intel®

iMMX™ 800
MULTIBUS® MESSAGE EXCHANGE
REFERENCE MANUAL

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

Order Number: 144912-001

iMMX™ 800
MULTIBUS® MESSAGE EXCHANGE REFERENCE MANUAL

Order Number: 144912-001

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

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

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

The information in this document is subject to change without notice.

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 be used to identify Intel products:

BXP	Intel	iSBC	Multibus
CREDIT	Int$_e$l	iSBX	Multichannel
i	Intelevision	iSXM	Multimodule
ICE	Intellec	Library Manager	Plug-A-Bubble
iCS	Intellink	MCS	PROMPT
i$_m$	iOSP	Megachassis	RMX/80
iMMX	iPDS	Micromainframe	System 2000
Insite	iRMX	Micromap	UPI

A702/882/3K

PREFACE

This manual describes how to use iMMX 800 software to augment iRMX 80-, iRMX 88-, or iRMX 86-based application systems to allow tasks on different iSBC boards to communicate over the Multibus system bus.

It is assumed that readers of this manual already are familiar with either the iRMX 80 or iRMX 88 Executive or the iRMX 86 Operating System.

The manuals listed below provide reference information concerning Intel hardware and software products with which the iMMX 800 modules may be used:

- Introduction to the iRMX™ 80/88 Real-Time Multitasking Executives, Order Number: 143238
- iRMX™ 80 User's Guide, Order Number: 9800522
- iRMX™ 80 Installation Instructions, Order Number: 9803087
- iRMX™ 80/88 Interactive Configuration Utility User's Guide, Order Number: 142603
- iRMX™ 88 Reference Manual, Order Number: 143232
- iRMX™ 88 Installation Instructions, Order Number: 143241
- Guide to Writing Device Drivers for the iRMX™ 86 and iRMX™ 88 I/O Systems, Order Number: 142926
- Introduction to the iRMX™ 86 Operating System, Order Number: 9803124
- iRMX™ 86 Nucleus Reference Manual, Order Number: 9803122
- iRMX™ 86 Terminal Handler Reference Manual, Order Number: 143323
- iRMX™ 86 Debugger Reference Manual, Order Number: 143324
- iRMX™ 86 Basic I/O System Reference Manual, Order Number: 9803123
- iRMX™ 86 Extended I/O System Reference Manual, Order Number: 143308
- iRMX™ 86 System Programmer's Reference Manual, Order Number: 142721
- iRMX™ 86 Configuration Guide, Order Number: 9803126
- iRMX™ 86 Installation Guide, Order Number: 9803125

- PL/M-80 Programming Manual, Order Number: 9800268
- PL/M-86 Programming Manual for 8080/8085-Based Development Systems, Order Number: 9800466
- PL/M-86 Compiler Operating Instructions for 8080/8085-Based Development Systems, Order Number: 9800478
- PL/M-86 User's Guide for 8086-Based Development Systems, Order Number: 121636
- ISIS-II User's Guide, Order Number: 9800306
- User's Guide for the iSBC® 957B iAPX 86, 88 Interface and Execution Package, Order Number: 143979
- iSBC® 80/24 Hardware Reference Manual, Order Number: 142648
- iSBC® 80/30 Hardware Reference Manual, Order Number: 9800611
- iSBC® 86/05 Hardware Reference Manual, Order Number: 143153
- iSBC® 86/12A Hardware Reference Manual, Order Number: 9803074
- iSBC® 86/14 and iSBC® 86/30 Single Board Computer Hardware Reference Manual, Order Number: 144044
- iSBC® 88/25 Single Board Computer Hardware Reference Manual, Order Number: 143825
- iSBC® 88/40 Measurement and Control Computer Hardware Reference Manual, Order Number: 142978
- iSBC® 88/45 Advanced Data Communications Processor Board Hardware Reference Manual, Order Number: 143824
- iSBC® 544 Intelligent Communications Controller Board Hardware Reference Manual, Order Number: 9800616
- iSBC® 550 Ethernet* Communications Controller Hardware Reference Manual, Order Number: 121746
- Ethernet Communications Controller Programmer's Reference Manual, Order Number: 121769
- iSBC® 569 Intelligent Digital Controller Board Hardware Reference Manual, Order Number: 9800845

* Ethernet is a trademark of the Xerox Corporation.

CONTENTS

PAGE

CHAPTER 1
INTRODUCTION TO THE iMMX 800 SOFTWARE
iMMX 800 Application Example..... 1-1
Hardware Environment..... 1-3
Software Requirements for iMMX 800-Based Systems..... 1-4
How This Manual is Organized..... 1-4

CHAPTER 2
The iMMX 800 INTERDEVICE COMMUNICATION MODEL
Intertask Message Sender/Receiver Model..... 2-1
Interdevice Message Transfers..... 2-2
  System and Local Ports..... 2-2
  Channels..... 2-3
iMMX 800 Message Exchange Services..... 2-3
  Message-Transfer Protocol..... 2-4
    The Find Port (CQFIND) Service..... 2-4
    The Transfer Message (CQXFER) Service..... 2-4
    The Lose Port (CQLOSE) Service..... 2-4
  Message-Reception Protocol..... 2-4
    The Activate Port (CQACTV) Service..... 2-5
    Standard Message-Reception Calls..... 2-5
    The Deactivate Port (CQDACT) Service 2-5
Interdevice Message-Exchange Protocol..... 2-5
Intertask Message Exchanges on a Single Device..... 2-7
iMMX 800 Memory Configuration and Management..... 2-7
The Mechanics of Message Transfers..... 2-8

CHAPTER 3
MMX 80 PROCEDURE CALLS
PL/M-80 Language Interface..... 3-1
iRMX 80 Message Structure..... 3-1
Condition Codes..... 3-1
MMX 80 Procedure Summary..... 3-1
  Find Port..... 3-3
  Transfer Message..... 3-5
  Lose Port..... 3-9
  Activate Port..... 3-10
  Message Reception..... 3-12
  Deactivate Port..... 3-13
  MMX 80 Usage Examples..... 3-14

CHAPTER 4
MMX 88 PROCEDURE CALLS
PL/M-86 Language Interface..... 4-1
A Notational Convention for MMX 88 Discussions..... 4-1
iRMX 88 Message Structure..... 4-2
Condition Codes..... 4-2

CONTENTS (continued)

PAGE

CHAPTER 4 (continued)
MMX 88 Procedure Summary..... 4-2
Find Port..... 4-4
Transfer Message..... 4-6
Lose Port..... 4-11
Activate Port..... 4-12
Message Reception..... 4-14
Deactivate Port..... 4-15
MMX 88 Usage Examples..... 4-16

CHAPTER 5
MMX 86 PROCEDURE CALLS
PL/M-86 Language Interface..... 5-1
Condition Codes..... 5-1
MMX 86 Procedure Summary..... 5-1
Find Port..... 5-3
Transfer Message..... 5-5
Lose Port..... 5-9
Activate Port..... 5-10
Message Reception..... 5-12
Deactivate Port..... 5-14
MMX 86 Usage Examples..... 5-15

CHAPTER 6
PARTITIONED MEMORY MANAGER
Memory Pools..... 6-1
Using the Free Space Pool..... 6-2
Using Pools 0 Through N..... 6-3
Requesting Memory..... 6-3
Returning Allocated Memory..... 6-5
Creating Memory Pools Dynamically..... 6-6

CHAPTER 7
CONFIGURING YOUR APPLICATION SYSTEM
Software Configuration..... 7-1
Decisions that Provide Information Needed for Configuration..... 7-1
System-Level Decisions..... 7-1
Device-Level Decisions..... 7-2
Port-Level Decision..... 7-4
Variables and Data Structures That Must Be Assigned Values..... 7-4
Device Description (CQDVCS)..... 7-4
Channel Description (DCMROM, DCMRAM)..... 7-4
Port Descriptions (CQPRTS, LPTROM, LPTRAM)..... 7-6
Address Description (CQSKTS, DSDT)..... 7-7
Attribute Description (SFT, CQITWT, CQMDLY, CQIDPD, CQSGLV, CQLMEX, MCBI)..... 7-8

PAGE

CHAPTER 7 (continued)
Memory Description (CQIDSS, IDST)..... 7-13
Memory Assignment (CQPLHS, PLHTBL, CQBLKS, BKLTBL)..... 7-14
A Comprehensive View of the System Data Structures..... 7-15
An Example of iMMX 800 Configuration..... 7-15
Making the Decisions..... 7-15
Filling the Structures..... 7-20
Linking and Locating iMMX 800 Application Systems..... 7-33
Linking and Locating for MMX 80..... 7-34
Linking and Locating for MMX 88..... 7-36
Linking and Locating for MMX 86..... 7-38
Hardware Configuration..... 7-42
iSBC 544 Device Interrupt Generation..... 7-42
iSBC 544 Device Interrupt Reception..... 7-42
iSBC 569 Device Interrupt Generation..... 7-42
iSBC 569 Device Interrupt Reception..... 7-42
iSBC 80/24 Device Interrupt Generation..... 7-42
iSBC 80/24 Device Interrupt Reception..... 7-43
iSBC 80/30 Device Interrupt Generation..... 7-43
iSBC 80/30 Device Interrupt Reception..... 7-43
iSBC 86/05 Device Interrupt Generation..... 7-43
iSBC 86/05 Device Interrupt Reception..... 7-43
iSBC 86/12A Device Interrupt Generation..... 7-43
iSBC 86/12A Device Interrupt Reception..... 7-44
iSBC 86/14 and iSBC 86/30 Device Interrupt Generation..... 7-44
iSBC 86/14 and iSBC 86/30 Device Interrupt Reception..... 7-44
iSBC 88/25 Device Interrupt Generation..... 7-44
iSBC 88/25 Device Interrupt Reception..... 7-44
iSBC 88/40 Device Interrupt Generation..... 7-45
iSBC 88/40 Device Interrupt Reception..... 7-45
iSBC 88/45 Device Interrupt Generation..... 7-45
iSBC 88/45 Device Interrupt Reception..... 7-45

CHAPTER 8
PERFORMANCE CONSIDERATIONS
Avoid Unnecessary Traffic on the Multibus Interface..... 8-1
Minimize the Number of Times that Messages Must be Copied..... 8-1
Distribute the Workload Among the Boards in Your System..... 8-2
Minimize the Number of Message Transfers by Using Large Messages... 8-2
Experiment with Various Interrupt Mechanisms and Polling Periods... 8-2
Experiment with Various Hardware and Software Configurations..... 8-2

APPENDIX A
MULTIBUS INTERPROCESSOR PROTOCOL (MIP)
What is MIP..... A-1
Implementing MIP..... A-2
The MIP Model..... A-2

CONTENTS (continued)

PAGE

APPENDIX A (continued)
Three-Level Interface Structure..... A-4
  Physical Level..... A-5
  Logical Level..... A-7
  Virtual Level..... A-7
Memory Management..... A-8
Buffer Movement..... A-10
Signalling..... A-10
Error Handling..... A-10
MIP Functional Specification..... A-11
Procedural Specification..... A-11
  Data Types..... A-11
  Processor-Dependent Subroutines..... A-11
  PTR$ADD..... A-11
  CONVERT$LOCAL$ADR..... A-12
  CONVERT$SYSTEM$ADR..... A-12
  TIME$WAIT..... A-13
Physical Level Specification..... A-14
  Request Queue Descriptor..... A-14
  Request Queue Entry..... A-15
  Queue Procedure Returns..... A-16
  INIT$REQUEST$QUEUE..... A-16
  TERM$REQUEST$QUEUE..... A-16
  QUEUE$GIVE$STATUS..... A-17
  REQUEST$GIVE$POINTER..... A-18
  RELEASE$GIVE$POINTER..... A-19
  REQUEST$TAKE$POINTER..... A-20
  RELEASE$TAKE$POINTER..... A-21
Logical Level Database..... A-22
  Configuration Constants..... A-22
  Destination Socket Descriptor Table (DSDT)..... A-23
  Local Port Table (LPT)..... A-23
  Device to Channel Map (DCM)..... A-23
  Inter-Device Segment Table (IDST)..... A-24
  Response Queue List (RQL)..... A-25
Logical Level Algorithms..... A-25
  DYING$CHANNEL..... A-25
  SERVE$TURNAROUND$QUEUE..... A-26
  SERVE$COMMAND$QUEUE..... A-27
  OUT$TASK..... A-28
  RECEIVE$COMMAND..... A-30
  RECEIVE$RESPONSE..... A-32
  IN$TASK..... A-33
Virtual Level..... A-35
  Status Constants..... A-35
  FIND$SYSTEM$PORT..... A-35
  TRANSFER$BUFFER..... A-36
  ACTIVATE$SYSTEM$PORT..... A-38
  DEACTIVATE$SYSTEM$PORT..... A-39
  RECEIVE$BUFFER..... A-40

PAGE

APPENDIX B
COMMUNICATION WITH AN iSBC 550 ETHERNET COMMUNICATIONS CONTROLLER
Ethernet-Related Intel Hardware and Software Products..... B-1
Putting the Hardware Together..... B-3
Writing Tasks that Communicate on an Ethernet Network..... B-4
  Building and iSBC 550 Request Block..... B-5
  Sending the Request Block to the iSBC 550 Controller..... B-5
  The Ethernet Tasks° Environment and Duties..... B-6
    Using the iRMX 86 Basic I/O System..... B-6
    Using the iRMX 86 Extended I/O System..... B-9
    Using the iRMX 88 I/O System..... B-11
Configuring, Linking, and Locating an iRMX 86 or 88 I/O System
    for Use with iSBC 550 Controllers..... B-14
  Configuring an iRMX 86 I/O System for Use with iSBC 550 Controllers B-14
    A Sample Basic I/O System Configuration File..... B-17
    A Sample MMX 86 Configuration File for the Host Device..... B-21
  Linking and Locating the Configured iRMX 86 I/O System..... B-25
  Configuring the iRMX 88 I/O System for Use with
      iSBC 550 Controllers..... B-26
    Responding to ICU Prompts..... B-27
    Modifying Files Produced by the ICU..... B-28
  A Sample MMX 88 Configuration File for the Host Device..... B-30

APPENDIX C
MMX 80 DIAGNOSTICS
RQPBHX Port Diagnostic..... C-1
MEM$INIT$STATUS Diagnostic..... C-1

APPENDIX D
iMMX 800 CONDITION CODES.....D-1

FIGURES

		PAGE
1-1.	iMMX 800-Based Application Example	1-2
2-1.	Sending and Receiving Task Models	2-2
2-2.	Dedicated Channel Example	2-3
2-3.	iMMX 800 Message Exchange Calls	2-6
2-4.	Message Transfer Diagram	2-8
3-1.	Sending Task Program Example	3-14
3-2.	Receiving Task Program Example	3-15
4-1.	Sending Task Program Example	4-16
4-2.	Receiving Task Program Example	4-17
5-1.	Sending Task Program Example	5-15
5-2.	Receiving Task Program Example	5-16
7-1.	A Level-Oriented Representation of Configuration Structures.	7-5
7-2.	The Principal iMMX 800 Configuration Data Structures	7-15
7-3.	Example Target System	7-16
7-4.	Example Target System with Channels	7-17
7-5.	Initial Allocation of Memory	7-18
7-6.	Memory Map for the Example	7-19
A-1.	MIP System Example	A-1
A-2.	System Port Configuration Example	A-4
A-3.	MIP Model Data Flow Example	A-5
A-4.	Request Queue Format	A-6
A-5.	Conceptual Structure of a Channel	A-6
A-6.	Example of Inter-Device Memory Segments	A-8
B-1.	Hardware for a System Communicating with Ethernet	B-2
B-2.	Software for a System Communicating with Ethernet	B-2
C-1.	MEM$INIT$STATUS Diagnostic Example	C-2

TABLES

3-1.	MMX 80 Procedures Summary	3-2
4-1.	MMX 88 Procedures Summary	4-3
5-1.	MMX 86 Procedures Summary	5-2
A-1.	System Inter-Device Segment Table	A-9
D-1.	iMMX 800 Condition Codes	D-1

CHAPTER 1. INTRODUCTION TO THE iMMX™ 800 SOFTWARE

The iMMX 800 software extends the communications capabilities normally available to iRMX 80-, iRMX 88-, or iRMX 86-based applications. In these applications, tasks reside on the same iSBC board and communicate with the assistance of a single operating system. The iMMX 800 software provides communication capabilities between tasks residing on different iSBC boards. The only restriction is that the boards must all have access to the same Multibus system bus. The boards can be of different types and the tasks on the boards can be executing under different operating systems. For example, with the aid of iMMX 800 software, tasks running on an iSBC 80/30 board under the supervision of the iRMX 80 Executive can communicate with tasks running on an iSBC 86/12A board under the supervision of the iRMX 86 Operating System.

Tasks executing in the iMMX 800 environment communicate by means of messages, and the format requirements (if any) of the messages are identical to those of the iRMX operating system supporting the iMMX 800 software.

There are three implementations of the iMMX 800 software and each is fully compatible with one of the iRMX 80, iRMX 88, and iRMX 86 operating systems. The three implementations are called MMX 80, MMX 88, and MMX 86, respectively. With them, you can design powerful systems that take advantage of the differing capabilities of the various iSBC boards and iRMX operating systems.

For convenience, an iSBC board with an iRMX operating system and iMMX 800 software is known as a device.

iMMX 800 APPLICATION EXAMPLE

Some of the design flexibility provided by iMMX 800 software can be seen in the example illustrated in Figure 1-1. The example shows that application tasks on each device are serviced by device-resident iMMX 800 software for interdevice message transfers.

For purposes of the example, assume you are designing a database application that allows operators at two terminals to access and modify data files.

The terminals are under the control of iRMX 80 Executives residing on their respective iSBC 80/24 processor boards, along with MMX 80 software for interdevice message transfers.

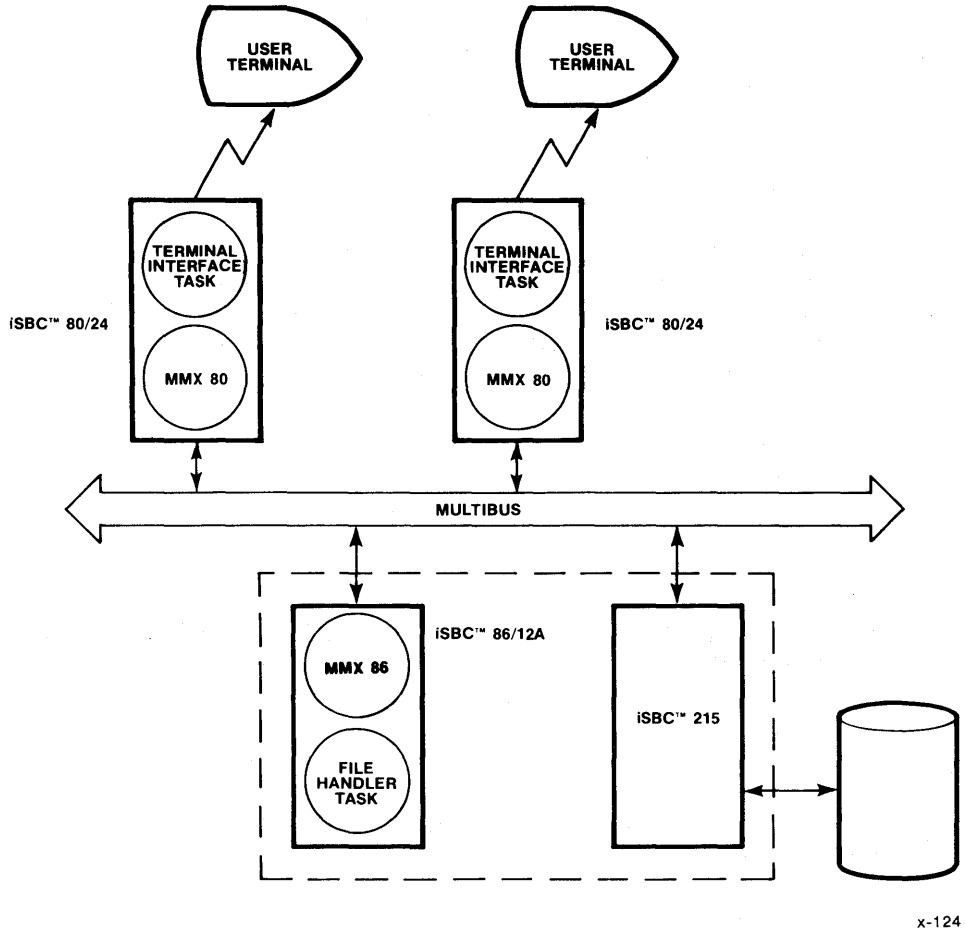


Figure 1-1. iMMX™ 800-Based Application Example

The operators have access to files on a Winchester disk, with the iRMX 86 I/O System handling requests from the terminals and performing the necessary I/O.

When an operator enters a request at a terminal, the following sequence of events occurs:

1. A task on the iSBC 80/24 board in the terminal builds a message that meets iRMX 80 message-format requirements and issues a CQXFER call to the device-resident copy of MMX 80. (CQXFER is the name of the iMMX 800 transfer procedure.)

2. MMX 80 transfers the message to MMX 86 on the iSBC 86/12A board.

3. MMX 86 reformats the message and passes it to an iRMX 86 task.

4. The I/O System performs the necessary I/O operations for the iRMX 86 task.

5. The iRMX 86 task puts the data in a message that satisfies format conventions and issues a CQXFER call to MMX 86.

6. MMX 86 transfers the message to MMX 80 on the iSBC 80/24 board.

7. MMX 80 reformats the message to meet iRMX 80 format requirements and passes it to the iRMX 80 task.

8. The iRMX 80 task extracts the data from the message and sends it to the terminal.

For a more detailed description and also for configuration procedures for this example, refer to Chapter 7.

HARDWARE ENVIRONMENT

Hardware systems supporting the iMMX 800 software are limited to three bus masters with serial bus arbitration, or 16 bus masters with parallel bus arbitration. An iMMX 800-based hardware system can employ a combination of any of the following processor and intelligent slave boards:

- iSBC 80/24 processor board, which must be a Multibus master.
- iSBC 80/30 processor board, which can be either a Multibus master or a Multibus slave.
- iSBC 86/05 processor board, which must be a Multibus master.
- iSBC 86/12A processor board, which must be a Multibus master.
- iSBC 86/14 processor board, which must be a Multibus master.
- iSBC 86/30 processor board, which must be a Multibus master.
- iSBC 88/25 processor board, which must be a Multibus master.
- iSBC 88/40 processor board, which can be either a Multibus master or a Multibus slave.
- iSBC 88/45 processor board, which can be either a Multibus master or a Multibus slave.
- iSBC 544 Intelligent Communications Controller, which can be a Multibus unimaster or slave but not a multimaster.
- iSBC 550 Ethernet* Communications Controller, which must be a Multibus master.

* Ethernet is a trademark of the Xerox Corporation.

- iSBC 569 Intelligent Digital Controller, which can be a Multibus unimaster or slave but not a multimaster.

The terms "slave," "unimaster," and "multimaster" are defined in the hardware reference manuals for the various boards.

SOFTWARE REQUIREMENTS FOR iMMX 800-BASED SYSTEMS

For each device serviced by an iMMX 800 implementation, the required software modules are as follows:

- One copy of iRMX 80 or iRMX 88 Executive software without the Free Space Manager -- the FSM is replaced by the Partitioned Memory Manager that is supplied with the iMMX 800 software -- or one copy of the iRMX 86 Operating System, depending upon the device and the application requirements.
- One copy of the MMX 80, MMX 88, or MMX 86 software.
- Any desired application tasks.

HOW THIS MANUAL IS ORGANIZED

Chapter 2 of this manual contains general descriptions of the procedure calls that are part of each implementation of the iMMX 800 software.

Chapters 3, 4, and 5 give the specific descriptions and calling sequences for the procedures as they are implemented in MMX 80, MMX 88, and MMX 86, respectively. The semantics of the procedures in the different implementations are very similar, but the syntax requirements are somewhat diverse, due to the differences among the iRMX operating systems.

Chapter 6 discusses the Partitioned Memory Manager, which is part of each implementation of the iMMX 800 software. The Partitioned Memory Manager is much like the iRMX 80 and iRMX 88 Free Space Managers, and, in fact, replaces the FSM in applications using MMX 80 or MMX 88.

Chapter 7 explains how to configure your hardware and software for iMMX 800 applications. It also expands upon the example of this chapter, giving the example's configuration files for MMX 80 and MMX 86.

Appendix A describes the MIP (Multibus Interprocessor Protocol) that the iMMX 800 services follow.

Appendix B discusses using the iMMX 800 software to build a Multibus-based system that can communicate with an Ethernet controller.

Appendix C describes two diagnostic tools for debugging MMX 80-based applications.

Appendix D gives the mnemonics and numeric values of the condition (status) codes that the iMMX 800 procedure calls can return.

CHAPTER 2. THE iMMX™ 800 INTERDEVICE COMMUNICATION MODEL

This chapter introduces you to the intertask message-exchange model, protocols, and memory structures for interdevice message transfers. You need a good understanding of this architecture before you can design and create an iMMX 800-based application system.

In the iMMX 800 context, a "device" refers to a single iSBC board that contains its own copy of the iMMX 800 software, an iRMX operating system that controls the device, and application tasks. In an interdevice message transfer, an application task on one device sends a message to an application task on another device.

INTERTASK MESSAGE SENDER/RECEIVER MODEL

The message sender/receiver model should be familiar to most iRMX operating system users. It defines a simple system that consists of two types of tasks: those that receive data, and those that transfer data.

A message-receiving task waits for a message to be posted at a particular exchange or mailbox and takes control of the processor only when it has received a message. This task performs an action that might be based on the content of the message and then waits until it receives another message. Usually, the receiving task acknowledges completion of its function by returning the message to an exchange or mailbox where the sending task is waiting for a response.

A message-sending task initiates its function by transferring a message to an exchange or mailbox. The task can wait until it receives a response to its message, or it can continue to run while the receiving task processes the message.

Generally, the distinction between message-sending and message-receiving tasks is not absolute, because many tasks both send and receive messages. However, the sender/receiver concept presented in Figure 2-1 helps clarify the general interaction of tasks.

Because intertask communication is through an exchange or mailbox, messages containing data are queued automatically. Thus, a sending task can be allowed to "get ahead" of a receiving task without loss of data.

The iRMX software supports the sender/receiver communication model on a single device. The iMMX 800 software services generalize the model, supporting it for communication between devices.

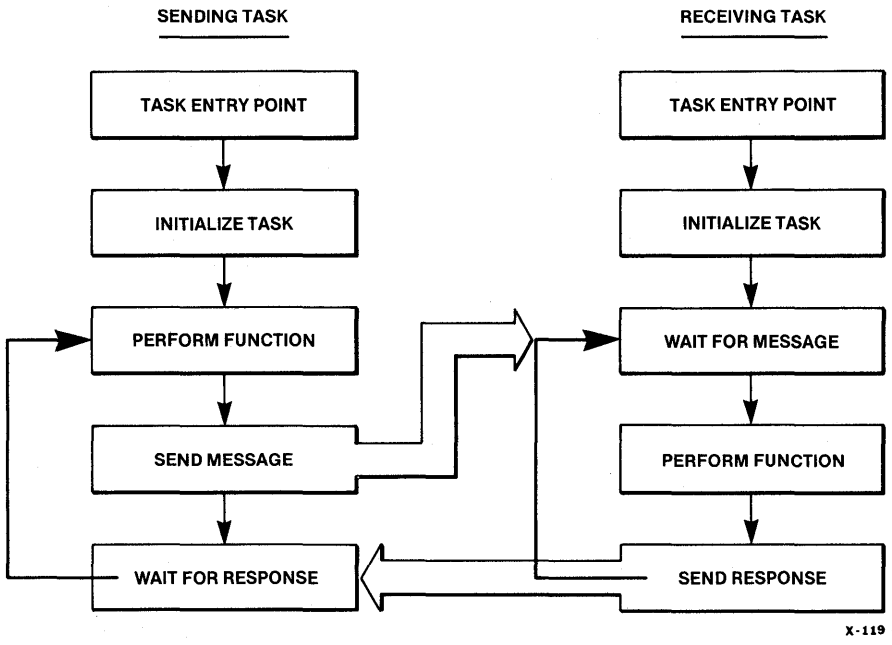


Figure 2-1. Sending and Receiving Task Models

INTERDEVICE MESSAGE TRANSFERS

Tasks using the iMMX 800 services to transfer and receive messages see those services as procedures that are associated with ports. The link between the port known to the sending tasks and the port known by the receiving task is a channel.

SYSTEM AND LOCAL PORTS

In iRMX-based applications, tasks send and receive messages through exchanges or mailboxes. In iMMX 800-based systems, a task might not know which operating system supports the task with which it is communicating, so the iMMX 800 software provides ports, which are similar to exchanges and mailboxes.

A port can be viewed by a task in two ways, depending upon the task's intentions. A task intending to transfer a message to another task by means of iMMX 800 services views a message's destination as a system port. On the other hand, a task intending to receive a message views the same port as a local port that resides on the same device as does the task. On devices controlled by iRMX 80 or iRMX 88 Executives, local ports are exchanges. On devices controlled by iRMX 86 Operating Systems,

local ports are mailboxes. In either case, the resident operating system provides the software support for the local port, so tasks on a device receive all messages, regardless of their origin, by using the message-reception system call provided by the operating system on the device.

CHANNELS

The iMMX 800 software supports the association between system and local ports by providing a channel. Although a channel can be thought of as an interdevice pipeline through which messages can be transferred (see Figure 2-2), it is actually a pair of single-direction request queues in memory shared by the two devices. See Appendix A for a more detailed description of request queues.

The concept of "channel" is used to emphasize that the request-queue pair is dedicated to the exclusive use of these two devices for interdevice message transfers. No other device in an application has access to that channel. If one of the devices also communicates with another device in the system, the two devices use another channel for message transfers.

iMMX 800 MESSAGE EXCHANGE SERVICES

The iMMX 800 software provides services that application tasks see as procedures for transferring and receiving messages.

The following sections provide a general description of the services, and describe how they are used in combination to accomplish message transfer and reception.

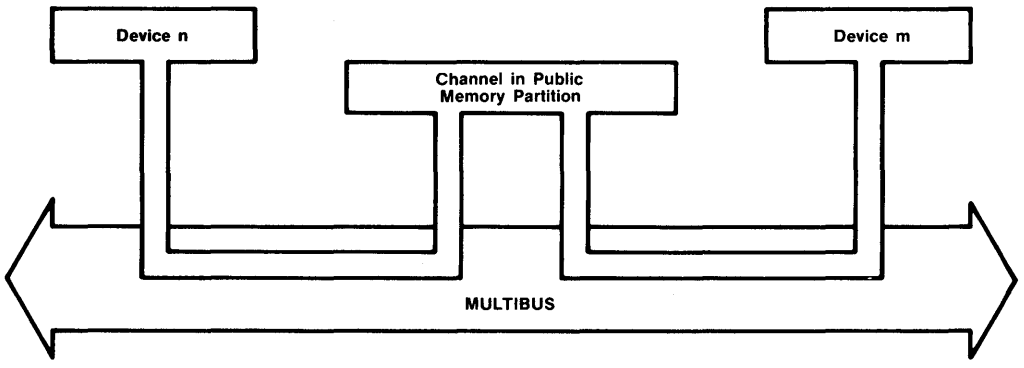


Figure 2-2. Dedicated Channel Example

MESSAGE TRANSFER PROTOCOL

The following sections discuss the services that the sending task uses.

The FIND PORT (CQFIND) Service

When an application task wishes to send a message to a task on another device, it must first "locate" the destination system port. The task does this by invoking the FIND PORT service (CQFIND procedure). The task identifies the system port by its system-port name. This system-port name, which is defined during software configuration (see Chapter 7), distinguishes the system port from all other system ports in the entire application.

The call to CQFIND causes the iMMX 800 software to return a unique connection to the calling task. The sending task uses the connection as a parameter when it subsequently calls the CQXFER procedure to send messages.

Each task wanting to send a message to the same system port must invoke CQFIND in order to get its own connection to that port.

The TRANSFER MESSAGE (CQXFER) Service

If the call to CQFIND is successful, the sending task calls the TRANSFER MESSAGE service (CQXFER procedure) to transfer messages, using the connection returned by CQFIND as a parameter. The task can use the connection to send as many messages as it wants to send.

The LOSE PORT (CQLOSE) Service

If and when a task has no further messages to transfer to the system port by means of CQXFER calls, it invokes the CQLOSE procedure to release the system resources that have been used to support the task's message transfers. After the call to CQLOSE, the connection no longer refers to the associated system port and cannot be used again by the sending task.

If the task later wishes to transfer more messages to that system port, it must again invoke the CQFIND procedure to obtain a new connection to the port.

MESSAGE-RECEPTION PROTOCOL

The following sections discuss the services that the receiving task uses.

The ACTIVATE PORT (CQACTV) Service

Until a port is activated by some task resident on a device, tasks cannot receive messages at that system port. Activating a port enables iMMX 800 services to deliver messages to the local exchange or mailbox associated with that port.

Some device-resident task initially activates the local port by invoking the CQACTV procedure. A parameter in the call to CQACTV identifies the port by means of its system-port name. Once the port is activated, iMMX 800 services deliver all messages sent to that port to the associated exchange or mailbox.

The CQACTV procedure returns an exchange address or a mailbox token, depending upon which iRMX operating system is resident on the device. The requesting task uses the returned address or token as input in subsequent iRMX calls for message reception.

Standard Message-Reception Calls

Receiving tasks invoke the standard message-reception calls provided by the device-resident operating system: RQWAIT and RQACPT for iRMX 80 and iRMX 88 Executives, and RQ$RECEIVE$MESSAGE for iRMX 86 Operating Systems.

Once a device-resident task has activated a local port, all other tasks on that device can receive messages at that port. Note, however, that receiving tasks cannot distinguish between messages that come from device-resident tasks and those that come from tasks on other devices, unless the application has made special provisions that make this distinction for the receiving tasks.

The DEACTIVATE PORT (CQDACT) Service

If and when it is appropriate, a task on the same device as a system port deactivates (CQDACT) that port. The task identifies the port by means of that port's system-port name. Once the port is deactivated, no further messages can be sent to it. Thereafter, any task attempting to receive a message at that port receives an exceptional condition. However, transferred messages that are already queued at the exchange or mailbox when the call to CQDACT is made are not affected, and these messages can be received by tasks.

INTERDEVICE MESSAGE-EXCHANGE PROTOCOL

The flow diagram illustrated in Figure 2-3 shows the various interfaces used in the interdevice message-exchange protocol. Note that the message travels only one way, and that the receiving task on device B does not return a reply to the sending task. If the sending task on device A expects a reply, it must invoke (prior to its first CQXFER call) a call to CQACTV to activate a system port on its own device for a reply

message. Similarly, the receiving task on device B must invoke the CQFIND and CQXFER calls to send a reply to device A before waiting for further messages.

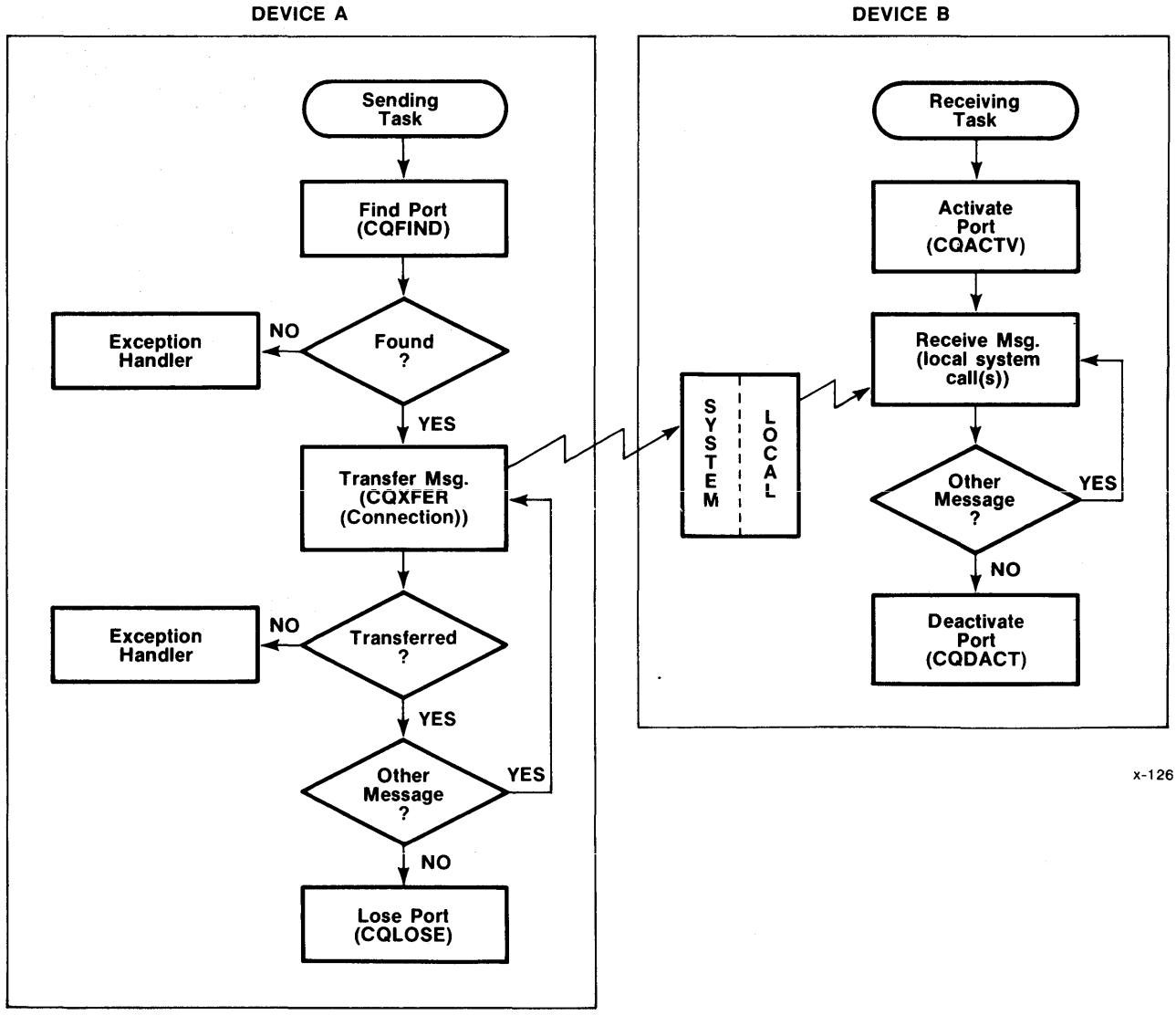


Figure 2-3. iMMX™ 800 Message Exchange Calls

INTERTASK MESSAGE EXCHANGES ON A SINGLE DEVICE

The iMMX 800 message-exchange services can optionally be used for local intertask message transfers on a single device. One possible reason for doing so is on-board emulation of an entire application during system development. That is, the total application is resident on a single processor board for testing and debugging and, once debugged, is off-loaded to its intended devices.

iMMX 800 MEMORY CONFIGURATION AND MANAGEMENT

In any system that employs the iMMX 800 services, some of the RAM on or accessible by each device must be managed by the Partitioned Memory Manager that resides on that device. Memory that must be managed is divided into pools, where a pool is a contiguous area of RAM. If a pool is to be shared between devices, it must lie in an interdevice segment, which is a contiguous area of RAM with the following characteristics:

- It consists entirely of non-overlapping pools.
- All of it must be addressable by both devices. An example of such memory is the dual-port RAM on an iSBC 80/30 board.

This means that there are two kinds of pools that the Partitioned Memory Manager on a device manages: pools that lie entirely inside an interdevice segment and are used for message-passing between devices or occasionally between tasks on the device; and pools that lie entirely outside of all interdevice segments and are used for message-passing between tasks on the device.

The Partitioned Memory Manager on a device can manage up to 255 memory pools. Each of those pools is identified by a pool id that you specify during configuration. Each pool's id must be in the range 0 through 254. The pool denoted by the zero (0) pool id is defined as the "Free Space Pool." The PMM supports allocation and reclamation of memory from this Free Space Pool in a manner that is compatible with the iRMX 80 and iRMX 88 Free Space Managers. Existing iRMX 80- or iRMX 88-based applications designed to use the Free Space Manager can use the PMM without requiring any changes.

The iSBC 80/30, iSBC 86/12A, iSBC 86/14, iSBC 86/30, iSBC 88/40, iSBC 544, and iSBC 569 boards each contain dual-port RAM. This memory is accessible through both the processor's local bus and the Multibus system bus. When two or more devices access a given memory location they need not do so by using the same address. Instead, they can use "alias addressing," where an on-board processor accesses a range of dual-port memory locations by one set of addresses, and other processors access the same range of dual-port memory locations by a different set of addresses. You define alias addresses at iMMX 800 configuration time.

