The RadiSys logo is a blue rectangular box with the word "RadiSys." in white serif font. A thin black line extends from the right side of the box, ending in a small circle, and then continues vertically down the page.

RadiSys.

iRMX[®]

Network User's Guide and Reference

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com
07-0625-01
December 1999

EPC, iRMX, INtime, Inside Advantage, and RadiSys are registered trademarks of RadiSys Corporation. Spirit, DAI, DAQ, ASM, Brahma, and SAIB are trademarks of RadiSys Corporation.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation and Windows 95 is a trademark of Microsoft Corporation.

IBM and PC/AT are registered trademarks of International Business Machines Corporation.

Microsoft Windows and MS-DOS are registered trademarks of Microsoft Corporation.

Intel is a registered trademark of Intel Corporation.

All other trademarks, registered trademarks, service marks, and trade names are property of their respective owners.

December 1999

Copyright © 1999 by RadiSys Corporation

All rights reserved.

Quick Contents

- Chapter 1. Introduction
- Chapter 2. iRMX-NET Overview
- Chapter 3. Network Access Using iRMX-NET
- Chapter 4. Using the Network
- Chapter 5. Example: Configuring an Administrative Unit
- Chapter 6. Examples: Configuring Multiple Operating Systems
- Chapter 7. Network Software Implementation
- Chapter 8. iNA 960 Topology and Addressing
- Chapter 9. The Multibus II Subnet and Routing Between Subnets
- Chapter 10. The Programmatic Interface
- Chapter 11. Using and Programming the Name Server
- Chapter 12. Programming the Transport Layer
- Chapter 13. Programming the Data Link Layer
- Chapter 14. Using the Network Management Facility
- Chapter 15. Remote Booting
- Chapter 16. Internetwork Routing
- Appendix A. iRMX-NET and iNA 960 Transport Configuration Values
- Appendix B. Data Flow in MIP and COMMputer Jobs
- Appendix C. iNA 960 Network Objects
- Appendix D. Related Documentation
- Appendix E. Network Error Messages
- Glossary
- Index

Notational Conventions

The references to system calls in the text and graphics use C syntax instead of PL/M (for example, the system call **receive_message** instead of **receive\$message**). If you are working in C, use the C header files, *rmx_c.h*, *udi_c.h* and *rmx_err.h*. If you are working in PL/M, use the *rmxplm.ext* and *error.lit* header files and use dollar signs (\$) in the system calls. Additional header files for network programs are listed in this manual.

This manual uses the following conventions:

- User input, command syntax and computer output are printed like this, in regular monospaced text.
- In examples combining user input and computer output, user input is printed like this, in bold monospaced text.
- System call names and command names appear in bold, like this.
- All numbers are decimal unless otherwise stated. Hexadecimal numbers include the H radix character (for example, 0FFH).
- Bit 0 is the low-order bit unless otherwise stated.
- The following iRMX Operating System layer abbreviations are used. The Nucleus layer is unabbreviated.

AL	Application Loader
BIOS	Basic I/O System
EIOS	Extended I/O System
HI	Human Interface
UDI	Universal Development Interface



Note

A note calls attention to an important fact.



CAUTION

A caution points out something that could damage your hardware or data.

Contents

1 Introduction

How to Use This Book	1
Networking Concepts and Terminology	2
Network Software Choices	3
iNA 960 Programmatic Interfaces	3
iRMX-NET	3
TCP/IP and NFS	5

2 iRMX-NET Overview

iRMX-NET Client and Server	7
Network Operation	8
The Name Server	8
The User Definition File	11
The Client Definition File	11
Network Security	12
Client-based Protection	12
Server-based Protection	13

3 Network Access Using iRMX-NET

Overview	17
Adding a Server to the Name Server Object Table	18
Choosing a Server Name	18
Entering Information Into the Object Table	18
Defining Network Users in the UDF	20
Accessing Other AUs	21
Backing Up the Master UDF File	21
Adding a Client to the CDF	22
Diagnostics	23
What's Next?	23

4	Using the Network	
	Accessing Remote Files	27
	Connecting to a File Server	27
	Attaching a Server.....	27
	Detaching a Server	28
	Listing Remote Connections	28
	Using Remote Files	28
	Copying Files Across the Network.....	29
	Copying the Master UDF	29
	Copying the CDF	30
	Making Local Files Accessible to Other Nodes.....	31
	Setting Up Public Directories.....	33
	Listing Public Directories.....	33
	Removing Public Directories	34
	Protecting Files on a Server.....	34
	What's Next?	35

