                        73      EXTRN INIT_HARD:NEAR
                        74      EXTRN BEGIN_AIM:NEAR
                        75      EXTRN SYS_TO_LOC_ADDR:NEAR
                        76
                        77      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        78      ;;      SETS THE MBUS WINDOW AND RETURNS
                        79      ;;      PTR TO ACCESS THE LOCATION
                        80      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        81
                        82
0000                    83      SET_WINDOW:
0000 5E                 84        POP   SI    ; THE RETURN ADDRESS
0001 58                 85        POP   AX    ; THE LSW
0002 5A                 86        POP   DX    ; THE MSW
                        87
0003 BF0040             88        MOV   DI, BASE_552    ; SET THE MBUS
                        WINDOW
0006 8EC7               89        MOV   ES, DI
0008 BF0221             90        MOV   DI, MB_WINDOW
000B 268815             91        MOV   BYTE PTR ES:[DI], DL
                        92
000E B208               93        MOV   DL, 8     ; MBUS MAPPING AT 80000?
                        94
0010                    95      STLA_1:
0010 B90C00             96        MOV   CX, 12    ; CONVERT LOW(DL):BX TO
                        POINTER
0013 50                 97        PUSH AX         ; SAVE LSW
                        98
                        99      ; GET BASE OF THE POINTER IN THE DX
                        REGISTER
0014                   100      STLA_2:
0014 F8                101        CLC
0015 D1D0              102        RCL   AX, 1
0017 D1D2              103        RCL   DX, 1
0019 E2F9              104        LOOP STLA_2
                       105
                       106      ; GET THE OFFSET OF THE POINTER IN AX
001B 58                107        POP   AX
001C 250F00            108        AND   AX, 0FH
                       109
001F 8BD8              110        MOV   BX, AX    ; RETURN AS POINTER
0021 8EC2              111        MOV   ES, DX
                       112
0023 FFE6              113        JMP   SI        ; RETURN
                       114
                       115      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       116      ;; THE BEGINNING
                       117      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       118      PUBLIC BOOT_START_ENTRY
0025                   119      BOOT_START_ENTRY:
0025 FA                120        CLI
                       121
0026 B8----        R   122        MOV   AX, DGROUP    ; INIT SS,DS AND SP
```

```
0029 8ED8                123         MOV  DS, AX
002B 8ED0                124         MOV  SS, AX
002D BC5300          R   125         MOV  SP, OFFSET DGROUP:MY_STACK_TOP
                         126
                         127         ; CALL CONFIDENCE TESTS
0030 50                  128         PUSH AX                  ; SAVE RESULT
                         129
0031 B80000          E   130         MOV  AX, ENABLE_CA       ; UNMASK CA
0034 BA28FF              131         MOV  DX, PIC_MASK_P
0037 EF                  132         OUT  DX, AX
                         133
0038 BA24FF              134         MOV  DX, PIC_POLL_P
003B                     135       WAIT_CA:
003B ED                  136         IN   AX, DX
003C 81F20000        E   137         XOR  DX, INT_FROM_USER
0040 75F9                138         JNE  WAIT_CA
                         139
0042 B80000          E   140         MOV  AX, CA_INT_VECTOR_TYPE ; EOI THE
                           PIC
0045 BA22FF              141         MOV  DX, PIC_EOI_P
0048 EF                  142         OUT  DX, AX
                         143
0049 BB0040              144         MOV  BX, SCP_BASE        ; GET SCP AS
                           DWORD
004C 8EC3                145         MOV  ES, BX
004E BB0008              146         MOV  BX, SCP_0_OFF
0051 268A07              147         MOV  AL, BYTE PTR ES:[BX]
0054 BB0010              148         MOV  BX, SCP_1_OFF
0057 268A27              149         MOV  AH, BYTE PTR ES:[BX]

005A BB0018              150         MOV  BX, SCP_2_OFF
005D 268A17              151         MOV  DL, BYTE PTR ES:[BX]
0060 32F6                152         XOR  DH, DH
                         153
0062 52                  154         PUSH DX      ; GET PTR TO ISCP
0063 50                  155         PUSH AX
0064 E899FF              156         CALL SET_WINDOW
                         157
0067 58                  158         POP  AX      ; RESULT OF CONFIDENCE TEST
0068 26884701            159         MOV  ES:[BX].STATUS, AL
                         160
006C 268A4708            161         MOV  AL, ES:[BX].INT_TYPE ; SAVE INT
                           INFO
0070 A20000          R   162         MOV  SAVED_INT_TYPE, AL
0073 268B470A            163         MOV  AX, ES:[BX].INT_ADDR
0077 A30100          R   164         MOV  SAVED_INT_ADDR, AX
                         165
007A 268B4704            166         MOV  AX, ES:[BX].SCB_BASE_1 ; GET BC
                           ADDR
007E 268A5706            167         MOV  DL, ES:[BX].SCB_BASE_2
0082 32F6                168         XOR  DH, DH
0084 26034702            169         ADD  AX, ES:[BX].SCB_OFFSET
0088 83D200              170         ADC  DX, 0
                         171
008B A35700          R   172         MOV  BCB_LOC, AX ; AND SAVE IT
008E 89165900        R   173         MOV  BCB_LOC+2, DX
                         174
0092 C70653000000    R   175         MOV  INIT_WINDOW, 0 ; NOTE THE VALUE
                           OF INITIAL WINDOW
```

```
0098 89165500      R   176         MOV  INIT_WINDOW+2, DX
                       177
009C 26C60700          178         MOV  ES:[BX].BUSY, 0; CLEAR BUSY BIT
                       179
00A0 52                180         PUSH DX              ; SET WINDOW TO
                      BCB
00A1 50                181         PUSH AX
00A2 E85BFF            182         CALL SET_WINDOW
                       183
00A5 E90000        E   184         JMP  BEGIN_AIM       ; START COMM
                      SYSTEM
                       185
                       186     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       187     ;; GENERATES INT TO HOST
                       188     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       189     PUBLIC GEN_INT
00AB                   190     GEN_INT:
00A8 BB0040            191         MOV  BX, BASE_552       ; INIT ES WITH
                      4000H
00AB 8EC3              192         MOV  ES, BX
                       193
00AD A00000        R   194         MOV  AL, SAVED_INT_TYPE
00B0 8B160100      R   195         MOV  DX, SAVED_INT_ADDR
00B4 FEC8              196         DEC  AL                 ; IF AL=1 THEN
                      MEM MAPPED
00B6 740B              197         JZ   MEM_MAPPED_INT
00B8 FEC8              198         DEC  AL                 ; IF AL=2 THEN
                      I/O MAPPED
00BA 741D              199         JZ   IO_MAPPED_INT