THE MECHANICS OF MESSAGE TRANSFERS

If special arrangements were not provided, a message created by a sending task might be inaccessible by the intended receiving task. Of course, this would prevent the message transfer from being successful. And this is not the only possible obstacle to a successful message transfer. Because the receiving task does not know which operating system controlled activities on the sending device, it cannot know whether the sending task put header information at the beginning of the message, nor can it know how much header information there is. Because of these potential obstacles to message transfers, every message transfer must have the following properties:

- At some stage of the transfer process, the message must be in memory that is accessible by both the sending and receiving devices.
- When the message is accessible by both devices, it must have a (generic) form that is completely independent of both the sending and receiving operating systems.

To ensure that these properties are always in evidence, the iMMX 800 software requires that every message be copied into shared memory and that all header information be stripped in the process. This is the first copy operation required by the iMMX 800 software, and it can be performed in either of two ways. If the task requests in its call to CQXFER that the iMMX 800 software perform the copy operation, the transfer is called transparent; otherwise, it is non-transparent.

The second copy operation is always performed by the iMMX 800 software, and it always copies the message from shared memory into memory that is accessible only by the receiving device. If the receiving device is controlled by iRMX 80 or iRMX 88 executive, then the required header information is added on during the second copy operation.

Figure 2-4 illustrates a typical message transfer from a task on an iSBC 80/30 board to a task on an iSBC 86/12A board.

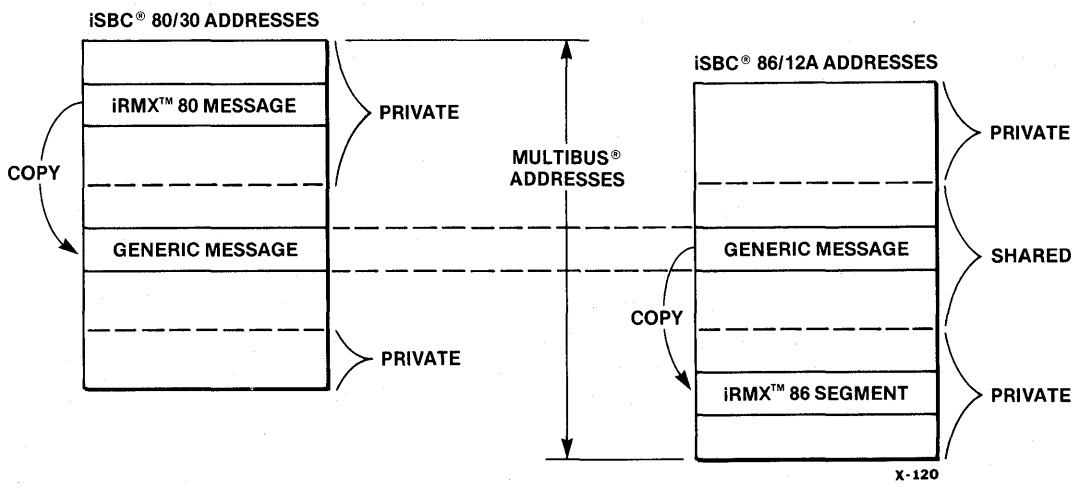


Figure 2-4. Message Transfer Diagram

CHAPTER 3. MMX 80 PROCEDURE CALLS

The procedure calls described in this chapter apply only to tasks running under the supervision of the iRMX 80 Executive. Although the iMMX 800 software is a single product, in the MMX 80 implementation, the syntax requirements of procedure calls are different than the syntax requirements of corresponding calls in the MMX 88 and MMX 86 implementations.

PL/M-80 LANGUAGE INTERFACE

The MMX 80 procedures described in this chapter are defined in PL/M-80. See the section of Chapter 7 entitled "Linking and Locating iMMX 800 Application Systems" for the names of files containing EXTERNAL declarations of the procedures.

iRMX 80 MESSAGE STRUCTURE

The iRMX 80 message structure has the following fields in the following order:

```
LINK            ADDRESS
LENGTH          ADDRESS
TYPE            BYTE
HOME$EX         ADDRESS
RESP$EX         ADDRESS
MSG$AREA(*)     BYTE
```

CONDITION CODES

After each call to an MMX 80 procedure, MMX 80 returns to the calling task a status value called a condition code. The condition code reflects the success or failure of the call. In case of failure, the code indicates the reason for the failure. Consequently, tasks should always check the condition code immediately after issuing an MMX 80 call.

MMX 80 PROCEDURE SUMMARY

Table 3-1 provides a summary description of the MMX 80 procedures for fast reference.

Table 3-1. MMX 80 Procedures Summary

Procedure	Parameters	Description
FIND PORT CQFIND	Input Values: sys$port$name condition$ptr Returned Value: connection	Furnishes a connection for sending messages to the system port represented by the specified system-port name.
TRANSFER MESSAGE CQXFER	Input Values: connection message$ptr condition$ptr	Delivers the iRMX 80 message to the system port associated with the connection.
LOSE PORT CQLOSE	Input Value: connection	Releases the memory and connection previously acquired through a call to CQFIND. The task can no longer use the connection for message transfers to the system port.
ACTIVATE PORT CQACTV	Input Values: sys$port$name condition$ptr Returned Value: exchange$ptr	Activates a local iRMX 80 exchange that serves as the system port represented by the specified system-port name. Messages transferred to the system port are delivered to this exchange by the MMX 80 software.
WAIT FOR MESSAGE RQWAIT	Input Values: exchange$ptr time$limit Returned Value: message$ptr	Standard iRMX 80 operation that tasks use to receive messages at exchanges representing system ports. If desired, tasks can specify a waiting period.
ACCEPT MESSAGE RQACPT	Input Value: exchange$ptr Returned Value: message$ptr	Standard iRMX 80 operation that tasks use to receive messages at exchanges representing system ports. Tasks cannot specify a waiting period.
DEACTIVATE PORT CQDACT	Input Values: sys$port$name condition$ptr	Deactivates a system port that had been activated earlier by a call to CQACTV. Messages from another device can no longer be transferred to that system port. Messages still queued there can still be received by local tasks.

FIND PORT

The CQFIND procedure returns a connection for a system port. The calling task can use the connection to transfer messages to tasks on another (or the same) device.

```
connection = CQFIND (sys$port$name, condition$ptr);
```

sys$port$name	An ADDRESS containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The ADDRESS of a BYTE where MMX 80 returns the condition code for the call.
connection	An ADDRESS whose value is returned for use only by the calling task. The task uses the connection when invoking CQXFER to transfer messages to the specified system port.

DESCRIPTION

When configuring MMX 80 for this device, you specify the name and location of every system port to which tasks on this device transfer messages. CQFIND returns to the calling task a connection that identifies the system port whose name is specified in the call. The task can use the connection in calling CQXFER. If and when the task is finished making CQXFER calls with the connection, the task can call CQLOSE to return the connection to the system.

CQFIND initiates the allocation of a 32-byte block of memory from the Free Space Pool for internal needs, and also creates an exchange for MMX 80 use. The resources allocated to the calling task by means of the CQFIND procedure are returned to the system if and when the task calls the CQLOSE procedure.

The connection returned by CQFIND should be used by the task to which it is issued. If more than one iRMX 80 task on the same device needs to send messages to the same system port, each task should invoke the CQFIND procedure to obtain its own connection.

CONDITION CODES

SYSTEM$SERVICE$READY	CQFIND executed without error.
INSUFFICIENT$MEMORY	There is insufficient memory in the Free Space Pool to meet the requirements of CQFIND.
UNKNOWN$SYSTEM$PORT	The iMMX 800 software did not recognize the system-port name that the calling task supplied.

TRANSFER MESSAGE

The CQXFER procedure transfers an iRMX 80 message to the system port associated with the specified connection.

```
CALL CQXFER (connection, message$ptr, condition$ptr);
```

connection	An ADDRESS whose value identifies the system port where the specified message is to be transferred.
message$ptr	The ADDRESS of an iRMX 80 message that is to be sent to the specified system port.
condition$ptr	The ADDRESS of a BYTE where MMX 80 returns the condition code for the call.

DESCRIPTION

The TRANSFER MESSAGE service transfers a message to the system port specified by the connection. The task with the connection must invoke CQXFER for each message sent to the system port.

An application task invoking the CQXFER procedure is suspended until the message is delivered and queued at the destination port, or until the MMX 80 software detects an error while attempting to deliver the message.

The LENGTH field of the message specifies the size of the message block that was allocated by the PMM. The number of bytes actually transferred by CQXFER equals the number specified in the LENGTH field in the header of the message being transferred, minus the nine bytes of the header itself.

Do not send a zero-length message to a system port. Doing so causes unpredictable results in the device-resident iRMX 80 Executive.

The TYPE field should normally be set to MMXANYTYPE (=0) or to MMXPRELOC$TYPE (=48). Setting the TYPE field to MMXANYTYPE causes the source message's contents to be copied into a buffer that can be accessed by the destination device. Setting the TYPE field to MMXPRELOC$TYPE prevents the message from being copied. Use of MMXPRELOC$TYPE assumes that the message contents are accessible by the destination device. The following table summarizes the effects of TYPE field options for each kind of device. In the table, these statements apply:

(1) "Peer Device" refers to a device characteristic that is defined for each device during configuration for that device.

(2) Names of returned condition codes assume that no other errors occurred in the call.

(3) "A copy" is shorthand for "A copy of the message."

Destination Device	MMXANYTYPE	MMXPRELOC$TYPE
Peer device with the ability to make copies	MMX 80 makes a copy in memory that is accessible by the destination device and returns the system$message$copy$delivered condition code to the calling task. Because MMX 80 returns the message to the PMM, the message area is not free for reuse.	MMX 80 doesn't make a copy and returns the system$message$copy$delivered condition code to the calling task. When control returns to the calling task, the message area is not free for reuse.
Peer device without the ability to make copies	MMX 80 makes a copy in memory that is accessible by the destination device and returns the system$message$delivered condition code to the calling task. Because MMX 80 returns the message to the PMM, the message area is not free for reuse.	MMX 80 doesn't make a copy and returns the system$message$delivered condition code to the calling task. When control returns to the calling task, the message area is not free for reuse.

When the message block was allocated, the PMM set the HOME$EX field of the message. When the message is successfully delivered, the original copy of the message is sent for reclamation to the specified home exchange, so the task must not alter this field. If an exceptional condition arises during the transfer process, CQXFER returns an exceptional condition and MMX 80 does not send the original message to the home exchange.

The RESP$EX field is undefined for use with messages passed via the CQXFER procedure. That is, the sending task cannot use this field to tell the receiving task where to return a response.

All other fields within the message are as defined by the iRMX 80 Executive. See Chapter 6 for a more detailed description of the message fields.

If it is necessary for communicating tasks to pass additional information concerning a message block, then some user-defined convention can be adopted that utilizes a "subheader" within the message itself for conveying such information. This subheader is considered part of the message's data and will be transferred by CQXFER.

CAUTION

When you CQXFER a message from an iRMX 80-based system to an iRMX 88- or iRMX 86-based system, the MMX 88 or MMX 86 facility at the receiving end increases the size of the message in order to meet local iRMX 88 or iRMX 86 requirements. Consequently, if you use the iMMX 800 software to shuttle information back and forth between such systems many times, as in a "do forever" loop, and the task at each end always "sends" the same buffer that it just "received", then the buffers -- there are at least two, because the iMMX 800 software always make a copy on the destination device -- will grow beyond the limits of your system's memory. To prevent this from happening, one or more of the tasks should take responsibility for controlling the size of the buffers. A task using MMX 80 can exercise this control by always doing the following: (a) obtain a new memory block of the required size; (b) copy the data into this new block; (c) dispose of (reclaim) the old block; and (d) use the new block for the message transfer.

CONDITION CODES

SYSTEM$PORT$DEAD	MMX 80 has concluded that the indicated destination device is dead and therefore cannot receive transferred messages. The message$ptr remains valid.
SYSTEM$MESSAGE$COPY$DELIVERED	The destination device copied the message before it was successfully delivered. The message$ptr is no longer valid.
SYSTEM$MESSAGE$DELIVERED	The message was successfully delivered to the destination system port without being copied by the destination device. The message$ptr is no longer valid.
INSUFFICIENT$MEMORY	Not enough memory was available for local or destination buffers. The message$ptr remains valid.

SYSTEM$PORT$INACTIVE	The destination port currently is not active, so the message is not deliverable. The message$ptr remains valid.
UNKNOWN$SYSTEM$PORT	The specified connection is not valid, so the CQXFER call was not successful. The message$ptr remains valid.

LOSE PORT

The CQLOSE procedure allows a task to release resources that were previously allocated by the CQFIND procedure. After the CQLOSE call, the connection can no longer be used to transfer messages.

```
CALL CQLOSE (connection);
```

connection	An ADDRESS whose value was returned by CQFIND to the calling task, for the purpose of using CQXFER to transfer messages.

DESCRIPTION

When an iRMX 80 task no longer wishes to send messages to a specified system port, the LOSE PORT service lets the task return to the system the resources previously allocated for message transfers. The calling task surrenders the following resources when it invokes the CQLOSE call:

- Connection - the calling task can no longer use the connection to transfer messages to the system port.
- Free Space memory - the 32-byte memory block previously allocated for system use is returned to the Free Space Pool.
- An exchange - the exchange previously created by CQFIND is deleted.

CAUTION

The MMX 80 software does no validity checking when the CQLOSE procedure is called. Consequently, specifying an improper connection, or one that was invalidated by a previous CQLOSE call, causes unpredictable results in the device-resident iRMX 80 Executive.

ACTIVATE PORT

The CQACTV procedure activates a system port and creates a device-resident iRMX 80 exchange for message reception at the specified system port.

```
exchange$ptr = CQACTV (sys$port$name, condition$ptr);
```

sys$port$name	An ADDRESS containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The ADDRESS of a BYTE where MMX 80 returns the condition code for the call.
exchange$ptr	The ADDRESS of the iRMX 80 exchange that MMX 80 creates. Local tasks, including the calling task, use exchange$ptr in calls to RQWAIT and RQACPT in order to receive messages.

DESCRIPTION

The MMX 80 services do not deliver messages to a system port until that port has been activated by a call to CQACTV. When called, the CQACTV procedure attempts to associate the specified system-port name with a device-resident system port. If the system port is defined for this device and the port is not already activated, CQACTV activates the port and returns an exchange$ptr for the associated iRMX 80-exchange address.

If other device-resident tasks are to receive messages at this iRMX 80 exchange, the task calling CQACTV must pass the iRMX 80 exchange address to those other tasks.

An activated system port remains active (that is, able to receive messages) until it is deactivated by a call to the CQDACT procedure.

Although an application task can invoke the iRMX 80 system call RQCXCH, to dynamically create exchanges for communication between tasks residing on the same device, application tasks cannot call RQCXCH to create exchanges for interdevice communication. Only the system ports (which you define at iMMX 800 configuration time) can be used as exchanges for interdevice communication and each must be activated by a call to CQACTV.

CONDITION CODES

SYSTEM$SERVICE$READY	Service completed without error.
SYSTEM$PORT$ACTIVE	The indicated port is already activated.
UNKNOWN$SYSTEM$PORT	MMX 80 did not find the specified system port name when it searched the local system port table.

MESSAGE RECEPTION

RQWAIT and RQACPT are standard iRMX 80 system calls that tasks use to receive messages at exchanges. In particular, tasks use RQWAIT and RQACPT to receive messages at exchanges representing activated system ports.

```
message$ptr = RQWAIT (exchange$ptr, time$limit);
                    or
message$ptr = RQACPT (exchange$ptr);
```

exchange$ptr	The ADDRESS of an iRMX 80 exchange previously created by the CQACTV procedure.
time$limit	An ADDRESS (used in calls to RQWAIT only) whose value is the length of time (in iRMX 80 system time units) that the calling task is willing to wait for a message to arrive.
message$ptr	Normally the ADDRESS of the message at the front of the exchange's message queue. However, if the task called RQWAIT and then "timed out", message$ptr contans the address of a five-byte message of type TIMEOUTTYPE (=3).

DESCRIPTION

An application task receives messages sent to an iRMX 80 exchange by invoking the iRMX 80 system calls RQWAIT and RQACPT. The exchange is identified in the calls by exchange$ptr. If the exchange represents a system port, the exchange pointer was previously returned to an application task by the CQACTV procedure.

After a task calls RQWAIT or RQACPT, it must ascertain whether the call was successful. If it calls RQWAIT, the task receives the address of a message. The task must check the TYPE field of that message to learn whether the message is what the task was waiting for. If the value in that field is three (3), the message is from the iRMX 80 Executive and indicates that the specified time limit expired before a message from another task arrived at the exchange. Otherwise, the message is from another task.

If the task calls RQACPT, and the returned message$ptr value is zero (0), then no message was queued at the exchange. Otherwise, the value is the address of a message.

NOTE

The resp$ex field is undefined in iRMX 80 messages delivered to a system port by MMX 80 services. This field should not be used by receiving tasks.

DEACTIVATE PORT

The CQDACT procedure deactivates the specified system port. Messages are no longer delivered to that port by the device-resident MMX 80 software.

```
CALL CQDACT (sys$port$name, condition$ptr);
```

sys$port$name	An ADDRESS containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The ADDRESS of a BYTE where MMX 80 returns the condition code for the call.

DESCRIPTION

The DEACTIVATE PORT service allows an application task to deactivate a system port. After the port is deactivated, messages can no longer be sent to that port until it is re-activated by the same or another device-resident task. A SYSTEM$PORT$INACTIVE exceptional condition is returned to tasks attempting to send further messages to the deactivated port.

CQDACT does not affect messages already queued at the iRMX 80 exchange representing the system port when the CQDACT request is made. Such messages remain available to tasks on the device. MMX 80 deletes the exchange when the last remaining message is received by a task.

CONDITION CODES

SYSTEM$SERVICE$READY	Service completed without error.
UNKNOWN$SYSTEM$PORT	The iMMX 800 software did not recognize the system port name supplied by the calling task.

MMX 80 USAGE EXAMPLES

The program examples in Figures 3-1 and 3-2 show typical usage of the MMX 80 interdevice message-transfer services. The program given in Figure 3-1 represents portions of a task that sends messages to a task on another device. The task that sends messages is called the MMX$producer$task. The task to which it sends messages is similarly portrayed in Figure 3-2 and is called the MMX$consumer$task.

In the examples, the data types of the variables can be derived from context.

```
MMX$producer$task:
   DO;
       DECLARE condition$code BYTE;
       DECLARE consumer$connection ADDRESS;
       DECLARE (consumer$sys$port$name,
                producer$sys$port$name) ADDRESS EXTERNAL;
           :
   consumer$connection = CQFIND (consumer$sys$port$name,
                                 .condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   producer$exch = CQACTV (producer$sys$port$name,
                           .condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   CALL generate (.producer$message); /* generate a message */

   producer$message.type = MMX$ANY$TYPE;

   CALL CQXFER(consumer$connection,
               .producer$message,
               .condition$code);

   IF NOT (condition$code = (SYSTEM$MESSAGE$DELIVERED OR
                             SYSTEM$MESSAGE$COPY$DELIVERED))
       THEN CALL problem$handler;

   consumer$reply$message$ptr = RQWAIT (producer$exch,
                                        some$delay);
           :
   CALL CQDACT(producer$sys$port$name,
               .condition$code);
           :
END MMX$producer$task;
```

Figure 3-1. Sending Task Program Example

```
MMX$consumer$task:
   DO;
       DECLARE condition$code BYTE;
       DECLARE producer$connection ADDRESS;
       DECLARE (consumer$sys$port$name,
                producer$sys$port$name) ADDRESS EXTERNAL;
           :
   consumer$exch = CQACTV (consumer$sys$port$name,
                           .condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   producer$message$ptr = RQWAIT (consumer$exch, some$delay);

   IF producer$message.type = TIME$OUT$TYPE
       THEN CALL problem$handler;

   producer$connection = CQFIND (producer$sys$port$name,
                                 .condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   CALL generate (.reply$message); /*generate a reply*/

   producer$msg.type = MMX$ANY$TYPE;

   CALL CQXFER (producer$connection,
                .reply$message,
                .condition$code);

   IF NOT (condition$code = SYSTEM$MESSAGE$DELIVERED OR
                            SYSTEM$MESSAGE$COPY$DELIVERED)
       THEN CALL problem$handler;
           :
   CALL CQDACT (consumer$sys$port$name,
                .condition$code);
           :
END MMX$consumer$task;
```

Figure 3-2. Receiving Task Program Example

CHAPTER 4. MMX 88 PROCEDURE CALLS

The procedure calls described in this chapter apply only to tasks running under the supervision of the iRMX 88 Executive. Although the iMMX 800 software is a single product, in the MMX 88 implementation the syntax requirements of procedure calls are different than the syntax requirements of corresponding calls in the MMX 80 and MMX 86 implementations.

PL/M-86 LANGUAGE INTERFACE

The MMX 88 procedures described in this chapter are defined in PL/M-86. See the section of Chapter 7 entitled "Linking and Locating iMMX 800 Application Systems" for the names of files containing EXTERNAL declarations of the procedures.

A NOTATIONAL CONVENTION FOR MMX 88 DISCUSSIONS

Because two addressing modes -- megabyte and non-megabyte -- are available to iRMX 88 Executive users, and these modes affect MMX 88 differently, this manual uses the convention stated as follows at the beginning of the iRMX 88 REFERENCE MANUAL:

> The addressing mode for a module is determined conditionally when you compile the module. You can inform the compiler of your intentions by inserting two statements at the beginning of your source module. First, insert either
>
> ```
> $SET megabyte
> ```
>
> or
>
> ```
> $RESET megabyte
> ```
>
> depending upon whether you want megabyte or non-megabyte addressing, respectively. Then insert
>
> ```
> $IF megabyte
>     DECLARE LOCATION$OF LITERALLY '@';
>     DECLARE LOCATION LITERALLY 'POINTER';
> $ELSE
>     DECLARE LOCATION$OF LITERALLY '.';
>     DECLARE LOCATION LITERALLY 'ADDRESS';
> $ENDIF
> ```

The last of these instructions can be found in the INCLUDE file named LOCATE.LIT, which is on your Nucleus diskette. Place the directive $INCLUDE(:Fn:LOCATE.LIT) at the beginning of each task's module, where n is the number of the disk drive with the LOCATE.LIT file.

This combination of instructions to the compiler enables you to use the designations "location$of" and "location" in your code to achieve the intended mode of addressing. For example, assuming that the appropriate INCLUDE files have been specified and the structure EXCHANGE has been declared, the following code will correctly invoke the RQCXCH procedure.

```
CALL RQCXCH( LOCATION$OF EXCHANGE )
```

This procedure requires an argument that is either an address or a pointer, depending upon whether non-megabyte or megabyte addressing is being used, respectively.

Throughout this chapter, the generic term LOCATION is used in place of ADDRESS and POINTER, except in those few instances where ADDRESS or POINTER applies independently of the addressing mode.

In later chapters, this manual occasionally deviates from this convention, because some discussions of MMX 88 are combined with discussions of MMX 80, to which the convention does not apply. Such deviations should not cause you any misunderstanding.

iRMX 88 MESSAGE STRUCTURE

The iRMX 88 message structure has the following fields in the following order:

LINK	LOCATION
LENGTH	WORD
TYPE	BYTE
HOME$EX	LOCATION
RESP$EX	LOCATION
MSG$AREA(*)	BYTE

CONDITION CODES

After each call to an MMX 88 procedure, MMX 88 returns to the calling task a status value called a condition code. The condition code reflects the success or failure of the call. In case of failure, the code indicates the reason for the failure. Consequently, tasks should always check the condition code immediately after issuing an MMX 88 call.

MMX 88 PROCEDURE SUMMARY

Table 4-1 provides a summary description of the MMX 88 procedures for fast reference.

Table 4-1. MMX 88 Procedures Summary

Procedure	Parameters	Description
FIND PORT CQFIND	Input Values: sys$port$name condition$ptr Returned Value: connection	Furnishes a connection for sending messages to the system port represented by the specified systemport name.
TRANSFER MESSAGE CQXFER	Input Values: connection message$ptr xfer$flag xfer$length condition$ptr	Delivers the iRMX 88 message to the system port associated with the connection.
LOSE PORT CQLOSE	Input Value: connection	Releases the memory and connection previously acquired through a call to CQFIND. The task can no longer use the connection for message transfers to the system port.
ACTIVATE PORT CQACTV	Input Values: sys$port$name condition$ptr Returned Value: exchange$ptr	Activates a local iRMX 88 exchange that serves as the system port represented by the specified system-port name. Messages transferred to the system port are delivered to this exchange by the MMX 88 software.
WAIT FOR MESSAGE RQWAIT	Input Values: exchange$ptr time$limit Returned Value: message$ptr	Standard iRMX 88 operation that tasks use to receive messages at exchanges representing system ports. If desired, tasks can specify a waiting period.
ACCEPT MESSAGE RQACPT	Input Value: exchange$ptr Returned Value: message$ptr	Standard iRMX 88 operation that tasks use to receive messages at exchanges representing system ports. Tasks cannot specify a waiting period.
DEACTIVATE PORT CQDACT	Input Values: sys$port$name condition$ptr	Deactivates a system port that had been activated earlier by a call to CQACTV. Messages from another device can no longer be transferred to that system port. Messages still queued there can still be received by local tasks.

FIND PORT

The CQFIND procedure returns a connection for a system port. The calling task can use the connection to transfer messages to tasks on another (or the same) device.

```
connection = CQFIND (sys$port$name, condition$ptr);
```

sys$port$name	A WORD containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The LOCATION of a BYTE where MMX 88 returns the condition code for the call.
connection	The LOCATION of a WORD where a connection is returned. The task uses the connection when invoking CQXFER to transfer messages to the specified system port. No other task should use this connection.

DESCRIPTION

When configuring MMX 88 for this device, you specify the name and location of every system port to which tasks on this device transfer messages. CQFIND returns to the calling task a connection that identifies the system port whose name is specified in the call. The task can use the connection in calling CQXFER. If and when the task is finished making CQXFER calls with the connection, the task can call CQLOSE to return the connection to the system.

CQFIND allocates a block of memory (32 bytes in the non-megabyte version; 48 bytes in the megabyte version) from the Free Space Pool for internal needs, and also creates an exchange for MMX 88 use. The resources allocated to the calling task by means of the CQFIND procedure are returned to the system if and when the task calls the CQLOSE procedure.

The connection returned by CQFIND should be used by the task to which it is issued. If more than one iRMX 88 task on the same device needs to send messages to the same system port, each task should invoke the CQFIND procedure to obtain its own connection.

CONDITION CODES

SYSTEM$SERVICE$READY	CQFIND executed without error.
INSUFFICIENT$MEMORY	There is insufficient memory in the Free Space Pool to meet the requirements of CQFIND.
UNKNOWN$SYSTEM$PORT	The iMMX 800 software did not recognize the system-port name that the calling task supplied.

TRANSFER MESSAGE

The CQXFER procedure transfers an iRMX 88 message to the system port associated with the specified connection.

```
CALL CQXFER (connection, message$ptr, xfer$flag, xfer$length,
             condition$ptr);
```

connection — The LOCATION of the system port to which the specified message is to be transferred.

message$ptr — The LOCATION of an iRMX 88 message that is to be sent to the specified port.

xfer$flag — A BYTE that specifies the transmission mode for the message transfer. The possible values and their mnemonics are as follows:

Numeric Code	Mnemonic
0H (000B)	nloc$full$delivery
1H (001B)	nloc$full$transfer
2H (010B)	nloc$partial$delivery
3H (011B)	nloc$partial$transfer
4H (100B)	ploc$full$delivery
5H (101B)	ploc$full$transfer
6H (110B)	ploc$partial$delivery
7H (111B)	ploc$partial$transfer

The general meanings of the mnemonics follow. Much more detailed explanations of nloc, ploc, delivery, and transfer are in the description portion of this section.

- nloc and ploc - determine whether or not MMX 88 is to obtain a buffer (in an area that is accessible by the destination device) and place a copy of the message in the buffer. If MMX 88 is not directed to do this, the calling task must already have done so by the time it issues the CQXFER call, and message$ptr must point to the copy. "nloc" means MMX 88 should make a copy, and "ploc" means the task has already made a copy.

- full and partial - determine whether the entire message block or only the message portion (without the header) is to be transmitted. "full" means the entire block, and "partial" means only the first n bytes of the message portion, where n is the value specified in the message header or the value of xfer$length, whichever is smaller.

xfer$flag (continued)

- transfer and delivery - determine whether the calling task plans to reuse the memory, perhaps to broadcast the message to several devices. "transfer" means the task plans to reuse the memory, and "delivery" means the task does not intend to use the memory again.

xfer$length — A WORD whose value specifies the length, in bytes, of the message to be delivered by MMX 88. If bit 1 of the xfer$flag is zero (meaning the entire message block is sent), the xfer$length parameter is ignored by CQXFER. (See the CAUTION in the following DESCRIPTION section.) Otherwise, the length of the message to be sent is equal to xfer$lenth or to the length of the entire message block, whichever is smaller.

condition$ptr — A LOCATION of a BYTE where MMX 88 returns the condition code for the call.

DESCRIPTION

The TRANSFER MESSAGE service transfers a message to the system port specified by the connection. The task with the connection must invoke a separate call to CQXFER for each message sent to the system port.

An application task invoking the CQXFER procedure is suspended until the message is delivered and queued at the destination port, or until the MMX 88 software detects an error while attempting to deliver the message.

The xfer$flag parameter specifies how the calling task wants the message transmission to be handled. A table describing the full effects of the options (except for "partial" and "full"), with some preliminary notes, is as follows:

- "Message area" is the area defined by message$ptr and xfer$length.
- "Peer device" and "slave device" refer to device characteristics that you define for each device during iMMX 800 configuration for that device.
- Names of returned condition codes assume that no other errors occurred in the call.
- "A copy" is shorthand for "a copy of the message." Where the copy is made (locally or remotely) is either stated or is clear from context.
- "When control returns" is shorthand for "when control returns to the calling task."

Destination Device		Transfer	Deliver
Peer device with the ability to make copies	nloc	MMX 88 makes a copy in memory accessible by the destination device and returns the system$message$copy$delivered condition code. The message area is free for reuse.	MMX 88 makes a copy in memory accessible by the destination device and returns the system$message$copy$delivered condition code. Because MMX returns the message area to the PMM, the message area is not free for reuse.
	ploc	MMX 88 doesn't make a copy and returns the system$message$copy$delivered condition code. When control returns, a copy has been queued at the appropriate exchange or mailbox on the desination device, and the message area is free for reuse.	MMX 88 doesn't make a copy and returns the system$message$copy$delivered condition code. When control returns, the message area is not free for reuse.
Slave device	nloc	This is an error condition because MMX 88 does not make copies when transmitting messages to slave devices. MMX 88 returns the xfer$flag$error condition code.	This is an error condition because MMX 88 does not make copies when transmitting messages to slave devices. MMX 88 returns the xfer$flag$error condition code.
	ploc	This is an error condition because a task that elects to transfer a message expects to be able to use the message area immediately upon regaining control. MMX 88 returns the xfer$flag$error condition code.	MMX doesn't make a copy and returns the system$message$delivered condition code. When control returns, the message area is not free for reuse.
Peer device without the ability to make copies	nloc	MMX 88 makes a copy in memory accessible by the destination device and returns the system$message$delivered condition code. When control returns, the message area is free for reuse.	MMX 88 makes a copy in memory accessible by the destination device, returns the message area to the PMM, and returns the system$message$delivered condition code. When control returns, the message area is not free for reuse.
	ploc	MMX 88 doesn't make a copy and returns the system$message$delivered condition code. When control returns, a copy has been queued at the appropriate exchange or mailbox on the destination device, but the message area is not free for reuse.	MMX 88 doesn't make a copy and returns the system$message$delivered condition code. When control returns, the message area is not free for reuse.

In the event that a task chooses to let MMX 88 return the message block to PMM (that is, if it selects the "delivery" option in the xfer$flag parameter), the task must not alter the HOME$EX field of the message.

The RESP$EX field of the message is undefined for use with messages passed by means of the CQXFER procedure. That is, the sending task cannot use this field to signify to the receiving task where to return a response.

All other fields within the message are as defined by the iRMX 88 Executive. See Chapter 6 for a more detailed description of the message fields.

If it is necessary for communicating tasks to pass additional information concerning a message block, then some convention can be adopted that utilizes a "subheader" within the message itself for conveying such information. This subheader is considered part of the message's data and will be transferred by CQXFER.

CAUTION

When you use the full delivery mode to CQXFER a message from an iRMX 88-based system to an iRMX 80- or iRMX 86-based system, the MMX 80 or MMX 86 facility at the receiving end increases the size of the message in order to meet local iRMX 80 or iRMX 86 requirements. Consequently, if you use the iMMX 800 software to shuttle information back and forth between such systems many times, as in a "do forever" loop, and the task at each end always "sends" the same buffer that it just "received", then the buffers -- there are at least two, because the iMMX 800 software always make a copy on the destination device -- will grow beyond the limits of your system's memory. To prevent this from happening, one or more of the tasks should take responsibility for controlling the size of the buffers. A task using MMX 88 can exercise this control by using the partial$delivery mode, with the xfer$length field set to the appropriate value.

CONDITION CODES

SYSTEM$MESSAGE$COPY$DELIVERED	The destination device copied the message before it was successfully delivered. The message$ptr may or may not be valid.
SYSTEM$MESSAGE$DELIVERED	The message was successfully delivered to the destination system port without being copied by the destination device. The message$ptr may or may not be valid.
INSUFFICIENT$MEMORY	Not enough memory was available for local or destination buffers. The message$ptr remains valid.
SYSTEM$PORT$DEAD	MMX 80 has concluded that the indicated destination device is dead and therefore cannot receive transferred messages. The message$ptr remains valid.
SYSTEM$PORT$INACTIVE	The destination port currently is not active, so the message is not deliverable. The message$ptr remains valid.
UNKNOWN$SYSTEM$PORT	The specified connection is not valid, so the CQXFER call was not successful. The message$ptr remains valid.
XFER$FLAG$ERROR	The value of xfer$flag was not in the range 0 through 7, or an incorrect value was specified for the destination device.

LOSE PORT

The CQLOSE procedure allows a task to release resources that were previously allocated by the CQFIND procedure. After the CQLOSE call, the connection can no longer be used to transfer messages.

```
CALL CQLOSE (connection);
```

connection	The LOCATION of a WORD containing a connection that was previously returned to the calling task by the CQFIND service, for the purpose of using CQXFER to transfer messages.

DESCRIPTION

When an iRMX 88 task no longer wishes to send messages to a specified system port, the LOSE PORT service lets the task return to the system the resources previously allocated for message transfers. The calling task surrenders the following resources when it invokes the CQLOSE call:

- Connection - the calling task can no longer use the connection to transfer messages to the system port.
- Free Space memory - the memory block (32 bytes in the non-megabyte version; 48 bytes in the megabyte version) previously allocated for system use is returned to the Free Space Pool.
- An exchange - the exchange previously created by CQFIND is deleted.

CAUTION

The MMX 88 software does no validity checking when the CQLOSE procedure is called. Consequently, specifying an improper connection, or one that was invalidated by a previous CQLOSE call, causes unpredictable results in the device-resident iRMX 88 Executive.

ACTIVATE PORT

The CQACTV procedure activates the specified system port and creates a device-resident iRMX 88 exchange for message reception at the specified system port.

```
exchange$ptr = CQACTV (sys$port$name, condition$ptr);
```

sys$port$name	A WORD containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The LOCATION of a BYTE where MMX 88 returns the condition code for the call.
exchange$ptr	The LOCATION of the iRMX 88 exchange that MMX 88 creates. Local tasks, including the calling task, use exchange$ptr in calls to RQWAIT and RQACPT in order to receive messages from a task on another device.

DESCRIPTION

The MMX 88 services do not deliver messages to a system port until that port has been activated by a call to CQACTV. When called, the CQACTV procedure attempts to associate the specified system-port name with a device-resident system port. If the system port is defined for this device and the port is not already activated, CQACTV activates the port and returns an exchange$ptr for the associated iRMX 88-exchange address.

If other device-resident tasks are to receive messages at this iRMX 88 exchange, the task calling CQACTV must pass the iRMX 88 exchange address to those other tasks.

An activated system port remains active (that is, able to receive messages) until it is deactivated by a call to the CQDACT procedure.

Although an application task can invoke the iRMX 88 system call RXCXCH, to dynamically create exchanges for communication between tasks residing on the same device, application tasks cannot call CQCXCH to create exchanges for interdevice communication. Only the system ports (which you define at iMMX 800 configuration time) can be used as exchanges for interdevice communication, and each must be activated by a call to CQACTV.

CONDITION CODES

SYSTEM$SERVICE$READY	Service completed without error.
SYSTEM$PORT$ACTIVE	The indicated port is already activated.
UNKNOWN$SYSTEM$PORT	MMX 88 did not find the specified system port name when it searched the local system port table.

MESSAGE RECEPTION

RQWAIT and RQACPT are standard iRMX 88 system calls that tasks use to receive messages at exchanges. In particular, tasks use RQWAIT and RQACPT to receive messages at exchanges representing activated system ports.

```
message$ptr = RQWAIT (exchange$ptr, time$limit);
                    or
message$ptr = RQACPT (exchange$ptr);
```

exchange$ptr	The LOCATION of an iRMX 88 exchange previously created by the CQACTV procedure.
time$limit	A WORD (used in calls to RQWAIT only) whose value is the length of time (in iRMX 88 system time units) that the calling task is willing to wait for a message to arrive.
message$ptr	Normally the LOCATION of the message at the front of the exchange's message queue. However, if the task called RQWAIT and then "timed out", message$ptr contains the address of a five-byte (non-megabyte version) or seven-byte (megabyte version) message of type TIMEOUTTYPE (=3).

DESCRIPTION

An application task receives messages sent to an iRMX 88 exchange by invoking the iRMX 88 system calls RQWAIT and RQACPT. The exchange is identified in the calls by exchange$ptr. If the exchange represents a system port, the exchange location was previously returned to an application task by the CQACTV procedure.

After a task calls RQWAIT or RQACPT, it must ascertain whether the call was successful. If it calls RQWAIT, the task receives the location of a message. The task must check the TYPE field of that message to learn whether the message is what the task was waiting for. If the value in that field is three (3), the message is from the iRMX 88 Executive and indicates that the specified time limit expired before a message from another task arrived at the exchange. Otherwise, the message is from another task.

If the task calls RQACPT, and the returned message$ptr value is zero (0), then no message was queued at the exchange. Otherwise, the value is the location of a message from another task.

NOTE

The resp$ex field is undefined in iRMX 88 messages delivered to a system port by MMX 88 services. This field should not be used by receiving tasks.

DEACTIVATE PORT

The CQDACT procedure deactivates the specified system port. Messages are no longer delivered to that port by the device-resident MMX 88 software.

```
CALL CQDACT (sys$port$name, condition$ptr);
```

sys$port$name	A WORD containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The LOCATION of a BYTE where MMX 88 returns the condition code for the call.

DESCRIPTION

The DEACTIVATE PORT service allows an application task to deactivate the specified system port. After the port is deactivated, messages can no longer be sent to that port until it is re-activated by the same or another device-resident task. A SYSTEM$PORT$INACTIVE exceptional condition is returned to tasks attempting to send further messages to the deactivated port.

CQDACT does not affect messages already queued at the iRMX 88 exchange representing the system port when the CQDACT request is made. Such messages remain available to tasks on the device. MMX 88 deletes the exchange when the last remaining message is received by a task.

CONDITION CODES

SYSTEM$SERVICE$READY	Service completed without error.
UNKNOWN$SYSTEM$PORT	The iMMX 800 software did not recognize the system port name supplied by the calling task.

MMX 88 USAGE EXAMPLES

The program examples in Figures 4-1 and 4-2 show typical usage of the MMX 88 interdevice message-transfer services. The program given in Figure 4-1 represents portions of a task that sends messages to a task on another device. The task that sends messages is called the MMX$producer$task. The task to which it sends messages is similarly portrayed in Figure 4-2 and is called the MMX$consumer$task.