5	Example: Configuring an Administrative Unit	
	Configuring the Systems	38
	Configuring the Master Node.....	38
	System 1: iRMX for PCs Node	38
	ICU-configurable Master Node.....	39
	Configuring the Other Nodes	39
	System 2: Multibus I, System 320.....	40
	System 3: Multibus II, System 520	40
	System 4: Multibus II, System 520	41
	System 5: PC Bus Platform.....	42
	Setting Up the Administrative Unit.....	43
	System 1	43
	Modcdf Example.....	44
	Systems 2 through 5	45

6	Example: Configuring Multiple Operating Systems	
	The DOS System.....	49
	Connecting a DOS Client to an iRMX Server.....	49
	On the iRMX Server	49
	On the DOS Client	50
	iRMX and DOS Interoperability	51
	The PCL2 Name Server	51
	DOS Client Restrictions	51

The UNIX System	53
Connecting a UNIX Client to an iRMX Server	53
On the iRMX Server	53
On the UNIX Client	53
Connecting an iRMX Client to a UNIX Server	54
On the UNIX Server	54
On the iRMX Client	54
Setting Up the Administrative Unit	54
iRMX and UNIX Nodes in Separate AUs	55
iRMX and UNIX Nodes in the Same AU	55
iRMX and UNIX Interoperability	56
SV-OpenNET Server Features and Restrictions	56
iRMX Server Restrictions	58
Connecting to Nodes on Older Versions of SV-OpenNET	58

7 Network Software Implementation

Hardware Environments	59
Software COMMputer and MIP Environments	60
Overview of iNA 960 Software	61
The iNA Layers	62
The Name Server	63
The Transport Layer	63
The Network Layer	63
The Data Link Layer	63
The Network Management Facility	64
The Programmatic Interface	64
Overview of iRMX-NET Software	64
Data Flow Through iRMX-NET and iNA 960 Software	66
Configuring the MIP	67

8 iNA 960 Topology and Addressing

The iNA 960 Network Topology	69
General Subnetwork Types	70
iNA 960 Subnetworks	71
Network Addressing	71
Network Service Access Point (NSAP) Address	72
Subnet Address	73
Internetwork Routing	73
iNA 960 Network Layer Addressing Schemes	74
Null2 Network Addressing	74
Static Internetwork Addressing	75

End System to Intermediate System (ES-IS) Network Addressing.....	75
Choosing a Network Layer Configuration.....	76

8 The Multibus II Subnet and Routing Between Subnets

Configuring Networks with the Multibus II Subnet.....	77
Routing Between Subnets.....	78
Definition of a Router.....	78
ES-IS vs. Null2 Jobs.....	78
ES-IS Routing.....	79
Ethernet Addresses in the Multibus II Subnet.....	80
Data Link Subsystem ID for the Multibus II Subnet.....	80
Name Server Search Domain.....	81
Overview of Setting up the Multibus II Subnet.....	81
Step 1: Mapping the Network.....	82
Using Only TCP/IP Outside the Multibus II Subnet.....	85
Step 2: Choosing the iNA 960 Jobs.....	86
Step 3: Configuring Jobs in the ICU.....	88
Step 4: Creating a Loadable Network Job.....	89
Step 5: Using Loadable Jobs.....	90
Step 6: Changing Subnet IDs on Other Systems.....	91
Step 7: Modifying the net/data File.....	92
Step 8 - 10 Overview: Configuring iNA 960 Routing.....	93
Using Inamon to Configure Routing.....	93
Step 8: Establishing ES and IS Hellos.....	94
Step 9: Getting the NET and Subnet Information.....	96
Step 10: Setting Up the iNA 960 Static Routing Tables.....	98
Step 11: TCP/IP Configuration.....	104
Increasing Performance for Remotely-Booted Boards.....	105

10 The Programmatic Interface

Referencing Data Buffers in Request Blocks.....	107
Using Addresses in iNA 960 Request Blocks.....	108
Translating Pointers.....	108
Limitations on Buffer Size.....	108
Interface Libraries and Link Sequences.....	109
Include Files.....	109
Programming with Structures.....	110
Using the cq_ System Calls.....	111
Exception Handling.....	112
System Calls to iNA 960.....	113
cq_comm_multi_status.....	114

cq_comm_ptr_to_dword.....	116
cq_comm_rb.....	117
cq_comm_status.....	121
cq_create_comm_user.....	123
cq_create_multi_comm_user.....	124
cq_delete_comm_user.....	126