                       200
00BC BB0A21            201         MOV  BX, MB_INT_ENABLE     ; ELSE LEVEL
                      MODE
00BF 26881F            202         MOV  BYTE PTR ES:[BX], BL
00C2 C3                203         RET
                       204
00C3                   205     MEM MAPPED INT:
00C3 A15700        R   206         MOV  AX, BCB_LOC
00C6 8B0E5900      R   207         MOV  CX, BCB_LOC+2
00CA 03C2              208         ADD  AX, DX
00CC 83D100            209         ADC  CX, 0
00CF 51                210         PUSH CX
00D0 50                211         PUSH AX
00D1 E80000        E   212         CALL SYS_TO_LOC_ADDR
00D4 26C607FF          213         MOV  BYTE PTR ES:[BX], 0FFH
00D8 C3                214         RET
                       215
00D9                   216     IO_MAPPED_INT:
00D9 BB0421            217         MOV  BX, MB_IO_ENABLE
00DC 26881F            218         MOV  BYTE PTR ES:[BX], BL
00DF EE                219         OUT  DX, AL
00E0 BB0621            220         MOV  BX, MB_IO_DISABLE
00E3 26881F            221         MOV  BYTE PTR ES:[BX]. BL
00E6 C3                222         RET
                       223
                       224     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       225     ;; CLEARS THE INT GENED TO HOST
                       226     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       227     PUBLIC CLEAR_INT
```

```
00E7                    228     CLEAR_INT:
00E7 A00000       R     229       MOV   AL, SAVED_INT_TYPE
00EA 0AC0               230       OR    AL, AL
00EC 750B               231       JNZ   CI_1
                        232
00EE BB0040             233       MOV   BX, BASE_552
00F1 8EC3               234       MOV   ES, BX
00F3  BB0821            235       MOV   BX, MB_INT_DISABLE
00F6 26881F             236       MOV   BYTE PTR ES:[BX], BL
                        237
00F9                    238     CI_1:
00F9 C3                 239       RET
                        240
----                    241     CODE ENDS
                        242
                        243     END BOOT_START_ENTRY

ASSEMBLY COMPLETE, NO ERRORS FOUND


SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE
INITIALIZE_552
NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM86.86 :F1:INI552.A86 PAGEWIDTH(78) NOOJ

LOC  OBJ             LINE     SOURCE

                        1     NAME   INITIALIZE_552
                        2     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        3     ;;
                        4     ;;      MODULE NAME:
                        5     ;;           INITIALIZES THE 552 BOARD.
                        6     ;;
                        7     ;;      FUNCTION: THIS MODULE
                        8     ;;          INIT_HARD- INITIALIZES THE
                        9     ;;           PCS AND MCS OF THE 80186
                       10     ;;
                       11     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       12
                       13     CGROUP GROUP CODE
                       14     ASSUME CS:CGROUP
                       15
                       16     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       17     ;;   LITERAL DEFS FOR HARDWARE
                       18     ;;   INITIALIZATION
                       19     ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  FFA0                 20     UMCS_P            EQU       0FFA0H
  C038                 21     UMCS_VAL          EQU       0C038H
                       22
  FFA2                 23     LMCS_P            EQU       0FFA2H
  3FF8                 24     LMCS_VAL          EQU       03FF8H
                       25
  FFA8                 26     MPCS_P            EQU       0FFA8H
  81F8                 27     BLOCK_SIZE_8K     EQU       081F8H
                       28
  FFA6                 29     MMCS_P            EQU       0FFA6H
  41F8                 30     MCS_BASE          EQU       041F8H
                       31
  FFA4                 32     PACS_P            EQU       0FFA4H
```

```
  4228                    33      PCS_BASE          EQU       04228H
                          34
----                      35      CODE SEGMENT BYTE PUBLIC 'CODE'
                          36      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                          37      ;;    INITIALIZES THE 552 HARDWARE, THE
                          38      ;;    PCS AND MCS
                          39      ;;    this consists of initializing the
                          40      ;;    chip selects lines of the 80186,
                          41      ;;    sets the esi chip
                          42      ;;    into non loop back mode.
                          43      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                          44      PUBLIC  INIT_HARD
0000                      45      INIT_HARD:
0000 B838C0               46                        MOV   AX, UMCS_VAL
0003 BAA0FF               47                        MOV   DX, UMCS_P
0006 EF                   48                        OUT   DX, AX
                          49
0007 B8F83F               50                        MOV   AX, LMCS_VAL
000A BAA2FF               51                        MOV   DX, LMCS_P
000D EF                   52                        OUT   DX, AX
                          53
000E B8F881               54                        MOV   AX, BLOCK_SIZE_8K
0011 BAA8FF               55                        MOV   DX, MPCS_P
0014 EF                   56                        OUT   DX, AX
                          57
0015 B8F841               58                        MOV   AX, MCS_BASE
0018 BAA6FF               59                        MOV   DX, MMCS_P
001B EF                   60                        OUT   DX, AX
                          61
001C B82842               62                        MOV   AX, PCS_BASE
001F BAA4FF               63                        MOV   DX, PACS_P
0022 EF                   64                        OUT   DX, AX
                          65
0023 C3                   66                        RET
                          67
----                      68      CODE ENDS
                          69
                          70      END

ASSEMBLY COMPLETE, NO ERRORS FOUND


SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE
LOC_TO_SYS_ADR_FOR_552
NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM86.86 :F1:LTSA.A86 PAGEWIDTH(78) NOOJ