In the examples, the data types of the variables can be derived from context.

```
MMX$producer$task:
   DO;
       DECLARE condition$code BYTE;
       DECLARE consumer$connection LOCATION;
       DECLARE (consumer$sys$port$name,
                producer$sys$port$name) WORD EXTERNAL;
            :
   consumer$connection = CQFIND (consumer$sys$port$name,
                                 LOCATION$OF condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   producer$exch = CQACTV (producer$sys$port$name,
                           LOCATION$OF condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   CALL generate (LOCATION$OF producer$message); /* generate a message */

   CALL CQXFER(consumer$connection,
               LOCATION$OF producer$message,
               nloc$partial$deliver,
               xfer$length,
               LOCATION$OF condition$code);

   IF NOT ((condition$code = SYSTEM$MESSAGE$COPY$DELIVERED) OR
           (condition$code = SYSTEM$MESSAGE$DELIVERED))
       THEN CALL problem$handler;

   consumer$reply$message$ptr = RQWAIT (LOCATION$OF producer$exch,
                                        some$delay);
            :
   CALL CQDACT(producer$sys$port$name,
               LOCATION$OF condition$code);
            :
END MMX$producer$task;
```

Figure 4-1. Sending Task Program Example

```
MMX$consumer$task:
   DO;
       DECLARE condition$code BYTE;
       DECLARE producer$connection LOCATION;
       DECLARE (consumer$sys$port$name,
                producer$sys$port$name) WORD EXTERNAL;
            :
   consumer$exch = CQACTV (consumer$sys$port$name,
                           LOCATION$OF condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   producer$message$ptr = RQWAIT (consumer$exch,
                                  some$delay);

   IF producer$message.type = TIME$OUT$TYPE
       THEN CALL problem$handler;

   producer$connection = CQFIND (producer$sys$port$name,
                                 LOCATION$OF condition$code);

   IF NOT (condition$code = SYSTEM$SERVICE$READY)
       THEN CALL problem$handler;

   CALL generate (LOCATION$OF reply$message);  /*generate a reply*/

   CALL CQXFER (producer$connection,
                LOCATION$OF reply$message,
                nloc$partial$delivery,
                xfer$length,
                LOCATION$OF condition$code);

   IF NOT ((condition$code = SYSTEM$MESSAGE$COPY$DELIVERED) OR
           (condition$code = SYSTEM$MESSAGE$DELIVERED))
       THEN CALL problem$handler;
            :
   CALL CQDACT (consumer$sys$port$name,
                LOCATION$OF condition$code);
            :
END MMX$consumer$task;
```

Figure 4-2. Receiving Task Program Example

CHAPTER 5. MMX 86 PROCEDURE CALLS

The procedure calls described in this chapter apply only to tasks running under the supervision of the iRMX 86 Operating System. Although the iMMX 800 software is a single product, in the MMX 86 implementation, the syntax requirements of procedure calls are different than the syntax requirements of corresponding calls in the MMX 80 and MMX 88 implementations.

For iRMX 86 tasks, having different iMMX 800 implementations on the various devices in an application has the following implications:

- Except for CQXFER calls, an iRMX 86 task calling an iMMX 800 procedure is serviced only by the MMX 86 software resident on its own device. (CQXFER calls require interaction between iMMX 800 implementations residing on the source and destination devices.)
- An iRMX 86 task that sends messages to other devices need not concern itself with which iMMX 800 implementation provides services at the receiving devices.
- An iRMX 86 task that receives messages from another device need not concern itself with the origin of those messages; the receiving task's message-reception calls are serviced by MMX 86 and iRMX 86 software residing on its own device.

PL/M-86 LANGUAGE INTERFACE

The MMX 86 procedures described in this chapter are defined in PL/M-86. See the section of Chapter 7 entitled "Linking and Locating iMMX 800 Application Systems" for the names of files containing EXTERNAL declarations of the procedures.

CONDITION CODES

After each call to an MMX 86 procedure, MMX 86 returns to the calling task a status value called a condition code. The condition code reflects the success or failure of the call. In case of failure, the code indicates the reason for the failure. Consequently, tasks should always check the condition code immediately after issuing an MMX 86 call.

MMX 86 PROCEDURE SUMMARY

Table 5-1 provides a summary description of the MMX 86 procedures for fast reference.

Table 5-1. MMX 86 Procedures Summary

Procedure	Parameters	Description
FIND PORT CQFIND	Input Values: sys$port$name condition$ptr Returned Value: connection	Furnishes a connection for sending messages to the system port represented by the specified system-port name.
TRANSFER MESSAGE CQXFER	Input Values: connection msg$token xfer$flag msg$length condition$ptr	Delivers the iRMX 86 message to the system port associated with the connection.
LOSE PORT CQLOSE	Input Value: connection	Releases the memory and connection previously acquired through a call to CQFIND. The task can no longer use the connection for message transfers to the system port.
ACTIVATE PORT CQACTV	Input Values: sys$port$name condition$ptr Returned Value: exchange$ptr	Activates a local iRMX 86 exchange that serves as the system port represented by the specified system-port name. Messages transferred to the system port are delivered to this exchange by the MMX 86 software.
RECEIVE MESSAGE RQ$-RECEIVE$-MESSAGE	Input Values: mailbox$token time$limit response$ptr condition$ptr Returned Value: msg$token	Standard iRMX 86 operation that tasks use to receive objects at mailboxes representing system ports. If desired, tasks can specify a waiting period.
DEACTIVATE PORT CQDACT	Input Values: sys$port$name condition$ptr	Deactivates a system port that had been activated earlier by a call to CQACTV. Messages from another device can no longer be transferred to that system port. Messages still queued there can still be received by local tasks.

FIND PORT

The CQFIND procedure returns a connection for a system port. The calling task can use the connection to transfer messages to tasks on another (or the same) device.

```
connection = CQFIND (sys$port$name, condition$ptr);
```

sys$port$name	A WORD containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	The POINTER to a WORD where MMX 86 returns the condition code for the call.
connection	A TOKEN whose value is returned for use only by the calling task. The task uses the connection when invoking CQXFER to transfer messages to the specified system port.

DESCRIPTION

When configuring MMX 86 for this device, you specify the name and address of every system port to which tasks on this device transfer messages. CQFIND returns to the calling task a connection that identifies the system port whose name is specified in the call. The task can use the connection in calling CQXFER. If and when the task is finished making CQXFER calls with the connection, the task can call CQLOSE to return the connection to the system.

CQFIND initiates the allocation of a 32-byte segment from the Free Space Pool for internal needs, and also creates a mailbox for MMX 86 use. The resources allocated to the calling task by means of the CQFIND procedure are returned to the system if and when the task calls the CQLOSE procedure.

The connection returned by CQFIND should be used by the task to which it is issued. If more than one iRMX 86 task on the same device needs to send messages to the same system port, each task should invoke the CQFIND procedure to obtain its own connection.

Each call to CQFIND increases the object count for the task's job by 2.

CONDITION CODES

E$OK	The CQFIND call was successful and the returned connection is valid.
E$LIMIT	The CQFIND call was unsuccessful because to complete the call would have exceeded the object limit for the calling task's job.
E$MEM	The CQFIND call was unsuccessful because there is insufficient free space in the job containing the calling task.
E$UNKNOWN$SYSTEM$PORT	The CQFIND call was unsuccessful because the iMMX 800 software did not recognize the system-port name that the calling task supplied.

TRANSFER MESSAGE

The CQXFER procedure transfers a message to the system port associated with the specified connection.

```
CALL CQXFER (connection, msg$token, xfer$flag, xfer$length,
             condition$ptr);
```

---

connection — A TOKEN whose value identifies the system port where the specified message is to be transferred.

msg$token — A TOKEN for the segment containing the message that is to be sent to the specified port.

xfer$flag — A WORD that specifies the transmission mode for the message transfer. The two low-order bits determine the mode, as follows:

- Bit 0 - Specifies whether the calling task expects to reuse the message segment. The value 1 means that the task does expect to reuse the segment, and 0 means that it does not. See the description section below for more detail on this.
- Bit 1 - Specifies the amount of data to be transmitted. 0 means transmit the entire segment, and 1 means transmit n bytes, where n is the size of the segment or the value of xfer$length, whichever is smaller.

xfer$length — A WORD whose value specifies the length, in bytes, of the message to be delivered by MMX 86. If bit 1 of the xfer$flag is zero (meaning the entire segment is sent), the xfer$length parameter is ignored by CQXFER. (See the CAUTION in the following DESCRIPTION section.) Otherwise, the length of the message to be sent is equal to xfer$length or to the length of the entire segment, whichever is smaller.

condition$ptr — A POINTER to a WORD where MMX 86 returns the condition code for the call.

DESCRIPTION

The TRANSFER MESSAGE service transfers a message to the system port identified by the connection. The task with the connection must issue a separate call to CQXFER for each message sent to the system port.

An application task invoking the CQXFER procedure is suspended until the message is delivered and queued at the destination port, or until an exceptional condition is detected during the execution of the call.

The xfer$flag parameter specifies the mode of the message transmission. The following table, which is preceded by some preliminary notes, describes the full significance of bit 0 of xfer$flag. In that table,

- "Peer device" and "Slave device" refer to device characteristics that are defined for each device during iMMX 800 configuration for the device.
- Names of returned condition codes assume that no other errors occurred in the call.
- "Message segment" is the segment whose token is msg$token.

Destination Device	Transfer	Deliver
Peer device	MMX 86 makes a copy of the message in memory accessible by the destination device and returns the E$OK condition code. When control returns to the calling task, the message segment is free for reuse.	MMX 86 makes a copy of the message in memory accessible by the destination device, deletes the memory segment, and returns the E$OK condition code. When control returns to the calling task, the message segment is not free for reuse.
Slave device	This is an error condition, because MMX 86 does not make copies when transmitting to slave devices. MMX 86 returns the E$CONTEXT exceptional condition to the calling task.	MMX 86 doesn't make a copy of the message, and returns the E$OK condition code to the calling task. When control returns to the calling task, the message segment is not free for reuse.

CAUTION

When you CQXFER a message from an iRMX 86-based system to an iRMX 80- or iRMX 88-based system, the MMX 80 or MMX 88 facility at the receiving end increases the size of the message in order to meet local iRMX 80 or iRMX 88 requirements. Consequently, if you use the iMMX 800 software to shuttle information back and forth between such systems many times, as in a "do forever" loop, and the task at each end always "sends" the same buffer that it just "received", then the buffers -- there are at least two, because the iMMX 800 software always make a copy on the destination device -- will grow beyond the limits of your system's memory. To prevent this from happening, one or more of the tasks should take responsibility for controlling the size of the buffers. A task using MMX 86 can exercise this control by setting the xfer$flag parameter to 2 and the xfer$length parameter to the appropriate value.

CONDITION CODES

E$OK	The CQXFER call was successful. If bit 0 of xfer$flag was 1, msg$token is still a valid token for the message segment; otherwise msg$token is not valid.
E$CONTEXT	The CQXFER call was unsuccessful and the message was not delivered. The call attempted to transfer a segment to a "slave"-type device. Msg$token remains valid.
E$DESTINATION$CHANNEL$MEM	The CQXFER call was not successful because there was insufficient memory space on the destination device to make a copy of the message. Msg$token remains valid.
E$EXIST	The CQXFER call was unsuccessful because either connection or msg$token is not a token for an existing object. Msg$token remains valid.

E$LIMIT	The CQXFER call was unsuccessful because completing the call would have exceeded the object limit for the calling task's job. Msg$token remains valid.
E$MEM	The CQXFER call was unsuccessful because there is not sufficient free space in the calling task's job to provide the work space that MMX 86 requires. Msg$token remains valid.
E$SOURCE$CHANNEL$MEM	The CQXFER call was unsuccessful because there is not sufficient free space in the shared memory space to make a local copy of the message. Msg$token remains valid.
E$SYSTEM$PORT$DEAD	The CQXFER call was unsuccessful because the destination device failed to respond to a signal within a time period that was specified during configuration and consequently was declared dead. Subsequent attempts to communicate with that device are blocked. Msg$token remains valid.
E$SYSTEM$PORT$INACTIVE	The CQXFER call was unsuccessful because the destination system port was not activated (via a CQACTV call) by some task on the destination device prior to the attempted message transfer. Msg$token remains valid.
E$TYPE	The specified connection is a valid token for an object that is not a segment.
E$UNDEFINED$POOL	The CQXFER call was unsuccessful because the pool specified for the destination device was incorrectly specified during configuration (of the DSDT table for source device.) Msg$token remains valid.
E$UNKNOWN$SYSTEM$PORT	The CQXFER call was unsuccessful because the specified connection does not refer to a valid system port on the destination device. Msg$token remains valid.

LOSE PORT

The CQLOSE procedure allows an iRMX 86 task to release resources that were previously allocated by the CQFIND procedure. After the CQLOSE call, the connection can no longer be used to transfer messages.

```
CALL CQLOSE (connection, condition$ptr);
```

connection	A TOKEN whose value was returned by CQFIND to the calling task, for the purpose of using CQXFER to transfer messages.
condition$ptr	A POINTER to a WORD where MMX 86 returns the condition code for the call.

DESCRIPTION

When an iRMX 86 task no longer wishes to send messages to a system port, the LOSE PORT service lets the task return to the system the resources previously allocated for message transfers. The calling task surrenders the following resources when it invokes the CQLOSE call:

- Connection - the calling task can no longer use the connection to transfer messages to the system port.
- Free space memory - the 32-byte segment previously allocated for system use is returned to the Free Space Pool.
- A mailbox - the mailbox previously created by CQFIND is deleted.

CONDITION CODES

E$OK	The call to CQLOSE was successful and the connection is valid.
E$EXIST	The call to CQLOSE was unsuccessful because the specified connection is not a token for an existing object.
E$TYPE	The call to CQLOSE was unsuccessful because the specified connection is a token for an object that is not a connection object.

ACTIVATE PORT

The CQACTV procedure activates the specified system port and creates a device-resident iRMX 86 mailbox for message reception at the specified system port.

```
mailbox$token = CQACTV (sys$port$name, condition$ptr);
```

sys$port$name	A WORD containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	A POINTER to a WORD where MMX 86 returns the condition code for the call.
mailbox$token	A TOKEN to which MMX 86 returns a token for an iRMX 86 mailbox. This mailbox is used by the calling task (and all other device-resident tasks that access the same mailbox for message reception) in subsequent RQ$RECEIVE$MESSAGE calls.

DESCRIPTION

The MMX 86 services do not deliver messages to a system port until that system port has been activated by a call to CQACTV. When called, the CQACTV procedure attempts to associate the specified system-port name with a device-resident system port. If the system port is defined for this device and the port is not already activated, CQACTV activates the port and returns a token for the associated iRMX 86 mailbox.

If other device-resident tasks are to receive messages at this mailbox, the task calling CQACTV must pass the token for the mailbox to those other tasks.

An activated system port remains active (that is, able to receive messages) until it is deactivated by a call to the CQDACT procedure.

Although an application task can invoke the iRMX 86 system call RQ$CREATE$MAILBOX, to dynamically create mailboxes for communication between tasks residing on the same device, application tasks cannot call RQ$CREATE$MAILBOX to create mailboxes for interdevice communication. Only the system ports (which you define at iMMX 800 configuration time) can be used as mailboxes for interdevice communication and each must be activated by a call to CQACTV.

CONDITION CODES

E$OK	The CQACTV call was successful and the returned mailbox$token is valid.
E$SYSTEM$PORT$ACTIVE	The CQACTV call was unsuccessful because the indicated port is already activated.
E$UNKNOWN$SYSTEM$PORT	The CQACTV call was unsuccessful because MMX did not recognize the specified system port name when it searched the local system port table.

MESSAGE RECEPTION

RQ$RECEIVE$MESSAGE is a standard iRMX 86 system call that tasks use to receive objects at mailboxes. In particular, tasks use RQ$RECEIVE$MESSAGE to receive messages at mailboxes representing activated system ports.

```
msg$token = RQ$RECEIVE$MESSAGE (mailbox$token, time$limit,
                                response$ptr, condition$ptr);
```

mailbox$token	A TOKEN containing a token for a mailbox previously created by the CQACTV procedure.
time$limit	A WORD which, • if zero, indicates the calling task is not willing to wait. • if 0FFFFH, indicates the task will wait as long as is necessary. • if between 0 and 0FFFFH, is the number of clock intervals the task is willing to wait. The length of the clock interval is configurable. Refer to the iRMX 86 CONFIGURATION GUIDE for further information.
response$ptr	A POINTER to a WORD in which the system always returns a value of zero, since response$ptr is not supported in MMX 86 implementations.
condition$ptr	A POINTER to a WORD where MMX 86 returns the condition code for the call.
msg$token	A TOKEN containing the token for the message segment being received.

DESCRIPTION

An application task receives messages sent to an iRMX 86 mailbox by invoking the RQ$RECEIVE$MESSAGE system call. The mailbox is identified in the call by mailbox$token. If the mailbox represents a system port, the mailbox token was previously returned to an application task by a call to the CQACTV procedure.

When used in conjunction with MMX 86 software, the RQ$RECEIVE$MESSAGE system call behaves as expected, except that the value returned to the WORD pointed to by the response$ptr is always 0. This is because MMX 86 does not know where the message came from.

CONDITION CODES

E$OK	The RQ$RECEIVE$MESSAGE call was successful and the message has been received.
E$EXIST	The RQ$RECEIVE$MESSAGE call was unsuccessful because either: • mailbox$token was not a token for an existing object or • the local iRMX 86 Operating System deleted the MMX 86 job on the device while the requesting task was waiting.
ENOTCONFIGURED	The RQ$RECEIVE$MESSAGE call was unsuccessful because RQ$RECEIVE$MESSAGE was excluded during iRMX 86 configuration.
E$TIME	The RQ$RECEIVE$MESSAGE call was unsuccessful because the calling task either: • specified a time$limit of 0 and no messages were queued at the mailbox or • specified a non-zero time$limit that was less than 0FFFFH and then "timed out".
E$TYPE	The RQ$RECEIVE$MESSAGE call was unsuccessful because mailbox$token is a token for an object that is not a mailbox.

DEACTIVATE PORT

The CQDACT procedure deactivates the specified system port. Messages are no longer delivered to that port by the device-resident MMX 86 software.

```
CALL CQDACT (sys$port$name, condition$ptr);
```

sys$port$name	A WORD containing the two-byte ASCII name of a system port. You assign names to system ports during iMMX 800 configuration.
condition$ptr	A POINTER to a WORD where MMX 86 returns the condition code for the call.

DESCRIPTION

The DEACTIVATE PORT service allows an application task to deactivate the specified system port. After the port is deactivated, messages can no longer be sent to that port until it is re-activated by the same or another device-resident task calling the CQACTV procedure. An exceptional condition is returned to tasks attempting to send further messages to the port.

CQDACT does not affect messages already enqueued at the iRMX 86 mailbox representing the system port when the CQDACT request is made. Such messages remain available to tasks on the device. MMX 86 deletes the mailbox when the last remaining message is received by a task.

CONDITION CODES

E$OK	The call to CQDACT was successful.
E$UNKNOWN$SYSTEM$PORT	The CQACTV call was unsuccessful because MMX 86 did not recognize the specified system port name when it searched the local system port table.

MMX 86 USAGE EXAMPLES

The program examples in Figures 5-1 and 5-2 show typical usage of the MMX 88 interdevice message-transfer services. The program given in Figure 5-1 represents portions of a task that sends messages to a task on another device. The task that sends messages is called the MMX$producer$task. The task to which it sends messages is similarly portrayed in Figure 5-2 and is called the MMX$consumer$task.

In the examples, the data types of the variables can be derived from context.

```
MMX$producer$task:
   DO;
       DECLARE condition$code WORD;
       DECLARE consumer$connection TOKEN;
       DECLARE (consumer$sys$port$name,
                producer$sys$port$name) WORD EXTERNAL;
           :
   consumer$connection = CQFIND (consumer$sys$port$name,
                                 @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;

   producer$mbox = CQACTV (producer$sys$port$name,
                           @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;

   CALL generate (@producer$message);

   CALL CQXFER(consumer$connection,
               producer$message,
               full$deliver,  /* transfer and delete whole segment */
               OFFFFH,
               @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;

   msg$token = RQ$RECEIVE$MESSAGE (producer$mbox,
                                   some$delay,
                                   @response,
                                   @condition$code);
           :
   CALL CQDACT(producer$sys$port$name,
               @condition$code);
           :
END MMX$producer$task;
```

Figure 5-1. Sending Task Program Example

```
MMX$consumer$task:
   DO;
       DECLARE condition$code WORD;
       DECLARE producer$connection TOKEN;
       DECLARE (consumer$sys$port$name,
                producer$sys$port$name) WORD EXTERNAL;
           :
   consumer$mbox = CQACTV (consumer$sys$port$name,
                           @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;

   msg$token = RQ$RECEIVE$MESSAGE(consumer$mbox,
                                  some$delay,
                                  @response
                                  @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;

   producer$connection = CQFIND (producer$sys$port$name,
                                 @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;

   CALL generate (@reply$message);

   CALL CQXFER (producer$connection,
                reply$message,
                full$deliver,  /* transfer and delete whole segment */
                OFFFFH,
                @condition$code);

   IF NOT (condition$code = E$OK)
       THEN CALL problem$handler;
           :
   CALL CQDACT (consumer$sys$port$name,
                @condition$code);
           :
END MMX$consumer$task;
```

Figure 5-2. Receiving Task Program Example

CHAPTER 6. PARTITIONED MEMORY MANAGER

The Partitioned Memory Manager (PMM) is provided to manage one or more contiguous blocks of RAM. The PMM allocates memory to tasks on request and accepts memory from tasks if and when the tasks no longer need the memory. In systems that use it, the PMM replaces the Free Space Manager (FSM).

NOTE

In iRMX 86-based applications, tasks obtain memory for local needs by calling RQ$CREATE$SEGMENT, and they return memory by calling RQ$DELETE$SEGMENT, so FSM-like features are not needed. Furthermore, MMX 86 manages all pools on behalf of tasks that communicate with other devices. Consequently, MMX 86 users do not explicitly use the PMM and need not read this chapter.

MEMORY POOLS

The contiguous blocks of memory that the PMM manages are called pools. You define the pools for each MMX80- or MMX88-based device during the configuration process for the device. You also assign numbers called pool id's to the pools during configuration. The pool id's for each device usually start with 0 and continue upward sequentially, such as 0 through 10. For convenience, we refer to the pools by their id numbers 0 through N. There can be up to 255 pools per device, and pool id's can range from 0 to 254.

There can be many reasons for separating memory into pools. What the reasons are and how you do the separating depend upon the requirements of your application. How your tasks use the PMM also depends upon your application and its requirements. The iMMX 800 software, like your application tasks, is a PMM user; the primary use it has for the PMM is to transfer messages between devices.

Memory pools are important when transferring messages between devices. Suppose there is a channel for communication between task A on device A and task B on device B. To support message transfers from task A to task B, there must be a memory pool (pool A), managed by the PMM on device A, that is accessible by both devices. Messages from task A to task B must be put into pool A. Similarly, to support message transfers from task B to task A, there must be a pool (pool B), managed by the PMM on device B, and it too must be accessible by both devices. And, similarly, messages from task B to task A must be put into pool B.

When task A sends a message to task B, task A has two choices as to how the message is put into pool A. Task A can itself put the message into pool A and then call CQXFER in such a way that the iMMX 800 software does not make a separate copy of the message. Or task A can request that CQXFER make a copy in pool A before transferring the message. Therefore, one of the benefits of having the PMM procedures available to application tasks is that the tasks can prevent the PMM from making the extra copy of messages and using the extra memory that doing so entails.

In iRMX 80- and iRMX 88-based applications, pool 0 on each device is the Free Space Pool for the device. Memory in a device's Free Space Pool is usually dedicated to on-board needs, but, if desired, it can also be used to transfer messages between the device and other devices, provided that all of the memory in the Free Space Pool is accessible by all of those devices.

In contrast, the memory in the other pools for the device is reserved for transferring messages between the device and other devices. Each channel between the device and another device uses one or more pools on the device for transferring messages through that channel, provided that all of the memory in each of the pools is accessible by the other device. The other device also uses one or more pools for transferring messages through the channel. If desired, the memory in a single pool on a device can be used for transferring messages through more than one channel.

USING THE FREE SPACE POOL

In iRMX 80- and iRMX 88-based applications, the PMM manages the Free Space Pool in the same way that the iRMX 80 and iRMX 88 Free Space Managers (FSM) manage their respective pools of memory. That is, tasks obtain memory by sending request messages to the RQFSAX exchange, and they return memory by sending it to the RQFSRX exchange. Consequently, on iRMX 80- and iRMX 88-based devices, the PMM replaces the FSM. Reference material describing the FSM's can be found in the iRMX 80 USER'S GUIDE and the iRMX 88 REFERENCE MANUAL. If you are planning to use only FSM functions, you can read about them in the appropriate manual, instead of reading the remainder of this chapter.

PMM management of the Free Space Pool on a device differs from FSM memory management in the following ways:

- A PMM memory block must always begin on a paragraph boundary (that is, its starting address must be a multiple of 16), and its length, in bytes, is always a multiple of 16. If a task sends a memory request to the RQFSAX exchange with the LENGTH field set to a value that is not a multiple of 16, the PMM will round that value upward to the nearest multiple of 16 before acting on the request. Therefore, a task that always requests memory in multiples of 16 bytes can see no difference between the allocation algorithms of the PMM and the FSM.

- The memory for the Free Space Pool is normally defined during configuration. When the application system begins to run, the PMM automatically initializes the Free Space Pool. In contrast, the FSM does not establish its own pool of memory, but requires that some task give it memory that has been reserved by some method, such as by being declared an array.

  Tasks in the PMM environment must not send memory to the RQFSRX exchange when that memory had not been obtained through the RQFSAX exchange. This is because there is no way to guarantee that such memory begins on a paragraph boundary. The rounding process that takes place can cause up to 15 bytes to be chopped off each end of such a memory block. For the same reason, tasks normally should not return part of a memory block to the RQFSRX exchange, even when the memory was obtained through the RQFSAX exchange. However, if a task makes certain that the part of a block being returned starts on a paragraph boundary and has a length that is a multiple of 16, there should not be any problems.

NOTE

Memory sent to the RQFSRX exchange in the MMX 88 environment will not be reclaimed unless it starts on a paragraph boundary and has a length that is a multiple of 16. Instead, the memory is "lost."

In the MMX 88 environment, memory must not be sent to RQFSRX exchange unless it had previously been allocated from the RQFSAX exchange.

USING POOLS 0 THROUGH N

Tasks needing memory for interdevice message transfers obtain memory through the RQFLMX exchange. When the memory is no longer needed, it is usually returned to the appropriate pool, either by an application task or by MMX 80 or MMX 88.

REQUESTING MEMORY

The RQFLMX exchange has slightly different format requirements than do the RQFSAX and RQFSRX exchanges. The message structures associated with requesting memory from the RQFLMX exchange are as follows:

```
DECLARE MSG$HEADER LITERALLY
     'LINK          ADDRESS,    /* LOCATION if MMX 88 */
     LENGTH         WORD,
     TYPE           BYTE,
     HOME$EX        ADDRESS,    /* LOCATION if MMX 88 */
     RESP$EX        ADDRESS';   /* LOCATION if MMX 88 */


DECLARE PMM$REQ$STRUC LITERALLY
   '(MSG$HEADER,
     NEEDED$SIZE    WORD,
   /* FILLER         WORD,    if megabyte MMX 88 */
     MEMORY$POOL    BYTE)';


DECLARE PMM$GOT$BLK$STRUC LITERALLY
   '(MSG$HEADER,
     BLK$PTR        ADDRESS,    /* LOCATION if MMX 88 */
     MEMORY$POOL    BYTE)';


DECLARE PMM$NOT$ALLOC$STRUC LITERALLY
   '(MSG$HEADER,
     BIGGEST$BLK    WORD)';
```

Messages of the form PMMREQSTRUC are sent to the RQFLMX exchange in order to request memory. The fields that the requesting task must fill in are the following:

TYPE must be either PMMGETBLK$TYPE (=4H) or PMM$UCGETBLK$TYPE (=5H). PMM$GETBLKTYPE signifies a conditional request, which means that the task wants the requested memory but is not willing to wait if a sufficiently large block is not available. PMMUCGETBLKTYPE, on the other hand, signifies that the task must have the memory and will wait indefinitely for it.

RESP$EX must contain the address of the exchange where the requesting task will wait for a response to its request.

NEEDED$SIZE must contain the number of bytes being requested. This value must be large enough to accommodate both the body of the message and the message header (12 bytes in MMX 80 and the non-megabyte version of MMX 88; 20 bytes in the megabyte version of MMX 88) that the PMM places at the beginning of the allocated memory.

MEMORY$POOL must contain the pool id of the memory pool that is dedicated to the channel that the requesting task is planning to use. If the task doesn't know the number of the pool, it can obtain that value by means of the CQGDPA procedure, as follows:

```
MEMORY$POOL = CQGDPA(CONNECTION);
```

where CONNECTION is the connection previously obtained for the channel by a call to CQFIND.

After the requesting task sends the request message, it must wait at the response exchange indicated in the request. When the task receives the response message, the TYPE field reveals the disposition of the request, as follows:

- If the value in the TYPE field is PMM$ERROR$TYPE (=6H), the response message is of the PMMNOTALLOC$STRUC type and the MEMORY$POOL field of the request message did not contain a valid pool id value, so the request is denied.
- If the value in the TYPE field is PMMNOSPACE$TYPE (=2BH), there was not sufficient memory at the time of the request, so the request is denied. In this case, the response message is of the PMMNOTALLOC$STRUC type and the BIGGEST$BLK field contains the number of bytes in the largest block that could have been allocated. Note that there is no guarantee that a block of that size still remains in the pool.
- Otherwise, the value in the TYPE field is the same as the value in the TYPE field of the request message, and the request is granted. In this case, the response message is of the PMMGOTBLK$STRUC type, the BLK$PTR field contains the address of the allocated memory block, and the LENGTH and HOME$EX fields should not be altered.

RETURNING ALLOCATED MEMORY

If and when a block of memory is no longer needed, a task can return the memory to the PMM by sending the memory to the exchange whose address is in the HOME$EX field of the memory block's message header.

If, for some reason, the task wants to return the memory to a different pool than the pool from which the memory was allocated, the task can easily do so, although this practice is not recommended in the MMX 80 environment and is absolutely forbidden in the MMX 88 environment. The task must first put the appropriate pool id in the MEMORY$POOL field of the memory block and the PMM$FREE$BLK$TYPE (=28H) in the TYPE field. Then the task sends the memory block (as a message) to the RQFLMX exchange.

When a block is sent to the RQFLMX exchange, the RESP$EX field is ignored by the PMM, so if an error occurs, the task does not learn of it. In iRMX 80-based applications, if the error is that a non-existent pool was specified, the PMM has sent the memory to an exchange called RQPBHX. Application tasks can do an RQACPT operation at that exchange to see whether any blocks have been improperly reclaimed. In iRMX 88-based applications, memory that is sent to the RQFLMX exchange with an invalid pool id is "lost".

CREATING MEMORY POOLS DYNAMICALLY

For iRMX 80 applications, it is not necessary for all pools to be defined during configuration. A task can request that a pool be created dynamically by a process similar to that used for requesting memory from an existing pool.

First, the task prepares a message of the PMMREQSTRUC type. The TYPE field of the message must contain the value PMM$CREATE$POOL$TYPE (=29H). The RESP$EX field must contain the address of the exchange where the task will wait for a response to its request. The MEMORY$POOL field must contain the pool id of the pool that is to be created.

After preparing the message, the task must send it to the PMM exchange RQPMX, and then the task must wait at the specified response exchange.

When the task receives the response message, the TYPE field reveals the disposition of the request, as follows:

- If the value in the TYPE field is PMM$ERROR$TYPE (=6H), there is already a pool with the specified pool id, so the request is denied.
- If the value in the TYPE field is PMMNOSPACE$TYPE (=2BH), either there was not sufficient memory available to form a pool or the required 32 bytes of work area was not available in the Free Space Pool, so the request is denied.
- Otherwise, the value in the TYPE field is the same as that in the request, namely PMM$CREATE$POOL$TYPE (=29H), and the request is granted.

Once the task has confirmed that the pool has been created, it must give memory to the new pool. It prepares to do so by obtaining the memory from the Free Space Pool. Then the task sends the memory to the RQFLMX exchange with the TYPE field set to PMM$FREE$BLK$TYPE (=28H) and the MEMORY$POOL field set to the pool id of the new pool.

CHAPTER 7. CONFIGURING YOUR APPLICATION SYSTEM

After you have done high-level design of your hardware and written and compiled your tasks, it is time for you to configure your system. Configuring your hardware consists of making the final adjustments, such as jumpering, that prepare your hardware for use with both the iRMX operating system(s) and the iMMX 800 software. Configuring your software involves describing the resources, including the hardware and memory partitions, that the iMMX 800 services have at their disposal. These descriptions take the form of files of PL/M declarations, and most of this chapter is concerned with the process of declaring the appropriate variables and data structures and assigning values to them. The remainder of the chapter describes how to compile the iMMX 800 configuration file, how to link it to the iMMX 800 software and to the iRMX tasks, and how to configure your hardware for the use of the iMMX 800 services.

SOFTWARE CONFIGURATION

The hard part of software configuration consists of making a number of decisions concerning the requirements of your application system. The easy part consists of translating these decisions into the variables and data structures that support the iMMX 800 internal control structures.

DECISIONS THAT PROVIDE INFORMATION NEEDED FOR CONFIGURATION

The decisions that you must make fall into three categories, depending upon their scope: system-level decisions, device-level decisions, and port-level decisions. So that you can use the following lists of decisions as convieniently as possible, the name(s) of the variables and data structures that are affected by each decision are listed immediately after the description of the issue requiring the decision.

System-Level Decisions

(1) What types of devices make up the system, and how many are there of each type? (CQDVCS)

(2) What pairs of devices require interdevice communication? (CQPRTS, LPT$ROM, CQSKTS, DSDT)

(3) What are the ID's for the devices and interdevice segments (IDS's) that are involved in iMMX 800-supported interdevice communication? An ID is a non-negative integer that identifies the device or IDS. More important, the iMMX 800 software uses the ID's as indexes for arrays of data structures pertaining to devices or IDS's. Both sets of ID's must begin with zero and must increase sequentially. The ID's in each set can be assigned in any order. (DSDT.DESTDEVID and DSDT.SRCDEVID)

(4) What are the addresses of the request queue descriptiors and how many entries can each queue accommodate? Each request queue descriptor is eight bytes long and is followed immediately by the memory that is reserved for the queued entries. Each queued entry occupies 16 bytes. If you decide to place all of your request queues consecutively in the same area of memory, you might want to skip the next eight bytes after each queue so that each request queue descriptor can start on a paragraph boundary. (DCM$ROM.RQD$OUT$PTR and DCM$ROM.RQDINPTR)

(5) What do you want to call the ports in the system? Each port must have a two-character name that uniquely represents it throughout the whole system. (LPT$ROM)

(6) How many interdevice segments are there in the system, and where are they? (CQIDSS, IDST)

Device-Level Decisions

(7) For how long a time period will the iMMX 800 software on the device wait before beginning to communicate? (CQITWT)

(8) For how long a time period will the device wait for a response from other devices? (CQMDLY)

(9) For each device, which of the following schemes is used to alert the device to the occurrence of an external event: Multibus interrupt, I/O-mapped interrupt, memory-mapped interrupt, or polling? A Multibus interrupt travels to its destination along the Multibus interface and has the disadvantage that it can interrupt every device in the system. An I/O-mapped interrupt arrives at a device through an I/O port. A memory-mapped interrupt arrives at a device in the device's memory. (SFT.INTR$TYPE)

(10) For devices that are interrupted through the Multibus interface:

(a) To which bit (0-7) of port C of an 8255 Programmable Peripheral Interface should a value be written to generate an interrupt? (SFT.INTR$VALUE)

(b) What I/O control port generates a Multibus interrupt for the device? (SFT.INTR$LOCATION)

(11) For devices that receive I/O-mapped interrupts:

(a) Which I/O port receives interrupts? (SFT.INTR$LOCATION)

(b) What value is sent to the port to generate an interrupt? (SFT.INTR$VALUE)

(12) For devices that receive memory-mapped interrupts:

(a) What memory location receives the interrupt? (SFT.INTR$LOCATION)

(b) What value will be written to that memory location to generate an interrupt? (SFT.INTR$VALUE)

(13) For each (MMX 86) device, what method does the device use to clear interrupts that it has generated? (SFT.CLROUTTYPE)

(14) For each (MMX 86) device that is responsible for clearing interrupts that it has generated:

(a) Which I/O port or memory location is associated with interrupt clearance? (SFT.CLROUTINTR$LOCATION)

(b) What value is sent to the I/O port or memory location to clear the interrupt? (SFT.CLROUTINTR$VALUE)

(15) For devices that are responsible for clearing interrupts that they have received:

(a) What method does the device use to clear interrupts that it has received? (SFT.CLR$INTR$TYPE for MMX 88; SFT.CLRINTYPE for MMX 86)

(b) Which I/O port or memory location is associated with interrupt clearance? (SFT.CLR$INTR$LOCATION for MMX 88; SFT.CLRININTR$LOCATION for MMX 86)

(c) What value is sent to the I/O port or memory location to clear the interrupt? (SFT.CLR$INTR$VALUE for MMX 88; SFT.CLRININTR$VALUE for MMX 86)

(16) What is the device's polling period? (CQIDPD)

(17) What is the interrupt level, if any, that the iMMX 800 software uses to interrupt the device? (CQSGLV)

(18) For iRMX 80- and iRMX 88-based devices, what is the address of the interrupt exchange for the interrupt level that the iMMX 800 software uses? (CQLMEX)

(19) What are the ID's for the device's ports and memory pools? As is the case for device and interdevice segment ID's, port and pool ID's are used as indexes into arrays of data structures pertaining to ports and memory pools. For each device in the system, its set of port and pool ID's each begin with zero and increase sequentially. (DSDT.DEST$PORT$ID, DSDT.POOLID, LPTROM, BLKTBL)

(20) What are the addresses of the interdevice segments as they would be addressed by the device? (IDST)

(21) What are the locations and sizes of memory pools on the device? (BLKTBL)

Port-Level Decision

(22) For each port on iRMX 80- and iRMX 88-based devices, what is the ID of the pool into which messages destined for that port are to be copied? (LPT$ROM)

VARIABLES AND DATA STRUCTURES THAT MUST BE ASSIGNED VALUES

After you have made all of the necessary decisions, it is time to place the appropriate values in the variables and data structures that describe the system for the iMMX 800 software. Figure 7-1 shows the most important of these data structures and some of the relationships among them. The placement of the structures in the figure reflects a top-down approach to configuration.

In the paragraphs that follow, the numbers of the decisions previously listed are referenced to help you see more clearly the relationship between the decision-making phase of configuration and the structure-filling phase. As you read these paragraphs, remember that we are discussing the configuration of software that will run on a particular system device. Similar configuration has to be done for every other device in the system.

Device Description (CQDVCS)

The number of devices in the entire system (decision 1) must be assigned to the variable CQDVCS, which is of the BYTE data type.

Channel Description (DCMROM, DCMRAM)

The DCM$ROM (Device-to-Channel Map) array of structures has an entry for each device in the system. The array is indexed by the device ID's. Each entry represents the channel (or lack of a channel -- decision 2) between the device for which this configuration is being done and another (or the same) device in the system. In the case of an entry that represents a channel, the entry contains the locations and sizes of the request queues (decision 4) associated with the channel. Each entry in this table has the following format:

0	RQDOUTPOINTER	
4	RQOUTSIZE	
5	RQEOUTSIZE	
6	RQDINPOINTER	
10	RQINSIZE	
11	RQEINSIZE	

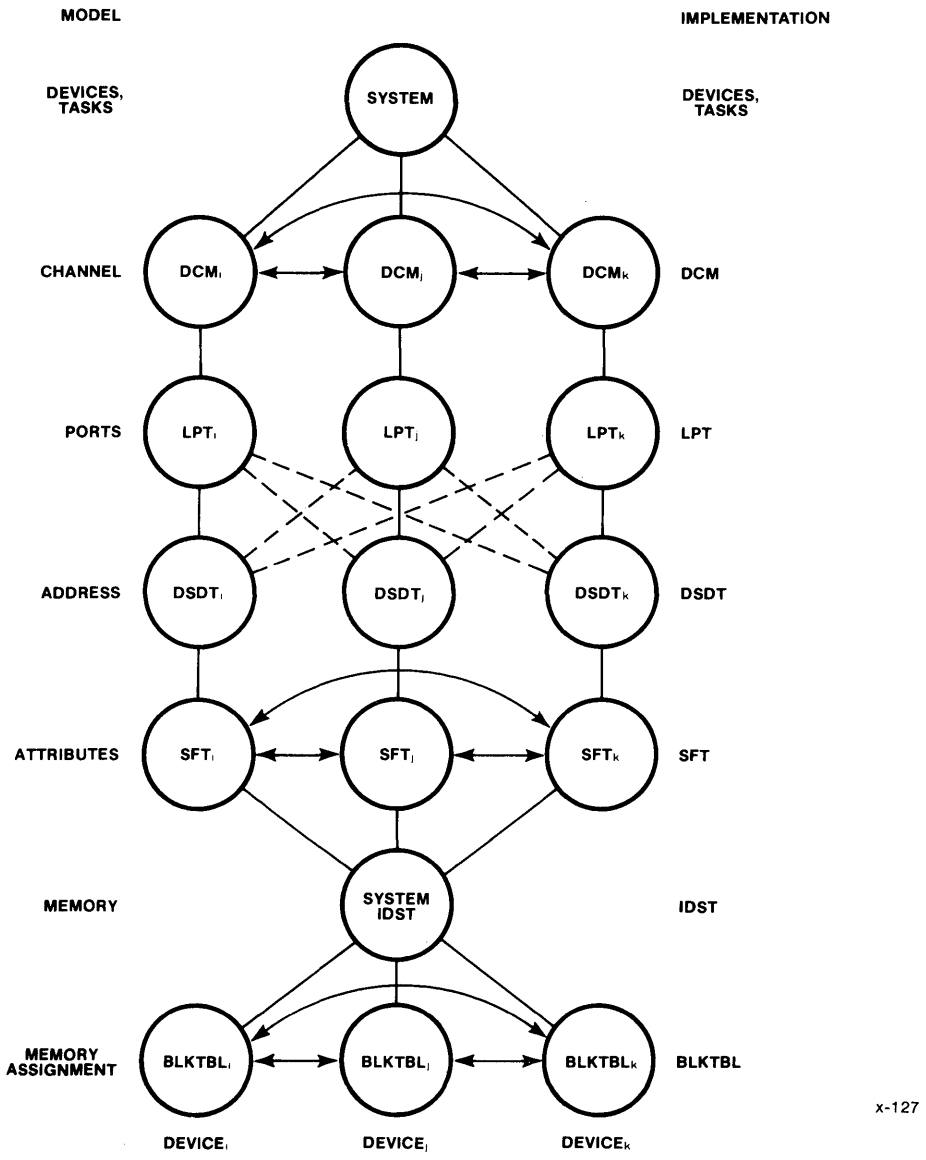


Figure 7-1. A Level-Oriented Representation of Configuration Structures

where:

RQDOUTPOINTER and RQDINPOINTER are POINTERs (ADDRESSes in MMX 80) to the outbound and inbound request queue descriptors, respectively. If the entry is for the device for which this configuration is being done, and this device will use an iMMX 800 channel to communicate with itself, these two pointers should contain the same address. This is because a channel between a device and itself requires only one queue. In case the entry represents the lack of a channel between the two devices, these fields must each contain the value OFFFFH.