11 Using and Programming the Name Server

The Name Server Object Table.....	131
Adding an Object to the Name Server Object Table.....	134
Loading Objects from the :sd:net/data File.....	135
Editing the :sd:net/data.ex File.....	136
Syntax of the :sd:net/data File.....	138
Other Name Server Operations.....	141
Deleting an Object from the Name Server Object Table.....	141
Obtaining Local Name Server Information.....	141
Obtaining Remote Name Server Information.....	141
Object Table Entries at Initialization.....	142
Location of the Name Server.....	146
Request Block Arguments.....	146
Example Software.....	147
Name Server Commands.....	148
ADD_NAME.....	150
ADD_SEARCH_DOMAIN.....	153
CHANGE_VALUE.....	155
DELETE_NAME.....	157
DELETE_PROPERTY.....	159
DELETE_SEARCH_DOMAIN.....	161
GET_NAME.....	163
GET_SEARCH_DOMAIN.....	166
GET_SPOKESMAN.....	168
GET_VALUE.....	170
LIST_TABLE.....	173

12 Programming the Transport Layer

Transport Services.....	175
Virtual Circuit Service.....	177
Example Software.....	178
Datagram Service.....	178
Buffers.....	178
Buffer Addressing.....	178

TSAP Address Buffer.....	179
Contiguous Buffers.....	184
Noncontiguous Buffers.....	184
ISO Reason Codes.....	185
Virtual Circuit Commands.....	185
Commands to Establish a Connection	186
Commands for the Data Transfer Phase	186
Posting Receive Buffers for Virtual Circuits.....	187
Commands to Terminate a Connection	188
Datagram Commands	188
Posting Receive Buffers for Datagrams	189
Transport Service Commands	189
ACCEPT_CONNECT_REQUEST	190
AWAIT_CLOSE	193
AWAIT_CONNECT_REQUEST_TRAN	
AWAIT_CONNECT_REQUEST_CLIENT	196
CLOSE	206
OPEN	209
RECEIVE_ANY	211
RECEIVE_DATA	214
RECEIVE_DATAGRAM.....	217
RECEIVE_EXPEDITED_DATA	220
SEND_CONNECT_REQUEST.....	223
SEND_DATA/SEND_EOM_DATA	229
SEND_DATAGRAM	233
SEND_EXPEDITED_DATA.....	236
STATUS.....	239
WITHDRAW_DATAGRAM_RECEIVE_BUFFER.....	249
WITHDRAW_EXPEDITED_BUFFER.....	251
WITHDRAW_RECEIVE_BUFFER.....	253

13 Programming the Data Link Layer

Overview of the Data Link Layer.....	255
The External Data Link (EDL) Interface.....	256
The RawEDL Interface	256
iNA 960-Supported Hardware Subnets and Protocols	257
LSAP Identifiers.....	258
Data Link Commands.....	260
CONFIGURE.....	264
CONNECT.....	266
DISCONNECT.....	269
FLUSH	271

IA_SETUP.....	272
MC_ADD	274
MC_REMOVE.....	276
POST_RPD	278
RAW_POST_RECEIVE	282
RAW_TRANSMIT	286
READ_CLOCK.....	288
TRANSMIT.....	289

14 Using the Network Management Facility

NMF Services.....	293
NMF Operation	295
Managers and Agents	295
Local Versus Remote NMF Operation	297
Local Operation.....	297
Remote Operation	298
NMF Communications Services.....	298
Using NMF Commands.....	300
Net Agent Connection Commands	300
Layer Management Commands.....	301
NMF Object IDs.....	301
Using Layer Management Commands	302
Event Notification	302
NMF Events	303
Debugging Commands	303
Maintenance Commands	303
Remote Load Operations	304
The NMF Commands	304
ATTACH_AGENT	306
AWAIT_EVENT.....	309
DETACH_AGENT	312
DUMP	313
ECHO.....	316
READ_AND_CLEAR_OBJECT	318
READ_MEMORY/SET_MEMORY	319
READ_OBJECT/SET_OBJECT READ_AND_CLEAR_OBJECT	321
SET_MEMORY	327
SET_OBJECT	328
SUPPLY_BUFFER	329
TAKEBACK_BUFFER	332