LOC  OBJ             LINE      SOURCE

                        1      NAME   LOC_TO_SYS_ADR_FOR_552
                        2      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        3      ;;
                        4      ;;  MODULE NAME:
                        5      ;;  LOCAL TO SYSTEM ADDRESS
                        6      ;;
                        7      ;;  FUNCTION:
                        8      ;;  CONVERTS PLM86 POINTER TO A 32
                        9      ;;  BIT ABSOLUTE ADDRESS AND THEN
                       10      ;;  SETS BIT 25 TO 1
```

```
                        11      ;;
                        12      ;;  CALLING SEQUENCE:
                        13      ;;      ADDR = LOC_TO_SYS_ADDR( PTR );
                        14      ;;
                        15      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        16      CGROUP GROUP CODE
                        17      ASSUME CS:CGROUP
                        18
----                    19      CODE SEGMENT BYTE PUBLIC 'CODE'
                        20
                        21      PUBLIC LOC_TO_SYS_ADDR
0000                    22      LOC_TO_SYS_ADDR:
0000 5F                 23          POP   DI            ; THE RETURN ADDRESS
0001 5E                 24          POP   SI            ; THE OFFSET
0002 58                 25          POP   AX            ; THE BASE
                        26
0003 33D2               27          XOR   DX, DX        ; CONVERT BASE TO
                                DWORD
0005 B90400             28          MOV   CX, 4
0008                    29      LTSA_1:
0008 F8                 30          CLC
0009 D1D0               31          RCL   AX, 1
000B D1D2               32          RCL   DX, 1
000D E2F9               33          LOOP  LTSA_1
                        34
000F 03C6               35          ADD   AX, SI        ; AND ADD IT TO THE
                                OFFSET
0011 81D20001           36          ADC   DX, 100H      ; SETTING BIT 25 TO 1

                        37
0015 8BD8               38          MOV   BX, AX        ; RETURN AS POINTER AS
                                WELL
0017 8EC2               39          MOV   ES, DX
                        40
0019 FFE7               41          JMP   DI            ; RETURN
                        42
----                    43      CODE ENDS
                        44      END

ASSEMBLY COMPLETE, NO ERRORS FOUND


SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.1 ASSEMBLY OF MODULE COPY
NO OBJECT MODULE REQUESTED
ASSEMBLER INVOKED BY:  ASM86.86 :F1:COPY.A86 PAGEWIDTH(78) NOOJ

LOC  OBJ             LINE      SOURCE

                        1      NAME    COPY
                        2      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        3      ;; THIS MODULE CONTAINS THE ROUTINES:
                        4      ;;      1. SYS_TO_LOC_ADDR
                        5      ;;      2. SAVE_ADDRESS_SPACE
                        6      ;;      3. RESTORE_ADDRESS_SPACE
                        7      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  4000                  8      BASE_552          EQU       4000H
  2102                  9      MB_WINDOW         EQU       2102H
                       10
                       11      CGROUP GROUP CODE
```

```
                        12      DGROUP GROUP DATA
                        13      ASSUME CS:CGROUP, DS:DGROUP
                        14
----                    15      DATA SEGMENT PUBLIC 'DATA'
0000 ??                 16      CUR_WINDOW              DB ?
                        17      ; TO SAVE THE VALUE OF THE CURRENT
                        18      ; MBUS WINDOW
----                    19      DATA ENDS
                        20
                        21
----                    22      CODE SEGMENT BYTE PUBLIC 'CODE'
                        23
                        24      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        25      ;;      FUNCTION:
                        26      ;; IF ADDRESS > 16 MBYTES
                        27      ;;  CONVERTS 24 BIT ADDRESS TO PLM86
                        28      ;;  POINTER
                        29      ;; ELSE SETS MULTIBUS WINDOW TO ACCESS
                        30      ;;  24 BIT ADDRESS AND RETURNS POINTER
                        31      ;;  TO ACCESS THE ADDRESS
                        32      ;;
                        33      ;;      CALLING SEQUENCE:
                        34      ;; PTR = SYS_TO_LOC_ADDR( ADDRESS );
                        35      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                        36      PUBLIC SYS_TO_LOC_ADDR
0000                    37      SYS_TO_LOC_ADDR:
0000 5E                 38        POP   SI        ; THE RETURN ADDRESS
0001 58                 39        POP   AX        ; THE LSW OF THE ADDRESS
0002 5A                 40        POP   DX        ; THE MSW OF THE ADDRESS
                        41
0003 0AF6               42        OR    DH, DH    ; GREATER THAN 16 MBYTES
0005 7511               43        JNZ   STLA_1    ; IT NOT THEN GOTO STLA_1
                        44
0007 BF0040             45        MOV   DI, BASE_552 ; SET THE MBUS
                                WINDOW
000A 8EC7               46        MOV   ES, DI
000C BF0221             47        MOV   DI, MB_WINDOW
000F 268815             48        MOV   BYTE PTR ES:[DI], DL
0012 88160000      R    49        MOV   CUR_WINDOW, DL ; NOTE THE ADDRESS
                                 OF CURRENT WIDOW
0016 B208               50        MOV   DL, 8     ; MBUS MAPPING AT 8000:?
                        51
0018                    52      STLA_1:
0018 B90C00             53        MOV   CX, 12      ; CONVERT LOW(DL):BX
                                TO POINTER
001B 50                 54        PUSH AX           ; SAVE LSW
                        55
                        56      ; GET BASE OF THE POINTER IN THE DX
                                REGISTER
001C                    57      STLA 2:
001C F8                 58        CLC
001D D1D0               59        RCL   AX, 1
001F D1D2               60        RCL   DX, 1
0021 E2F9               61        LOOP STLA_2
                        62
                        63      ; GET THE OFFSET OF THE POINTER IN AX
0023 58                 64        POP   AX
0024 250F00             65        AND   AX, 0FH
                        66
```