RQOUTSIZE and RQINSIZE are BYTEs containing the maximum allowable numbers of entries in the outbound and inbound request queues, respectively. Each of these values must be a power of two. In addition, in systems that intermix Release 2.0 and Release 3.0 versions of the iMMX 800 software, RQOUTSIZE and RQINSIZE should each be greater than the total number of tasks (on both devices) that will be using the channel corresponding to this DCM$ROM array entry.

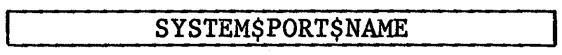
RQEOUTSIZE and RQEINSIZE are BYTEs containing the value 4. This signifies that the entries in each of the queues are 16 bytes long. The value 4 is used because it is the base 2 logarithm of 16.

When the iMMX 800 software goes through its initialization phase, it builds a number of data structures in RAM. One of these is DCM$RAM, an array of areas (20 bytes for each device in configurations of MMX 80, MMX 86, or the non-megabyte version of MMX 88; 24 bytes for each device in configurations of the megabyte version of MMX 88) for which you must provide space in storage. The number of areas in the array is the number of devices in the system.

Port Descriptions (CQPRTS, LPTROM, LPTRAM)

The variable CQPRTS, which is of the BTYE data type, contains the number of ports (decision 2) resident on the device for which this configuration is being performed.

The array LPT$ROM of structures has an entry for each port on the device being configured. Each entry contains the unique name (decision 5) of one of the ports on the device. The array is indexed by the port ID's (decision 19) for the device. In MMX 86 systems, each entry has the following format:



and in MMX 80 and MMX 88 systems, each entry has the following format:

where:

SYSTEM$PORT$NAME is a WORD containing the unique two-character name of the port that the entry represents.

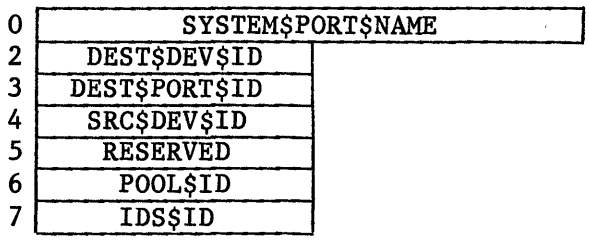
POOL$ID is a BYTE containing the ID of the pool (decision 22) into which messages destined for the port are copied.

Another of the data structures that the iMMX 800 software builds during its initialization phase is an array called LPT$RAM, and you must provide room for it in storage. In MMX 86 applications, each entry in the array consists of a BYTE followed by a WORD, in MMX 80 and MMX 88 (non-megabyte version) applications each entry consists of 11 BYTEs, and in MMX 88 (megabyte version), each entry consists of 21 BYTEs. As in the case of LPT$ROM, the array is indexed by the port ID's for the device.

Address Description (CQSKTS, DSDT)

The variable CQSKTS, which is of type BYTE, contains the number of system ports (decision 2) to which the device being configured sends messages.

The DSDT (Destination System Port Descriptor Table) array of structures has an entry for each port to which the device being configured sends messages. Each entry in this table associates a system port with the device where it resides. Each entry has the following format:



where:

SYSTEM$PORT$NAME is a WORD containing the unique, two-character name of the port (decision 5) to which this DSDT array entry corresponds.

SRCDEVID and DESTDEVID are BYTEs containing the device ID's (decision 3) of the source and destination devices, respectively. The source device is the device for which this configuration is being done.

DEST$PORT$ID is a BYTE containing the port ID (decision 19) of the destination port.

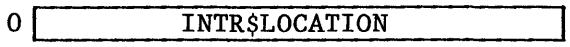
POOL$ID is a BYTE containing the ID of the pool (decision 19) into which the iMMX 800 software will copy messages in preparation for transferring messages between the specified source and destination devices.

IDS$ID is a BYTE containing the ID of the interdevice segment (decision 3) that contains the pool specified by POOL$ID.

Attribute Description (SFT, CQITWT, CQMDLY, CQIDPD, CQSGLV, CQLMEX, MCBI)

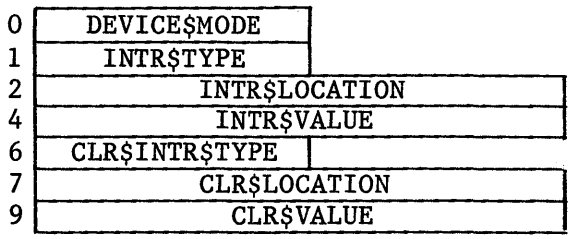
The array SFT of structures has an entry for each device in the system and is indexed by device ID's. Each entry describes the physical characteristics of a device and how to generate an interrupt to that device. The form of the structure depends upon whether the device is being configured for MMX 80, MMX 88, or MMX 86.

If the device is being configured for MMX 80, each SFT entry has the following format:



where INTR$LOCATION is a word whose meaning depends upon whether the device corresponding to this array entry has memory-mapped interrupts. If it does have memory-mapped interrupts, INTR$LOCATION contains the device's wake-up address (decision 12). Otherwise, INTR$LOCATION should contain 0FFFFH.

If the device is being configured for MMX 88, each SFT entry has the following form:



where:

DEVICE$MODE is a BYTE containing a code that defines a characteristic of a device in the system. The possible code values and their literal equivalents are:

0 -- NO$DEVICE	The device being configured will not communicate with the device corresponding to this array entry.
1 -- SLAVE$DEVICE	The device corresponding to this array entry either is an iSBC 550 ethernet controller or sends messages to itself.
2 -- PEER$DEVICE	The device corresponding to this entry is not an iSBC 550 Ethernet controller.

INTR$TYPE is a BYTE containing a code for the interrupt scheme (decision 9) that the device being configured will use to interrupt the device corresponding to this array entry. The possible code values and literal equivalents of them are:

0 -- NO$INTERRUPT	The device corresponding to this array entry uses polling and cannot be interrupted.
1 -- MB$INTERRUPT	Multibus interrupts.
2 -- MM$INTERRUPT	Memory-mapped interrupts.
3 -- IO$INTERRUPT	I/O-mapped interrupts.

INTR$LOCATION is an ADDRESS containing the location where interrupts are to be generated. For Multibus interrupts (decision 10), this is the address of the I/O control port on the 8255 Programmable Peripheral Interface that is to generate an interrupt through the Multibus. For I/O-mapped interrupts (decision 11), this is the address of the I/O port that is to generate interrupts. For memory-mapped interrupts (decision 12), this is the base address of the memory location that is to generate interrupts.

INTR$VALUE is a WORD containing the value that is to be used to interrupt the device corresponding to this array entry. For Multibus interrupts (decision 10), the value (0 through 7) specifies which bit of Port C of an 8255 Programmable Peripheral Interface will generate interrupts onto the Multibus interface. For memory-mapped interrupts (decision 12), the value is to be written to the device's memory. For I/O-mapped interrupts (decision 11) the value is to be written to an I/O port.

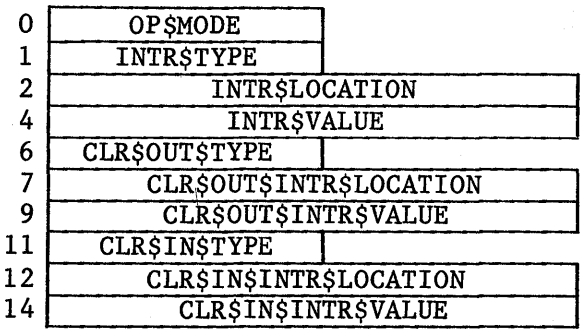
CLR$INTR$TYPE is a BYTE containing a code indicating the manner (decision 13) in which interrupts generated by the device are to be cleared. The possible code values and their literal equivalents are:

0 -- NO$INTR$CLEARED	It is not necessary to clear received interrupts.
1 -- MEMORY$READ$CLR	By reading from the memory location specified in the SFT structure.
2 -- MEMORY$WRITE$CLR	By writing to a memory location.
3 -- IO$READ$CLR	By reading from the I/O port specified in the SFT structure.
4 -- IO$WRITE$CLR	By writing to an I/O port.

CLR$LOCATION is an ADDRESS containing the base address (decision 14) of the memory location or the I/O port address to which the specified value is to be written, in order to clear interrupts.

CLR$VALUE is a WORD containing the value that is to be written to the specified I/O port or memory location, in order to clear interrupts.

If the device is being configured for MMX 86, each SFT entry has the following form:



where:

OP$MODE is a BYTE containing a code that defines a characteristic of a device in the system. The possible code values and their literal equivalents are:

0 -- NO$DEVICE	The device being configured will not communicate with the device corresponding to this array entry.
1 -- SLAVE$DEVICE	The device corresponding to this array entry either is an iSBC 550 ethernet controller or sends messages to itself.
2 -- PEER$DEVICE	The device corresponding to this entry is not an iSBC 550 Ethernet controller.

INTR$TYPE is an encoded BYTE indicating the manner (decision 9) in which the device being configured will interrupt the device corresponding to this array entry. The bit-level fields and their meanings are:

Bits 0-3 specify the type of interrupt, as follows:

0 -- No interrupts

1 -- Multibus interrupts through port C of the 8255 Programmable Peripheral Interface.

2 -- I/O-mapped interrupts by writing.

3 -- Memory-mapped interrupts by writing.

4 -- I/O-mapped interrupts by reading.

5 -- Memory-mapped interrupts by reading.

Bit 4 is reserved.

Bit 5 specifies whether the value in the INTR$VALUE field is to be written into INTR$LOCATION as a BYTE of data or as a WORD of data.

0 -- BYTE

1 -- WORD

Bits 6-7 are reserved.

INTR$LOCATION is a WORD containing the location where interrupts are to be generated. For Multibus interrupts (decision 10), this is the address of the I/O control port on the 8255 Programmable Peripheral Interface that is to generate an interrupt through the Multibus. For I/O-mapped interrupts (decision 11), this is the address of the I/O port that is to generate interrupts. For memory-mapped interrupts (decision 12), this is the base address of the memory location that is to generate interrupts.

INTR$VALUE is a WORD containing the value that is to be used to interrupt the device corresponding to this array entry. For Multibus interrupts (decision 10), the value (0 through 7) specifies which bit of Port C of an 8255 Programmable Peripheral Interface will generate interrupts onto the Multibus interface. For memory-mapped interrupts (decision 12), the value is to be written to the device's memory. For I/O-mapped interrupts (decision 11) the value is to be written to an I/O port.

CLROUTTYPE is an encoded BYTE indicating the manner (decision 13) in which interrupts generated by the device are to be cleared. The bit-level fields and their meanings are:

Bits 0-3 indicate the method of clearing the interrupts, as follows:

0 -- It is not necessary to clear generated interrupts.

1 -- By writing to an I/O port.

2 -- By writing to a memory location.

3 -- By reading from the I/O port specified in the SFT structure.

4 -- By reading from the memory location specified in the SFT structure.

Bit 4 is reserved.

Bit 5 specifies whether the value in the CLROUTINTR$VALUE field is to be written into CLROUTINTR$LOCATION as a BYTE of data or as a WORD of data. (0 means BYTE; 1 means WORD)

Bits 6-7 are reserved.

CLROUTINTR$LOCATION is a WORD that specifies the location to which the value in the CLROUTINTR$VALUE is to be written, in order to clear interrupts generated by the device. If the interrupt was I/O-mapped, this value is an I/O port address. If the interrupt was memory-mapped, this value is a base address.

CLROUTINTR$VALUE is a WORD containing the value that is to be written to the specified I/O port or memory location, in order to clear interrupts generated by the device.

CLRINTYPE is an encoded BYTE indicating the manner (decision 15) in which interrupts received by the device are to be cleared. The bit-level fields and their meanings are:

Bits 0-3 indicate the method of clearing the interrupts, as follows:

0 -- It is not necessary to clear received interrupts.

1 -- By writing to an I/O port.

2 -- By writing to a memory location.

3 -- By reading from the I/O port specified in the SFT structure.

4 -- By reading from the memory location specified in the SFT structure.

Bit 4 is reserved.

Bit 5 specifies whether the value in the CLRININTR$VALUE field is to be written into CLRININTR$LOCATION as a BYTE of data or as a WORD of data. (0 means BYTE; 1 means WORD)

Bits 6-7 are reserved.

CLRININTR$LOCATION is a WORD that specifies the location to which the value in the CLRININTR$VALUE is to be written, in order to clear interrupts received by the device. If the interrupt was I/O-mapped, this value is an I/O port address. If the interrupt was memory-mapped, this value is a base address.

CLRININTR$VALUE is a WORD containing the value that is to be written to the specified I/O port or memory location, in order to clear interrupts received by the device.

The variable CQITWT, which is of type WORD, contains the number of system time units (decision 7) that the device being configured will wait before beginning to communicate.

The variable CQMDLY, which is of type WORD, contains the number of system time units (decision 8) that the device being configured will wait for a response from another device.

The variable CQIDPD, which is of type WORD, specifies the device's polling period (decision 16).

The variable CQSGLV, which is of type BYTE (WORD in MMX 86), contains the interrupt level (decision 17), if any, that the iMMX 800 software uses to interrupt the device being configured. In iRMX 80- and iRMX 88-based applications, CQSGLV can be any value from 0 to 7 (0 to 0AH if the device is an iSBC 544 or 569 board, where 9, 0AH, and 0BH correspond to INT 7.5, 6.5, and 5.5, respectively), whereas in iRMX 86-based applications, it can range from 0 to 63. In the latter case, you must encode the level by first expressing it in octal, then using the following encoding scheme (bit 15 is the high-order bit):

Bits	Value
15-7	0
6-4	First octal digit (0-7) of the interrupt level
3	If 1, the level is a master level and bits 6-4 specify the entire level number If 0, the level is a slave level (refer to the iRMX 86 NUCLEUS REFERENCE MANUAL) and bits 2-0 specify the second octal digit
2-0	Second digit (0-7) of the interrupt level if bit 3 is 0; ignored otherwise

The variable CQLMEX is used only by MMX 80 and MMX 88 devices. In MMX 80 it is of type ADDRESS; in MMX 88 it is of type LOCATION. When used, it contains the address (decision 18) of the interrupt exchange for the interrupt level that the iMMX 800 software uses.

The array MCBI of structures has an entry for every iRMX 80-based device in the system. Each entry is 23 bytes in length. This array is used by the iMMX 800 software for internal communication between its tasks, so it does not need to be filled in during configuration.

Memory Description (CQIDSS, IDST)

The variable CQIDSS, which is of type BYTE, contains the number of interdevice segments (decision 6) in the system.

The array IDST of structures has an entry for each interdevice segment in the system and is indexed by the interdevice segment ID's. Each entry contains the location of an interdevice segment (decision 6) and has the following format:

where:

OFFSET is a WORD containing the offset of an interdevice segment.

PAGE is a WORD containing the page address of a 64K-byte page containing the interdevice segment whose IDST entry this is.

NOTE

If the device cannot address the IDS, the values of OFFSET and PAGE are each 0FFFFH.

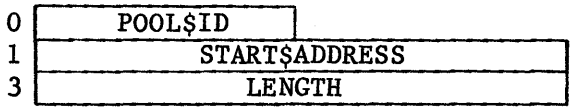
Memory Assignment (CQPLHS, PLHTBL, CQBLKS, BKLTBL)

The variable CQPLHS, which is of type BYTE, contains the number of memory pools available to the Partitioned Memory Manager residing on the device being configured.

The array PLHTBL of structures has an entry for every memory pool that is available to the Partitioned Memory Manager residing on the device being configured. The structures are filled in by the iMMX 800 software, but space for them must be declared during configuration. The structures vary in size, depending upon the operating system. For MMX 80, each structure is one byte in length; for MMX 86, each structure is two bytes in length; for the non-megabyte version of MMX 88, each structure is 29 bytes in length; and for the megabyte version of MMX 88, each structure is 53 bytes in length.

Each memory pool that is available to the Partitioned Memory Manager that resides on the device being configured consists of one or more contiguous blocks of memory. The variable CQBLKS, which is of type BYTE, contains the total number of such blocks in all such pools.

Each of the contiguous blocks (decision 21) just discussed is represented by an entry in an array of structures called BLKTBL. The entries each have the following format:



where:

POOL$ID is a BYTE containing the ID of the memory pool of which the block is a part.

START$ADDRESS is an ADDRESS in MMX 80 and a SELECTOR in both MMX 88 and MMX 86, and contains the address of the first byte of the block. (All blocks begin on paragraph boundaries.)

LENGTH is a WORD (ADDRESS in MMX 80) containing the length of the block. This length is expressed as a number of bytes in MMX 80 and as a number of (16-byte) paragraphs in MMX 88 and MMX 86.

A COMPREHENSIVE VIEW OF THE SYSTEM DATA STRUCTURES

The various data structures are interrelated in ways that are easy to understand but fairly hard to visualize. Figure 7-2 is provided to help you form a complete mental picture of these structures.

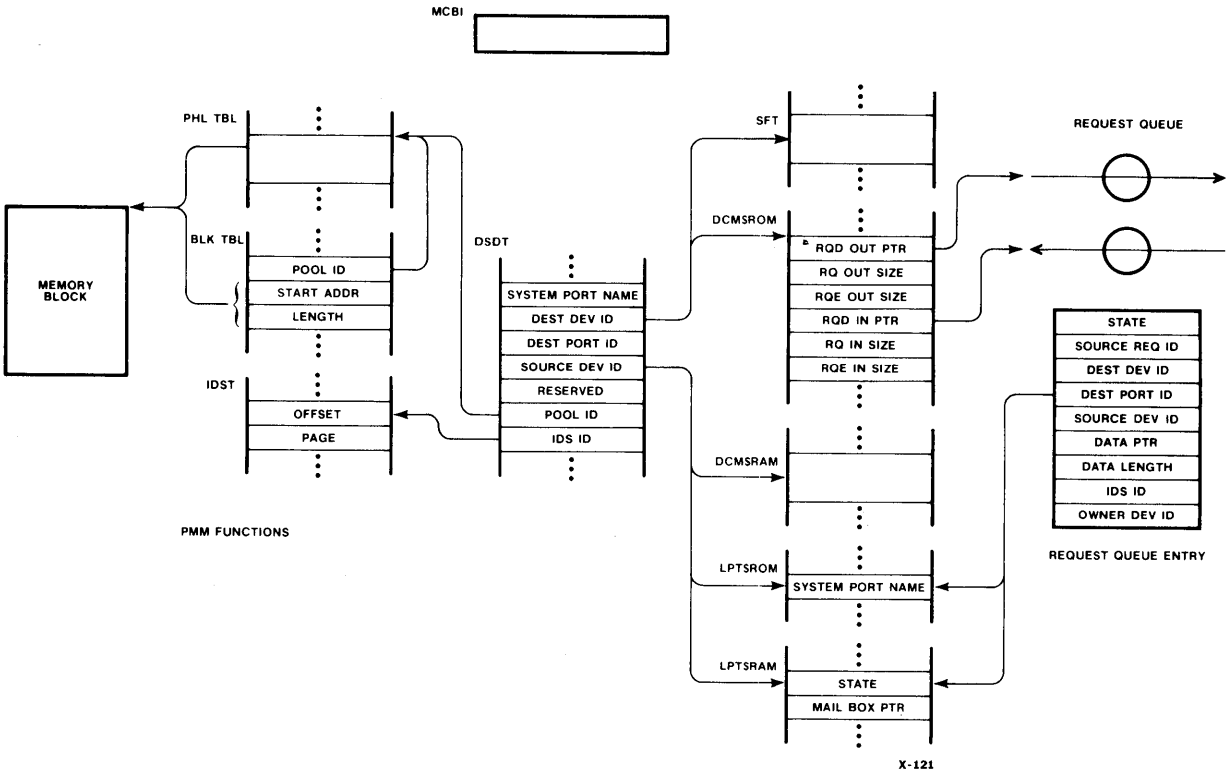


Figure 7-2. The Principal iMMX™ 800 Configuration Data Structures

AN EXAMPLE OF iMMX 800 CONFIGURATION

The example of this section shows the process of creating a configuration file for a small iMMX 800 application.

Making the Decisions

As is shown in Figure 7-3, the intended hardware configuration has a pair of iSBC 80/24 devices each communicating with an iSBC 86/12A device through the Multibus interface.

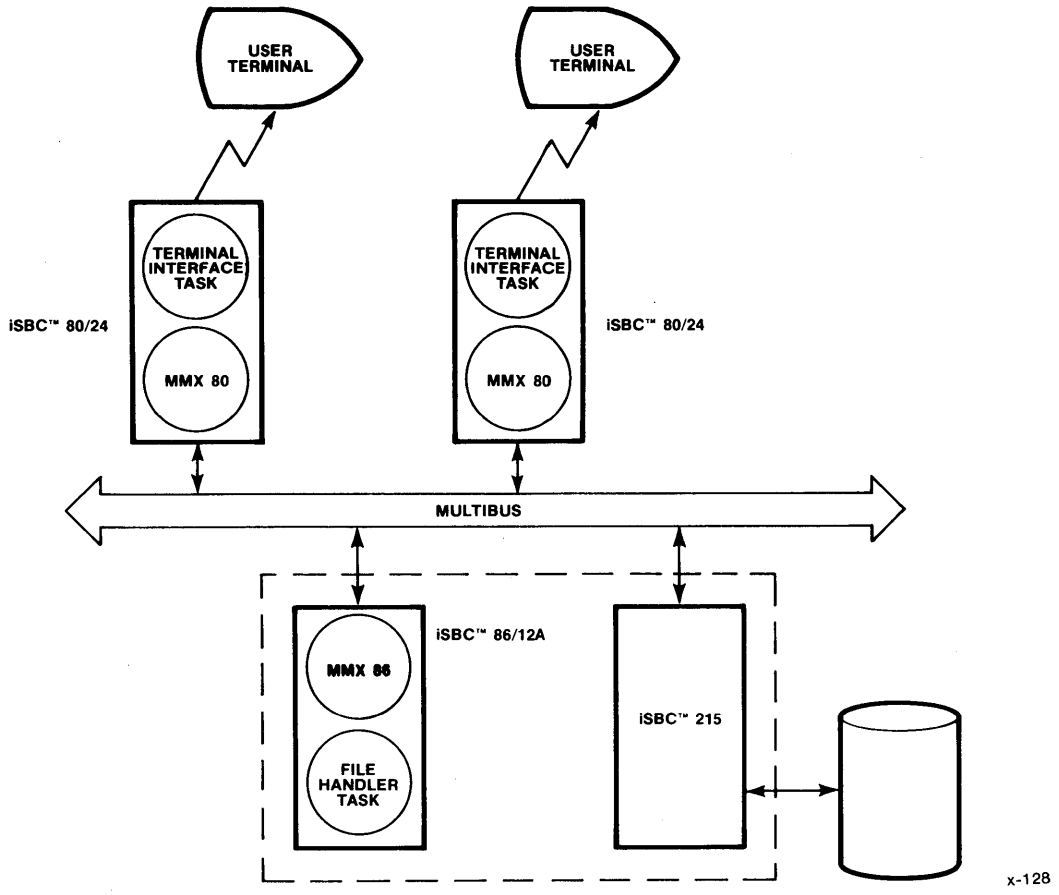


Figure 7-3. Example Target System

A terminal is connected to each of the iSBC 80/24 devices, with a supporting software interface. The iSBC 86/12A device is connected to an iSBC 215 Winchester Disk Controller that it uses for file access.

In order to support the desired interdevice communication, we need two channels, each connecting one of the iSBC 80/24 devices with the iSBC 86/12A device. Figure 7-4 shows this arrangement schematically.

Tasks in the iSBC 80/24 devices receive requests from the terminals and, with the support of MMX 80, pass the requests to a task on the iSBC 86/12A device. That task uses the requests to update files on the Winchester disk or to obtain information from the disk. The task on the iSBC 86/12A device then uses MMX 86 either to return a message of acknowledgment or to pass data back to a task on the iSBC 80/24 devices.

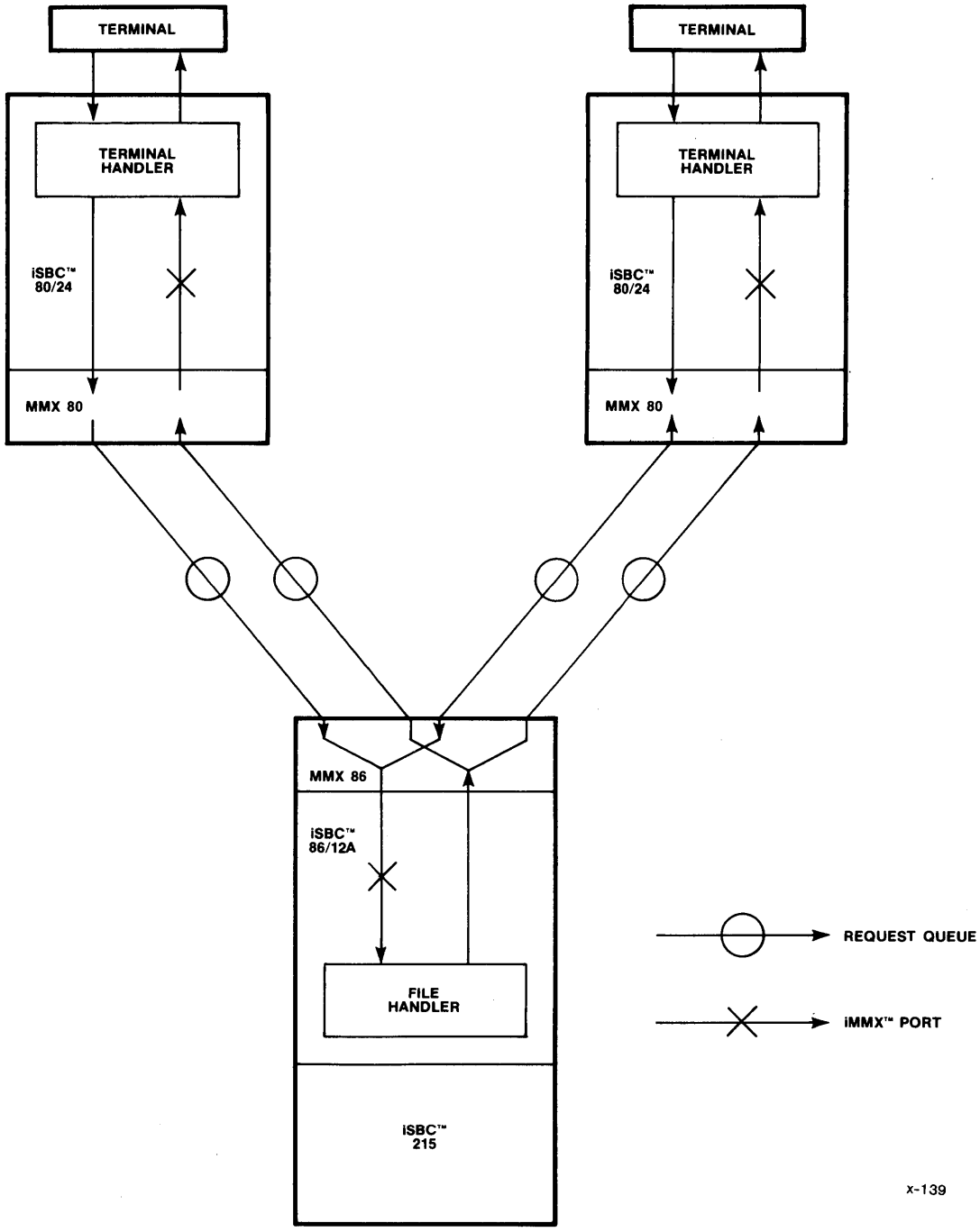


Figure 7-4. Example Target System with Channels

Information from both Terminal Handlers is delivered to the same system port on the iSBC 86/12A device. A user-defined message protocol identifies both the originator and the type of each message. Even though there is no direct connection between the iSBC 80/24 devices, the tasks on the two iSBC 80/24 devices can communicate with each other by way of the iSBC 86/12A device.

All three devices are interrupted by means of Multibus interrupts. Each device uses level 4 interrupts, so more devices can easily be added later, if necessary.

On each iSBC 80/24 device, we dedicate 16K of ROM to code for user tasks, the iRMX 80 software to support them, and the Terminal Handler. In addition, we set aside 8K of RAM for task stacks and tables, as well as for other dynamic needs, such as messages.

The iSBC 86/12A device (with an iSBC 300 RAM expansion module) has 64K of RAM and 16K of ROM. The 16K of ROM is for user tasks and the iRMX 86 and MMX 86 software that supports those tasks. Of the 64K of RAM, 44K is for the iRMX 86 I/O System interface with the iSBC 215 controller, 16K is for local use by the I/O System and other on-board tasks, and 4K is to be shared by all devices for communicating by means of the iMMX 800 services. Off-board ROM and RAM is used to meet local memory requirements where on-board memory does not suffice.

Figure 7-5 illustrates the allocation of these sections of memory and gives their addresses.

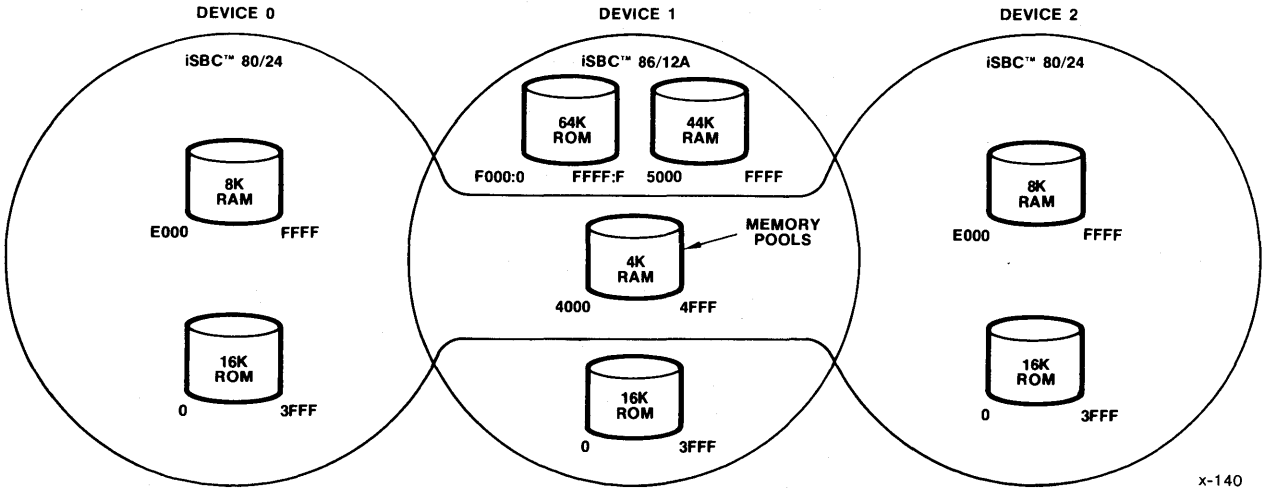


Figure 7-5. Initial Allocation of Memory

The 4K of shared RAM includes space for the two pairs of request queues and three pools of memory. Even though this memory is resident on the iSBC 86/12A device, two of the pools are each managed by one of the iSBC 80/24 devices, while the iSBC 86/12A device manages the other pool. The pools managed by the iSBC 80/24 devices are used for communication from the iSBC 80/24 devices to the iSBC 86/12A device, whereas the pool managed by the iSBC 86/12A device is used for communication in the other direction.

Figure 7-6 shows the final layout of memory for the three boards. It also shows the device and pool ID's for the example. Notice that each of the iSBC 80/24 devices has a private memory pool in addition to the pool in shared memory. These private pools are used by the MMX 80 Partitioned Memory Managers to allocate local memory to tasks.

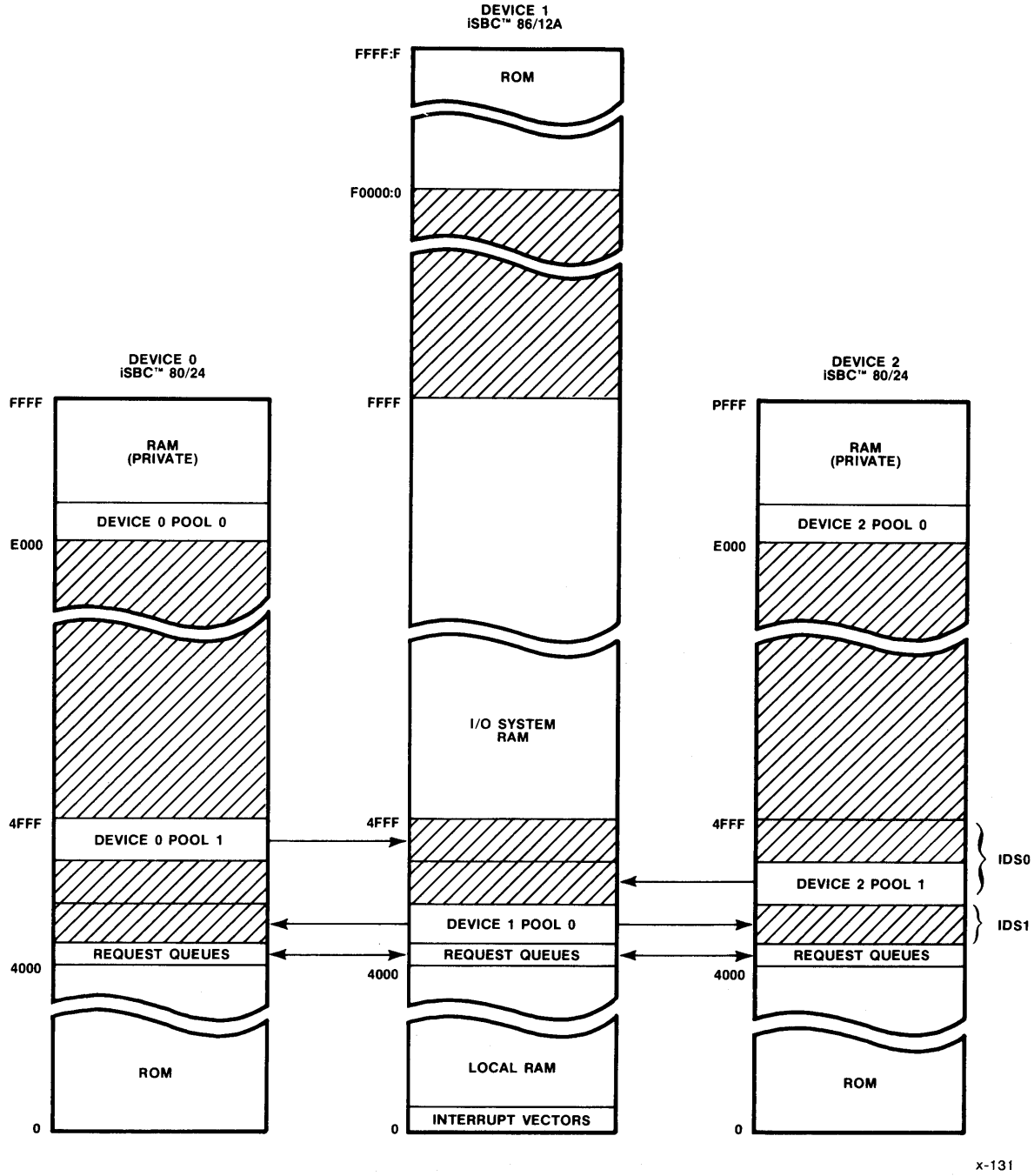


Figure 7-6. Memory Map for the Example

Notice that Figure 7-6 shows two interdevice segments and that their ID's are defined, as well. One IDS contains the two shared pools that are managed by the iSBC 80/24 devices, and the other contains the shared pool managed by the iSBC 86/12A device. Because the two IDS's are adjacent, they could be one IDS; we chose to use two IDS's for purposes of illustration.

Each request queue has an eight-byte descriptor followed by eight 16-byte slots for request queue entries, making 136 bytes in all. Rounding this up to 144 bytes so that each queue can start on a paragraph boundary, the addresses of the request queues are 4000H, 4090H, 4120H, and 41B0H. This leaves addresses 4240H through 4FFFH, a total of 3520 bytes, for the three shared pools. These three pools can then be 1168 bytes long, with 16 bytes being unused. Consequently, the pools begin at address 4240H, 46D0H, and 4B60H, and the IDS's begin at addresses 4240H and 46D0H.

The local pools on the iSBC 80/24 devices are each 1K bytes in length.

Filling the Structures

Having gotten this far, we have only minor decisions to make, so we can begin to fill in the data structures and provide declarations for the areas of RAM needed by the iMMX 800 software.