15	Remote Booting	
	Hardware and Software Requirements	333
	Overview of Remote Booting	335
	Configuring the Load Files	337
	Operating System Boot File	338
	Generating an OS Boot File	338
	Load-time Configuration File	340
	Remote Third Stage Bootstrap Loader	341
	iNA 960 Load File	342
	Generating a First Stage EPROM for the Boot Client	342
	Creating a First Stage for EtherExpress 16 or EWENET	343
	Using the iPPS PROM Programmer	345
	Installing the EPROM	346
	Configuring the Remote Boot Server	346
	Creating the <i>ccinfo</i> File	346
	Class Codes	347
	Generating the <i>ccinfo</i> File	348
	Loading the Boot Server	349
	Installing the Load Files	350
	Configuring the File Server	350
	Loading Server Names into the Name Server Database	351
	Adding Client Names to the CDF	352
	Adding Server Names to the <i>:config:terminals</i> File	352
	Remote Boot Start	353
	Booting Multibus I Systems	353
	Booting Multibus II or PC Bus Systems	353
	System Initialization on a Diskless Node	353
	If Remote Booting Fails	355
	Troubleshooting	355
	Creating Custom Server Applications	358
	Boot Request and Response	358
	Loading Operation	359
	Boot Module Format	361
	Using SUPPLY_BUFFER and TAKEBACK_BUFFER	362

16	Internetwork Routing	
	Internetwork Routing Protocols	365
	Static Routing	365
	ES-IS Routing	365
	Using Static and ES-IS Routing Together	366
	Routing Tables	366

Application Access to Routing Tables.....	367
Reading and Setting Static Routing Objects.....	368
Command and Response Buffers for Static Routing	368
Command Buffer.....	369
Response Buffer	370
Field Descriptions for Command and Response Buffers.....	371
Reading and Setting ES-IS Routing Objects.....	374
Command and Response Buffers for ES-IS Routing.....	376
Command Buffer.....	376
Response Buffer	377
Field Descriptions for Command and Response Buffers.....	377
The Local End System Table Structure.....	379
The Intermediate System Table Structure	380
The Static Intermediate System Table Structure	381
The Reachable NSAP Address Table Structure	382
The Subnet Table Structure.....	383
The Local NSAP Address Table Structure.....	384

A iRMX-NET and iNA 960 Transport Configuration Values

Files Containing iNA 960 Transport Software	385
iNA 960 Download Files.....	385
iNA 960 COMMputer Jobs.....	388
Configuration of iNA 960 MIP Jobs.....	388
Configuration of iRMX-NET Jobs	388

B Data Flow in MIP and COMMputer Jobs

Data Interchange with the MIP.....	391
Multibus I and PC Bus MIP	393
Multibus II MIP.....	394
Data Interchange in a COMMputer Job.....	395