```
0027 8BD8                67        MOV   BX, AX     ; RETURN AS POINTER
0029 8EC2                68        MOV   ES, DX
                         69
002B FFE6                70        JMP   SI         ; RETURN
                         71
                         72      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                         73      ;;     SAVES THE PARM PERTAINING TO THE
                         74      ;;     CURRENT ADDRESS SPACE ON THE STACK
                         75      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                         76      PUBLIC SAVE_ADDRESS_SPACE
002D                     77      SAVE_ADDRESS_SPACE:
002D 5F                  78        POP   DI         ; THE RETURN ADDRESS
002E FF360000      R     79        PUSH WORD PTR CUR_WINDOW ; THE CURRENT
                               ADDR SPACE
0032 FFE7                80        JMP   DI         ; RETURN
                         81
                         82      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                         83      ;;      RESTORES THE ADDR SPACE SAVED BY
                         84      ;;      THE PRECEEDING CALL
                         85      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                         86      PUBLIC RESTORE_ADDRESS_SPACE
0034                     87      RESTORE_ADDRESS_SPACE:
0034 5F                  88        POP   DI         ; THE RETURN ADDRESS
0035 58                  89        POP   AX         ; THE OLD ADDR SPACE
                         90
0036 BB0040              91        MOV   BX, BASE_552 ; NOW SET THE
                               ADDRESS SPACE OF
0039 8EC3                92        MOV   ES, BX       ; THE 552 TO THAT
                               STORED IN AX
003B BB0221              93        MOV   BX, MB_WINDOW
003E 268807              94        MOV   BYTE PTR ES:[BX], AL
                         95
0041 FFE7                96        JMP   DI           ; RETURN
                         97
----                     98      CODE ENDS
                         99      END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

APPENDIX D
CONFIGURE COMMAND PARAMETERS

The data link CONFIGURE command is used to load the 82568 (or the iSBC 550 FW) with its operating parameters. In addition, the DL_CONF configuration macro is used to load the operating parameters in a similar manner during initialization. This appendix gives the format for the CONFIGURE command argument field and describes the command parameters. For more information on the particular parameters, see the *82586 Reference Manual* or the *iSBC 550 Ethernet Controller Kit Programmer's Manual.*

Figure D-1 shows the format of the CONFIGURE command argument field. This field has the same format for both the 82586 and iSBC 550FW controllers.

Table D-1 lists the CONFIGURE command parameters. The iSBC 550FW controller only recognizes a subset of the 82586 configuration parameters. Those relevant to the iSBC 550FW controller are identified by an asterisk beside the parameter mnemonic.

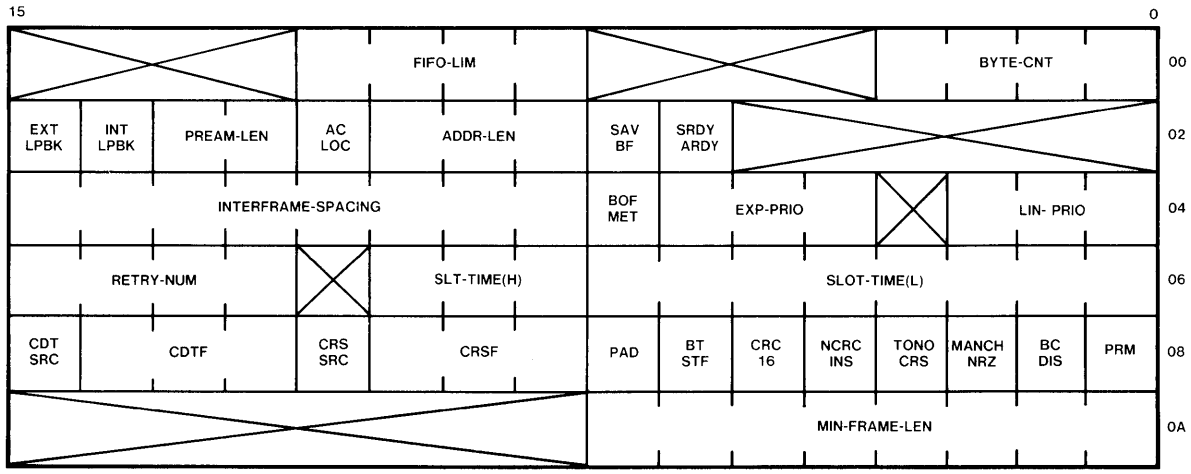

Figure D-1. CONFIGURE Command Argument Field

122193-10

Table D-1. Communications Controller Configuration Parameters

Byte	Bit(s)	Mnemonic	Default	Description
1	0–3	BYTE-CNT*	—	Byte count. Number of bytes, including this one, that hold parameters to be configured. A value greater than 12 is truncated to 12. A value less than 4 is interpreted as 4. In word mode, if the value is an odd number, the last byte is truncated.
2	0–3	FIFO-LIM	8	FIFO limit value.
3	6	SRDY/$\overline{\text{ARDY}}$	0	0 — SRDY/$\overline{\text{ARDY}}$ pin operates as $\overline{\text{ARDY}}$ (internal synchronization). 1 — SRDY/$\overline{\text{ARDY}}$ pin operates as SRDY (external synchronization).
	7	SAV-BF*	0	0 — Received bad frames are not saved in memory. 1 — Received bad frames are saved in memory.
4	0–2	ADDR-LEN	6	Number of address bytes. Note: 7 is interpreted as 0.
	3	AC-LOC*	0	0 — Address and Type fields are separated from data and associated with Transmit Command Block or Receive Frame Descriptor. For transmitted frame, the Source Address is generated by the 82586. 1 — Address and Type fields are part of the transmit/receive data buffers, including the Source Address.
	4–5	PREAM-LEN	2	Preamble Length, including the Beginning of Frame indicator. Values are as follows: 00 — 2 bytes 01 — 4 bytes 10 — 8 bytes 11 — 16 bytes
	6	INT-LPBK*	0	0 — No internal loopback. 1 — Internal loopback (if bit 7 = 1).
	7	EXT-LPBK	0	0 — No external loopback. 1 — External loopback.
5	0–2	LIN-PRIO	0	Linear Priority.
	4–6	EXP-PRIO	0	Exponential Priority.
	7	BOF-MET	0	Exponential Backoff Method. 0 — Ethernet. 1 — Short Topology and/or Low Bit Rate (Interframe Spacing shorter than the Slot Time).
6	0–7	IF-SPACING	96	Interframe Spacing in T×C period units. A value less than 32 is interpreted as 32.
7	0–7	SLT-TM(L)	0	Slot Time number, low byte.
8	0–2	SLT-TM(H)	1 (512)	Slot Time number, high byte. Slot Time is the number of T×C period units. Slot Time number = 0 is interpreted as 2048 (211).
	4–7	RETRY-NUM*	15	Number of transmission retries on collisions.
9	0	PRM*	0	Promiscuous Mode. 0 — Non-promiscuous address filtering mode. 1 — Promiscuous mode.
	1	BC-DIS*	0	Broadcast Disable. 0 — Broadcasted frames accepted. 1 — Broadcasted frames rejected.
	2	MANCH/NRZ	0	Manchester or NRZ encoding/decoding. 0 — NRZ. 1 — Manchester.