Before actually filling in the configuration structures, however it is to our advantage to prepare a list of literal declarations containing all the data that is to go into those structures. The benefit of this approach is that, if any of the data change in the future, only the literal statements need to be changed.

```
/* DEVICE DESCRIPTION */

          /* Three devices are present in the system. */

        DECLARE MMX$DEVICES LITERALLY '3';


          /* Each of the three devices requires an ID. */

        DECLARE MMX$DEVICE$0 LITERALLY '0';
        DECLARE MMX$DEVICE$1 LITERALLY '1';
        DECLARE MMX$DEVICE$2 LITERALLY '2';


/* CHANNEL DESCRIPTION */

          /* The DCM$ROM table for each device's configuration has an
             entry for each device in the system.  Each entry contains
             the addresses of the inbound and outbound queues, the size
             of each queue, and the size of the entry slots in each
             queue.  We define four request queues with a single entry
             size for all queues. */
```

```
DECLARE REQUEST$QUEUE$ENTRY$SIZE LITERALLY '04H';
DECLARE RQ$ADDR$DEVICE$0$TO$DEVICE$1 LITERALLY '4000H';
DECLARE RQ$SIZE$DEVICE$0$TO$DEVICE$1 LITERALLY '08H';

DECLARE RQ$ADDR$DEVICE$1$TO$DEVICE$0 LITERALLY '4090H;
DECLARE RQ$SIZE$DEVICE$1$TO$DEVICE$0 LITERALLY '08H';

DECLARE RQ$ADDR$DEVICE$1$TO$DEVICE$2 LITERALLY '4120H';
DECLARE RQ$SIZE$DEVICE$1$TO$DEVICE$2 LITERALLY '08H';

DECLARE RQ$ADDR$DEVICE$2$TO$DEVICE$1 LITERALLY '41B0H';
DECLARE RQ$SIZE$DEVICE$2$TO$DEVICE$1 LITERALLY '08H';


/* PORT DESCRIPTION */

   /* Each device has one local port and therefore one entry in each
      LPT$ROM table. */

DECLARE PORT$0$DEVICE$0 LITERALLY '00H';
DECLARE PORT$0$DEVICE$1 LITERALLY '00H';
DECLARE PORT$0$DEVICE$2 LITERALLY '00H';


   /* The following specify that there is one port on each device. */

DECLARE SOURCE$PORTS$DEVICE$0 LITERALLY '1';
DECLARE SOURCE$PORTS$DEVICE$1 LITERALLY '1';
DECLARE SOURCE$PORTS$DEVICE$2 LITERALLY '1';


   /* There is one system port on each device and each needs a
      unique name. */

DECLARE SYSTEM$PORT$NAME$DEVICE$0 LITERALLY '5030H';   /* P0 */
DECLARE SYSTEM$PORT$NAME$DEVICE$1 LITERALLY '5031H';   /* P1 */
DECLARE SYSTEM$PORT$NAME$DEVICE$2 LITERALLY '5032H';   /* P2 */


/* ADDRESS DESCRIPTION */

   /* Devices 0 and 2 each communicate with a single system port
      (the one on device 1), while device 1 communicates with two
      ports. */

DECLARE DESTINATION$PORT$DEVICE$0 LITERALLY '1';
DECLARE DESTINATION$PORT$DEVICE$1 LITERALLY '2';
DECLARE DESTINATION$PORT$DEVICE$2 LITERALLY '1';

   /* No literals are shown here for the DSDT entries, because all
      of the information for them either has already been declared
      (system port names, device and port ID's) or will be declared
      later (pool and interdevice segment ID's). */
```

```
/* ATTRIBUTE DESCRIPTIONS */

        /* Each entry in an MMX 86 SFT table requires ten values to
           describe the interrupt characteristics of a device.  The
           following declare these values for each of the iSBC 80/24
           devices.  Because the iSBC 86/12A device does not interrupt
           itself, we don't need to define values for its entry. */

      DECLARE OP$MODE$DEVICE$0 LITERALLY 'PEER$DEVICE';
      DECLARE INTR$TYPE$DEVICE$0 LITERALLY '00000001B';
      DECLARE INTR$LOCATION$DEVICE$0 LITERALLY '00CEH';
      DECLARE INTR$VALUE$DEVICE$0 LITERALLY '0005H';
      DECLARE CLR$OUT$TYPE$DEVICE$0 LITERALLY '00H';
      DECLARE CLR$OUT$INTR$LOCATION$DEVICE$0 LITERALLY '0000H';
      DECLARE CLR$OUT$INTR$VALUE$DEVICE$0 LITERALLY '0000H';
      DECLARE CLR$IN$TYPE$DEVICE$0 LITERALLY '00H';
      DECLARE CLR$IN$INTR$LOCATION$DEVICE$0 LITERALLY '0000H';
      DECLARE CLR$IN$INTR$VALUE$DEVICE$0 LITERALLY '0000H;

      DECLARE OP$MODE$DEVICE$2 LITERALLY 'PEER$DEVICE';
      DECLARE INTR$TYPE$DEVICE$2 LITERALLY '00000001B';
      DECLARE INTR$LOCATION$DEVICE$2 LITERALLY '00CEH';
      DECLARE INTR$VALUE$DEVICE$2 LITERALLY '0005H';
      DECLARE CLR$OUT$TYPE$DEVICE$2 LITERALLY '00H';
      DECLARE CLR$OUT$INTR$LOCATION$DEVICE$2 LITERALLY '0000H';
      DECLARE CLR$OUT$INTR$VALUE$DEVICE$2 LITERALLY '0000H';
      DECLARE CLR$IN$TYPE$DEVICE$2 LITERALLY '00H';
      DECLARE CLR$IN$INTR$LOCATION$DEVICE$2 LITERALLY '0000H';
      DECLARE CLR$IN$INTR$VALUE$DEVICE$2 LITERALLY '0000H;


        /* Each entry in an MMX 80 SFT table requires only one value.
           The following define that value for the devices. */

      DECLARE MM$INTERRUPT$ADDRESS$DEVICE$0 LITERALLY '0FFFFH';
      DECLARE MM$INTERRUPT$ADDRESS$DEVICE$1 LITERALLY '0FFFFH';
      DECLARE MM$INTERRUPT$ADDRESS$DEVICE$2 LITERALLY '0FFFFH';


        /* CQITWT defines the initial time period a device waits
           before beginning interdevice communications.  The following
           specify 2.56 seconds for each device, assuming that the
           system time unit is 10 milliseconds. */

      DECLARE INITIAL$DELAY$DEVICE$0 LITERALLY '0100H';
      DECLARE INITIAL$DELAY$DEVICE$1 LITERALLY '0100H';
      DECLARE INITIAL$DELAY$DEVICE$2 LITERALLY '0100H';


        /* CQMDLY defines the amount of time a device waits for a
           response from another device before timing out and
           declaring the other device dead.  This value must be larger
           than the initial delay value for CQITWT.  The following
           specify a 40.96-second time period for the devices. */
```

```
DECLARE DEAD$DELAY$PERIOD$DEVICE$0 LITERALLY '1000H';
DECLARE DEAD$DELAY$PERIOD$DEVICE$1 LITERALLY '1000H';
DECLARE DEAD$DELAY$PERIOD$DEVICE$2 LITERALLY '1000H';

  /* On interruptible devices CQIDPD is an interrupt timeout
     period.  On polling devices it is a polling period.  The
     following specify .16 seconds for each device. */

DECLARE POLLING$PERION$DEVICE$0 LITERALLY '0010H';
DECLARE POLLINR$PERION$DEVICE$1 LITERALLY '0010H';
DECLARE POLLING$PERIOD$DEVICE$2 LITERALLY '0010H';

  /* CQSGLV defines the interrupt level that the iMMX 800
     software uses to interrupt each device.  The following
     define interrupt level 4 for each device. */

DECLARE INTERRUPT$LEVEL$DEVICE$0 LITERALLY '4H';
DECLARE INTERRUPT$LEVEL$DEVICE$1 LITERALLY '48H';
DECLARE INTERRUPT$LEVEL$DEVICE$2 LITERALLY '4H';

  /* CQLMEX, which does not apply to MMX 86, contains the
     address of the interrupt exchange for the device for which
     it is defined.  The following establish the level 4
     interrupt exchange addresses for the iSBC 80/24 devices. */

DECLARE RQL4EX ADDRESS EXTERNAL:

DECLARE EXCHANGE$ADDRESS$DEVICE$0 LITERALLY '.RQL4EX';
DECLARE EXCHANGE$ADDRESS$DEVICE$2 LITERALLY '.RQL4EX';


/* MEMORY DESCRIPTION */

  /* The number of interdevice segments is a global value in any
     iMMX 800-based system.  There are two in the example. */

DECLARE MMX$INTERDEVICE$SEGMENTS LITERALLY '2';

  /* There are two IDS's in the example and no alias
     addressing.  For each device the IDST is the same. */

DECLARE IDS$0$ID LITERALLY '0';
DECLARE IDS$0$OFFSET LITERALLY '46D0H';
DECLARE IDS$0$PAGE LITERALLY '0000H';
DECLARE IDS$1$ID LITERALLY '1';
DECLARE IDS$1$OFFSET LITERALLY '4240H';
DECLARE IDS$1$PAGE LITERALLY '0000H';
```

```
/* MEMORY ASSIGNMENT */

        /* There are five pools in the example.  One is for MMX 86 to
           manage and there are two for each copy of MMX 80.  Of the
           two pools that each MMX 80 manages, one is for private use
           on the iSBC 80/24 device and is managed by the PMM, while
           the other is for shared use on the iSBC 86/12A device.  The
           following values are used in CQPLHS */

      DECLARE POOLS$DEVICE$0 LITERALLY '2';
      DECLARE POOLS$DEVICE$1 LITERALLY '1';
      DECLARE POOLS$DEVICE$2 LITERALLY '2';

        /* There is one block of memory for each pool. The following
           values are used in CQBLKS. */

      DECLARE BLOCKS$DEVICE$0 LITERALLY '2';
      DECLARE BLOCKS$DEVICE$1 LITERALLY '1';
      DECLARE BLOCKS$DEVICE$2 LITERALLY '2';


        /* The BLKTBL table for each copy of MMX 80 has two entries,
           while for MMX 86 there is one entry.  The values for those
           entries are defined as follows. */

      DECLARE POOL$0$ID$DEVICE$0 LITERALLY '0';
      DECLARE ADDR$BLK$0$DEVICE$0 LITERALLY 'E000H';
      DECLARE LNGTH$BLK$0$DEVICE$0 LITERALLY '0400H';

      DECLARE POOL$1$ID$DEVICE$0 LITERALLY '1';
      DECLARE ADDR$BLK$1$DEVICE$0 LITERALLY '4B60H';
      DECLARE LNGTH$BLK$1$DEVICE$0 LITERALLY '0490H';

      DECLARE POOL$0$ID$DEVICE$1 LITERALLY '0';
      DECLARE ADDR$BLK$0$DEVICE$1 LITERALLY '424H';
      DECLARE LNGTH$BLK$0$DEVICE$1 LITERALLY '049H';

      DECLARE POOL$0$ID$DEVICE$2 LITERALLY '0';
      DECLARE ADDR$BLK$0$DEVICE$2 LITERALLY 'E000H';
      DECLARE LNGTH$BLK$0$DEVICE$2 LITERALLY '0400H';

      DECLARE POOL$1$ID$DEVICE$2 LITERALLY '1';
      DECLARE ADDR$BLK$1$DEVICE$2 LITERALLY '46D0H';
      DECLARE LNGTH$BLK$1$DEVICE$2 LITERALLY '0490H';
```

That completes the declarations of the literals. Now we can begin the actual process of configuring each of the three portions of the overall system.

```
/* DEVICE 0 CONFIGURATION DECLARATIONS -- MMX 80 DEVICE */

     DECLARE CQDVCS BYTE PUBLIC DATA (MMX$DEVICES);

     DECLARE DM$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
          RQD$OUT                   ADDRESS,
          RQ$OUT$SIZE               BYTE,
          RQE$OUT$SIZE              BYTE,
          RQD$IN                    ADDRESS,
          RQ$IN$SIZE                BYTE,
          RQE$IN$SIZE               BYTE)';

     DECLARE NO$SYSTEM$CHANNEL LITERALLY '
          0FFFFH,
          00H,
          00H,
          0FFFFH,
          00H,
          00H';

     DECLARE DCM$ROM (MMX$DEVICES) DM$ROM$ENTRY$TYPE PUBLIC DATA (
          NO$SYSTEM$CHANNEL,  /* No path between dev 0 and dev 0 */
          RQ$ADDR$DEVICE$0$TO$DEVICE$1,
               RQ$SIZE$DEVICE$0$TO$DEVICE$1,
               REQUEST$QUEUE$ENTRY$SIZE,
               RQ$ADDR$DEVICE$1$TO$DEVICE$0,
               RQ$SIZE$DEVICE$1$TO$DEVICE$0,
               REQUEST$QUEUE$ENTRY$SIZE,
          NO$SYSTEM$CHANNEL);  /* No path between dev 0 and dev 2 */


     DECLARE DM$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
          ENTRY(20)                 BYTE)';

     DECLARE DCM$RAM(MMX$DEVICES) DM$RAM$ENTRY$TYPE PUBLIC;

     DECLARE CQPRTS BYTE PUBLIC DATA (SOURCE$PORT$DEVICE$0);

     DECLARE LPT$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
          SYSTEM$PORT$NAME          ADDRESS,
          POOL$ID                   BYTE)';

     DECLARE LPT$ROM(SOURCE$PORT$DEVICE$0) LPT$ROM$ENTRY$TYPE PUBLIC
                                                          DATA (
          SYSTEM$PORT$NAME$DEVICE$0,
          POOL$1$ID$DEVICE$0);

     DECLARE LPT$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
          ENTRY(11)                 BYTE)';

     DECLARE LPT$RAM(SOURCE$PORT$DEVICE$0) LPT$RAM$ENTRY$TYPE PUBLIC;

     DECLARE CQSKTS BYTE PUBLIC DATA (DESTINATION$PORT$DEVICE$0);
```

```
DECLARE DSD$ENTRY$TYPE LITERALLY 'STRUCTURE(
     SYSTEM$PORT$NAME          ADDRESS,
     DEST$DEV$ID               BYTE,
     DEST$PORT$ID              BYTE,
     SRC$DEV$ID                BYTE,
     RESERVED                  BYTE,
     POOL$ID                   BYTE,
     IDS$ID                    BYTE)';

DECLARE DSDT(DESTINATION$PORT$DEVICE$0) DSD$ENTRY$TYPE PUBLIC
                                                    DATA(
     SYSTEM$PORT$NAME$DEVICE$1,
     MMX$DEVICE$1,
     PORT$0$DEVICE$1,
     MMX$DEVICE$0,
     0,
     POOL$1$ID$DEVICE$0,
     IDS$0$ID);

DECLARE CQITWT WORD PUBLIC DATA (INITIAL$DELAY$DEVICE$0);

DECLARE CQMDLY WORD PUBLIC DATA (
     DEAD$DELAY$PERIOD$DEVICE$0);

DECLARE MCBI$ENTRY$TYPE LITERALLY 'STRUCTURE(
     ENTRY(23)                  BYTE)';

DECLARE MCBIT(MMX$DEVICES) MCBI$ENTRY$TYPE PUBLIC;

DECLARE SFT$ENTRY$TYPE LITERALLY 'ADDRESS';

DECLARE SFT(MMX$DEVICES) SFT$ENTRY$TYPE PUBLIC DATA (
     MM$INTERRUPT$ADDRESS$DEVICE$0,
     MM$INTERRUPT$ADDRESS$DEVICE$1,
     MM$INTERRUPT$ADDRESS$DEVICE$2);

DECLARE CQIDPD WORD PUBLIC DATA (POLLING$PERIOD$DEVICE$0);

DECLARE CQSGLV BYTE PUBLIC DATA (INTERRUPT$LEVEL$DEVICE$0);

DECLARE CQLMEX ADDRESS PUBLIC DATA (EXCHANGE$ADDRESS$DEVICE$0);

DECLARE CQIDSS BYTE PUBLIC DATA (MMX$INTERDEVICE$SEGMENTS);

DECLARE IDS$ENTRY$TYPE LITERALLY 'STRUCTURE(
     OFFSET                     ADDRESS,
     PAGE                       ADDRESS)';

DECLARE IDST(MMX$INTERDEVICE$SEGMENTS) IDST$ENTRY$TYPE PUBLIC
                                                    DATA(
     IDS$0$OFFSET,
          IDS$0$PAGE,
     IDS$1$OFFSET,
          IDS$1$PAGE),

DECLARE CQPLHS BYTE PUBLIC DATA (POOLS$DEVICE$0);
```

```
    DECLARE PLHTBL$ENTRY$TYPE LITERALLY 'BYTE';

    DECLARE PLHTBL(POOLS$DEVICE$0) PLHTBL$ENTRY$TYPE PUBLIC;

    DECLARE CQBLKS BYTE PUBLIC DATA (BLOCKS$DEVICE$0);

    DECLARE BLKTBL$ENTRY$TYPE LITERALLY 'STRUCTURE(
         POOL$ID                          BYTE,
         START$ADDRESS                    ADDRESS,
         LENGTH                           ADDRESS)';

    DECLARE BLKTBL(BLOCKS$DEVICE$0) BLKTBL$ENTRY$TYPE PUBLIC DATA (
         POOL$0$ID$DEVICE$0,
              ADDR$BLK$0$DEVICE$0,
              LNGTH$BLK$0$DEVICE$0,
         POOL$1$ID$DEVICE$0,
              ADDR$BLK$1$DEVICE$0,
              LNGTH$BLK$1$DEVICE$0);

/* DEVICE 1 CONFIGURATION DECLARATIONS -- MMX 86 DEVICE */

    DECLARE CQDVCS BYTE PUBLIC DATA (MMX$DEVICES);

    DECLARE DM$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
         RQD$OUT                          POINTER,
         RQ$OUT$SIZE                      BYTE,
         RQE$OUT$SIZE                     BYTE,
         RQD$IN                           POINTER,
         RQ$IN$SIZE                       BYTE,
         RQE$IN$SIZE                      BYTE)';

    DECLARE NO$SYSTEM$CHANNEL LITERALLY '
         0FFFFH,
         00H,
         00H,
         0FFFFH,
         00H,
         00H';

    DECLARE DCM$ROM (MMX$DEVICES) DM$ROM$ENTRY$TYPE PUBLIC DATA (
         RQ$ADDR$DEVICE$1$TO$DEVICE$0,
              RQ$SIZE$DEVICE$1$TO$DEVICE$0,
              REQUEST$QUEUE$ENTRY$SIZE,
              RQ$ADDR$DEVICE$0$TO$DEVICE$1,
              RQ$SIZE$DEVICE$0$TO$DEVICE$1,
              REQUEST$QUEUE$ENTRY$SIZE,
         NO$SYSTEM$CHANNEL,
         RQ$ADDR$DEVICE$1$TO$DEVICE$2,
              RQ$SIZE$DEVICE$1$TO$DEVICE$2,
              REQUEST$QUEUE$ENTRY$SIZE,
              RQ$ADDR$DEVICE$2$TO$DEVICE$1,
              RQ$SIZE$DEVICE$2$TO$DEVICE$1,
              REQUEST$QUEUE$ENTRY$SIZE);

    DECLARE DM$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
         ENTRY(20)                        BYTE)';
```

```
DECLARE DCM$RAM(MMX$DEVICES) DM$RAM$ENTRY$TYPE PUBLIC;

DECLARE CQPRTS BYTE PUBLIC DATA (SOURCE$PORT$DEVICE$0);

DECLARE LPT$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
     SYSTEM$PORT$NAME              WORD)';

DECLARE LPT$ROM(SOURCE$PORT$DEVICE$1) LPT$ROM$ENTRY$TYPE PUBLIC DATA (
     SYSTEM$PORT$NAME$DEVICE$1);

DECLARE LPT$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
     ENTRY(3)                      BYTE)';

DECLARE LPT$RAM(SOURCE$PORT$DEVICE$1) LPT$RAM$ENTRY$TYPE PUBLIC;

DECLARE CQSKTS BYTE PUBLIC DATA (DESTINATION$PORT$DEVICE$1);

DECLARE DSD$ENTRY$TYPE LITERALLY 'STRUCTURE(
     SYSTEM$PORT$NAME              WORD,
     DEST$DEV$ID                   BYTE,
     DEST$PORT$ID                  BYTE,
     SRC$DEV$ID                    BYTE,
     RESERVED                      BYTE,
     POOL$ID                       BYTE,
     IDS$ID                        BYTE)';

DECLARE DSDT(DESTINATION$PORT$DEVICE$1) DSD$ENTRY$TYPE PUBLIC
                                                     DATA(
     SYSTEM$PORT$NAME$DEVICE$0,
          MMX$DEVICE$0,
          PORT$0$DEVICE$0,
          MMX$DEVICE$1,
          0,
          POOL$0$ID$DEVICE$1,
          IDS$1$ID,
     SYSTEM$PORT$NAME$DEVICE$2,
          MMX$DEVICE$2,
          PORT$0$DEVICE$2,
          MMX$DEVICE$1,
          0,
          POOL$0$ID$DEVICE$1,
          IDS$1$ID);

DECLARE CQITWT WORD PUBLIC DATA (INITIAL$DELAY$DEVICE$1);

DECLARE CQMDLY WORD PUBLIC DATA (
     DEAD$DELAY$PERIOD$DEVICE$1);
```

```
DECLARE SFT$ENTRY$TYPE LITERALLY 'STRUCTURE(
     OP$MODE                         BYTE,
     INTR$TYPE                       BYTE,
     INTR$LOCATION                   WORD,
     INTR$VALUE                      WORD,
     CLR$OUT$TYPE                    BYTE,
     CLR$OUT$INTR$LOCATION           WORD,
     CLR$OUT$INTR$VALUE              WORD,
     CLR$IN$TYPE                     BYTE,
     CLR$IN$INTR$LOCATION            WORD,
     CLR$IN$INTR$VALUE               WORD)';

DECLARE NOT$INTERRUPTED LITERALLY '
     00H,
     00H,
     0000H,
     0000H,
     00H,
     0000H,
     0000H.
     00H,
     0000H,
     0000H';

DECLARE SFT(MMX$DEVICES) SFT$ENTRY$TYPE PUBLIC DATA (
     OP$MODE$DEVICE$0,
          INTR$TYPE$DEVICE$0,
          INTR$LOCATION$DEVICE$0,
          INTR$VALUE$DEVICE$0,
          CLR$OUT$TYPE$DEVICE$0,
          CLR$OUT$INTR$LOCATION$DEVICE$0,
          CLR$OUT$INTR$VALUE$DEVICE$0,
          CLR$IN$TYPE$DEVICE$0,
          CLR$IN$INTR$LOCATION$DEVICE$0,
          CLR$IN$INTR$VALUE$DEVICE$0,
     NOT$INTERRUPTED,    /* Device 1 doesn't interrupt itself */
     OP$MODE$DEVICE$2,
          INTR$TYPE$DEVICE$2,
          INTR$LOCATION$DEVICE$2,
          INTR$VALUE$DEVICE$2,
          CLR$OUT$TYPE$DEVICE$2,
          CLR$OUT$INTR$LOCATION$DEVICE$2,
          CLR$OUT$INTR$VALUE$DEVICE$2,
          CLR$IN$TYPE$DEVICE$2,
          CLR$IN$INTR$LOCATION$DEVICE$2,
          CLR$IN$INTR$VALUE$DEVICE$2);

DECLARE CQIDPD WORD PUBLIC DATA (POLLING$PERIOD$DEVICE$1);

DECLARE CQSGLV WORD PUBLIC DATA (INTERRUPT$LEVEL$DEVICE$1);

DECLARE CQIDSS BYTE PUBLIC DATA (MMX$INTERDEVICE$SEGMENTS);

DECLARE IDS$ENTRY$TYPE LITERALLY 'STRUCTURE(
     OFFSET                          WORD,
     PAGE                            WORD)';
```

```
DECLARE IDST(MMX$INTERDEVICE$SEGMENTS) IDST$ENTRY$TYPE PUBLIC
                                                        DATA(
     IDS$0$OFFSET,
          IDS$0$PAGE,
     IDS$1$OFFSET,
          IDS$1$PAGE);

DECLARE CQPLHS BYTE PUBLIC DATA (POOL$DEVICE$1);

DECLARE PLHTBL$ENTRY$TYPE LITERALLY 'STRUCTURE(
     ENTRY(2)                       BYTE)';

DECLARE PLHTBL(POOL$DEVICE$1) PHLTBL$ENTRY$TYPE PUBLIC;

DECLARE CQBLKS BYTE PUBLIC DATA (BLOCKS$DEVICE$1);

DECLARE BLKTBL$ENTRY$TYPE LITERALLY 'STRUCTURE(
     POOL$ID                        BYTE,
     START$ADDRESS                  WORD,
     LENGTH                         WORD)';

DECLARE BLKTBL(BLOCKS$DEVICE$1) BLKTBL$ENTRY$TYPE PUBLIC DATA(
     POOL$0$ID$DEVICE$1,
          ADDR$BLK$0$DEVICE$1,
          LNGTH$BLK$0$DEVICE$1);


/* DEVICE 2 CONFIGURATION DECLARATIONS -- MMX 80 DEVICE */

     DECLARE CQDVCS BYTE PUBLIC DATA (MMX$DEVICES);

     DECLARE DM$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
          RQD$OUT                   ADDRESS,
          RQ$OUT$SIZE               BYTE,
          RQE$OUT$SIZE              BYTE,
          RQD$IN                    ADDRESS,
          RQ$IN$SIZE                BYTE,
          RQE$IN$SIZE               BYTE)';

     DECLARE NO$SYSTEM$CHANNEL LITERALLY '
          0FFFFH,
          00H,
          00H,
          0FFFFH,
          00H,
          00H';
```

```
DECLARE DCM$ROM (MMX$DEVICES) DM$ROM$ENTRY$TYPE PUBLIC DATA (
     NO$SYSTEM$CHANNEL,  /* No path between dev 0 and dev 2 */
     RQ$ADDR$DEVICE$2$TO$DEVICE$1,
          RQ$SIZE$DEVICE$2$TO$DEVICE$1,
          REQUEST$QUEUE$ENTRY$SIZE,
          RQ$ADDR$DEVICE$1$TO$DEVICE$2,
          RQ$SIZE$DEVICE$1$TO$DEVICE$2,
          REQUEST$QUEUE$ENTRY$SIZE,
     NO$SYSTEM$CHANNEL);  /* No path between dev 2 and dev 2 */

DECLARE DM$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
     ENTRY(20)                BYTE)';

DECLARE DCM$RAM(MMX$DEVICES) DM$RAM$ENTRY$TYPE PUBLIC;

DECLARE CQPRTS BYTE PUBLIC DATA (SOURCE$PORT$DEVICE$2);

DECLARE LPT$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
     SYSTEM$PORT$NAME         ADDRESS,
     POOL$ID                  BYTE)';

DECLARE LPT$ROM(SOURCE$PORT$DEVICE$2) LPT$ROM$ENTRY$TYPE PUBLIC
                                                        DATA (
     SYSTEM$PORT$NAME$DEVICE$2,
     POOL$1$ID$DEVICE$2);

DECLARE LPT$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
     ENTRY(11)                BYTE)';

DECLARE LPT$RAM(SOURCE$PORT$DEVICE$2) LPT$RAM$ENTRY$TYPE PUBLIC;

DECLARE CQSKTS BYTE PUBLIC DATA (DESTINATION$PORT$DEVICE$2);

DECLARE DSD$ENTRY$TYPE LITERALLY 'STRUCTURE(
     SYSTEM$PORT$NAME         ADDRESS,
     DEST$DEVICE$ID           BYTE,
     DEST$PORT$ID             BYTE,
     SRC$DEV$ID               BYTE,
     RESERVED                 BYTE,
     POOL$ID                  BYTE,
     IDS$ID                   BYTE)';

DECLARE DSDT(DESTINATION$PORT$DEVICE$0) DSD$ENTRY$TYPE PUBLIC
                                                      DATA(
     SYSTEM$PORT$NAMEDEVICE$1,
     MMX$DEVICE$1,
     PORT$1$DEVICE$1,
     MMX$DEVICE$2,
     0,
     POOL$1$ID$DEVICE$2,
     IDS$0$ID);

DECLARE CQITWT WORD PUBLIC DATA (INITIAL$DELAY$DEVICE$2);

DECLARE CQMDLY WORD PUBLIC DATA (
     DEAD$DELAY$PERIOD$DEVICE$2);
```

```
DECLARE MCBI$ENTRY$TYPE LITERALLY 'STRUCTURE(
     ENTRY(23)                    BYTE)';

DECLARE MCBIT(MMX$DEVICES) MCBI$ENTRY$TYPE;

DECLARE SFT$ENTRY$TYPE LITERALLY 'ADDRESS';

DECLARE SFT(MMX$DEVICES) SFT$ENTRY$TYPE PUBLIC DATA (
     MM$INTERRUPT$ADDRESS$DEVICE$0,
     MM$INTERRUPT$ADDRESS$DEVICE$1,
     MM$INTERRUPT$ADDRESS$DEVICE$2);

DECLARE CQIDPD WORD PUBLIC DATA (POLLING$PERIOD$DEVICE$2);

DECLARE CQSGLV BYTE PUBLIC DATA (INTERRUPT$LEVEL$DEVICE$2);

DECLARE CQLMEX ADDRESS PUBLIC DATA (EXCHANGE$ADDRESS$DEVICE$2);

DECLARE CQIDSS BYTE PUBLIC DATA (MMX$INTERDEVICE$SEGMENTS);

DECLARE IDS$ENTRY$TYPE LITERALLY 'STRUCTURE(
     OFFSET                       ADDRESS,
     PAGE                         ADDRESS)';

DECLARE IDST(MMX$INTERDEVICE$SEGMENTS) IDST$ENTRY$TYPE PUBLIC
                                                     DATA(
     IDS$0$OFFSET,
          IDS$0$PAGE,
     IDS$1$OFFSET,
          IDS$1$PAGE),

DECLARE CQPLHS BYTE PUBLIC DATA (POOLS$DEVICE$2);

DECLARE PLHTBL$ENTRY$TYPE LITERALLY 'BYTE';

DECLARE PLHTBL(POOLS$DEVICE$2) PLHTBL$ENTRY$TYPE PUBLIC;

DECLARE CQBLKS BYTE PUBLIC DATA (BLOCKS$DEVICE$2);

DECLARE BLKTBL$ENTRY$TYPE LITERALLY 'STRUCTURE(
     POOL$ID                      BYTE,
     START$ADDRESS                ADDRESS,
     LENGTH                       ADDRESS)';

DECLARE BLKTBL(BLOCKS$DEVICE$2) BLKTBL$ENTRY$TYPE PUBLIC DATA (
     POOL$0$ID$DEVICE$2,
          ADDR$BLK$0$DEVICE$2,
          LNGTH$BLK$0$DEVICE$2,
     POOL$1$ID$DEVICE$2,
          ADDR$BLK$1$DEVICE$2,
          LNGTH$BLK$1$DEVICE$2);
```

This completes the example of how to create a configuration file. Note that the literal file R1CNFG.LIT could have been INCLUDEd at the beginning of the MMX 80 files. This would have eliminated the need to write out each literal declaration explicitly. Similarly, R4CNFG.LIT could have been INCLUDEd at the beginning of the MMX 86 files. If there had been MMX 88 files, R2CNFG.LIT (for non-megabyte), R3CNFG.LIT (for megabyte, COMPACT size control), or R5CNFG.LIT (for megabyte, LARGE size control) could have been INCLUDEd at the beginning of them.

Another aid that we could have used (but didn't, for purposes of illustration) is files of non-literal declarations with the data missing. These files are R1CNFG.P80 (for MMX 80), R2CNFG.P86 (for MMX 88 non-megabyte), R3CNFG.P86 (for MMX 88 megabyte, COMPACT size control), R4CNFG.P86 (for MMX 86), and R5CNFG.P86 (for MMX 88 megabyte, LARGE size control), and they contain such things as

```
DECLARE BLKTBL( ) BLKTBL$ENTRY$TYPE PUBLIC DATA(
```

Typical usage of these files is as follows:

1. Place the instruction $INCLUDE (RnCNFG.LIT) at the beginning of the RnCNFG.P8x file (where n = 1, 2, 3, 4, or 5, and x = 0 or 6.)

2. Fill in the data that is missing from the RnCNFG.P8x file.

3. Compile the file by entering either

```
    PLM80 R1CNFG.P80 (cr)              (for MMX 80)
or
    PLM86 RnCNFG.P86 COMPACT ROM (cr)  (for MMX 86 or MMX 88)
or
    PLM86 RnCNFG.P86 LARGE ROM (cr)    (for MMX 86 or MMX 88)
```

The result is the file RnCNFG.OBJ. We will say more about this file later.

LINKING AND LOCATING iMMX 800 APPLICATION SYSTEMS

This section assumes that you have compiled or assembled those tasks (or jobs) that will run on the device for which this configuration is being performed. In addition, it is assumed that you have compiled your configuration module. If you have not yet done these things, the following INCLUDE files can help you to save time and code space:

R1XMGR.LIT R2XMGR.LIT R3XMGR.LIT R4COM.LIT R5XMGR.LIT	INCLUDE files containing declarations of constants that pertain to MMX 80, non-megabyte MMX 88, megabyte MMX 88 with the COMPACT size control, MMX 86, and megabyte MMX 88 with the LARGE size control, respectively.
R1XMGR.EXT R2XMGR.EXT R3XMGR.EXT R4XINF.EXT R5XMGR.EXT	INCLUDE files containing external declarations of the procedures in MMX 80, non-megabyte MMX 88, megabyte MMX 88 with the COMPACT size control, MMX 86, and megabyte MMX 88 with the LARGE size control, respectively.

R1PMM.EXT R2PMM.EXT R3PMM.EXT R5PMM.EXT	INCLUDE files containing external declarations of the PMM exchanges for MMX 80, non-megabyte MMX 88, megabyte MMX 88 with the COMPACT size control, and megabyte MMX 88 with the LARGE size control, respectively.
R1PMM.LIT R2PMM.LIT R3PMM.LIT R5PMM.LIT	PMM INCLUDE files containing message structure literals for MMX 80, non-megabyte MMX 88, megabyte MMX 88 with the COMPACT size control, and megabyte MMX 88 with the LARGE size control, respectively.

If you INCLUDE any of these files and your version of the PL/M-86 compiler does not recognize the SELECTOR data type, you should INCLUDE the file R4SELC.LIT (from the iMMX 800 diskette) ahead of the other INCLUDEs in every module in which the other INCLUDEs appear.

Linking and Locating for MMX 80

The linking and locating operations that relate to MMX 80 usage involve using the Interactive Configuration Utility (ICU80) for the iRMX 80 Executive. You respond to ICU80 prompts by entering descriptive information about your hardware and software. The result of doing this is a submit file that you SUBMIT to produce a linked and located system.

When you use ICU80 to accomplish linking and locating for an MMX 80-based device, you must respond to certain prompts in certain ways. This section explains the special actions that you must perform. One of these is that, when you are prompted for the data concerning your tasks, you must enter certain tasks in a particular order. The following excerpts from an ICU80 dialogue illustrate this and the other requirements:

```
FSM: NO

        [other prompts and responses here]

TASK NAME:  CQDRVR
        ENTRY POINT:    CQDRVR
        STK LENGTH:     150
        PRIORITY:       129
        DFLT EXCHG:
        TASK DESCRIPTOR:     [your choice]
        EXTRA:     0

TASK NAME:  CQINTM
        ENTRY POINT:    CQINTM
        STK LENGTH:     75
        PRIORITY:       [determined by interrupt level used by MMX 80]
        DFLT EXCHG:
        TASK DESCRIPTOR:     [your choice]
        EXTRA:     0
```

```
TASK NAME:      RQPMT
     ENTRY POINT:   RQPMT
     STK LENGTH:    40
     PRIORITY:      131       [or whatever; must be same as priority
     DFLT EXCHG:    RQPMX                                    of RQFLMT]
     TASK DESCRIPTOR:    [your choice]
     EXTRA:    0

TASK NAME:      RQFLMT
     ENTRY POINT:   RQFLMT
     STK LENGTH:    40
     PRIORITY:      131       [or whatever; must be same as priority
     DFLT EXCHG:    RQFLMX                                   of RQPMT]
     TASK DESCRIPTOR:    [your choice]
     EXTRA:    0

          [user tasks here]

EXCHANGE:       RQFSAX
     SCOPE:     EXTERNAL
     INTERRUPT:      NO

EXCHANGE:       RQFSRX
     SCOPE:     EXTERNAL
     INTERRUPT:      NO

EXCHANGE:       RQPMX
     SCOPE:     EXTERNAL
     INTERRUPT:      NO

EXCHANGE:       RQFLMX
     SCOPE:     EXTERNAL
     INTERRUPT:      NO

EXCHANGE:       CQMXIX
     SCOPE:     EXTERNAL
     INTERRUPT:      NO

EXCHANGE:       CQMXRX
     SCOPE:     EXTERNAL
     INTERRUPT:      NO

EXCHANGE:       RQLnEX    [n = 0-7, 9, 0AH, 0BH, depending on interrupt
     SCOPE:     PUBLIC                                        level]
     INTERRUPT:      YES

          [user exchanges here or mixed with the other exchanges]

          [other prompts and responses here]

LINK:     [user modules]
LINK:     [compiled configuration module -- probably R1CNFG.OBJ]
LINK:     :Fn:R1XMGR.LIB
LINK:     :Fn:R1DRVR.LIB
LINK:     :Fn:R1PMM.LIB
LINK:     :Fn:R1????.LIB
```

where ???? is 8024 (for an iSBC 80/24 board), 8030 (for an iSBC 80/30 board), 544 (for an iSBC 544 board), or 569 (for an iSBC 569 board), depending upon the type of device on which the located code will run.

Note that the Partitioned Memory Manager can be configured into an application independently of the rest of MMX 80. To accomplish this, follow the procedure just described, except omit the definitions of the CQDRVR and CQINTM tasks, the CQMXIX, CQMXRX, and RQLnEX exchanges, and the R1XMGR, R1DRVR, and R1???? links.

Linking and Locating for MMX 88

The linking and locating operations that relate to MMX 88 usage involve using the Interactive Configuration Utility (ICU88) for the iRMX 88 Executive. You respond to ICU88 prompts by entering descriptive information about your hardware and software. The result of doing this is a submit file that you SUBMIT to produce a linked and located system.

When you use ICU88 to accomplish linking and locating for an MMX 88-based device, you must respond to certain prompts in certain ways. This section explains the special actions that you must perform. One of these is that, when you are prompted for the data concerning your tasks, you must enter certain tasks in a particular order. The following excerpts from an ICU88 dialogue illustrate this and the other requirements:

```
FSM: NO

          [other prompts and responses here]

TASK NAME:      CQDRVR
     ENTRY POINT:    CQDRVR
     STK LENGTH:     200
     PRIORITY:       131
     DFLT EXCHG:
     TASK DESCRIPTOR NAME:     [your choice]
     8087 NDP: [yes/no]

TASK NAME:      CQINTM
     ENTRY POINT:    CQINTM
     STK LENGTH:     200
     PRIORITY:                 [depends upon the interrupt level used
     DFLT EXCHG:                                              by MMX 88]
     TASK DESCRIPTOR NAME:     [your choice]
     8087 NDP: [yes/no]

TASK NAME:      RQFLMT
     ENTRY POINT:    RQFLMT
     STK LENGTH:     160
     PRIORITY:       131
     DFLT EXCHG:
     TASK DESCRIPTOR NAME:     [your choice]
     8087 NDP: [yes/no]
```

```
TASK NAME:      RQFSAT
     ENTRY POINT:   RQFSAT
     STK LENGTH:    160
     PRIORITY:      131
     DFLT EXCHG:    [your choice]
     TASK DESCRIPTOR NAME:    [your choice]
     8087 NDP: [yes/no]

TASK NAME:      RQFSRT
     ENTRY POINT:   RQFSRT
     STK LENGTH:    160
     PRIORITY:      131
     DFLT EXCHG:    [your choice]
     TASK DESCRIPTOR NAME:    [your choice]
     8087 NDP: [yes/no]

          [user tasks here]

EXCHANGE:       RQFSAX
     SCOPE:     EXTERNAL
     INTERRUPT LEVEL:    NONE

EXCHANGE:       RQFSRX
     SCOPE:     EXTERNAL
     INTERRUPT LEVEL:    NONE

EXCHANGE:       RQFLMX
     SCOPE:     EXTERNAL
     INTERRUPT LEVEL:    NONE

EXCHANGE:       CQMXIX
     SCOPE:     EXTERNAL
     INTERRUPT LEVEL:    NONE

EXCHANGE:       CQMXRX
     SCOPE:     EXTERNAL
     INTERRUPT LEVEL:    NONE

          [user exchanges here or mixed with the other exchanges]

          [other prompts and responses here]

LINK:       [user modules]
LINK:       [compiled configuration module -- probably RxCNFG.OBJ]
LINK:       :Fn:RxXMGR.LIB
LINK:       :Fn:RxDRVR.LIB
LINK:       :Fn:RxPMM.LIB
LINK:       :Fn:RxUTIL.LIB
LINK:       :Fn:Rx????.LIB
```

where:

x — A decimal digit that specifies the addressing mode for the application being linked and, if the addressing mode is megabyte, the compiler size control used, as follows:

- 2 — Non-megabyte addressing.
- 3 — Megabyte addressing and COMPACT.
- 5 — Megabyte addressing and LARGE.

???? — An identifier for the class of the device, as follows:

- 957B — You are going to use the iSBC 957B package to execute the module on the device using the parallel interface.
- INTR — You are going to use some other execution vehicle instead.

If you do not specify the NOTYPE switch, TYPE MISMATCH warning messages will appear, but you can ignore them.

Note that the Partitioned Memory Manager can be configured into an application independently of the rest of MMX 88. To accomplish this, do the above, except omit the definitions of the CQDRVR and CQINTM tasks, the CQMXIX, and CQMXRX exchanges, and the RxXMGR, RxDRVR, RxUTIL, and Rx???? links.

Linking and Locating for MMX 86

The linking and locating process for MMX 86 is done in the same way as any other iRMX 86 application, except that certain extra things have to be done to accommodate the MMX 86 software. This section assumes that you are familiar with the iRMX 86 configuration process, so that we can focus on the aspects of configuration that are peculiar to configuring an MMX 86 application. We will discuss linking and locating four kinds of modules: root job configuration file; the iRMX 86 Nucleus; the user configuration file and the MMX 86 job; and user code files.

ROOT JOB CONFIGURATION FILE. The root job configuration file (for which the supplied default version is called R4ROOT.A86) must have a %JOB macro for the MMX 86 job, and this macro must be the first one, other than macros for iRMX 86 system jobs, in the file. This ensures that the MMX 86 job will be the first user job (from the standpoint of the iRMX 86 software) to be initialized. The %JOB macro for the MMX 86 job is as follows:

```
%JOB(OBJ$DIR$SIZE,
     MIN$POOL$SIZE, MAX$POOL$SIZE,
     MAX$NBR$OBJ, MAX$NBR$TASKS,
     MAX$JOB$PRIORITY,
     EXCP$HNDLR$ADDR, EXCP$HNDLR$MODE,
     JOB$FLAGS,
     INIT$TASK$PRIORITY, INIT$TASK$START,
     DATA$SEG,
     INIT$TASK$STACK, INIT$TASK$STACK$SIZE,
     INIT$TASK$FLAGS)
```

where:

OBJDIRSIZE is the maximum number of objects that can be cataloged in the MMX 86 job object directory. MMX 86 does not catalog objects, so this value is 0.

MIN$POOL$SIZE is the minimum allowable size of the memory pool of the MMX 86 job. This value is the initial size of the pool and therefore must be sufficient for the start-up phase of the MMX 86 job. If memory space is at a premium, this value can be calculated from the number of local ports, the number of devices in the system, and the stack requirements of the tasks in the MMX 86 job. A recommended value is 0FFFFH.

MAX$POOL$SIZE is the maximum allowable size of the memory pool of the MMX 86 job. It must at least as large as MIN$POOL$SIZE. Because all messages received at MMX 86 ports on the device being configured are allocated from the MMX 86 job, this value is at least the minimum initial pool size plus the space necessary to hold a maximum load of concurrent incoming messages. A recommended value is 0FFFFH.