C iNA 960 Network Objects..... 397

D Related Documentation 415

E Network Error Messages

System Initialization Error Messages	417
MIP Error Codes.....	419

Glossary 433

Index 439

Tables

Table 7-1. iNA 960 Services and ISO Specifications	61
Table 9-1. iNA 960 COMMputer Jobs for the Multibus II Subnet	86
Table 9-2. Configuring ES and IS Hellos.....	94
Table 10-1. System Calls for Access to iNA 960 and the Name Server	113
Table 11-1. Property Types for the Name Server	133
Table 11-2. Object Table Entries	142
Table 11-3. Name Server Commands	148
Table 11-4. Name Server Response Codes	149
Table 12-1. Transport Layer Commands	176
Table 12-2. TSAP Address Buffer Field Values	182
Table 12-3. ISO Reason Codes	185
Table 12-4. Maximum Total Buffer Lengths	231
Table 13-1. Data Link Commands	261
Table 13-2. Data Link Subsystem IDs	262
Table 13-3. IEEE 802.3 Response Codes.....	262
Table 14-1. Network Management Facility Commands	305
Table 15-1. Boot Client Systems	334
Table 15-2. Load Files for Remote Booting.....	337
Table 15-3. ICU Definition Files for Remote Booting.....	338
Table 15-4. Remote Third Stage Bootstrap Loader Files	341
Table 15-5. Class Code Ranges and Defaults	347
Table 15-6. Remote Load File Translation	348
Table 15-7. Default Directories for Load Files	350
Table A-1. iNA 960 Download Files	386
Table A-2. iNA 960 Download File Configuration.....	387
Table A-3. MIP Job Configuration	388
Table A-4. iRMX-NET Configuration.....	389
Table C-1. 802.3 Data Link Objects	398
Table C-2. 802.3 Data Link Objects With the 825595TX Component	399
Table C-3. 802.3 Data Link Objects With the DEC21143 Component.....	400
Table C-4. 802.3 Data Link Objects for the Multibus II Subnet	401
Table C-5. IP Network Layer Objects.....	402
Table C-6. Router Objects - Static	402
Table C-7. Router Objects - ES-IS.....	403
Table C-8. Transport Layer Objects - Virtual Circuit Connection Independent.....	406
Table C-9. Map 2.1 Transport Objects.....	409
Table C-10. Map 2.1 Transport Objects - Virtual Circuit Connection Dependent.....	411
Table C-11. Map 2.1 Transport Objects - Transport Datagram	413
Table C-12. NMF Objects.....	413
Table C-13. Network Layer Events	413
Table C-14. Transport Layer Events.....	414
Table E-1. MIP Error Codes	419

Figures

Figure 1-1. iRMX-NET Interoperability with Other OpenNET Systems	4
Figure 2-1. Name Server Operation.....	9
Figure 2-2. Name Server Example.....	10
Figure 2-3. Client-based Protection Within an AU	12
Figure 2-4. Server-based Protection Across AUs	14
Figure 3-1. Network Setup.....	16
Figure 4-1. Remote File Access.....	26
Figure 4-2. Public Directories as Seen from a Client.....	31
Figure 5-1. Single Administrative Unit.....	37
Figure 6-1. The OpenNET Network	47
Figure 6-2. Multiple Operating System Network.....	48
Figure 7-1. ISO OSI Model	59
Figure 7-2. iNA 960 Software Layers.....	62
Figure 7-3. iRMX-NET Data Flow on COMMputer Systems	66
Figure 7-4. iRMX-NET Data Flow on COMMengine Systems.....	66
Figure 8-1. A Single Subnetwork	69
Figure 8-2. Two Interconnected Subnetworks	70
Figure 9-1. Mapping Subnets.....	82
Figure 9-2. Mapping Subnets with an Internetwork.....	84
Figure 9-3. Mapping Subnets for TCP/IP Access, but no iNA 960 Access	85
Figure 9-4. Example iset.csd File	94
Figure 9-5. Routing Information on a Single External Network	96
Figure 9-6. Example Routing Information on a Single External Network	100
Figure 9-7. Routing Information on Multiple External Networks.....	102
Figure 11-1. The Name Server Object Table	132
Figure 11-2. The :sd:net/data.ex File	136
Figure 12-1. TSAP Address Format	181
Figure 12-2. Connection Request Consideration Policy.....	205
Figure 13-1. Data Link Interface	259
Figure 14-1. A Typical Net Manager/Net Agent Interaction	296
Figure 14-2. A Typical iNA 960 Network	297
Figure 15-1. Remote Booting the iRMX III OS, Start and Finish.....	335
Figure 15-2. Remote Booting a Diskless Node	336
Figure 15-3. The :sd:net/ccinfo.bdf File	347
Figure B-1. MIP Protocol Model	391
Figure B-2. Multibus I and PC Bus MIP Model	393
Figure B-3. Multibus II MIP Model.....	394
Figure B-4. COMMputer MIP Model.....	395

Introduction 1

Network User's Guide and Reference presents a number of networking options for iRMX[®] computers. iRMX systems can access the network through iRMX-NET, standalone iNA 960, or TCP/IP software and NFS, all of which are provided with the iRMX Operating Systems (OSs). This manual is primarily an introduction and reference to iNA 960 and iRMX-NET.

See also: *TCP/IP and NFS for the iRMX Operating System*

How to Use This Book

This manual contains a variety of information for users of iRMX networks, application developers, and network administrators. Most of the earlier chapters cover the installation, configuration, and use of iRMX-NET, which provides transparent file access to any of Intel's family of OpenNET Local Area Network (LAN) products. Most of the later chapters cover the interface to iNA 960, which provides a programmatic interface to the ISO/OSI Transport software. iNA 960 is the underlying software in iRMX network jobs.