Table D-1. Communications Controller Configuration Parameters (Cont'd.)

Byte	Bit(s)	Mnemonic	Default	Description
	3	TONO-CRS	0	Transmit on No Carrier Sense. 0 — Cease transmission if CRS goes inactive during frame transmission (after preamble is sent). 1 — Continue transmission even if there is No Carrier Sense.
	4	NCRC-INS	0	No CRC Insertion. 0 — 82586 generates and appends CRC to transmitted frames. 1 — Disables the internal logic that generates CRC.
	5	CRC-16	0	CRC Type. 0 — 32-bit Autodin II CRC polynomial. 1 — 16-bit CCITT CRC polynomial.
	6	BT-STF	0	Bitstuffing. 0 — End of Carrier mode (Ethernet). 1 — HDLC-like Bitstuffing mode.
	7	PAD	0	Padding. Note: PAD has meaning only for Bitstuffing. In the End of Carrier mode, PAD is internally forced to 0. 0 — No Padding. 1 — Perform padding by transmitting flags for the rest of the Slot Time.
10	0–2	CRSF	0	Carrier Sense Filter bits.
	3	CRS-SRC	0	Carrier Sense Source. 0 — Carrier Sense signal externally generated. 1 — Carrier Sense signal internally generated.
	4–6	CDTF	0	Collision Detect Filter bits.
	7	CDT-SRC	0	Collision Detect Source. 0 — Collision Detect signal externally generated. 1 — Collision Detect signal internally generated. (Works for a transceiver that does not feed back the transmitted signal on the receive pair).
11	0–7	MIN-FRM-LEN	64	Minimum Frame Length in bytes. Frames shorter than MIN-FRM-LEN are treated as bad frames.

APPENDIX E
MULTIBUS® INTERPROCESSOR PROTOCOL (MIP)

E.1 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 becomes useful only 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 E-1.

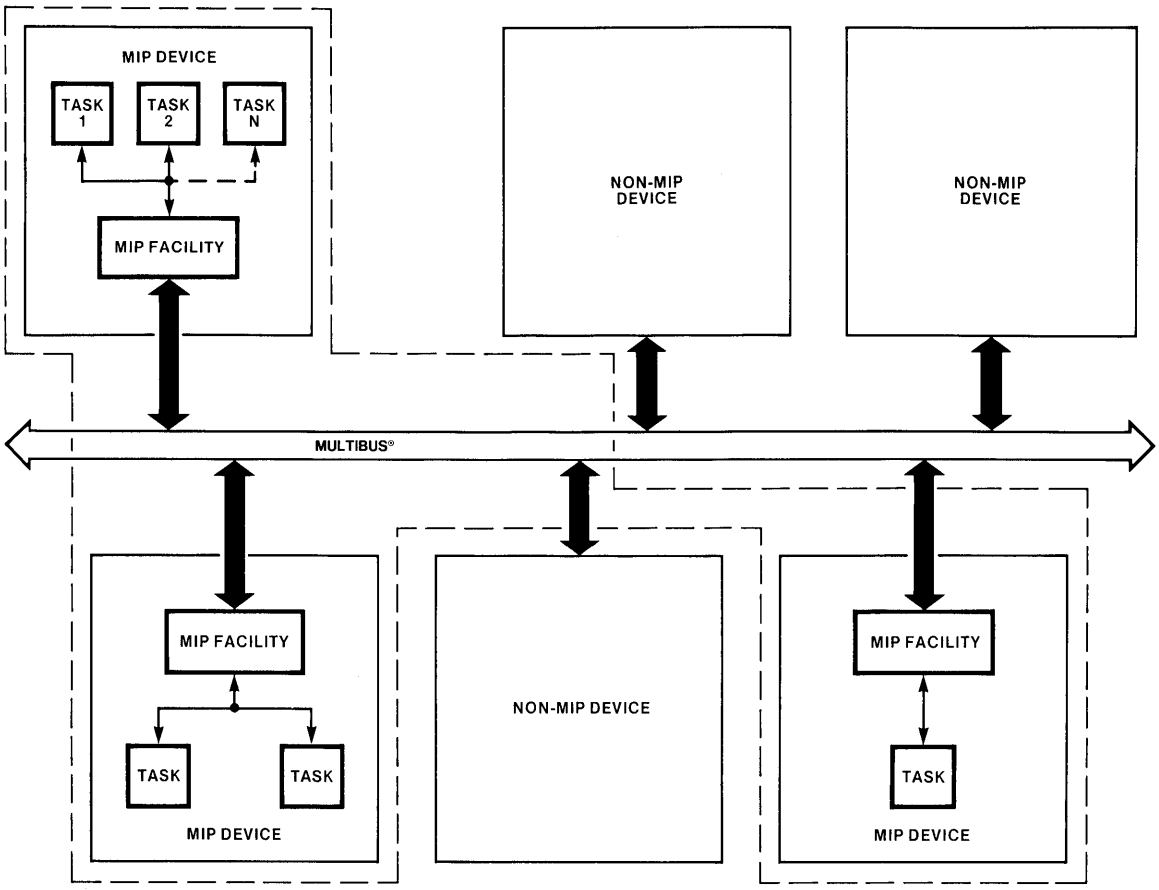

Figure E-1. A MIP System

121769-21

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 signaling 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, MIP facilities can run on other processors and under other operating systems.

E.2 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.

E.3 The MIP Model

E.3.1 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 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 E-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.

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.

E.3.2 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, which 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 as long as the information passed through 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 E-3 illustrates the three-level structure. Refer to this figure during the following discussion.