MAXNBROBJ is the maximum number of objects that can exist concurrently in the MMX 86 job. It can be calculated from the number of MMX 86 tasks, the number of local ports, the maximum number of incoming messages that can exist concurrently, and the number of other objects that the MMX 86 job requires. A recommended value is 0FFFFH.

MAXNBRTASKS is the number of tasks in the MMX 86 job. Its value is 3.

MAXJOBPRIORITY is the maximum allowable priority for tasks in the MMX 86 job. This priority must be greater than or equal to the interrupt priority being used by the signalling task in the MMX 86 job. A recommended value is 0.

EXCP$HNDLR$ADDR is the entry point for the MMX 86 job's exception handler. Because MMX 86 passes all exceptions back to the calling function, this value is 0:0.

EXCP$HNDLR$MODE is the exception handler mode. For the MMX 86 job, this mode is 0.

JOB$FLAGS defines the characteristics of the MMX 86 job. During system debugging, parameter validation should be used, so this value should be 0 until the system is debugged. At that time, this value can become 2.

INIT$TASK$PRIORITY is the priority of the initialization task and should be set to the priority of the message-handling function in the MMX 86 job. As a rule, this priority should be the highest among all non-interrupt tasks in the system.

INIT$TASK$START is the start address of the initialization task in the MMX 86 job. This address (which can be found on the LOCATE map of the MMX 86 job) is at the beginning of the MMX 86 job's code area, because the first byte there (normally in an area containing constants) contains a jump instruction to the actual start address.

DATA$SEG is the address of the data segment of the MMX 86 job. Because the MMX 86 job sets its own data segment dynamically, set this value to 0.

INIT$TASK$STACK is the address of the stack segment for the initialization task of the MMX 86 job. Because the iRMX 86 Operating System allocates this stack dynamically, set this value to 0:0.

INIT$TASK$STACK$SIZE is the size of the MMX 86 job's stack. While debugging, set this value to 180H. After debugging, you will have a better idea of the optimum value for this parameter.

INIT$TASK$FLAGS defines properties of the MMX 86 job's initialization task. Because this task does not use floating point instructions, this value can be set to 0.

In the root job configuration file (R4ROOT.A86), you must also use %SAB macros to delineate areas of memory that are to be used for the following:

- Code and data space for the MMX 86 job
- Local memory pool areas for the PMM to manage
- Request queues
- Any other space that is dedicated to other devices in the system

After the R4ROOT.A86 configuration file is complete, SUBMIT the file named CROOT.CSD. This will assemble the configuration module, producing a located file named R4ROOT.

NUCLEUS. To produce a located iRMX 86 Nucleus, you must specify two kinds of information:

- The Nucleus system calls that are used by the jobs that will run on this device.
- A description of this device.

Information of the first type is contained in a file called R4NUCL.A86, while information of the second type is contained in a file called NDEVCF.A86. NDEVCF.A86 is found on the iRMX 86 Nucleus diskette and R4NUCL.A86 is found on the iMMX 800 diskette. Copy these files, as well as a file named NUCLUS.CSD, to another diskette, maintaining the same names except that R4NUCL.A86 should be renamed NTABLE.A86. Then edit the copied NTABLE.A86 and NDEVCF.A86 files as described in the iRMX 86 CONFIGURATION GUIDE. After that is done, SUBMIT the file NUCLUC.CSD. This produces a located Nucleus.

THE USER CONFIGURATION FILE AND THE MMX 86 JOB. After you have compiled your configuration module (we will assume that the compiled module has the name R4CNFG.OBJ), you must link it to several MMX 86 modules. The link statement that accomplishes this is:

```
LINK86    R4DRVR.LIB(MBEGIN), &
          R4CNFG.OBJ,         &
          R4DRVR.LIB,         &
          R4XMGR.LIB,         &
          R4????.LIB,         &
          R4PMM.LIB,          &
          R4UTIL.LIB          &
          RPIFC.LIB           &
TO R4CNFG.LNK
```

where:

- R4DRVR.LIB, R4XMGR.LIB, R4????.LIB, R4PMM.LIB, and R4UTIL.LIB are included on the iMMX 800 product diskettes
- ???? defines the device type for which this configuration is being performed. Possible values are 957P, for any board using the iSBC 957B monitor with the parallel port, and INTR, for any other board.
- RPIFC.LIB is an iRMX 86 library for the PL/M-86 COMPACT model of segmentation.

If you do not specify the NOTYPE switch, TYPE MISMATCH warning messages will appear, but you can ignore them.

After linking, use the LOC86 command with the NOINITCODE control. The address specified in the INIT$TASK$START field of the first %JOB macro of the root job configuration file and the address specified when locating the R4NFG.LNK module must be the same. The NOINITCODE control prevents the locator from inserting several commands, beginning at location 200H.

USER CODE. Two interface libraries are available to use when you link your compiled code to MMX 86. They are R4CINF.LIB and R4LINF.LIB, for the COMPACT AND LARGE models of segmentation, respectively.

Before you compile your code, however, you might want to INCLUDE R4SELC.LIT or R4COM.LIT (both discussed earlier) or R4XINF.EXT in your source code. R4XINF.EXT contains external declarations of the MMX 86 system calls.

HARDWARE CONFIGURATION

If your system uses interrupts, you must jumper your hardware to provide for interrupt reception and generation. The required changes are described in the following paragraphs.

iSBC 544 DEVICE INTERRUPT GENERATION

The iSBC 544 device generates Multibus interrupts by means of its SOD output. Jumper the SOD output (post 80) to one of the Multibus interrupt lines INT0/-INT7/ (posts 82-89, respectively).

iSBC 544 DEVICE INTERRUPT RECEPTION

The iSBC 544 device is interrupted by means of its wake-up byte. Jumper the wake-up byte RST 5.5 (post 81) to one of the Multibus interrupt lines INT0/-INT7/ (posts 82-89, respectively).

iSBC 569 DEVICE INTERRUPT GENERATION

The iSBC 569 device generates Multibus interrupts by means of its programmable reset latch A7. Connect BUSINT to one of the Multibus interrupt lines INT0/-INT7/ (connection 138-139, 135-136, 132-133, 129-130, 126-127, 123-124, 120-121, or 117-118, respectively).

iSBC 569 DEVICE INTERRUPT RECEPTION

The iSBC 569 device is interrupted by means of its wake-up byte. No jumpering is required to support this arrangement, as it is hard-wired into the device.

iSBC 80/24 DEVICE INTERRUPT GENERATION

The iSBC 80/24 device generates Multibus interrupts through one of its I/O ports. To enable the iMMX 800 interface procedures to utilize I/O port E6, bit 7 to generate interrupts, disconnect jumper connection 35-50 and install connection 39-50. To select the correct interrupt line, connect INT0/-INT7/ (post 168, 169, 171, 173, 170, 172, 174, or 175) to INTROUT (post 166).

iSBC 80/24 DEVICE INTERRUPT RECEPTION

To receive interrupts from the Multibus interface, connect the Multibus interrupt line INT0/-INT7/ (post 108, 107, 106, 105, 109, 110, 111, or 112, respectively) to the local interrupt line IR0-IR6 of the on-board 8259A P.I.C. (post 102, 101, 100, 99, 98, 97, or 96, respectively) or to the local interrupt line on the 8259A P.I.C. (any post in the range 83-86).

iSBC 80/30 DEVICE INTERRUPT GENERATION

The iSBC 80/30 board generates Multibus interrupts through one of its I/O ports. To support this, jumper post 1 (INTROUT) to I/O post 9, and jumper one of INT0/-INT7/ (posts 181, 182, 183, 184, 187, 188, 189, or 190, respectively) to post 185 (INTROUT).

iSBC 80/30 DEVICE INTERRUPT RECEPTION

To receive interrupts from the Multibus interface, connect the Multibus interrupt line INT0/-INT7/ (post 148, 147, 152, 151, 150, 149, 146, or 136, respectively) to the local interrupt line IR0-IR7 of the on-board 8259A P.I.C. (post 133, 132, 131, 130, 129, 128, 127, or 126, respectively) or to the RST 5.5 or 6.5 interrupt line on the 8085 (post 140 or 139, respectively).

iSBC 86/05 DEVICE INTERRUPT GENERATION

The iSBC 86/05 device generates Multibus interrupts through port C of the 8255 P.P.I. Connect BUS INTR OUT (post 31) to one of the bits PC0-PC7 (post 50, 51, 52, 53, 45, 42, 43, or 44, respectively) of port C. Also connect the Multibus interrupt level INT0/-INT7 (post 199, 201, 195, 197, 194, 196, 200, or 198, respectively) to BUS INTR OUT (post 193).

iSBC 86/05 DEVICE INTERRUPT RECEPTION

For the iSBC 86/05 device to receive interrupts from the Multibus interface, one of the Multibus interrupt lines INT0/-INT7 (post 144, 136, 142, 143, 147, 146, 149, or 148, respectively) must be connected to one of the 8259A P.I.C. input lines IR0-IR7 (post 132, 133, 124, 131, 130, 145, 128, or 127, respectively).

iSBC 86/12A DEVICE INTERRUPT GENERATION

The iSBC 86/12A device generates Multibus interrupts through port C of the 8255 P.P.I. Connect BUS INTR OUT (post 9) to one of the bits PC0-PC7 (post 26, 28, 30, 32, 15, 19, 17, or 13, respectively) of port C. Also connect the Multibus interrupt level INT0/-INT7/ (post 141, 140, 139, 138, 137, 136, 135, or 134, respectively) to BUS INTR OUT (post 142).

iSBC 86/12A DEVICE INTERRUPT RECEPTION

For the iSBC 86/12A device to receive interrupts from the Multibus interface, one of the Multibus interrupt lines INT0/-INT7/ (post 73, 72, 71, 70, 69, 68, 66, or 65, respectively) must be connected to one of the 8259A P.I.C. input lines IR0-IR7 (post 81, 80, 79, 78, 77, 76, 75, 74, respectively).

iSBC 86/14 AND iSBC 86/30 DEVICE INTERRUPT GENERATION

The iSBC 86/14 and iSBC 86/30 devices generate interrupts through port C of the 8255 P.P.I. Connect BUS INTR OUT 1 (post 24) to one of the bits PC0-PC7 (post 57, 58, 59, 60, 56, 55, 54, or 53, respectively) of port C. Also, connect the Multibus interrupt level INT0/-INT7/ (post 253, 252, 251, 250, 249, 248, 247, or 246, respectively) to the buffered BUS INTR OUT 1 (post 244).

iSBC 86/14 AND iSBC 86/30 DEVICE INTERRUPT RECEPTION

For the iSBC 86/14 or iSBC 86/30 device to receive interrupts from the Multibus interface, one of the Multibus interrupt lines INT0/-INT7/ (post 160, 149, 148, 159, 162, 151, 150, or 161, respectively) must be connected to one of the 8259A P.I.C. interrupt lines IR0/-IR7/ (post 165, 164, 147, 136, 157, 152, 155, or 134, respectively).

iSBC 88/25 DEVICE INTERRUPT GENERATION

The iSBC 88/25 device generates Multibus interrupts through port C of the 3255 P.P.I. Connect BUS INTR OUT (post 29) to one of the bits PC0-PC7 (post 47, 48, 49, 50, 42, 39, 40, or 41, respectively) of port C. Also connect the Multibus interrupt level INT0/-INT7/ (post 178, 180, 174, 176, 173, 175, 179, or 177, respectively) to BUS INTR OUT (post 172).

iSBC 88/25 DEVICE INTERRUPT RECEPTION

For the iSBC 88/25 device to receive interrupts from the Multibus interface, one of the Multibus interrupt lines INT0/-INT7/ (post 132, 124, 130, 131, 135, 134, 137, or 136, respectively) must be connected to one of the 8259A P.I.C. input lines IR0-IR7 (post 116, 117, 133, 119, 120, 113, 122, or 121, respectively).

iSBC 88/40 DEVICE INTERRUPT GENERATION

The iSBC 88/40 device generates Multibus interrupts through port C of the 8255 P.P.I. Connect AUX INT (post 49) to one of the bits PC0-PC7 (post 60, 62, 64, 66, 58, 56, 54, or 52, respectively) of port C. Also connect the Multibus interrupt level INT0/-INT7/ (post 253, 252, 250, 249, 248, 246, 244, or 245, respectively) to AUX INT (post 247).

The iSBC 88/40 device generates memory-mapped interrupts by writing the value 01H into the first byte of dual-port memory on the iSBC 88/40 device. The interrupted device must clear the interrupt and does so by writing the value 00H via the system bus to that same byte. Connect INTERRUPT MASTER (post 251) to a Multibus interrupt line INT0/-INT7/ (post 253, 252, 250, 249, 248, 246, 244, or 245, respectively).

iSBC 88/40 DEVICE INTERRUPT RECEPTION

The iSBC 88/40 device receives interrupts when data (01H) is written from the Multibus interface into the first location in dual-port memory. The interrupt is cleared when data (00H) is written from the on-board CPU to the same location. One of the 8259A P.I.C. input lines IR0-IR7 (post 161, 157, 153, 149, 145, 141, 137, or 133, respectively) must be connected to INTERRUPT MASTER (post 251).

iSBC 88/45 DEVICE INTERRUPT GENERATION

The iSBC 88/45 device generates memory-mapped interrupts by writing the value 01H into the first byte of dual-port memory on the iSBC 88/45 device. The interrupted device must clear the interrupt and does so by writing the value 00H via the system bus to that same byte. Connect BUS FLAG INT (post 86) to a Multibus interrupt line INT0/-INT7/ (post 179, 180, 181, 182, 183, 184, 185, or 186, respectively).

iSBC 88/45 DEVICE INTERRUPT RECEPTION

The iSBC 88/45 device receives interrupts when data (01H) is written from the Multibus interface into the first location in dual-port memory. Interrupts are cleared when data (00H) is written to the same location. The FLAG INTERRUPT line is hard-wired to the 8259A P.I.C. interrupt line IR4, so no jumpering is required.

CHAPTER 8. PERFORMANCE CONSIDERATIONS

The purpose of this chapter is to suggest some basic principles that you can use to improve the performance of your iMMX 800-based application system. Because the iMMX 800 software operates in a multi-board environment, you have considerable freedom and can apply these principles in various combinations. To find the approach that works best for you, you will probably have to do some experimenting.

Remember that the goal is to improve overall system performance. Perhaps you will have to sacrifice efficiency in one part of your system in order to attain even greater efficiency in another part. Try to approach the task of maximizing your system's efficiency thoughtfully and with an open mind.

AVOID UNNECESSARY TRAFFIC ON THE MULTIBUS INTERFACE

Indiscriminant use of the Multibus interface can seriously degrade a system. You should be mindful of the ways in which the Multibus interface is being used in your application system. For example, suppose that a processor on one board is executing instructions that reside on another board. In this case, the required instruction fetches can place enormous demands on the system bus. To avoid this problem, try to design your system so that each processor executes only instructions that are local to that processor. If that is not possible, locate your code so that each processor uses local memory for its most-used routines or for routines that are most in need of speedy execution.

By designing your system with such matters in mind, you can make large strides toward the goal of optimum performance. Perhaps you can even achieve the ultimate in reduced Multibus traffic: a system that uses the Multibus interface only for message transfers.

MINIMIZE THE NUMBER OF TIMES THAT MESSAGES MUST BE COPIED

Whenever you use MMX 80 or MMX 88, you have, in theory at least, the option of building a message in memory that is accessible by the destination device. Then, when you invoke the CQXFER service, you can specify that the local iMMX 800 software should not make a copy when transferring the message. By doing this whenever possible, you can save processor time by eliminating unnecessary message copying operations.

DISTRIBUTE THE WORKLOAD AMONG THE BOARDS IN YOUR SYSTEM

A key to performance is the extent to which processors are kept busy. At the board level, the iRMX operating system assures this by providing logical concurrency. This means that, as long as at least one task on the board is in the ready state, some task will be executing.

In systems that are linked together by means of the iMMX 800 software, it is possible to achieve actual concurrency. This means that tasks on different boards can be executing at the same time.

By judiciously distributing the workload among the boards in your system, you can exploit the concurrency principle and work toward the goal of keeping all of your processors busy all of the time.

MINIMIZE THE NUMBER OF MESSAGE TRANSFERS BY USING LARGE MESSAGES

Because the overhead (aside from making copies) of a message transfer is the same for large messages as it is for small messages, one way of improving the performance of your system is to use a few large messages instead of many small messages.

EXPERIMENT WITH VARIOUS INTERRUPT MECHANISMS AND POLLING PERIODS

As you perform configuration, you must specify for each board the kind of interrupt mechanism that is to be used to interrupt the processor on that board. The possibilities include Multibus interrupts, I/O-mapped interrupts, and memory-mapped interrupts. You must also specify the length of the polling period for each board. By trying different interrupt mechanisms and by varying the length of the polling periods, you can experimentally find the optimum combination for your system.

EXPERIMENT WITH VARIOUS HARDWARE AND SOFTWARE CONFIGURATIONS

The iMMX 800 software is very flexible, so you can use it with many combinations of iSBC boards and iRMX operating systems. Because different boards have different capabilities and strengths, and because the same is true of operating systems, the optimum combination of boards and operating systems might not be obvious. Even though it seems drastic to do so, you might be able to improve the performance of your application system by making a change at the hardware or operating system level.

APPENDIX A. MULTIBUS® INTERPROCESSOR PROTOCOL (MIP)

WHAT IS MIP?

The Multibus Interprocessor Protocol (MIP) defines a set of mechanisms and protocols that provide a reliable and efficient exchange of data among tasks executing on various single-board computers connected to a common Multibus system bus. See Figure A-1 for an example of how MIP facilities are used in a Multibus configuration of single-board computers.

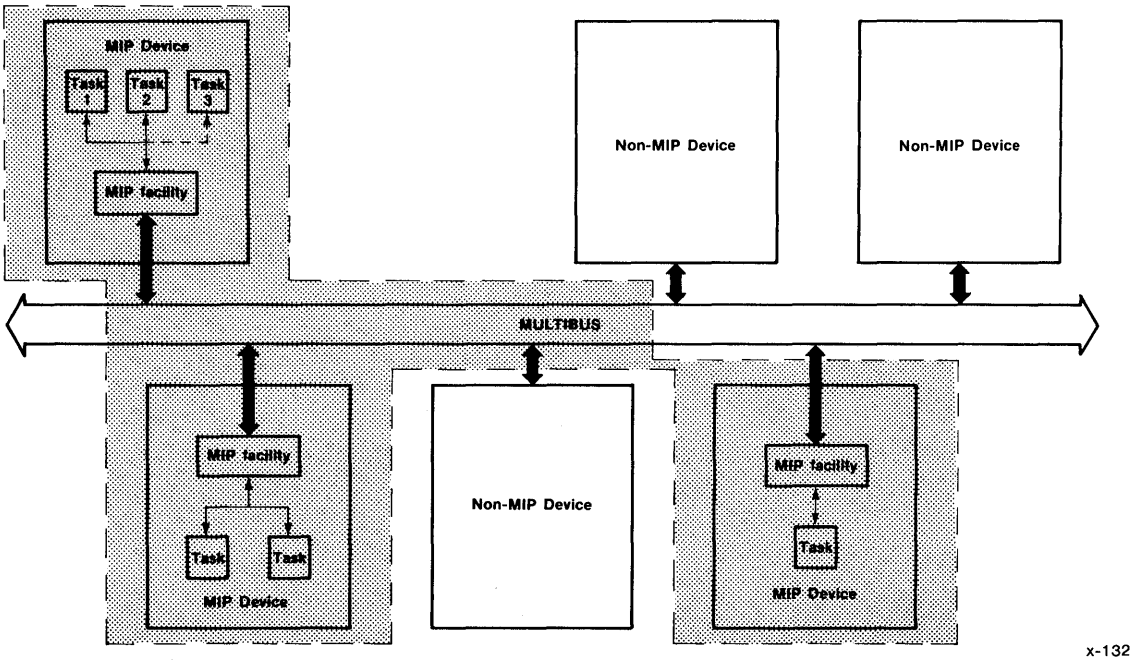


Figure A-1. MIP System Example

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

1. Tasks may be running on different kinds of processors.
2. Tasks may be running under different kinds of operating systems.
3. Different boards have different Multibus signalling mechanisms.
4. Not all boards share the same memory space.

5. Boards sometimes share memory but reference it by different addresses.

6. 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 (iSBC's) and intelligent device controllers. MIP can be used by any device on which a MIP implementation can be programmed. The MIP design does not limit the kinds of processors or operating systems that can execute MIP services.

MIP can be used by the MCS-85 or the iAPX-86 families of processors. The iMMX 800 Message Exchange, which is a MIP implementation, can run under the iRMX 80, iRMX 86, or iRMX 88 Operating Systems. You can also implement MIP facilities to run on other processors or under other operating systems.

IMPLEMENTING MIP

When using this specification as a guide for implementing MIP, be aware that it deals only with global concerns. Implementation details such as initialization or memory management are not addressed. You can add features that provide your implementation with a better interface with its local environment; for example, the processor, the operating system, or application tasks.

The MIP 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, provided that 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, its function in the model is merely descriptive.

- If an algorithm requires the cooperation of another communicating MIP facility, the algorithm must conform to the model.

THE MIP MODEL

The MIP specification defines several components that are required in all MIP implementations. This section describes these 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.

A MIP facility is an implementation of MIP. MIP facilities support communication among tasks executing on different iSBC devices that are attached to a common Multibus system bus. The set of intercommunicating tasks, along with associated iSBC devices, operating systems, and MIP facilities, is called a MIP system. 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 iSBC device in a MIP system. Each device has a device-ID, which is a number ranging from zero to the number of devices communicating in one MIP system (less 1).

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

Communications are delivered to tasks at system ports. A system port is a logical delivery mechanism that enables delivery in "first-in, first-out" (FIFO) order. In the MIP model, a system port is represented as a queue. In some operating systems, system ports are called "mailboxes" or "exchanges".

Each system port on a given device is identified by a port id, which is a number in the range zero to the number of ports (less 1) on the device. To provide system-wide addressability, a system port is also identified by a socket, which is an ordered pair (d,p), where "d" is the device-ID and "p" is the system-port-ID. Refer to Figure A-2 for a typical system port configuration.

In Figure A-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 identify system ports symbolically, so tasks that identify ports by their function-names are independent of changes in configuration.

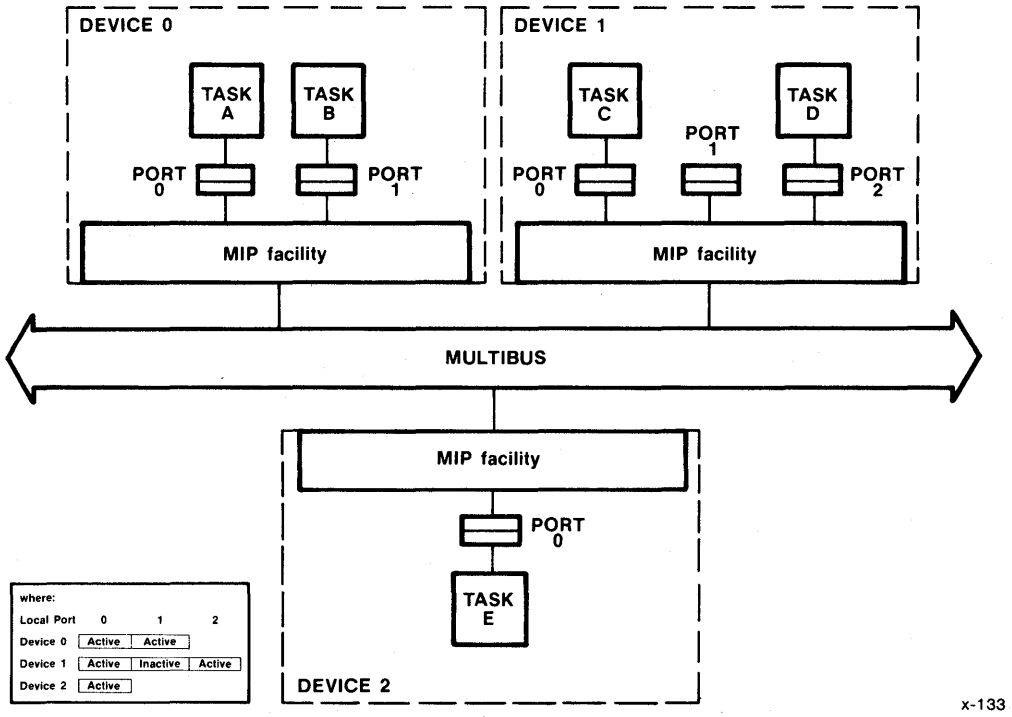


Figure A-2. System Port Configuration Example

THREE-LEVEL INTERFACE 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 the MIP facilities on different devices interact with each other.

3. The logical level, which associates the virtual level with the physical level.

At the physical level, a MIP implementation must adhere to the specified functions, structures, and constants. Any implementation that deviates from this requirement is not compatible with the MIP architecture, and might not be able to communicate with other MIP facilities.

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

The virtual level of the model simply suggests one way for tasks to view the MIP system. Any other viewpoint will work as well, provided the information passed through the virtual level 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 A-3 illustrates the three-level structure. Refer to this figure during the following discussion of all three levels.

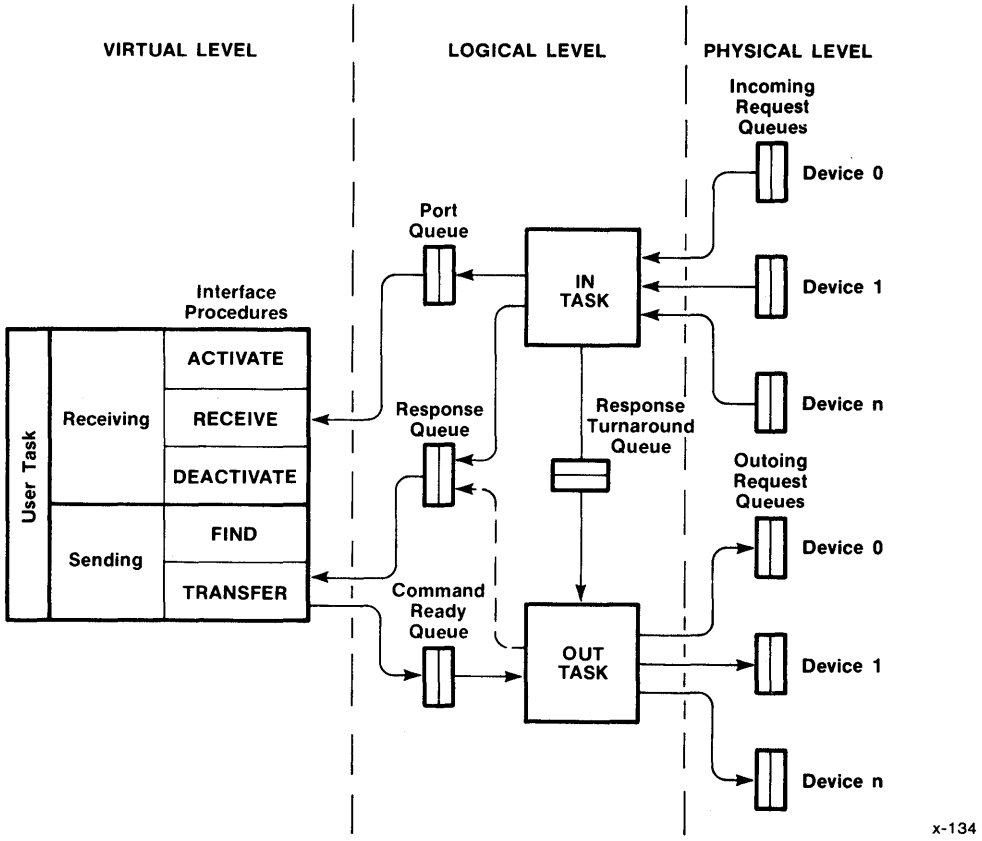


Figure A-3. MIP Model Data Flow Structure

PHYSICAL LEVEL

The physical communication mechanism between devices is a fixed size, one-direction, 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 A-4. The RQD keeps track of the "give" and "take" locations as well as other information about the queue.

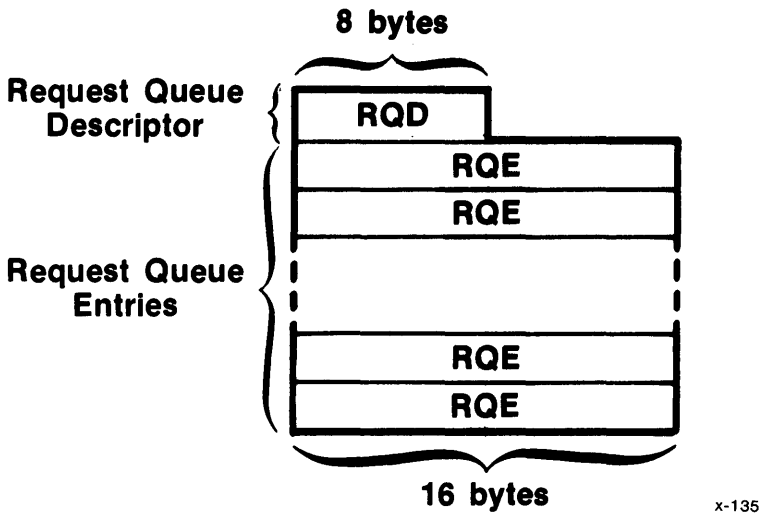


Figure A-4. Request Queue Format

Each Request Queue contains at least two RQE's, 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 A-5 for a conceptual diagram of a channel.

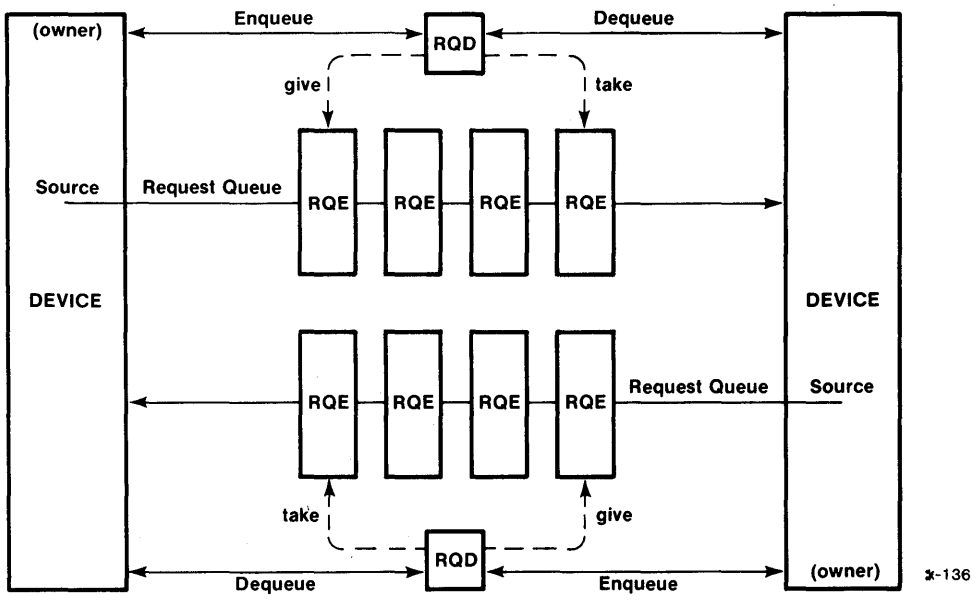


Figure A-5. Conceptual Structure of a Channel

LOGICAL LEVEL

The logical level of the MIP model uses Request Queues to transfer requests between source and destination MIP facilities. A request is either a command or a response. A command is an order sent from a source MIP facility to a destination facility. A response is returned from the destination facility to the source facility and indicates the results of an attempt to deliver a command. The Request Queues carry these requests and their associated parameters between MIP facilities.

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

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

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

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

VIRTUAL LEVEL

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

For sending buffers:

1. FIND--locates a port, given its function-name.

2. TRANSFER--initiates transfer of a buffer to a given port by placing a command in the destination device's Command Ready Queue. TRANSFER then waits for a response before allowing the sending task to continue.

For receiving buffers:

3. ACTIVATE--attaches a task to a port and enables reception of messages at that port.

4. RECEIVE--completes transfer of a buffer by taking a command from the task's Port Queue.

5. DEACTIVATE--disconnects a task from its port and terminates reception of commands at that port.

MEMORY MANAGEMENT

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

- The entire segment is continuously addressable within the address space of the device.

- The device cannot address any of the segment.

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

Figure A-6 presents a hypothetical memory configuration and shows how the address space is partitioned.

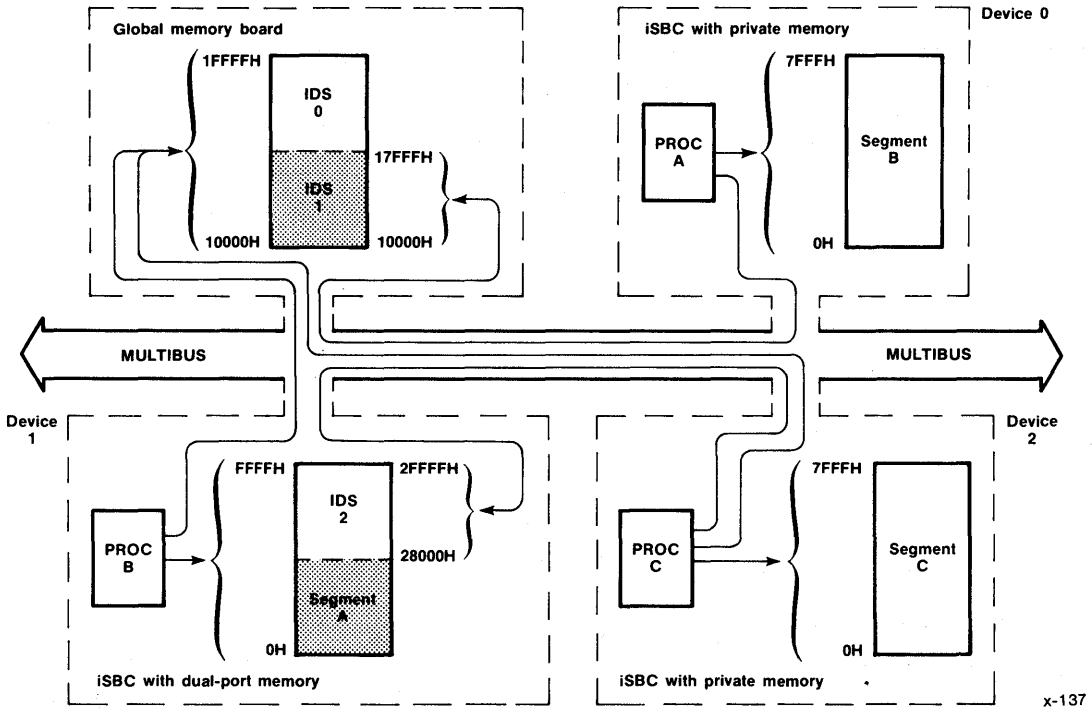


Figure A-6. Example of Inter-Device Memory Segments

In Figure A-6, processors A and C can communicate through IDS 1. Processors B and C can communicate through IDS's 0, 1, and 2. However, IDS 3 is a segment of dual-ported memory and is accessed by processor B by using a different range of addresses than those used by processor C. Memory segments A, B, and C cannot be used for inter-device communication.

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

Table A-1. System Inter-Device Segment Table

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

x-138

The MIP model contains special features for handling the "alias addressing" situation posed by dual-port memory. Dual-port memory may be addressed differently from the Multibus system bus than from its local processor.

The only case of a shared memory address in a MIP system is the buffer pointer in the RQE. This pointer is stored in a special format, called an IDS pointer, which 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. User tasks therefore need not be concerned with these addressing considerations.

BUFFER MOVEMENT

Generally, buffers are not physically moved from one memory location to another unless moving them is necessary. Instead, buffers are referenced by descriptors in the RQE's. 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.

SIGNALLING

MIP uses a signalling mechanism for efficient utilization of the inter-device request queues. The mechanism employed is a software handshake that uses flags in the signal bytes of the RQD's. 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$TASK does not poll outgoing request queues that are known to be full. When the IN$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.

IN$TASK and OUT$TASK poll their signal flags to detect changes in the states of their queues. Interrupts may be implemented to effect greater efficiency in polling the signals.

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. Sets flags to prevent any further activity on the channel.
2. Discards any responses destined for the dead device.
3. 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.

MIP FUNCTIONAL SPECIFICATION

The databases and algorithms described in the rest of this appendix define the attributes of the Multibus Interprocessor Protocol. Logically, the functional specification consists of the three levels described previously in this appendix.

PROCEDURAL SPECIFICATION

DATA TYPES

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

BYTE: Standard 8-bit variable.

WORD: Two-BYTE variable.

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

STATE: BYTE variable restricted to state constants.

POINTER: Device-dependent address reference.

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

PROCESSOR-DEPENDENT SUBROUTINES

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

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

PTR$ADD

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

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

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

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

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

 RETURN NEW$PTR;

END PTR$ADD;
```

CONVERT$LOCAL$ADR

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

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

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

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

END CONVERT$LOCAL$ADR;
```

CONVERT$SYSTEM$ADR

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

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

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

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

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

END CONVERT$SYSTEM$ADR;
```

TIME$WAIT

A destination device is assumed to be dead if it does not respond to a command within a reasonable period of time. How you detect a timeout depends upon the local processor's timing features

```
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;
```

PHYSICAL LEVEL SPECIFICATION

This level defines the structure and function of Request Queues. To ensure compatibility with the MIP architecture, an application must not deviate from the functions, structures, and constants presented in the Request Queue element descriptions provided under this heading.

REQUEST QUEUE DESCRIPTOR

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

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

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

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

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

RQE$LENGTH defines the number of bytes in a Request Queue Element (RQE). The number of elements is 2 to the power of RQE$LENGTH. For all queues shared between MIP facilities, RQE$LENGTH is 4 (that is, each entry is 16 bytes long).

GIVE$INDEX identifies the Request Queue Element available for enqueuing data.

TAKE$INDEX identifies the Request Queue Element available for dequeuing data.

GIVE$STATE and TAKE$STATE contain the booleans defined as follows (unused bits are reserved for future expansion):

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

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

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

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

For making comparisons between GIVE$INDEX and TAKE$INDEX, the following declaration is required:

```
DECLARE POINTER$MASK LITERALLY '7FH';
```

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:

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

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

DESTDEVID is the device identifier part of the destination socket.

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

SRCDEVID identifies the device from which a request is issued.

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

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

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

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

RSVRD is undefined space reserved for future expansion.

QUEUE PROCEDURE RETURNS

The following constants are used to return the results of procedures associated with the Request Queues:

```
DECLARE READY                          LITERALLY     '00H',
        FULL                           LITERALLY    '0FFH',
        EMPTY                          LITERALLY    '0FFH',
        FIRST$GIVE                     LITERALLY     '20H',
        FIRST$TAKE                     LITERALLY     '20H',
        HALTED                         LITERALLY     '40H',
        GIVE$DISABLED                  LITERALLY     '10H',
        TAKE$DISABLED                  LITERALLY     '10H';
```

INIT$REQUEST$QUEUE

This procedure enters a Request Queue Descriptor in memory, thereby initializing a Request Queue.

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

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

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

END INIT$REQUEST$QUEUE;
```

TERM$REQUEST$QUEUE

This procedure sets the Request Queue flags to prevent subsequent activity on a channel.

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

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

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

END TERM$REQUEST$QUEUE;
```

QUEUE$GIVE$STATUS

This procedure returns the status of a Request Queue without affecting the queue.

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

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

DECLARE STATUS              BYTE;             /* Output. */

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

END QUEUE$GIVE$STATUS;
```

REQUEST$GIVE$POINTER

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

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

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

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

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

END REQUEST$GIVE$POINTER;
```

RELEASE$GIVE$POINTER

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

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

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

DECLARE STATUS            BYTE,             /* Output. */
        GIVE$INDEX        BYTE;

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

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

END RELEASE$GIVE$POINTER;
```

REQUEST$TAKE$POINTER

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

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

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

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

  IF (RQD.GIVE$STATE AND GIVE$HALT) = GIVE$HALT
    THEN DO;
      RQD.TAKE$STATE = TAKE$DISABLED;
      STATUS = HALTED;
      RETURN;
    END /* THEN */;

  IF RQD.GIVE$INDEX = RQD.TAKE$INDEX
    THEN DO;
      STATUS = EMPTY;
      RETURN;
    END /* THEN */;

  STATUS = READY;
  RQE$PTR = SHL(DOUBLE(RQD.TAKE$INDEX AND POINTER$MASK),
                RQD.RQE$LENGTH)
            + 8 + RQD$PTR;
  RETURN;

END REQUEST$TAKE$POINTER;
```

RELEASE$TAKE$POINTER

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

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

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

DECLARE STATUS            BYTE,             /* Output. */
        MSB               BYTE;

  IF (RQD.GIVE$INDEX AND POINTER$MASK) =
     (((RQD.TAKE$INDEX AND POINTER$MASK) + 1) AND
      (RQD.RQ$SIZE - 1))

    THEN /* TAKE$FACTOR bit = GIVE$FACTOR bit. */
      MSB = RQD.GIVE$STATE AND GIVE$FACTOR

    ELSE MSB = 0;

  RQD.TAKE$INDEX = (((RQD.TAKE$INDEX AND POINTER$MASK) + 1) AND
                    (RQD.RQ$SIZE - 1)) OR MSB;

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

END RELEASE$TAKE$POINTER;
```

LOGICAL LEVEL DATABASE

CONFIGURATION CONSTANTS

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

```
DECLARE DEVICES        LITERALLY    'the number of devices in the MIP
                                    system',
        SOCKETS        LITERALLY    'the number of destination ports',
        PORTS          LITERALLY    'the number of local ports',
        HOME$DEVICE    LITERALLY    'the identifier of this device',
        TIME$DELAY     LITERALLY    'maximum time to wait for a
                                    response before a destination
                                    device is considered dead',
        IDS$S          LITERALLY    'the number of entries in the IDS
                                    table',
        RQL$S          LITERALLY    'the number of local response
                                    queues';
```

DESTINATION SOCKET DESCRIPTOR TABLE (DSDT)

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

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

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

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

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

LOCAL PORT TABLE (LPT)

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

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

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

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

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

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

DEVICE TO CHANNEL MAP (DCM)

The DCM table is used to route messages among inter-task and inter-device Request Queues and to manage the flow of messages into and out of the queues. Each MIP facility has one entry in its DCM for every device in the MIP system, including the device on which the MIP facility resides.