Use this guide to determine which parts of the manual you should read.

If you are:	Refer to:
An iRMX-NET user	Chapters 1-4
Using multiple operating systems	Interoperability information in Chapter 6, in addition to other chapters
A network administrator	Chapters 1-5, 7, 9, 11, 14-16, Appendices A and C
Managing multiple operating systems.....	Chapter 6, in addition to other chapters
An application developer	Chapters 1, 2, 7-16, Appendices A-E

Networking Concepts and Terminology

This manual uses these networking concepts and terminology:

A user has a login on a computer system, which is called a *node* in the network. The node you are logged into is the *local node*; any other one is a *remote node*. The nodes are connected into a *Local Area Network (LAN)* by some type of physical connection, such as Ethernet.

There are two levels of networking software that provide communications between nodes:

- *Transport software* like iNA 960 and TCP/IP allows you to write applications that transfer data between nodes, typically by establishing a *virtual circuit* between the nodes. TCP/IP software includes utilities to transfer files between nodes or to log in to a remote node.
- *Transparent file access* is provided by iRMX-NET running on iNA 960 software or NFS running on TCP/IP. Transparent file access is the ability to work on files on a remote node as if they were local to your own system.

The function of a network is to allow computers to share *resources*, such as files, printers, tape drives, diskette drives, and modems. A computer that enables other nodes to use its resources is a *server* (or server node). A computer that accesses the resources of another node is a *client* (or client node). A node may be both a server and a client.

A network can be set up with one or more *dedicated servers*, computers used exclusively to provide resources to the other nodes, which act as clients. Alternatively, all the nodes can provide resources for each other, so each one is both a server and a client, in a *peer-to-peer relationship*.

The glossary at the back of the manual provides additional terms and definitions.

See also: [Glossary, *Introducing the iRMX Operating Systems*](#)

Network Software Choices

These network software packages are provided with the OS:

Transport Software	iNA 960 programmatic interface to ISO OSI protocol	TCP/IP for Internet protocol (requires iNA 960 on iRMX systems)
Transparent File Access	iRMX-NET client and server (requires iNA 960)	NFS (requires TCP/IP)

iNA 960 Programmatic Interfaces

If you only want a programmatic interface to Transport services from your application, instead of transparent file access by users at the iRMX command line, use iNA 960 jobs without loading iRMX-NET jobs. By themselves, the iNA 960 jobs do not provide the iRMX-NET interface to the remote file driver, or network user administration.

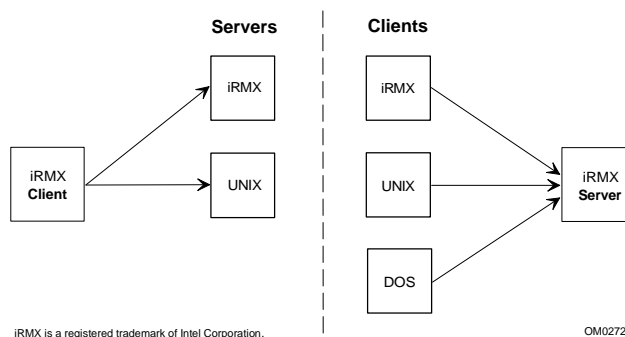
See also: Network Software Implementation, Chapter 7;
i.job, System Configuration and Administration*

iRMX-NET

iRMX-NET is distributed as part of the iRMX OS software, so this option is always available on any iRMX computer with a compatible hardware environment.

See also: Network Software Implementation, Chapter 7

One of the major features of iRMX-NET is that it enables iRMX computers to interconnect and interoperate easily with other OpenNET file servers and clients running a variety of OSs, as shown in Figure 1-1.



Arrows indicate flow of resource requests.

Figure 1-1. iRMX-NET Interoperability with Other OpenNET Systems

OpenNET cannot connect iRMX clients to DOS file servers. These software modules do not use the same communication protocols.

Each node on iRMX-NET can be used as a server and a client simultaneously. This provides great flexibility and saves the cost of a dedicated file server. On the other hand, it increases administrative duties and can slow response times.

iRMX-NET makes it possible for users to access remote files with the same Human Interface (HI) commands used for the equivalent operations on local files. This is called transparent file access.