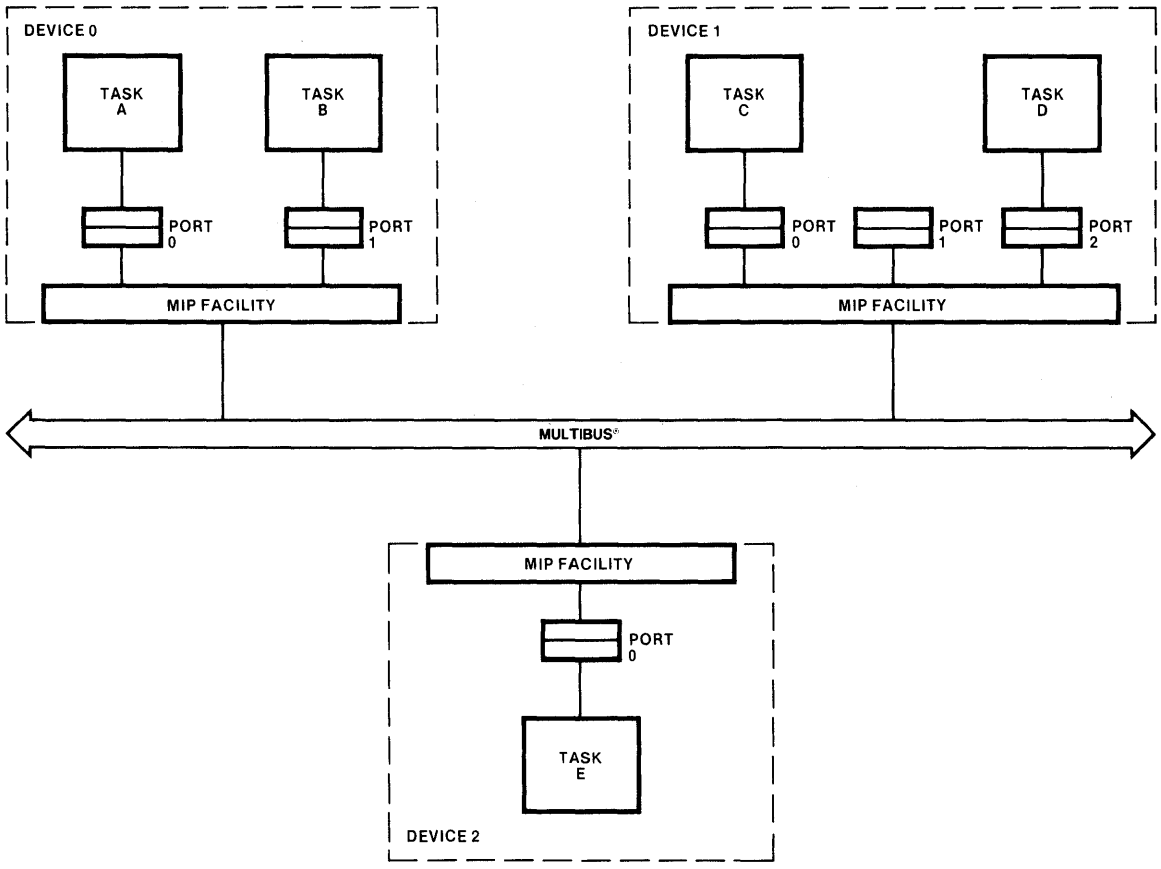

Figure E-2. A Configuration of Ports

121769-22

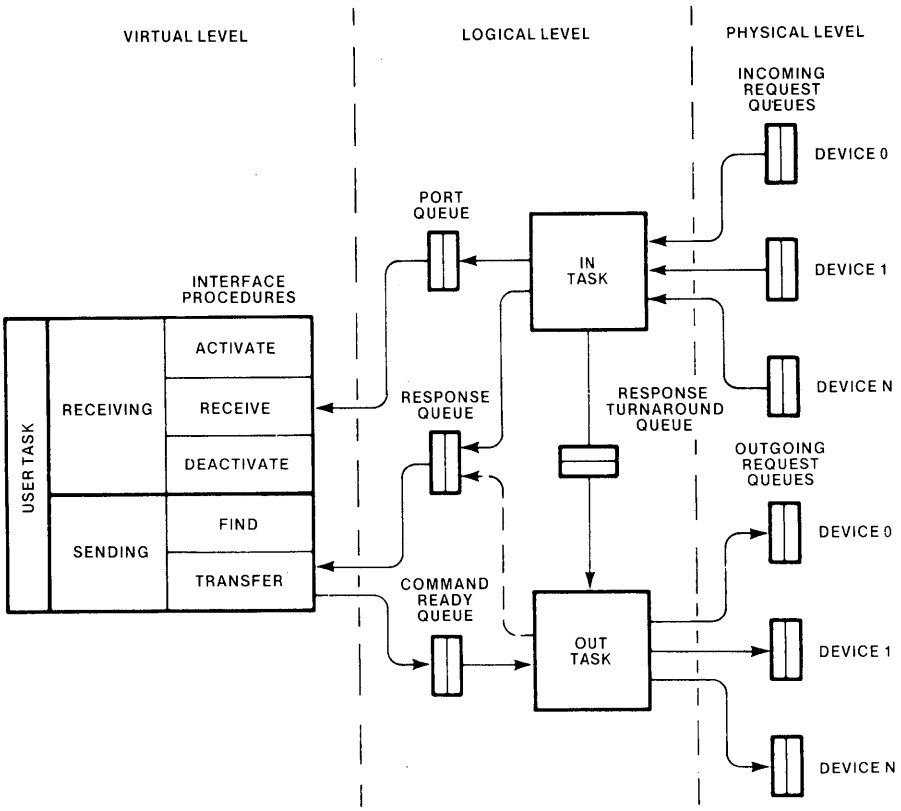

Figure E-3. Data-Flow Structure of the MIP Model 121769-23

E.3.3 Physical Level

The physical communication mechanism between devices is a fixed size, undirectional, 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 E-4. The RQD keeps track of the give and take locations, and 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 E-5 for a conceptual diagram of a channel.