The device identifier of a device is its index into the DCM. Each entry in a DCM represents a possible link between the home device and the device associated with that entry. If no such link exists, CHANNEL$STATE contains IDLE.

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

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

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

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

RQDOUTSIZE is the number of entries in this queue.

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

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

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

INTER-DEVICE SEGMENT TABLE (IDST)

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

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

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

RESPONSE QUEUE LIST (RQL)

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

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

LOGICAL LEVEL ALGORITHMS

DYING$CHANNEL

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

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

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

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

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

END DYING$CHANNEL;
```

SERVE$TURNAROUND$QUEUE

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

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

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

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

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

END SERVE$TURNAROUND$QUEUE;
```

SERVE$COMMAND$QUEUE

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

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

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

DECLARE STATUS                    BYTE;          /* Output. */

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

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

END SERVE$COMMAND$QUEUE;
```

OUT$TASK

This algorithm manages activity in the output request queues.

```
OUT$TASK: PROCEDURE;

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

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

/*  Transfer request loop.  */

 DO FOREVER;
  DO  DEVICE$INDEX = 0 TO DEVICES - 1;
   RQD$PTR = DCM(DEVICE$INDEX).RQD$IN$PTR;
   /* Read signal from in-RQD. */
   IF RQD.FULL$SIGNAL = FULL$NO$LONGER
    THEN DO;
     DCM(DEVICE$INDEX).CHANNEL$STATE =
      DCM(DEVICE$INDEX).CHANNEL$STATE OR RQD.FULL$SIGNAL;
     RQD.FULL$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
```

OUT$TASK (continued)

```
            THEN CALL SERVE$COMMAND$QUEUE (DEVICE$INDEX, STATUS);
          END /* THEN */;
        END /* THEN */;
      END /* ELSE */;
    END /* DO */;
  END /* FOREVER */;

END OUT$TASK;
```

RECEIVE$COMMAND

This IN$TASK subroutine 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$LENGTH, /* Copies the whole */
                RQE$MSG$PTR,        /* buffer into the  */
                LOCAL$DATA$PTR);    /* port queue.      */
     CALL RELEASE$GIVE$POINTER
           (LPT(IN$RQE.DEST$PORT$ID).PORT$QUEUE$PTR,
            STATUS);
     IN$RQE.REQUEST = MSG$DELIVERED$COPY;

     /*                             NOTE

         Instead of copying the whole buffer, you may copy
         only IN$RQE.DATA$PTR and IN$RQE.DATA$LENGTH.  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);
```

RECEIVE$COMMAND (continued)

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

     END RECEIVE$COMMAND;
```

RECEIVE$RESPONSE

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

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

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

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

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

END RECEIVE$RESPONSE;
```

IN$TASK

This algorithm manages activity in the incoming Request Queues.

```
     IN$TASK:  PROCEDURE;

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

      DO FOREVER;
       DO  DEVICE$INDEX = 0 TO DEVICES - 1;
        RQD$PTR = DCM(DEVICE$INDEX).RQD$IN$PTR;
        IF RQD.EMPTY$SIGNAL = EMPTY$NO$LONGER
         THEN DO;
          DCM(DEVICE$INDEX).CHANNEL$STATE =
           DCM(DEVICE$INDEX).CHANNEL$STATE OR RQD.EMPTY$SIGNAL;
          RQD.EMPTY$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;
```

IN$TASK (continued)

```
            END /* THEN */;
          END /* THEN */;
        END /* THEN */;
      END /* DO */;
    END /* FOREVER */;

  END IN$TASK;
```

VIRTUAL LEVEL

STATUS CONSTANTS

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

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

FIND$SYSTEM$PORT

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

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

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

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

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

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

END FIND$SYSTEM$PORT;
```

TRANSFER$BUFFER

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

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

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

DECLARE STATUS               BYTE;             /* Output. */

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

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

 CALL CONVERT$LOCAL$ADR (IDS$ID,
                         BUFFER$PTR,
                         RQE.DATA$PTR);
 RQE.DATA$LENGTH      = BUFFER$LENGTH;
 CALL RELEASE$GIVE$POINTER
       (DCM(SOCKET$DEVICE).COM$RDY$QUEUE$PTR,
        CALL$STATUS);

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

 CALL REQUEST$TAKE$POINTER (RQL(RQL$ID).RSP$QUEUE$PTR,
                            RQE$PTR,
                            CALL$STATUS);
 IF CALL$STATUS = EMPTY
```

TRANSFER$BUFFER (continued)

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

ACTIVATE$SYSTEM$PORT

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

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

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

DECLARE STATUS          BYTE;       /* Output. */

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

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

END ACTIVATE$SYSTEM$PORT;
```

DEACTIVATE$SYSTEM$PORT

This function terminates reception of messages at a port.

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

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

DECLARE STATUS        BYTE;     /* Output. */

DECLARE PORT$INDEX    BYTE;

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

END DEACTIVATE$SYSTEM$PORT;
```

RECEIVE$BUFFER

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

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

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

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

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

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

 RETURN;

END RECEIVE$BUFFER;
```

APPENDIX B. COMMUNICATION WITH AN iSBC® 550 ETHERNET* COMMUNICATIONS CONTROLLLER

The MMX 86 and MMX 88 software each allow your application system to communicate with an iSBC 550 Ethernet controller. The purpose of this appendix is to provide you with instructions for building an application system that communicates with an iSBC 550 Ethernet controller. This information falls into the following categories:

- A list of the Intel hardware and software products that you can use to build the application system.
- High-level directions for assembling the hardware.
- Special instructions for writing iRMX 86 and iRMX 88 tasks that can communicate with an iSBC 550 Ethernet Communications Controller.
- How to configure either the iRMX 86 Operating System or the iRMX 88 Executive and either MMX 86 or MMX 88, respectively, for communication with an Ethernet controller.

This appendix is designed to serve primarily as an overview. Although it contains some detailed information, when feasible it refers to other manuals rather than repeating information described elsewhere.

ETHERNET-RELATED INTEL HARDWARE AND SOFTWARE PRODUCTS

Figure B-1 shows the hardware of a system that communicates with the Ethernet network. In the figure and throughout the remainder of this appendix, assume, for simplicity, that the primary hardware elements of this system are an iSBC 86/12A or iSBC 86/30 computer (the host computer), an iSBC 550 Ethernet communications controller, and an ICS 80 system chassis. Note, however, that you can use any iAPX 86- or iAPX 88-based microcomputer as the host computer, and you can use any chassis that incorporates the Multibus interface.

The primary software elements of the system are your application tasks, the iRMX 86 Operating System or iRMX 88 Executive, and the MMX 86 or MMX 88 software. These are shown in Figure B-2.

The hardware and software interact as follows:

- The Multibus interface is the hardware link between the iSBC 550 Ethernet communications controller and the iSBC 86/12A or iSBC 86/30 single board computer.

*Ethernet is a trademark of the Xerox Corporation.

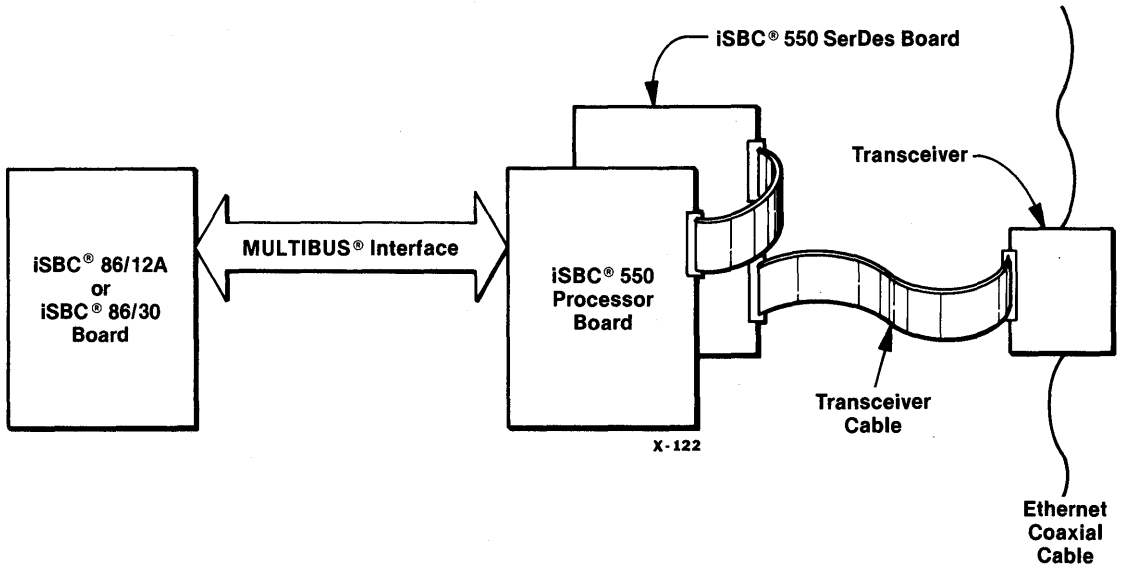


Figure B-1. Hardware for a System Communicating with Ethernet

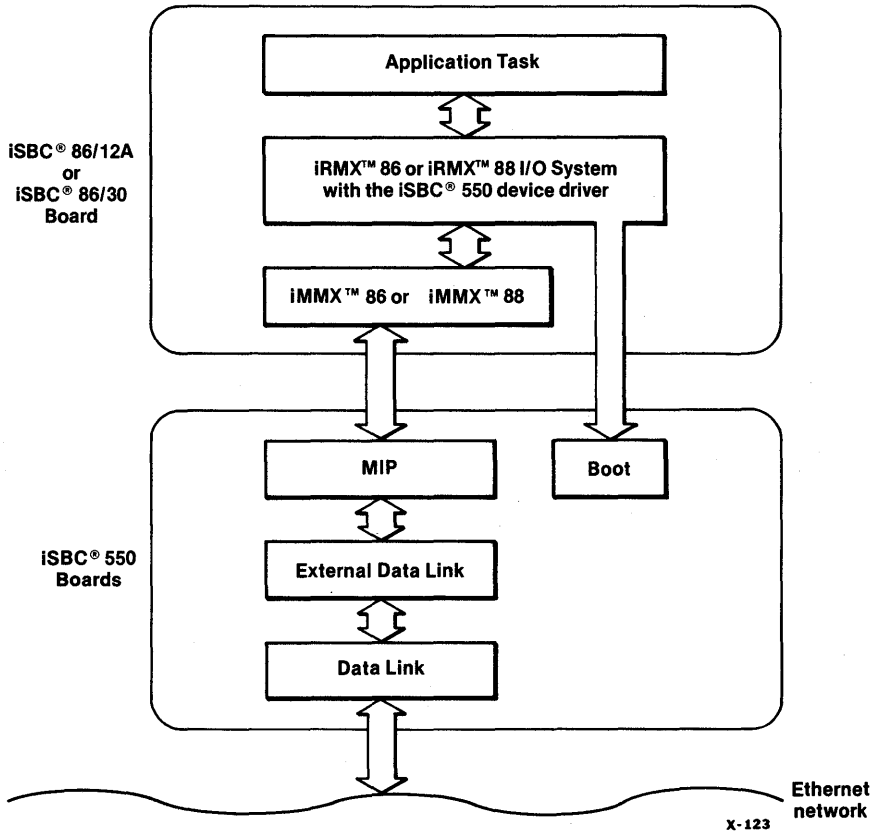


Figure B-2. Software for a System Communicating with Ethernet

- The iRMX 86 Basic or Extended I/O System or the iRMX 88 I/O System is the software interface between the tasks of your application system and the Ethernet network. Tasks communicate through the Ethernet network by:

    1. Explicitly formatting a message in an iRMX 86 segment or iRMX 88 memory block. This message tells the iSBC 550 Ethernet communications controller what actions to perform.

    2. Using the system calls of the iRMX 86 Basic or Extended I/O Systems (your choice) or the iRMX 88 I/O System to read information from or write information to the iSBC 550 Ethernet controller board, which then communicates with the network.

The interaction between the host computer and the communications controller is different than the interaction between a host computer and a non-intelligent controller in several respects:

- All the information sent between the host computer and the communications controller is passed via MMX 86 or MMX 88 software. However, your application tasks do not explicitly invoke MMX 86 or MMX 88 system calls. Instead, your tasks invoke only iRMX 86 or iRMX 88 I/O system calls that, in turn, use MMX 86 or MMX 88, respectively, to pass information.

- Your tasks can pass only information that is formatted for the iSBC 550 controller. The formats for the various iSBC 550 commands (CONNECT, DISCONNECT, ADDMCID, DELETEMCID, TRANSMIT, SUPPLYBUF, READ, and READC) are defined in the ETHERNET COMMUNICATIONS CONTROLLER PROGRAMMER'S REFERENCE MANUAL.

- Your application system must include either the iRMX 86 Operating System and MMX 86 or the iRMX 88 Executive and MMX 88. This appendix provides a sample configuration of each. These samples, which appear in the section entitled "Configuring, Linking, and Locating an I/O System for use with iSBC 550 Controllers," will simplify your configuration process.

PUTTING THE HARDWARE TOGETHER

There are several sources of information about assembling your hardware. The principal sources are:

- iSBC 550 ETHERNET COMMUNICATIONS CONTROLLER HARDWARE REFERENCE MANUAL

- iSBC 86/12A HARDWARE REFERENCE MANUAL or iSBC 86/14 AND iSBC 86/30 SINGLE BOARD COMPUTER HARDWARE REFERENCE MANUAL

- iRMX 86 INSTALLATION GUIDE or iRMX 88 INSTALLATION INSTRUCTIONS

- The configuration chapter (Chapter 7) of this manual.

You should also consult the hardware reference manuals for any other Intel hardware products that you are using.

WRITING TASKS THAT COMMUNICATE WITH AN ETHERNET NETWORK

The iSBC 550 Ethernet communications controller provides only basic services. It transmits information to the Ethernet network, and it receives information from the Ethernet network. It also does some message filtering by accepting from the network only messages of the requested Ethernet TYPE code.

Although the iSBC 550 controller does transmit, receive, and filter messages, there are many services that it does not perform. For example, it does not:

- Decide which task in your system is to receive a particular message.
- Add or remove header information that is required in each iSBC 550 request.

If your application system requires either of these or other similar high-level services, your tasks must explicitly provide the services.

For this reason, the remainder of this appendix utilizes a collection of three tasks to manage the iSBC 550 controller. The three tasks are an Initialization Task, a Reader Task, and a Writer Task. These tasks will be called Ethernet tasks, to distinguish them from the other tasks of an application system. After presenting some background information, this appendix describes the duties of each of the Ethernet tasks.

The Ethernet tasks insulate the rest of your application system from the details of the iSBC 550 controller. For example, the other tasks of your application system can send and receive (via iRMX 86 mailboxes or iRMX 88 exchanges) messages from the Ethernet network without having to add or remove the special header information required by the iSBC 550 Ethernet controller.

Another benefit of using Ethernet tasks is that you can implement high-level features on top of the Ethernet protocol. For example, you can design the Reader Task to examine a particular field of a received message and then route the message to the proper task within your application. If desired, you can also build special protocols to perform other duties.

The Ethernet tasks of your iRMX 86- or iRMX 88-based application system can communicate with an Ethernet network by using the system calls of the iRMX 86 Basic I/O System, the iRMX 86 Extended I/O System, or the iRMX 88 I/O System. Whenever one of your tasks uses an I/O System call to read or write to an Ethernet network, the following events occur:

- The I/O System uses MMX 86 or MMX 88 to communicate with the iSBC 550 communications controller.

- The iSBC 550 communications controller communicates directly with the Ethernet network.

Although the process requires MMX 86 or MMX 88, your Ethernet tasks do not explicitly invoke any MMX 86 or MMX 88 procedure calls. The I/O System that you have chosen will invoke any MMX 86 or MMX 88 procedure calls that are required. Of course, MMX 86 or MMX 88 services are still available to your application tasks.

However, whenever one of your Ethernet tasks communicates with the iSBC 550 communications controller by making calls to the I/O System, your task must explicitly set up an iSBC 550 request block that tells the iSBC 550 controller what to do.

BUILDING AN iSBC 550 REQUEST BLOCK

Whenever one of your Ethernet tasks sends information to (or receives information from) the iSBC 550 communications controller, the task must use an I/O System call to pass an iSBC 550 request block to the iSBC 550 controller. This request block is subject to two constraints:

1. With one exception, the request block must adhere to the format described in the ETHERNET COMMUNICATIONS CONTROLLER PROGRAMMER'S REFERENCE MANUAL. The exception is that your task need not fill in the RESPONSE SOCKET and PROCESSOR ID fields. The I/O System fills in these fields.

2. The iSBC 550 request block must be embedded in an iRMX 86 segment, with the first byte of the request block being the first byte of the segment or block. This means that your task cannot use an arbitrary block of memory as an iSBC 550 request block. Instead, the task must first create an iRMX 86 segment or allocate an iRMX 88 memory block and then construct the request block within the segment.

SENDING THE REQUEST BLOCK TO THE iSBC 550 CONTROLLER

Once your task has built the request block, it must send the block to the iSBC 550 controller. It does so by means of the RQAWRITE system call (of the iRMX 86 Basic I/O System), the RQSWRITE$MOVE system call (of the iRMX 86 Extended I/O System), or the DQ$WRITE system call (of the iRMX 88 I/O System), regardless of which iSBC 550 command is indicated in the request block. (Eight iSBC 550 commands are available. They are CONNECT, DISCONNECT, ADDMCID, DELETEMCID, TRANSMIT, SUPPLYBUF, READ, and READC.)

For any command that requires a response from the iSBC 550 controller, your task can find the response embedded in the same request block that was sent to the controller. If you have selected the iRMX 86 Basic I/O System, and the Writer Task is handling many write requests, the IORS's that are returned to the task might be returned in a different order than the order in which the corresponding RQAWRITE calls were issued. In this case, the Writer Task can find the correct token for each request block segment it has sent by looking in the BUFF$P field of the IORS that is returned in the designated response mailbox. The address of the segment is IORS.BUFF$P. If you have selected the iRMX 86 Extended I/O System or the iRMX 88 I/O System, the request blocks are returned in the same order in which they were sent, so it is not necessary for the requesting task to identify the request blocks that are returned.

When your Reader or Writer Task uses a writing system call to transfer a request block to the iSBC 550 controller, the task can receive exception codes other than those returned by the I/O System. The iRMX 86 tasks can receive MMX 86 exception codes and the iRMX 88 tasks can receive MMX 88 exception codes.

THE ETHERNET TASKS' ENVIRONMENT AND DUTIES

The three subsections of this section describe, separately for each of the I/O Systems of the iRMX 86 Operating System and iRMX 88 Executive, both the structure of the Ethernet tasks and special use restrictions regarding the sysyem calls of that I/O System. Many readers will need to read only one of these sections. Note that, in each high-level task description, some important elements, such as exception handling, have been omitted. This is intentional, so that you can more easily see the structure of the Ethernet tasks.

Using the iRMX 86 Basic I/O System

The following sections describe the Ethernet tasks and use restrictions pertaining to the iRMX 86 Basic I/O System.

The Ethernet Tasks. Ethernet Tasks that use the Basic I/O System have the following structures:

- Initialization Task

    1. Attach the iSBC 550 controller.
    2. Create the appropriate file on the device, using the device token obtained in step 1.
    3. Open the file for reading and writing, using the connection obtained in step 2.

4. Create a segment, build an iSBC 550 CONNECT request block in the segment, and use the RQAWRITE system call to send the segment to the iSBC 550 controller.

5. Create the Reader Task. Pass the connection to the Reader Task.

6. Create the Writer Task. Pass the connection to the Writer Task.

7. Suspend or delete itself.

- Reader Task

1. Create several segments containing iSBC 550 SUPPLYBUF request blocks and use the RQAWRITE system call to send them to the iSBC 550 controller, with the same response mailbox indicated in each call.

2. Wait at the response mailbox.

3. When a request block segment arrives at the response mailbox, the Reader Task creates a segment, copies the information from the block into the new segment, and sends the new segment to the appropriate application mailbox.

4. Call RQAWRITE to send the SUPPLYBUF request block back to the iSBC 550 controller, again indicating the same response mailbox.

5. Go to step 2.

- Writer Task

1. Wait at a previously-designated reception mailbox for write requests from application tasks and for returned iSBC 550 request block segments.

2. If a write request from an application task arrives at the mailbox, go to step 3. If an IORS arrives at the mailbox, go to step 5.

3. Create a segment, and build an iSBC 550 TRANSMIT request block with the data that is to be written. Specify that the segment is to be returned to the reception mailbox, and call RQAWRITE to send the segment to the iSBC 550 controller.

4. Go to step 1.

5. Delete the IORS and write request segment, and go to step 1.

Assume, when reading the following section about using the Basic I/O System, that we are referring to a system in which Ethernet tasks are being utilized as just outlined.

Use Restrictions. This section of the appendix assumes that you are already familiar with the iRMX 86 Basic I/O System. Consequently, rather than providing a tutorial on the Basic I/O System, this section of the appendix discusses only matters relating directly to using the iSBC 550 Ethernet controller. Reference material concerning the Basic I/O System is divided between the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL and the iRMX 86 SYSTEM PROGRAMMER'S REFERENCE MANUAL.

The Basic I/O System, when used for communication with an iSBC 550 controller, has different restrictions or behaves differently than it does in systems that do not support such communication. The differences fall into three areas.

First, the software link between the Basic I/O System and the iSBC 550 controller is implemented as a physical file. This means that your task should not use any of the system calls reserved for use only with stream files and/or named files. For example, if your task invokes the RQAGET$PATH$COMPONENT system call for this file, the Basic I/O System returns an E$SUPPORT exception code.

Second, even among the system calls that generally are useful for physical files, a few behave differently when used with the Ethernet controller. These are RQAREAD, RQASPECIAL, RQASEEK, RQATRUNCATE, RQAGET$CONNECTION$STATUS, and RQAGET$FILE$STATUS. The behavioral differences are as follows:

- If your task attempts to use the RQAREAD system call, the Basic I/O System responds as if the task had issued an RQAWRITE system call. The only prerequisite of the RQAREAD system call is that the connection be open for reading before your task invokes RQAREAD.

- If your task attempts to use the RQASPECIAL, RQASEEK, or RQATRUNCATE system call, the Basic I/O System returns an E$IDDR exception code, indicating that these system calls are not supported on the Ethernet controller.

- If your task invokes the RQAGET$FILE$STATUS or the RQAGET$CONNECTION$STATUS system call, some of the returned information is undefined. For the RQAGET$FILE$STATUS system call, the following fields are undefined:

  flags
  dev$gran
  dev$size

  For the RQAGET$CONNECTION$STATUS system call, the following fields are undefined:

  flags
  file$ptr
  access

Third, among the system calls that behave as expected, some of the input parameters have special restrictions:

- RQAATTACH$FILE -- The user and subpath parameters are ignored.
- RQACREATE$FILE -- The user, subpath, access, granularity, size, and must$create parameters are ignored. The prefix parameter must be a token for the device connection for the iSBC 550 controller.
- RQAREAD and RQAWRITE -- The buffer pointer must be a token for an iRMX 86 segment containing the request block. The count parameter is ignored, because the iSBC 550 controller ascertains the count from the request block.

All other Basic I/O System calls work exactly as expected.

Using the iRMX 86 Extended I/O System

The following sections describe the Ethernet tasks and use restrictions pertaining to the iRMX 86 Extended I/O System.

The Ethernet Tasks. Ethernet Tasks that use the Extended I/O System have the following structures:

- Initialization Task
    1. Attach the iSBC 550 controller.
    2. Create the appropriate file on the device, using the device token obtained in step 1.
    3. Open the file for reading and writing, using the connection obtained in step 2.
    4. Create a segment, build an iSBC 550 CONNECT request block in the segment, and use the RQSWRITE$MOVE system call to send the segment to the iSBC 550 controller.
    5. Create the Reader Task. Pass the connection to the Reader Task.
    6. Create the Writer Task. Pass the connection to the Writer Task.
    7. Suspend or delete itself.
- Reader Task
    1. Create a segment. (This is segment A.)

2. Build an iSBC SUPPLYBUF request block in segment A, and call RQSWRITE$MOVE to send it to the iSBC 550 controller.

3. (When control returns,) create another segment -- segment B -- of the required size, copy the data from segment A into segment B, and call RQ$SEND$MESSAGE to send segment B to the appropriate mailbox.

4. Go to step 2.

- Writer Task

1. Wait at a previously-designated reception mailbox for write requests from application tasks.

2. When a write request arrives, create a segment of the appropriate size, build an iSBC 550 TRANSMIT request block there with the data that is to be written, and call RQSWRITE$MOVE to send the segment to the iSBC 550 controller.

3. Go to step 1.

Assume, when reading the following section about using the Extended I/O System, that we are referring to a system in which Ethernet tasks are being utilized as just outlined.

Use Restrictions. This section of the appendix assumes that you are already familiar with the iRMX 86 Extended I/O System. Consequently, rather than providing a tutorial on the Extended I/O System, this section of the appendix only discusses matters relating directly to using the iSBC 550 Ethernet controller. Reference material concerning the Extended I/O System is contained in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

The Extended I/O System, when used for communication with an iSBC 550 controller, has different restrictions or behaves differently than it does in systems that do not support such communication. The differences fall into three areas.

First, the software link between the Extended I/O System and the iSBC 550 controller is implemented as a physical file. This means that your task should not use any of the system calls reserved for use only with stream files and/or named files. For example, if your task invokes the RQSCHANGE$ACCESS system call for this file, the Extended I/O System returns an E$SUPPORT exception code.

Second, even among the system calls that generally are useful for physical files, a few behave differently when used with the Ethernet controller. These are RQSREAD$MOVE, RQ$S$SPECIAL, RQ$S$SEEK, RQ$S$TRUNCATE, RQ$SGETCONNECTION$STATUS, RQ$SGETFILE$STATUS. The behavioral differences are as follows:

- If your task attempts to use the RQSREAD$MOVE system call, the Extended I/O System responds as if the task had issued an RQSWRITE$MOVE system call. The only prerequisite of the RQSREAD$MOVE system call is that the connection be open for reading before your task invokes RQSREAD$MOVE.

- If your task attempts to use the RQSSPECIAL, RQSSEEK, or RQSTRUNCATE$FILE system call, the Extended I/O System returns an E$IDDR exception code, indicating that these system calls are not supported on the Ethernet controller.

- If your task invokes the RQSGET$FILE$STATUS or the RQSGET$CONNECTION$STATUS system call, some of the returned information is undefined. For the RQSGET$FILE$STATUS system call, the following fields are undefined:

dev$gran	file$blocks	vol$size
dev$size	vol$name	accessor$count
file$gran	vol$gran	owner$access
owner$id		

  For the RQSGET$CONNECTION$STATUS system call, the following fields are undefined:

flags	num$buf
file$ptr	buf$size
access	

Third, among the system calls that behave as expected, some of the input parameters have special restrictions:

- RQSREAD$MOVE -- The buf$ptr parameter must be a token for an iRMX 86 segment containing the request block. The bytes$desired parameter is ignored because the iSBC 550 controller ascertains the number of bytes desired from the request block. For similar reasons, the bytes$read output parameter is undefined.

- RQSWRITE$MOVE -- The buf$ptr parameter must be a token for an iRMX 86 segment containing the request block. The count parameter is ignored because the iSBC 550 controller ascertains the count from the request block. For similar reasons, the bytes$read output parameter is undefined.

All other Extended I/O System calls work exactly as expected.

Using the iRMX 88 I/O System

The following sections describe the Ethernet tasks and use restrictions pertaining to the iRMX 88 I/O System.

The Ethernet Tasks. Ethernet Tasks that use the iRMX 88 I/O System have the following structures:

- Initialization Task

    1. Obtain a connection to the appropriate file by calling DQ$CREATE or DQ$ATTACH.

    2. Open the file for reading and writing, using the connection obtained in step 1.

    3. Obtain a memory block by calling DQ$ALLOCATE, build an iSBC 550 CONNECT request block in the memory, and call DQ$WRITE to send the request block to the iSBC 550 controller.

    4. Create the Reader Task.

    5. Create the Writer Task.

    6. Suspend or delete itself.

- Reader Task

    1. Call DQ$ALLOCATE to obtain a block of memory. (This is memory block A.)

    2. Build an iSBC 550 SUPPLYBUF request block in memory block A, and call DQ$WRITE to send it to the iSBC 550 controller. Reserve 15 bytes for the message header at the beginning of memory block A, in addition to the 12 bytes that are reserved in any SUPPLY$BUF request. When calling DQ$WRITE, set the length parameter to reflect the size of the entire message, including headers.

    3. (When control returns,) call DQ$ALLOCATE to obtain a block of memory -- memory block B -- of the required size, build an iRMX 88 message in memory block B, copy the data from memory block A to memory block B, and call RQSEND to send memory block B to the appropriate exchange.

    4. Go to step 2.

- Writer Task

    1. Wait at a previously-designated exchange for write requests from application tasks.

2. When a write request arrives, call DQ$ALLOCATE to obtain a block of the appropriate size, build an iSBC 550 TRANSMIT request block there with the data that is to be written, and call DQ$WRITE to send the request to the iSBC 550 controller. Reserve 15 bytes for the message header at the beginning of the memory block, in addition to the 12 bytes that are reserved in any TRANSMIT request. When calling DQ$WRITE, set the length parameter to reflect the size of the entire message, including headers.

3. Go to step 1.

Use Restrictions. This section of the appendix assumes that you are already familiar with the iRMX 88 I/O System. Consequently, rather than providing a tutorial on the iRMX 88 I/O System, this section of the appendix only discusses matters relating directly to using the iSBC 550 Ethernet controller. Reference material concerning the iRMX 88 I/O System is contained in the iRMX 88 REFERENCE MANUAL.

The iRMX 88 I/O System, when used for communication with an iSBC 550 controller, has different restrictions or behaves differently than it does in systems that do not support such communication. The differences fall into three areas.

First, the software link between the iRMX 88 I/O System and the iSBC 550 controller is implemented as a physical file. This means that your task should not use any of the system calls reserved for use only with named files. For example, if your task invokes the DQ$RENAME system call for this file, the iRMX 88 I/O System returns an E$SUPPORT exception code.

Second, even among the system calls that generally are useful for physical files, a few behave differently when used with the Ethernet controller. These are DQ$READ, DQ$SPECIAL, and DQ$SEEK. The behavioral differences are as follows:

- If your task attempts to use the DQ$READ system call, the iRMX 88 I/O System responds as if the task had issued an DQ$WRITE system call. The only prerequisite of the DQ$READ system call is that the connection be open for reading before your task invokes DQ$READ.

- If your task attempts to use the DQ$SPECIAL or DQ$SEEK system call, the iRMX 88 I/O System returns an E$IDDR exception code, indicating that these system calls are not supported on the Ethernet controller.

- If your task invokes the DQGETCONNECTION$STATUS system call, the following fields are undefined:

  access file$ptr seek

All other iRMX 88 I/O System calls work exactly as expected.

Third, RQFORMAT is not supported.

CONFIGURING, LINKING, AND LOCATING AN iRMX 86 OR 88 I/O SYSTEM FOR USE WITH iSBC 550 CONTROLLERS

The remainder of this appendix is devoted to describing the configuration, linking, and locating processes required to prepare an iRMX 86 or 88 I/O System for use with iSBC 550 controllers. Each of the following sections on configuring an iRMX 86 or 88 I/O System assumes that you are familiar with the general configuration process for that operating system. Consequently, these sections focus on Ethernet-related matters, insofar as such matters can be separated from other configuration issues. If you need to learn more about configuring an iRMX 86 or 88 system, refer to the iRMX 86 CONFIGURATION GUIDE or the iRMX 80/88 INTERACTIVE CONFIGURATION UTILITY USER'S GUIDE.

In the remainder of this appendix, there are several references to device-unit information blocks and device information tables. You don't necessarily have to know the meanings of these terms, but if you do need to, descriptions of them and other matters pertaining to device drivers can be found in the GUIDE TO WRITING DEVICE DRIVERS FOR THE iRMX 86 and iRMX 88 I/O SYSTEMS.

CONFIGURING AN iRMX 86 I/O SYSTEM FOR USE WITH iSBC 550 CONTROLLERS

Whether you are planning to use the Extended I/O System or not, you must configure the Basic I/O System. And it is in the configuration module for the Basic I/O System that you put the descriptive information about the iSBC 550 device. All that is needed in the configuration of the Extended I/O System, assuming that you have chosen to use it, is a %DEV_INFO_BLOCK macro for the iSBC 550 device.

Four INCLUDE files are used for adding configuration information concerning the iSBC 550 Ethernet controller to the Basic I/O System's standard device configuration file IDEVCF.A86. They are:

I550.EXT	External declarations of the names of the device driver routines that appear in the DUIB (device-unit information block) for the iSBC 550 device.
IEDUIB.LIT	A sample DUIB for an iSBC 550 device-unit.
IEDINF.INC	A declaration of the device information table structure for the iSBC 550 device.
IEDINF.LIT	A sample device information table for the iSBC 550 device.

The data in both IEDUIB.LIT and IEDINF.LIT can be modified to fit your special needs.

The DUIB for the iSBC 550 device is as follows:

```
define_duib
&   device name,
&   file drivers,
&   functions,
&   flags,
&   device gran,
&   low device size,
&   high device size,
&   device number,
&   unit number,
&   device unit number,
&   i550$init,
&   i550$finish,
&   i550$queue,
&   i550$cancel,
&   device info,
&   unit info,
&   update timeout,
&   number buffers,
&   priority
&
```

where the fields that you may or must change are:

device name	A one- to fourteen-character name that is unique among all device names in the system. This name, which must be preceded and followed by single quotes, is used in the call to RQAPHYSICAL$ATTACH$DEVICE, in order to identify the device to be attached.
device number	A BYTE containing the number of the device associated with this DUIB.
device unit number	A BYTE containing the number of the device-unit associated with this DUIB.
i550$init	A WORD containing the base address of the init$io routine that the I/O System calls.
i550$finish	A WORD containing the base address of the finish$io routine that the I/O System calls.
i550$queue	A WORD containing the base address of the queue$io routine that the I/O System calls.
i550$cancel	A WORD containing the base address of the cancel$io routine that the I/O System calls.
device info	A POINTER to the device information table for the iSBC 550 device.

The device information table has the following format:

```
define_devinf
&   init start addr
&   host device id
&   550 device id
&   RQ to 550
&   RQ from 550
&   IDS base strt addr
&   IDS base length
&   550 interrupt port
&   550 port name
&   host port info
&   timeout
&   priority
&   550 interrupt type
```

where the fields that you may or must change are as follows. An asterisk (*) indicates that the same value must also appear in the iMMX 800 configuration file for the "host device", that is, the device with which the iSBC 550 device communicates.

init start addr	A POINTER to the start address of the iSBC 550 communication area that is used for initialization. This address is a hardware configuration option on the iSBC 550 controller. See the ETHERNET COMMUNICATIONS CONTROLLER PROGRAMMER'S REFERENCE MANUAL for details concerning this address.
host device id*	A BYTE containing the device ID for the host (iSBC 86/12A or 86/30) device. This ID value goes into DSDT arrays.
550 device id*	A BYTE containing the device ID for the iSBC 550 device. This ID value goes into DSDT arrays.
RQ to 550*	A POINTER to the request queue for communication from the host device to the iSBC 550 device. This pointer goes into the DCM$ROM array.
RQ from 550*	A POINTER to the request queue for communication from the iSBC 550 device to the host device. This pointer goes into the DCM$ROM array.
IDS base strt addr*	A BYTE containing the start address of the IDS managed by the host device, as a multiple of 4K. This value goes into the IDST array.
IDS base length*	A BYTE specifying the size of the IDS managed by the host device, as a multiple of 4K.

550 interrupt port	A WORD containing the I/O port address used for waking up the iSBC 550 device during initialization. This port is a hardware configuration option on the iSBC 550 controller. See the iSBC 550 ETHERNET COMMUNICATIONS CONTROLLER HARDWARE REFERENCE MANUAL for details concerning this port. This value goes into the SFT array.
550 port name*	A WORD containing the system port name of the iSBC 550 port to which the iMMX 800 software will deliver messages. This name goes into the DSDT arrays.
host port info*	A WORD containing the socket on the host computer to which the iSBC 550 device will send messages. A socket consists of a device ID, port ID pair. For example, if the host device's ID were 1 and the port ID on the host device were 2, the socket would be 0102H. These device ID and port ID values go into an entry in the DSDT array.
timeout	A BYTE containing the time, in 52-millisecond units, that the iSBC 550 controller will wait for a response from MMX 86 before declaring the host device dead. The recommended value of 0FFH indicates that the iSBC 550 device will wait forever.
priority	A BYTE containing the priority of the Ethernet driver task that receives messages from MMX 86. A value of 129 is recommended.
550 interrupt type	A BYTE containing a code for the method used by the iSBC 550 device to interrupt the host device. The values for this field are defined in the ETHERNET COMMUNICATIONS CONTROLLER PROGRAMMER'S REFERENCE MANUAL.

A Sample Basic I/O System Configuration File

The following is a sample configuration file for the Basic I/O System. It also specifies a device driver for the iSBC 550 device.