On computers running DOSRMX, the iRMX-NET file server provides remote access to both the iRMX and the DOS file systems. Remote clients access the DOS file system through the iRMX-NET file server.

See also: iRMX-NET Overview, Chapter 2;
 Network Access Using iRMX-Net, Chapter 3;
 rnetserv.job and *remotefd.job*, *System Configuration and Administration*

TCP/IP and NFS

TCP/IP network software is available on any computer running the iRMX OS. TCP/IP utilities enable users to access other computers on the network to transfer files and log in to remote systems.

TCP/IP software provides the advantage of industry standard networking protocols that allow interoperability with most other OSs. Administrators of multiple OS networks, as well as many users, are likely to be familiar with TCP/IP networks.

For transparent file access using TCP/IP protocols, NFS software is also included with the iRMX OS.

The installation, configuration and use of TCP/IP and NFS software is covered in a separate manual.

See also: *TCP/IP and NFS for the iRMX Operating System; ip.job, rip.job, tcp.job, udp.job, mountd.job, nfsd.job, nfsd.job, and pmapd.job, System Configuration and Administration*



This chapter is a general description of how iRMX-NET works. It discusses the software, network operation, and security methods. If you are an iRMX-NET user, this chapter provides the background necessary for setting up the iRMX-NET software on your computer. If you are a network administrator or application developer, read this as an introduction to the more detailed discussions of these topics in later chapters.

See also: Programming the Name Server, Chapter 11;
Network Software Implementation, Chapter 7

iRMX-NET Client and Server

iRMX-NET includes these jobs, which you can configure into the OS or load separately:

iRMX-NET Client Contains the iRMX-NET file consumer and the remote file driver (RFD) The loadable version of this job is *remotefd.job*.

iRMX-NET Server The file server. The loadable version of this job is *rnetsrv.job*.

You can use either the client or the server separately or run both on the same system. These jobs require that you also run the appropriate iNA 960 job for your board.

See also: Network Software Implementation, Chapter 7;
Loading network jobs, *System Configuration and Administration*;
Network configuration, *ICU User's Guide and Quick Reference*

Network Operation

The operation of iRMX-NET depends on information maintained in three places:

- The *Name Server* maps the names of network services to their addresses. It contains information about all the servers on the network.
- The *User Definition File (UDF)* on each node contains names and other information about network users that can access that node. It is used to validate requests for remote access.
- The optional *Client Definition File (CDF)* on a server contains the client names and passwords of client nodes that can access that server. It is used to validate requests for remote access.

The Name Server

To be accessible, every server node must register its file server name with the Name Server. The Name Server is a subsystem of iRMX-NET that dynamically maps the names of network services to their addresses. This allows clients to find other computers on the network.

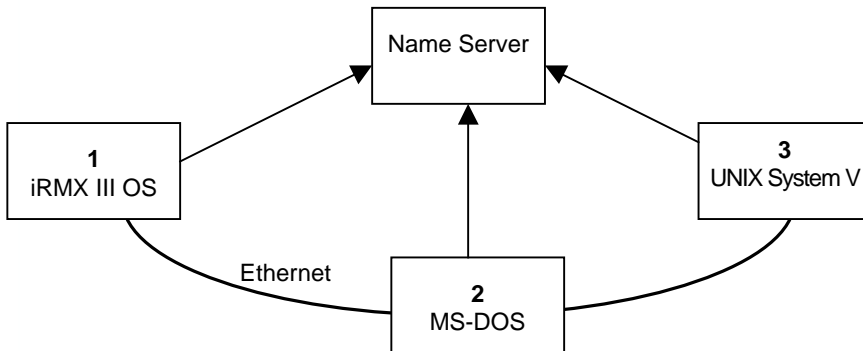
Dynamic name resolution means you can assign a server name to a computer at any time. If it already has another server name, you can add a second name. Once the new name is registered with the Name Server, remote users can access the node by using that name. They do not have to remember the network address, although they can obtain the address associated with this server name, if they wish. This process eliminates the administrative overhead of maintaining predetermined server names and network addresses in a central file.

A node that is used strictly as a client does not have to be registered with the Name Server. It may be, however, so remote users can find its net address.

Figure 2-1 shows the operation of the Name Server. Each system in the figure runs its own Name Server software. The various Name Servers work together across the network to provide name resolution to each node.