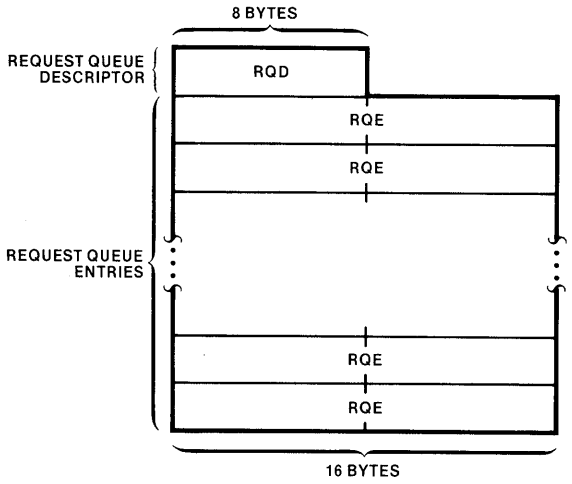

Figure E-4. Format of a Request Queue 121769-4

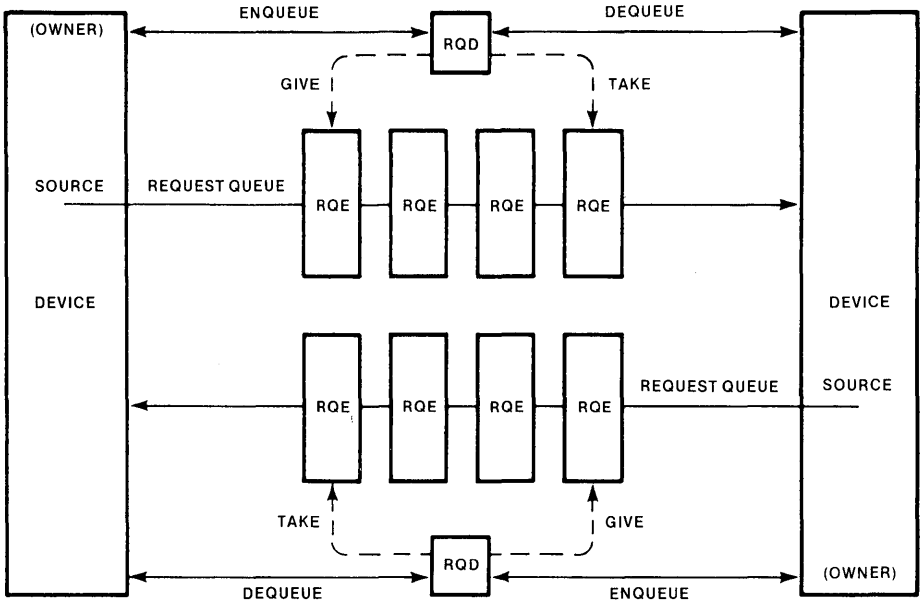

Figure E-5. Conceptual Structure of a Channel 121769-24

E.3.4 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 response is returned from the destination facility to the source facility and indicates the result of an attempt to deliver a command. The request queues carry these requests and their associated parameters between MIP facilities.

The primary procedures of a 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 multitasking environment. Because 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 E-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 tasks's device.

E.3.5 Virtual Level

User tasks interact with the MIP facility via the following 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.

E.3.6 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 so 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 *interdevice 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 E-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 interdevice communication.

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

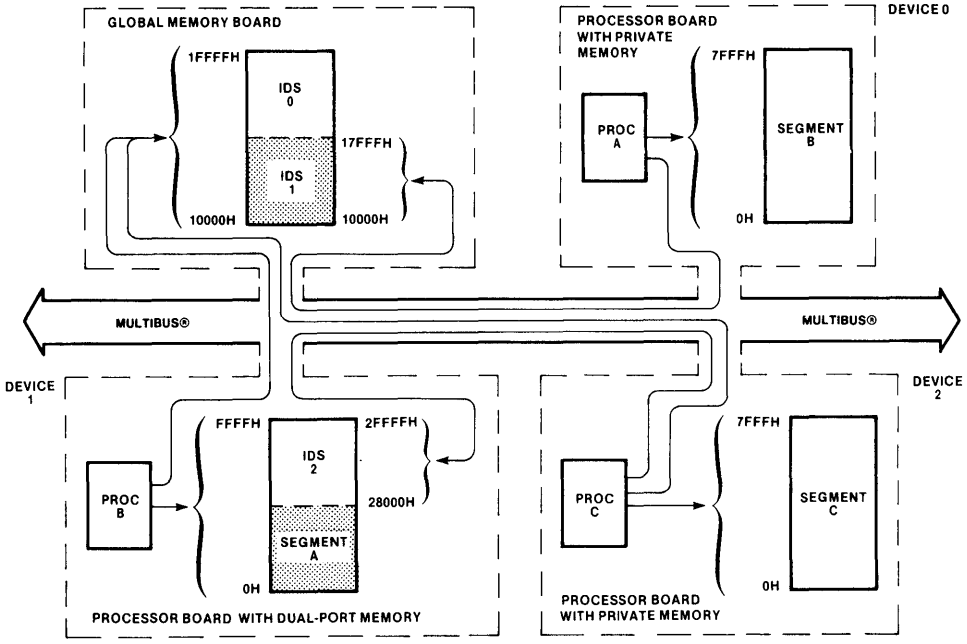

Figure E-6. Example of Interdevice Memory Segments 121769-6

Table E-1. System Interdevice Segment Table

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

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 RQED. 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.

E.3.7 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.

E.3.8 Signaling

MIP uses a signaling mechanism for efficient utilization of the interdevice 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 regularly.
- I-O MAPPED. Some devices (such as the iSBC 550 Ethernet Communications Board) recognize a write to a specific I-O port address as an interrupt. This technique is highly reliable; 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 technique is also reliable.

- 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 interrrupt line is shared by several devcies, the sending device must drop the line after a limited time to avoid continually reinterrupting 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.

E.3.9 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.

E.4 Procedural Specification

E.4.1 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.