```
1 $title(idevcf: I/O System Device-Driver Configuration Module)
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;
  ;  Note:
  ;       The I/O System has two configuration files: the itable.a86
  ;       for various interfaces and file-driver configurations; the
  ;       idevcf.a86 for device-driver configuration.
  ;
  ; idevcf.a86
  ;       Sample I/O System Device-Driver Configuration Module.
  ;
  ; Configures:
  ;
  ;       iSBC 550 Device Driver (using MMX 86)
  ;
  ;       Byte-Bucket Driver
  ;
  ;
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

                  name    idevcf

1 $include(:fl:idevcf.inc)
1 $save nolist
1 $nogen
  $include(:fl:iedinf.inc)

  ;
  ; specify iSBC 550 Ethernet custom device driver information
  ;
  i550_dev_info_struc
              dw      0                   ; 550 comm area offset
              dw      0                   ; 550 comm area base
              db      0                   ; host device id
              db      0                   ; 550 device id
              dw      0                   ; host rqd offset
              dw      0                   ; host rqd base
              dw      0                   ; 550 rqd offset
              dw      0                   ; 550 rqd base
              db      0                   ; ids base start addr
              db      0                   ; ids base length
              dw      0                   ; 550 interrupt port
              dw      0                   ; 550 port name
              dw      0                   ; host port name
              db      0                   ; timeout
              db      0                   ; priority
              db      0                   ; 550 interrupt type
  i550_dev_info  ends

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Define Device Driver External Procedures.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
code          segment
              assume  cs: cgroup
```

```
$include(:f1:i550.ext)
                extrn   i550init: near
                extrn   i550finish: near
                extrn   i550queue: near
                extrn   i550cancel: near

                extrn   bytebucketinitio: near
                extrn   bytebucketfinishio: near
                extrn   bytebucketqueueio: near
                extrn   bytebucketcancelio: near

code            ends
                assume  cs: nothing

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Define Device-Unit Information Blocks (DUIB's).
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

code            segment

duibtable       label   byte
                public  duibtable
$include(:f1:ieduib.lit)
;
; Ethernet iSBC 550 device, unit 0
;
define_duib     <
&               'E0',                   ; name (14)
&               001H,                   ; file$drivers
&               033H,                   ; functs
&               00H,                    ; flags
&               00,                     ; dev$gran
&               0H,0H,                  ; dev$size = 0
&               0,                      ; device
&               0,                      ; unit
&               0,                      ; dev$unit
&               i550init,               ; init$io
&               i550finish,             ; finish$io
&               i550queue,              ; queue$io
&               i550cancel,             ; cancel$io
&               dinfo_550,              ; device$info
&               0,                      ; unit$info
&               0ffffh,                 ; update$timeout
&               0,                      ; num$buffers
&               129                     ; priority
&>
```

```
;
; Byte-Bucket
;
define_duib     <
&               'BB',                   ; name(14)
&               03H,                    ; file$drivers
&               0F3H,                   ; functs
&               00H,                    ; flags
&               0,                      ; dev$gran
&               0,0,                    ; dev$size
&               2,                      ; device
&               0,                      ; unit
&               2,                      ; dev$unit
&               bytebucketinitio,       ; init$io
&               bytebucketfinishio,     ; finish$io
&               bytebucketqueueio,      ; queue$io
&               bytebucketcancelio,     ; cancel$io
&               0,                      ; device$info
&               0,                      ; unit$info
&               0FFFFH,                 ; update$timeout
&               0,                      ; num$buffers
&               129                     ; priority
&>

;
; Now is an appropriate time to define the number of DUIB's
;
NUM_DUIB        equ     (this byte - duibtable) / size define_duib

code            ends
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Define parameters and device tables.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

NUM_DEV_UNIT    equ     2 + 1           ; Max(dev$units in duib's) + 1
NUM_DEVICES     equ     2 + 1           ; Max(device #'s in duib's) + 1

%device_tables(NUM_DUIB,NUM_DEV_UNIT,NUM_DEVICES)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;       define device information
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
code            segment
$include(:f1:iedinf.lit)
;
;       iSBC 550 device information
;
```

```
dinfo_550 i550_dev_info <
&               0h,                 ; comm area start address offset
&               2000h,              ; comm area start address base
&               0h,                 ; host device id
&               01h,                ; 550 device id
&               0h,                 ; rqd to 550 offset
&               2010h,              ; rqd to 550 base
&               0h,                 ; rqd from 550 offset
&               2020h,              ; rqd from 550 base
&               0h,                 ; ids base start addr
&               4fh,                ; ids length
&               0a4h,               ; 550 interrupt port
&               0101h,              ; 550 port name
&               0000h,              ; host port name
&               0ffh,               ; timeout
&               129,                ; priority
&               03h                 ; 550 interrupt type
&>
code            ends

end
```

A Sample MMX 86 Configuration File for the Host Device

This section contains a sample MMX 86 configuration file for the (host) device that communicates with the iSBC 550 device. Note that this example exhibits the following properties, which are required of every MMX 86 configuration:

- The depth (RQINSIZE in DCM$ROM) of the request queue, for requests from the iSBC 550 device to the host device, must be 4.
- The system port name (SYSTEM$PORT$NAME in LPT$ROM) that the iSBC 550 device uses to reference the system port on the host device must be the same as that specified in the device information table for the host device.
- The device id (DEST$PORT$ID in DSDT) for the iSBC 550 device must not be zero. Port zero has special meaning to the iSBC 550 device and cannot be used by MMX 86.
- In the SFT structure corresponding to the iSBC 550 device, OP$MODE must be SLAVE$DEVICE (=01H), INTR$TYPE must be IO$INTERRUPT (=03H), and INTR$VALUE must be 02H.
- If the interrupt type field in the iSBC 550 Start Command request block for the iSBC 550 device is 3 or 4 -- where 3 is the recommended value if the iSBC 550 device is to interrupt the host device, and 0 is recommended otherwise -- the CLRINTYPE field in the SFT should be 01H and the INTR$VALUE field should be 02H. Moreover, the values for the INTR$LOCATION and CLR$IN$INTR$LOCATION fields of the SFT should correspond to the values for which the iSBC 550 device is jumpered.

- Memory on the host device must be mapped so that all of the iRMX 86 free space is addressable by the iSBC 550 device.

```
R4CNFG:
    DO;

$INCLUDE(:F1:R4CNFG.LIT)
    DECLARE DSD$ENTRY$TYPE LITERALLY 'STRUCTURE(
        SYSTEM$PORT$NAME        WORD,
        DEST$DEV$ID             BYTE,
        DEST$PORT$ID            BYTE,
        SRC$DEV$ID              BYTE,
        RESERVED                BYTE,
        POOL$ID                 BYTE,
        IDS$ID                  BYTE)';

    DECLARE LPT$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
        SYSTEM$PORT$NAME        WORD)';

    DECLARE LPT$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
        ENTRY(3)                BYTE)';

    DECLARE DM$ROM$ENTRY$TYPE LITERALLY 'STRUCTURE(
        RQD$OUT                 POINTER,
        RQ$OUT$SIZE             BYTE,
        RQE$OUT$SIZE            BYTE,
        RQD$IN                  POINTER,
        RQ$IN$SIZE              BYTE,
        RQE$IN$SIZE             BYTE)';

    DECLARE NO$SYSTEM$CHANNEL LITERALLY '0FFFFH,
                                           00H,
                                           00H,
                                        0FFFFH,
                                           00H,
                                           00H)';

    DECLARE DM$RAM$ENTRY$TYPE LITERALLY 'STRUCTURE(
        ENTRY(20)               BYTE)';

    DECLARE SFT$ENTRY$TYPE LITERALLY 'STRUCTURE(
        OP$MODE                 BYTE,
        INTR$TYPE               BYTE,
        INTR$LOCATION           WORD,
        INTR$VALUE              WORD,
        CLR$OUT$TYPE            BYTE,
        CLR$OUT$INTR$LOCATION   WORD,
        CLR$OUT$INTR$VALUE      WORD,
        CLR$IN$TYPE             BYTE,
        CLR$IN$INTR$LOCATION    WORD,
        CLR$IN$INTR$VALUE       WORD)';
```

```
DECLARE NO$DEVICE                LITERALLY '00H';
DECLARE SLAVE$DEVICE             LITERALLY '01H';
DECLARE PEER$DEVICE              LITERALLY '02H';

DECLARE NO$SYSTEM$SERVICE LITERALLY '00H,
                                     00H,
                                   0000H
                                   0000H,
                                     00H,
                                   0000H,
                                   0000H,
                                     00H,
                                   0000H,
                                   0000H';

DECLARE IDS$ENTRY$TYPE LITERALLY 'STRUCTURE(
    OFFSET                   WORD,
    PAGE                     WORD)';

DECLARE POOL$ENTRY$TYPE LITERALLY 'STRUCTURE(
    ENTRY(2)                 BYTE)';

DECLARE BLOCK$ENTRY$TYPE LITERALLY 'STRUCTURE(
    POOL$ID                  BYTE,
    START$ADR                SELECTOR,
    LENGTH                   WORD)';

DECLARE MMX$DEVICES                  LITERALLY '2';
DECLARE DEV$0$PORTS                  LITERALLY '1';
DECLARE DEV$0$PORT$0$NAME            LITERALLY '0000H';
DECLARE OUT$QUEUE$ADDRESS            LITERALLY '20100H';
DECLARE OUT$QUEUE$SIZE               LITERALLY '04H';
DECLARE OUT$QUEUE$ENTRY$SIZE         LITERALLY '04H';
DECLARE IN$QUEUE$ADDRESS             LITERALLY '20200H';
DECLARE IN$QUEUE$SIZE                LITERALLY '04H';
DECLARE IN$QUEUE$ENTRY$SIZE          LITERALLY '04H';
DECLARE DEV$0$DEST$PORTS             LITERALLY '1';
DECLARE DEV$1$PORT$1$NAME            LITERALLY '0101H';
DECLARE DEV$1$ID                     LITERALLY '1';
DECLARE DEV$1$PORT$1$ID              LITERALLY '1';
DECLARE DEV$0$ID                     LITERALLY '0';
DECLARE DEV$0$POOL$0$ID              LITERALLY '0';
DECLARE IDS$0$ID                     LITERALLY '0';
DECLARE COMMUNICATE$WAIT$TIME        LITERALLY '0000H';
DECLARE RESPONSE$WAIT$TIME           LITERALLY '0100H';
DECLARE DEV$1$OP$MODE                SLAVE$DEVICE;
DECLARE DEV$1$INTR$TYPE              LITERALLY '2';
DECLARE DEV$1$INTR$LOCATION          LITERALLY '0A4H';
DECLARE DEV$1$INTR$VALUE             LITERALLY '02H';
DECLARE DEV$1$CLR$OUT$TYPE           LITERALLY '0';
DECLARE DEV$1$CLR$OUT$INTR$LOCATION LITERALLY '0';
DECLARE DEV$1$CLR$OUT$INTR$VALUE     LITERALLY '0';
DECLARE DEV$1$CLR$IN$TYPE            LITERALLY '1';
DECLARE DEV$1$CLR$IN$INTR$LOCATION   LITERALLY '0A4H';
DECLARE DEV$1$CLR$IN$INTR$VALUE      LITERALLY '04H';
```

```
DECLARE DEV$0$INT$LEVEL              LITERALLY '0048H';
DECLARE DEV$0$POLLING$PERIOD         LITERALLY '200';
DECLARE MMX$INTERDEVICE$SEGMENTS     LITERALLY '01H';
DECLARE DEV$0$IDS$0$OFFSET           LITERALLY '0000H';
DECLARE DEV$0$IDS$0$PAGE             LITERALLY '0000H';
DECLARE DEV$0$POOLS                  LITERALLY '1';
DECLARE DEV$0$BLOCKS                 LITERALLY '1';
DECLARE DEV$0$POOL$0$ADDR            LITERALLY '2030H';
DECLARE DEV$0$POOL$0$LENGTH          LITERALLY '0020H';

DECLARE CQDVCS BYTE PUBLIC DATA(
    MMX$DEVICES);

DECLARE DCM$RAM(MMX$DEVICES) DM$RAM$ENTRY$TYPE PUBLIC;

DECLARE DCM$ROM(MMX$DEVICES) DM$ROM$ENTRY$TYPE PUBLIC DATA(
    NO$SYSTEM$CHANNEL,
    OUT$QUEUE$ADDRESS,
        OUT$QUEUE$SIZE,
        OUT$QUEUE$ENTRY$SIZE,
        IN$QUEUE$ADDRESS,
        IN$QUEUE$SIZE,
        IN$QUEUE$ENTRY$SIZE);

DECLARE CQPRTS BYTE PUBLIC DATA(
    DEV$0$PORTS);

DECLARE LPT$RAM(DEV$0$PORTS) LPT$RAM$ENTRY$TYPE PUBLIC;

DECLARE LPT$ROM(DEV$0$PORTS) LPT$ROM$ENTRY TYPE PUBLIC DATA(
    DEV$0$PORT$0$NAME);

DECLARE CQSKTS BYTE PUBLIC DATA(
    DEV$0$DEST$PORTS);

DECLARE DSDT(DEV$0$DEST$PORTS) DSD$ENTRY$TYPE PUBLIC DATA(
    DEV$1$PORT$1$NAME,
    DEV$1$ID,
    DEV$1$PORT$1$ID,
    DEV$0$ID,
    0,
    DEV$0$POOL$0$ID,
    IDS$0$ID);

DECLARE CQITWT WORD PUBLIC DATA(
    COMMUNICATE$WAIT$TIME);

DECLARE CQMDLY WORD PUBLIC DATA(
    RESPONSE$WAIT$TIME);
```

```
DECLARE SFT(MMX$DEVICES) SFT$ENTRY$TYPE PUBLIC DATA(
    NO$SYSTEM$SERVICE,
    DEV$1$OP$MODE,
        DEV$1$INTR$TYPE,
        DEV$1$INTR$LOCATION,
        DEV$1$INTR$VALUE,
        DEV$1$CLR$OUT$TYPE,
        DEV$1$CLR$OUT$INTR$LOCATION,
        DEV$1$CLR$OUT$INTR$VALUE,
        DEV$1$CLR$IN$TYPE,
        DEV$1$CLR$IN$INTR$LOCATION,
        DEV$1$CLR$IN$INTR$VALUE);

DECLARE CQSGLV WORD PUBLIC DATA(
    DEV$0$INT$LEVEL);

DECLARE CQIDPD BYTE PUBLIC DATA(
    DEV$0$POLLING$PERIOD);

DECLARE CQIDSS BYTE PUBLIC DATA(
    MMX$INTERDEVICE$SEGMENTS);

DECLARE IDST(MMX$INTERDEVICE$SEGMENTS) IDS$ENTRY$TYPE PUBLIC DATA(
    DEV$0$IDS$0$OFFSET,
    DEV$0$IDS$0$PAGE);

DECLARE CQPLHS BYTE PUBLIC DATA(
    DEV$0$POOLS);

DECLARE PLHTBL(DEV$0$POOLS) POOL$ENTRY$TYPE PUBLIC;

DECLARE CQBLKS BYTE PUBLIC DATA(
    DEV$0$BLOCKS);

DECLARE BLKTBL(DEV$0$BLOCKS) BLOCK$ENTRY$TYPE PUBLIC DATA(
    DEV$0$POOL$0$ID,
    DEV$0$POOL$0$ADDR,
    DEV$0$POOL$0$LENGTH);
```

LINKING AND LOCATING THE CONFIGURED iRMX 86 I/O SYSTEM

In order to link and locate the configured I/O System with the iSBC 550 device driver, you must make some modifications to the sample submit file IOS.CSD, which is provided with the iRMX 86 Operating System for linking and locating the I/O System. The modifications are the following:

I550.LIB, which contains the iSBC 550 device driver object code, must be linked after IDEVCF.OBJ.

R4CINF.LIB, which is the MMX 86 compact interface library, gets linked after IOS.LIB.

Here is a sample of IOS.CSD after it has been edited:

```
;
;   LINK AND LOCATE THE I/O SYSTEM
;
;       SUBMIT  :fx:ios( date, loc_adrH )
;
;   where:
;       date      = the date
;       loc_adr   = address where the IOS will be located
;
;
;   File-Drivers
;
ASM86 :f1:itable.a86 DATE(%0) PRINT(:f1:itable.lst) &
    WORKFILES(:f1:,:f1:) OBJECT(:f1:itable.obj)
;
;
;   Device-Drivers
;
ASM86 :f1:idevcf.a86 DATE(%0) PRINT(:f1:idevcf.lst) &
    WORKFILES(:f1:,:f1:) OBJECT(:f1:idevcf.obj)
;
LINK86                          &
       :f1:ios.lib(istart),     &
       :f1:itable.obj,          &
       :f1:idevcf.obj,          &
       :f1:i550.lib,            &
       :f1:iooptI.lib,          &
       :f1:ios.lib,             &
       :f1:r4cinf.lib,          &
       :f1:rpifc.lib,           &
       TO :f1:ios.lnk           &
       MAP PRINT(:f1:ios.mp1)
;
LOC86                                       &
       :f1:ios.lnk TO :f1:ios               &
       MAP PRINT(:f1:ios.mp2)               &
       OBJECTCONTROLS(NOLINES,NOCOMMENTS,NOPUBLICS,NOSYMBOLS) &
       SEGSIZE(stack(0))                    &
       ORDER(classes(code, data))           &
       ADDRESSES(classes(code(0%1))
```

CONFIGURING THE iRMX 88 I/O SYSTEM FOR USE WITH iSBC 550 CONTROLLERS

Configuring the iRMX 88 I/O System for use with an iSBC 550 controller is a three-stage process, which assumes that you have compiled your application code and your MMX 88 configuration module. (A discussion of the MMX 88 configuration module is at the end of this appendix.) In the first stage, you carry on a dialogue with the iRMX 88 Interactive Configuration Utility (the ICU). This stage produces several files, including a SUBMIT file. In the second stage, you modify some of these files, including the SUBMIT file. The third stage consists only of executing the SUBMIT file. The result of the third stage is a ready-to-test application system.

Responding to ICU Prompts

This section provides you with the answers, in the order in which they are requested, to ICU questions that pertain to an application system that communicates with an Ethernet network. The following list, which contains those answers, makes sense only in the context of an ICU session. Keep it handy while you are carrying on your ICU dialogue.

- Double buffering is not used.
- The named file driver is not used.
- RQ$FORMAT is not used.
- The physical file driver is used.
- Whole-sector I/O is not used.
- DQ$READ is used.
- DQ$WRITE is used.
- DQ$SEEK is not used.
- When prompted with "DEVICE TYPE -***", answer "CUSTOM" and then provide answers as follows:
  - For Level, specify 0
  - For Interrupt Task Priority, specify 0.
  - For Interrupt Task Stack Size, specify 0.
  - For Data Size, use the default value of 256.
  - For Number of Units, specify 1.
  - For each of Device Initialization, Device Finish, Device Start, Device Stop, and Device Interrupt, use the default value.
- The prompt "ADDITIONAL DEVICE INFORMATION TYPE -***" marks the beginning of a series of questions that the ICU uses to fill in a device information table for the iSBC 550 device. The fields of this table are defined earlier in this appendix under the heading "Configuring an iRMX 86 I/O System for Use with iSBC 550 Controllers". The questions in this series are asked in pairs, one pair per field. First, you are asked for the data type of the field, and then you are asked for the numerical value that is to go into that field. A summary of the requested information is as follows:

Data Type of Field	Name of Field
WORD	init start addr offset
WORD	init start addr base
BYTE	host device id
BYTE	550 device id
WORD	RQ to 550 offset
WORD	RQ to 550 base
WORD	RQ from 550 offset
WORD	RQ from 550 base
BYTE	IDS base strt addr
BYTE	IDS length
WORD	550 interrupt port
WORD	550 port name
WORD	host port info
BYTE	timeout
BYTE	priority
BYTE	550 interrupt type

- Next, you are prompted for some procedure names that will go into the unit information table for the iSBC 550 device. Proceed according to:

Prompt	Response
INIT IO	i550INIT
FINISH IO	i550FINISH
QUEUE IO	i550QUEUE
CANCEL IO	i550CANCEL

- When the ICU prompts you for timer data, specify the default values.

- When the ICU prompts you for information about the Free Space Manager, specify the default values and names. Later, you will remove this information, because the Partitioned Memory Manager is used in place of the Free Space Manager. (Note that you can't convince the ICU that you don't need the FSM. It assumes that you do need the FSM, because the I/O System is part of your system.)

- When the ICU prompts you for information about tasks and exchanges, supply the information that is given under the heading "Linking and Locating for MMX 88" in Chapter 7 of this manual.

This ends the ICU session and completes the first stage of the configuration process. The result is a collection of files, some of which you will modify in the second stage of configuration.

Modifying Files Produced by the ICU

In this, the second stage of the configuration process, you modify the following files, where this appendix assumes that MYSUB is the name you supplied to the ICU as the name of your SUBMIT file and configuration file:

- :Fx:DEVICE.A86, where DEVICE is the default name. This is the file that describes the I/O devices that the I/O System will be communicating with.

- :Fx:MYSUB.A86. This is the file that specifies which iRMX 88 modules are required by your application.

- :Fx:MYSUB.CSD. This is the SUBMIT file that links together all of the modules that you have specified during and after your ICU session, and then produces a located system, ready for testing.

Modifying the Device File. You must make the following changes to the :Fx:DEVICE.A86 file:

- Insert the following lines immediately after the line that reads "extrn radcancelio: near":

```
extrn i550INIT:   near
extrn i550FINISH: near
extrn i550QUEUE:  near
extrn i550CANCEL: near
```

- Substitute "0" for each of "defaultstart" and "defaultinterrupt".

- At the end of the file, remove the line "dd ***".

Modifying the Configuration File. In order to remove the Free Space Manager and all references to it, you must make the following changes:

- Remove all references to each of the following:

RQFSMSTACKSIZE	RQFSAX
RQFSMPRIORITY	RQFSRX
RQFMGR	RQRECLAIM
RQFMGRTD	RQRECLAIMTD
RQFMGRSTACK	RQRECLAIMSTACK

  The only exception is that references to RQFSAX and RQFSRX should not be deleted from the Initial Exchange Table (IET).

- Decrement by two the number of tasks in the Create Table RQCRTB.

Modifying the SUBMIT File. Before executing the :Fn:MYSUB.CSD SUBMIT file, you must make the following changes to it:

- Add the line ":Fn:I88ios.LIB(isleep), &" to the first list of modules that are to be linked together. If you are compiling your PL/M-86 modules using the COMPACT size control, put this line immediately after the line ":Fn:I88COM.LIB, &". If you are compiling your PL/M-86 modules using the LARGE size control, put this line immediately after the line ":Fn:I88LAR.LIB, &".

- If you are using the LARGE size control, add the line ":Fn:i5588L.LIB(i550II, i550FC, i550QI), &" immediately after the line ":Fn:I88IOS.LIB, &" in the first list of modules that are to be linked together.

- Add the following lines to the "no publics except" portion of the second list of modules that are to be linked together:

```
" rqdeletetask,      &
  rqdeletesegment,   &
  rqdeleteregion,    &
  iosdataseg,        &
  rqcreatesegment,   &
  rqcreateregion,    &
  rqcreatetask,      &
  rqsendcontrol,     &
  rqreceivecontrol,  &
  rqsendmessage,     &
  rqsleep,           &
  iors_enqueue,      &
  iors_dequeue,      &
  rqreceivemessage,  &
  psadd,             &
  gettaskparms,      &"
```

- Add the Ethernet device driver library (either i5588L.LIB or i5588C.LIB, depending upon whether your are using the LARGE or COMPACT size control, respectively) to the last list of modules that are to be linked together. The order of this list should be as follows:

  Configuration object module
  Your application object modules
  The link module produced by the second LINK86 command
  The appropriate Ethernet device driver library
  The MMX 88 configuration module
  The appropriate MMX 88 libraries
  The remaining libraries produced by the ICU

When this is done, you are ready to run your SUBMIT file :Fn:MYSUB.CSD.

A SAMPLE MMX 88 CONFIGURATION FILE FOR THE HOST DEVICE

This section contains a sample MMX 88 configuration file for the (host) device that communicates with the iSBC 550 device. Note that this example exhibits the following properties, which are required of every MMX 88 configuration:

- The depth (RQINSIZE in DCM$ROM) of the request queue, for requests from the iSBC 550 device to the host device, must be 4.

- The system port name (SYSTEM$PORT$NAME in LPT$ROM) that the iSBC 550 device uses to reference the system port on the host device must be the same as that specified in the device information table for the host device.

- The device id (DEST$PORT$ID in DSDT) for the iSBC 550 device must not be zero. Port zero has special meaning to the iSBC 550 device and cannot be used by MMX 88.

- In the SFT structure corresponding to the iSBC 550 device, DEVICE$MODE must be SLAVE$DEVICE (=01H), INTR$TYPE must be IO$INTERRUPT (=03H), and INTR$VALUE must be 02H.

- If the interrupt type field in the iSBC 550 Start Command request block for the iSBC 550 device is 3 or 4 -- where 3 is the recommended value if the iSBC 550 device is to interrupt the host device, and 0 is recommended otherwise -- the CLR$INTR$TYPE field in the SFT should be 04H. Further, the values for the INTR$LOCATION and CLR$LOCATION fields of the SFT should corrspond to the values for which the iSBC 550 device is jumpered.

- Memory on the host device must be mapped so that all of the iRMX 88 free space is addressable by the iSBC 550 device.

The following example of an iMMX 88 configuration file utilizes two files (R3XMGR.LIT and R3CNFG.LIT) that are included with iMMX 88 and are not listed here. This configuration applies whether the COMPACT or LARGE size control is used to compile this configuration file.

```
R3CNFG:
DO;

$INCLUDE(:F1:R3XMGR.LIT)

$INCLUDE(:f1:R3CNFG.LIT)

   /*****************************************************************
   :                                                                :
   :                  SAMPLE iMMX 88 CONFIGURATION                  :
   :                                                                :
    *****************************************************************/

DECLARE

    MMX$DEVICES                          LITERALLY '2',
    DEV$0$DEST$PORTS                     LITERALLY '1',
    DEV$0$PORTS                          LITERALLY '1',
```

```
/* The following values are used for the DCM$ROM table */

OUTPUT$QUEUE$ADDRESS                    LITERALLY '2F150H',
OUTPUT$QUEUE$SIZE                       LITERALLY '04H',
OUTPUT$QUEUE$ENTRY$SIZE                 LITERALLY '04H',
INPUT$QUEUE$ADDRESS                     LITERALLY '2F040H',
INPUT$QUEUE$SIZE                        LITERALLY '04H',
INPUT$QUEUE$ENTRY$SIZE                  LITERALLY '04H',

/* The following values are used in the DSDT and LPT$ROM tables */

DEV$1$PORT$NAME                         LITERALLY '0101H',
DEV$1$DEV$ID                            LITERALLY '1',
DEV$1$PORT$ID                           LITERALLY '1',
DEV$0$PORT$NAME                         LITERALLY '0000H',
DEV$0$DEV$ID                            LITERALLY '0',
DEV$0$POOL$ID                           LITERALLY '0',
DEV$0$IDS$ID                            LITERALLY '0',

/* The following are used for CQMDLY and CQITWT */

COMMUNICATE$WAIT$TIME                   LITERALLY '400',
RESPONSE$WAIT$TIME                      LITERALLY '0100H',

/* The following values are used in the SFT table */

DEV$1$MODE                              SLAVE$DEVICE,
DEV$1$INTR$TYPE                         IO$MAPPED$INTR,
DEV$1$INTR$LOCATION                     LITERALLY '0A4H',
DEV$1$INTR$VALUE                        LITERALLY '02H,'
DEV$1$CLR$INTR$TYPE                     IO$WRITE$CLR,
DEV$1$CLR$INTR$LOCATION                 LITERALLY '0A4H',
DEV$1$CLR$INTR$VALUE                    LITERALLY '04H',

/* The following are used for CQSGLV and CQIDPD */

DEV$0$INT$LEVEL                         LITERALLY '4',
DEV$0$POLLING$PERIOD                    LITERALLY '200',

/* the following are used for CQIDSS and the IDST table */

MMX$INTER$DEVICE$SEGMENTS               LITERALLY '1',
DEV$0$IDS$0$OFFSET                      LITERALLY '0000H',
DEV$0$IDS$0$BASE                        LITERALLY '0000H',

/* The following fields are used for CQPLHS, CQBLKS, and the
                    PHLTBL and BLKTBL tables. /*

DEV$0$POOLS                             LITERALLY '1',
DEV$0$BLOCKS                            LITERALLY '1',
DEV$0$POOL$0$ADDR                       LITERALLY '1000H',
DEV$0$POOL$0$LENGTH                     LITERALLY '1F00H';
```

```
DECLARE CQDVCS BYTE PUBLIC DATA (MMX$DEVICES);

DECLARE CQSKTS BYTE PUBLIC DATA (DEV$0$DEST$PORTS);

DECLARE CQPRTS BYTE PUBLIC DATA (DEV$0$PORTS);

DECLARE CQMDLY WORD PUBLIC DATA (RESPONSE$WAIT$TIME);

DECLARE CQITWT WORD PUBLIC DATA (COMMUNICATE$WAIT$TIME);

DECLARE DSDT (DEV$0$DEST$PORTS) DSDT$TYPE PUBLIC
        DATA (DEV$1$PORT$NAME,
              DEV$1$DEV$ID,
              DEV$1$PORT$ID,
              DEV$0$DEV$ID,
              0,
              DEV$0$POOL$ID,
              DEV$0$IDS$ID);

DECLARE LPT$ROM (DEV$0$PORTS) LPT$ROM$TYPE PUBLIC
        DATA (DEV$0$PORT$NAME,
              DEV$0$POOL$ID);

DECLARE LPT$RAM (dev$0$ports) LPT$RAM$TYPE PUBLIC;

DECLARE DCM$ROM (MMX$DEVICES) DCM$ROM$TYPE PUBLIC
        DATA   (0,
                0,
                0,
                0,
                0,
                0,
                OUTPUT$QUEUE$ADDRESS,
                OUTPUT$QUEUE$SIZE,
                OUTPUT$QUEUE$ENTRY$SIZE,
                INPUT$QUEUE$ADDRESS,
                INPUT$QUEUE$SIZE,
                INPUT$QUEUE$ENTRY$SIZE);

DECLARE DCM$RAM (MMX$DEVICES) DCM$RAM$TYPE PUBLIC:

DECLARE CQSGLV BYTE PUBLIC DATA (DEV$0$INT$LEVEL);

DECLARE RQL4EX (28) BYTE EXTERNAL;

DECLARE CQLMEX POINTER PUBLIC DATA (@RQL4EX);

DECLARE CQIDPD WORD PUBLIC DATA (DEV$0$POLLING$PERIOD);
```

```
DECLARE SFT (MMX$DEVICES) SFT$TYPE PUBLIC
        DATA   (0,
                0,
                0,
                0,
                0,
                0,
                0,
               DEV$1$MODE,
                 DEV$1$INTR$TYPE,
                 DEV$1$INTR$LOC,
                 DEV$1$INTR$VAL,
                 DEV$1$CLR$INTR$TYPE,
                 DEV$1$CLR$INTR$LOC,
                 DEV$1$CLR$INTR$VAL);

DECLARE CQIDSS BYTE PUBLIC DATA (MMX$INTER$DEVICE$SEGMENTS);

DECLARE IDST (MMX$INTER$DEVICE$SEGMENTS) IDS$TYPE PUBLIC
        DATA (DEV$0$IDS$0$OFFSET,
              DEV$0$IDS$0$BASE);

DECLARE CQPLHS BYTE PUBLIC DATA (DEV$0$POOLS);

DECLARE PHLTBL (DEV$0$POOLS) POOL$TABLE$TYPE PUBLIC;

DECLARE CQBLKS BYTE PUBLIC DATA (DEV$0$BLOCKS);

DECLARE BLKTBL (DEV$0$BLOCKS) BLOCK$TABLE$TYPE PUBLIC
        DATA (DEV$0$POOL$ID,
              DEV$0$POOL$0$ADDR,
              DEV$0$POOL$0$LENGTH);
END R3CNFG;
```

APPENDIX C. MMX 80 DIAGNOSTICS

Two MMX 80 diagnostics are provided for trouble-shooting during the development process.

RQPBHX PORT DIAGNOSTIC

The RQPBHX diagnostic provides you with a method of determining whether memory is being improperly sent for reclamation by application tasks. The RQPBHX port is an iRMX 80 exchange dedicated to use by the Partitioned Memory Manager (PMM).

If an application task attempts to use a PMM request of type PMM$FREE$BLK$TYPE and specifies a memory$pool identifier for a pool not previously created, the PMM sends the message block to the RQPBHX port.

In order to check whether a message block has been sent to the RQPBHX port, use the statement

```
IF NOT( RQACPT( .RQPBHX ) = 0 ) THEN ...
```

and put an error-handling block of code after "THEN".

MEM$INIT$STATUS DIAGNOSTIC

The mem$init$status diagnostic allows you to determine whether the PMM successfully allocated its initial memory blocks, as defined in the configuration table PHLTBL.

Use the following code outline as an example of using the mem$init$status diagnostic:

```
    :
    :
$include(:f1:R1PMM.LIT)
$include(:f1:R1PMM.EXT)
$include(:f1:R1DIAG.EXT)
    :
    :
DECLARE
    dummy           ADDRESS;
    status          BYTE;
    bad$block       BYTE;
```

```
DO WHILE (status := mem$init$status( .bad$block)) = 0;
     dummy = RQWAIT( some$exchange$ptr, one$clock$tick );
     END;

IF status = PMM$no$space$type
THEN DO
     /*At this point, the bad$block variable contains the index into
     the PHLTBL table of the next initial block that would have been
     processed if no error had occurred.  The problem was that there
     was not enough memory allocated to the Free Space Pool (pool 0)
     to process the remaining initial block declaration(s).

     Note that if an initial block declaration specifies a
     non-existent memory pool, that pool is automatically created and
     the initial memory block is allocated to it.  However, to create
     the new pool, a message block of at least 32 bytes must be
     available in the Free Space Pool for PMM overhead.  This is the
     reason the PMM$no$space$type error is returned.*/
     END;
ELSE /* status = PMM$ok$type */ DO;
     :
     :
     END;
     :
     :
```

Figure C-1. MEM$INIT$STATUS Diagnostic Example

APPENDIX D. iMMX™ 800 CONDITION CODES

When an application task calls an iMMX 800 procedure, status information is returned to the calling task in the form of a condition code that indicates the successful or unsuccessful completion of the service. In the case of unsuccessful completion, the code indicates the nature of the problem.

The condition code mnemonics and their hexadecimal values are listed in Table D-1. For the mnemonics and values of other condition codes that can be returned to an executing task, refer to the appropriate iRMX operating system manuals.

Table D-1. iMMX™ 800 Condition Codes

MMX 80 and MMX 88 Condition Codes:

Message	Value
SYSTEM$SERVICE$READY	00H
SYSTEM$MESSAGE$DELIVERED	30H
UNKNOWN$SYSTEM$PORT	31H
SYSTEM$MESSAGE$COPY$DELIVERED	32H
SYSTEM$PORT$ACTIVE	33H
XFLAG$ERROR	34H
INSUFFICIENT$MEMORY	35H
SYSTEM$PORT$INACTIVE	37H
SYSTEM$PORT$DEAD	39H

MMX 86 Condition Codes:

Message	Value
E$SYSTEM$MESSAGE$DELIVERED	130H
E$UNKNOWN$SYSTEM$PORT	131H
E$SYSTEM$MESSAGE$COPY$DELIVERED	132H
E$SYSTEM$PORT$ACTIVE	133H
E$DESTINATION$CHANNED$MEMORY	135H
E$SYSTEM$PORT$INACTIVE	137H
E$SYSTEM$PORT$DEAD	139H
E$SOURCE$CHANNEL$MEMORY	141H
E$UNDEFINED$POOL	143H

INDEX

Underscored entries are primary references.

8255 Programmable Peripheral Interface 7-2, 7-9

Activate Port service 2-5, 3-10, 4-10, 5-10
ACTIVATE$SYSTEM$PORT A-38
alias addressing 2-8, A-9
asynchronous tasks A-7

BLKTBL 7-3, 7-4, 7-14
buffer A-3

channel 2-2, 2-3, 6-2, 7-4, A-6
clearing interrupts 7-3, 7-9, 7-42
Command Ready Queue A-7
concurrency 8-2
condition code 3-1, 4-2, 5-1, D-1
configuration 6-1, 7-1, B-14
connection 2-4, 3-3, 4-4, 5-3, 6-4
CONVERT$LOCAL$ADR A-12
CONVERT$SYSTEM$ADR A-12
CQACTV 2-5, 3-10, 4-12, 5-10
CQBLKS 7-4
CQDACT 2-5, 3-14, 4-15, 5-14
CQDVCS 7-1, 7-4
CQFIND 2-4, 3-3, 4-4, 5-3
CQGDPA 6-4
CQIDPD 7-3, 7-13
CQIDSS 7-2, 7-13
CQITWT 7-2, 7-12
CQLMEX 7-3, 7-13
CQLOSE 2-4, 3-9, 4-11, 5-9
CQMDLY 7-2, 7-12
CQPLHS 7-14
CQPRTS 7-1, 7-6
CQSGLV 7-3, 7-13
CQSKTS 7-1, 7-7
CQXFER 2-4, 3-5, 4-6, 5-5
creating memory pools 6-6

data structures 7-4, 7-15
data types A-11
DCM$RAM 7-4
DCM$ROM 7-2, 7-4
Deactivate Port service 2-5, 3-14, 4-15, 5-14
DEACTIVATE$SYSTEM$PORT A-39
device 1-1, 7-2, 7-7, A-3
device id 7-3
diagnostics C-1
DSDT 7-1, 7-2, 7-7
dual-port memory 2-7, A-9
DYING$CHANNEL A-25

Ethernet 1-3, B-1
Ethernet tasks B-4
example 1-1, 3-14, 3-15, 4-16, 4-17, 5-15, 5-16, 7-15, B-17
exchange 2-1, 3-12, 4-14

Find Port service 2-4, 3-3, 4-4, 5-3
FIND$SYSTEM$PORT A-35
Free Space Manager 1-4, 6-1
Free Space Pool 2-7, 6-2

generating interrupts 7-42

hardware configuration 7-42

I/O port 7-2, 7-3, 7-9
I/O-mapped interrupt 7-2, 7-3, 7-9, 8-2
ICU 7-34, 7-36, B-26
IDS base address A-9
IDS pointer A-9
IDST 7-2, 7-4, 7-13, A-24
iMMX 800 1-1
IN$TASK A-7, A-10, A-30, A-33
INCLUDE files 7-33
INIT$REQUEST$QUEUE A-16
interdevice message transfer 2-2, 2-4, 3-5, 4-6, 5-5
interdevice segment 2-7, 7-2, 7-4, 7-13, A-8, A-9, A-24
interrupt exchange 7-3, 7-13
interrupt level 7-3, 7-13
interrupts 7-2, 7-8, 7-42, 8-2
iRMX 80 1-1, 3-1, 7-34
iRMX 86 1-1, 5-1, 7-38
iRMX 86 Basic I/O System B-6, B-14
iRMX 86 Extended I/O System B-9
iRMX 88 1-1, 4-1, 7-36
iRMX 88 I/O System B-11, B-26
iSBC 544 board 1-3, 2-7, 7-42
iSBC 550 board 1-3, B-1
iSBC 550 request block B-5
iSBC 569 board 1-3, 2-7, 7-42
iSBC 80/24 board 1-3, 7-42
iSBC 80/30 board 1-3, 2-7, 7-43
iSBC 86/05 board 1-3, 7-43
iSBC 86/12A board 1-3, 2-7, 7-43, B-1
iSBC 86/14 board 1-3, 2-7, 7-44
iSBC 86/30 board 1-3, 2-7, 7-44, B-1
iSBC 88/25 board 1-3, 7-44
iSBC 88/40 board 1-3, 2-7, 7-45
iSBC 88/45 board 1-3, 7-45
iSBC board 1-1

linking 7-34, 7-36, 7-38, B-25
local port 2-2, 3-10, 3-14, 4-12, 4-15, 5-10, 5-14, 7-6
locating 7-34, 7-36, 7-38, B-25
LOCATION 4-1
LOCATION$OF 4-1

Lose Port service 2-5, 3-9, 4-11, 5-9
LPT$RAM 7-6
LPT$ROM 7-1, 7-2, 7-3, 7-4

mailbox 2-1, 5-12
MCBI 7-13
megabyte addressing 4-1
MEM$INIT$STATUS C-1
memory allocation 6-3
memory management 6-1, 7-14
memory pool 2-7, 6-1, 7-14, 7-39
memory pool creation 6-6
memory-mapped interrupt 7-2, 7-3, 7-8, 8-2
memory reclamation 6-5
message copying 2-8, 8-1
message reception 2-4, 3-12, 4-14, 5-12
message reception protocol 2-5
message sender/receiver model 2-1
message structure 3-1, 4-2, 6-3
message transfer 2-2, 2-4, 3-5, 4-6, 5-5
message transfer mechanics 2-8, 8-1
message transfer protocol 2-2
MIP A-1
MIP pointer A-9
MMX 80 1-1, 3-1, 7-34
MMX 86 1-1, 5-1, 7-38, B-1
MMX 88 1-1, 4-1, 7-36, B-1
Multibus Interprocessor Protocol A-1
Multibus interrupt 7-2, 7-9, 8-2
Multibus system bus 1-1, 8-1, B-1

Nucleus configuration 7-40

obtaining memory 6-1, 6-3
OUT$TASK A-7, A-10, A-25, A-28

Partitioned Memory Manager 2-7, 6-1, 7-14
peer device 3-6, 4-7, 5-6, 7-8, 7-10
performance 8-1
PL/M-80 3-1
PL/M-86 4-1, 5-1
PHLTBL 7-14
polling period 7-2, 7-3, 7-13, 8-2
pool 2-7, 6-1, 7-14, 7-39
pool id 6-1, 7-3
port 2-2, 7-2
Port Queue A-7
PTR$ADD A-11

QUEUE$GIVE$STATUS A-40

RECEIVE$BUFFER A-40
RECEIVE$COMMAND A-30
RECEIVE$RESPONSE A-32
reclaiming memory 6-5

RELEASE$GIVE$POINTER A-19
RELEASE$TAKE$POINTER A-21
request queue 2-3, 7-2, A-5, A-6, A-7
REQUEST$GIVE$POINTER A-18
REQUEST$TAKE$POINTER A-20
requesting memory 6-3
response queue A-7, A-25
response turnaround queue A-7
returning allocated memory 6-1, 6-5
root job configuration 7-38
RQ$CREATE$SEGMENT 6-1
RQ$DELETE$SEGMENT 6-1
RQ$RECEIVE$MESSAGE 2-4, 5-12
RQACPT 2-5, 3-12, 4-14
RQCXCH 3-10, 4-11
RQFLMX 6-3
RQFSAX 6-2, 6-3
RQFSRX 6-2, 6-3
RQPBHX 6-5, C-1
RQWAIT 2-5, 3-12, 4-14

SERVE$COMMAND$QUEUE A-27
SERVE$TURNAROUND$QUEUE A-26
service 2-3
SFT 7-2, 7-8
slave device 4-7, 5-6, 7-8, 7-10
socket A-3, A-23
software configuration 7-1
software requirements 1-4
status constants A-35
system port 2-2, 3-3, 3-9, 4-4, 4-11, 5-3, 5-9, 7-7, A-3
system time unit 7-12

TERM$REQUEST$QUEUE A-16
TIME$WAIT A-13
Transfer Message service 2-4, 3-5, 4-6, 5-5
TRANSFER$BUFFER A-36
transparent message transfers 2-8, 6-2

virtual interface A-4, A-7, A-35

wake-up address 7-3, 7-8

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 THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
5200 N.E. Elam Young Pkwy.
Hillsboro, Oregon 97123

OMO Technical Publications

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.