E.4.2 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;
```

E.4.3 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 pointed 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;
```

E.4.4 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;
```

E.4.5 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;
```

E.4.6 GENERATE$INTERRUPT

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

```
GENERATE$INTERRUPT: PROCEDURE (DEVICES$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;
```

E.4.7 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;
```

E.5 Physical Level

E.5.1 Request Queue Descriptor

A request queue descriptor controls a request queue. The request queue descriptor 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$LENGTH defines the number of bytes in a request queue element (RQE). The number of elements is 2 to the power RQE$LENGTH. For all queues shared between MIP facilities, REQ$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.

```
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.

E.5.2 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 the 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 interdevice segment contains the buffer.

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

RSVRD is undefined space reserved for future expansion.

E.5.3 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',
        GIVE$DISABLED       LITERALLY    '10H',
        TAKE$DISABLED       LITERALLY    '10H',
        POINTER$MASK        LITERALLY    '7FH';
```

E.5.4 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;
```

E.5.5 TERM$REQUEST$QUEUE

This procedure sets the request queue flags to prevent subsequent acitivity 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;
```

E.5.6 QUEUE$GIVE$STATUS

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

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

DECLARE RQD$PTR     POINTER,     /* Input  */
        STATUS      BYTE;         /* Output */
```

```
DECLARE RQD BASED RQD$PTR RQD$STRUCTURE;

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

END QUEUE$GIVE$STATUS;
```

E.5.7 REQUEST$GIVE$POINTER

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

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

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

DECLARE RQD BASED RQD$PTR RQD$STRUCTURE;

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

END REQUEST$GIVE$POINTER;
```

E.5.8 RELEASE$GIVE$POINTER

This algorithm makes a previously give-requested RQE available for take.

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

DECLARE     RQD$PTR       POINTER,     /* Input  */
            STATUS        BYTE;        /* Output */

DECLARE RQD BASED RQD$PTR RQD$STRUCTURE,
            TEMP WORD;

      TEMP = RQD.GIVE$INDEX AND GIVE$FACTOR;

      IF ((RQD.TAKE$INDEX AND POINTER$MAKS) =
            (((RQD.GIVE$INDEX AND POINTER$MASK) + 1) AND
            (RQD.RQ$SIZE - 1)) THEN
            TEMP = (NOT (RQD.TAKE$INDEX AND TAKE$FACTOR)) AND
                  POINTER$MASK;
      RQD.GIVE$INDEX = (((RQD.GIVE$INDEX AND POINTER$MASK)
                  + 1) AND (RQD.RQ$SIZE - 1)) OR TEMP;
      IF (RQD.GIVE$INDEX AND POINTER$MASK) =
                  (((RQD.TAKE$INDEX AND POINTER$MASK) + 1)
                  AND (RQD.RQ$SIZE - 1)) THEN
            STATUS = FIRST$GIVE;
      ELSE
            STATUS = READY;
      RETURN;

END RELEASE$GIVE$POINTER;
```

E.5.9 REQUEST$TAKE$POINTER

This algorithm returns the address of a request queue element from the head (receive side) of a request queue.

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

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

DECLARE RQD BASED RQD$PTR RQD$STRUCTURE;

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

END REQUEST$TAKE$POINTER;
```

E.5.10 RELEASE$TAKE$POINTER

This algorithm makes a previously take-requested RQE available for give.

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

DECLARE    RQD$PTR POINTER,      /* Input  */
           STATUS BYTE;          /* Output */

DECLARE RQD BASED RQD$PTR RQD$STRUCTURE,
        TEMP WORD;

      TEMP = RQB.TAKE$INDEX AND TAKE$FACTOR;
      IF ((RQD.GIVE$INDEX AND POINTER$MASK) =
            (((RQD.TAKE$INDEX AND POINTER$MASK) + 1) AND
            (RQD.RQ$SIZE - 1))) THEN
            TEMP = RQD.GIVE$STATE AND GIVE$FACTOR;
      RQD.TAKE$INDEX = (((RQD.TAKE$INDEX
            AND POINTER$MASK) + 1)
            AND (RQD.RQ$SIZE - 1)) OR TEMP;
      IF (RQD.TAKE$INDEX AND POINTER$MASK) =
            (((RQD.GIVE$INDEX AND POINTER$MASK) + 1) AND
            (RQD.RQ$SIZE - 1)) THEN
            STATUS = FIRST$TAKE;
      ELSE
            STATUS = READY;
      RETURN;

END RELEASE$TAKE$POINTER;
```

E.6 Logical Level Database

E.6.1 Configuration Constants

The following constants define the system configuration. In place of the descriptions printed in italics, 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;
```

E.6.2 Destination Socket Descriptor Table (DSDT)

The DSDT contains information for locating sockets in a MIP system. Each entry associates a socket with a unique funtion-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 the algorithmic specification, DEST$PORT$ID is the index of the port in the Local Port Table on the destination device.

E.6.3 Local Port Table (LPT)

The Local Port Table is the list of ports and their parameters that are managed by a device. For the purpose 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 as follows:

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

E.6.4 Device to Channel Map (DCM)

The DCM table is used to route messages among intertask and interdevice 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 received device.

COMRDYQUEUE$PTR is the address of the local queue of responses 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.

E.6.5 Interdevice Segment Table (IDST)

The IDST defines the attributes of interdevice segments (IDSs). There is one entry for each IDC 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.

E.6.6 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);
```

E.7 Local Level Algorithms

E.7.1 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$PTR,
           STATUS);
  END /* THEN */;
  ELSE /* No more outstanding command. */ DO;
   DCM(DEVICE$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;
```

E.7.2 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;
```

E.7.3 SERVE$COMMAND$QUEUE

This subroutine of OUT$TASK transfers 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;
```

E.7.4 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;
```

E.7.5 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$LENGTH, /* Copies buffer */
           RQE$MSG$PTR, LOCAL$DATA$PTR); /* to port queue. */
      CALL RELEASE$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;
```

E.7.6 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$REQ.SRC$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;
```

E.7.7 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;
```

E.8 Virtual Level

E.8.1 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';
```

E.8.2 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;
```

E.8.3 TRANSFER$BUFFER

This function generates 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$ID        = SOCEKT$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;
```

E.8.4 ACTIVATE$SYSTEM$PORT

This funciton 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;
```

E.8.5 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;
```

E.8.6 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;
```

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating)._____

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP CODE _____

(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.