If your network has multiple subnets, the Name Server operates by default only on the range of subnet IDs preconfigured into each iNA 960 job. However, you can configure the default subnet IDs or extend the *domain* of subnets searched by the Name Server.

See also: Multibus II subnets, Chapter 9



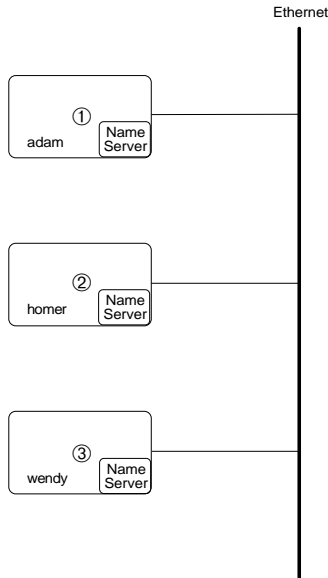
The Name Server consists of all the spokesman systems on the LAN. It maps system names to Ethernet addresses.

Figure 2-1. Name Server Operation

The Name Server works by allowing each node on the network to maintain a table of *network objects*. These objects are resources available to remote clients, like file servers, print servers, or virtual terminal servers. The object table lists the names of network objects and the attributes, or properties, associated with each object, such as its address and the type of service it offers. In addition to services provided by the local node, the object table can include the services of other nodes on the network. The only restriction is that no other object table on the network can have an object listing with the same name and property type (except for a few initial objects). A node that lists another computer's network services in its own Name Server object table is a *spokesman* for those services. A spokesman is required for any non-iRMX server that does not have Name Server capability.

A server name listed in a node's Name Server object table is visible to the entire network. When you ask for information about a server, your computer sends the request to the other computers on the network. They search their object tables for the requested information, and the computer that finds it sends the information back to your computer. Figure 2-2 shows an example of finding a server on the network through the Name Server.

See also: Programming the Name Server, Chapter 11



W-2950

1. System 1 sends a message for a system named "wendy".
System 1's Name Server broadcasts a message asking if there's a system named "wendy".
2. System 2's Name Server ignores the message, since it is not a spokesman for "wendy".
3. System 3's Name Server replies that it is named "wendy" and sends its Ethernet address.
System 1 establishes a connection to System 3, using the Ethernet addresses, then transfers the message.

Figure 2-2. Name Server Example

The User Definition File

Every iRMX computer has a UDF, which is not a special networking file, but part of the iRMX OS itself. The UDF is an ASCII text file with users' logon names, encrypted passwords, and user IDs. To facilitate connections to UNIX workstations on OpenNET networks, a user definition can also include a group ID, UNIX home directory, UNIX login shell, and a comment.

To access the network, you must be a *verified user*, that is, your logon name and password must be recognized as belonging to a valid network user. Your local node does this validation by checking for your name and password in the local UDF before

allowing you to log on. When you request network access, the remote server does another validation, in one of two ways. First it checks the identity of your local node in its own CDF. If it does not recognize the computer, then it checks its own UDF for your logon name and password.

See also: **password** command, *Command Reference*;
UDF, *System Configuration and Administration*

The Client Definition File

You can access a remote server without being checked for a valid logon name and password if your local node is a *verified client*, that is, its client name and password are recognized by the server as belonging to a valid client. The server does this validation by checking for the client name and password in its CDF.

The CDF is an ASCII text file maintained on every server, listing the client names and encrypted passwords of clients that access the server. CDFs can be used to simplify network maintenance when multiple nodes are grouped together into one *Administrative Unit* (AU). Clients in other AUs are not listed in the CDF, so connections across AUs require user validation.

You can change the default client name and password in the *rmx.ini* file or in the ICU, if there is one. A node's client name is not necessarily the same as its server name, which is listed in its Name Server object table.

See also: **modcdf** command, *Command Reference*;
CDF, *System Configuration and Administration*

Some other OSs (such as UNIX) use the term *subnet* or *subnetwork* in the way that the iRMX OS uses AU. In general, you can think of an AU as a subnet. However, the term subnet in iRMX documentation also refers to the iNA subnet address that is defined by the Network Layer in the iNA 960 Transport Software. There is no relation between the two uses of subnet.

See also: iNA Topology and Addressing, Chapter 8

Network Security

iRMX-NET provides two levels of network security: client-based protection and server-based protection.

Client-based Protection

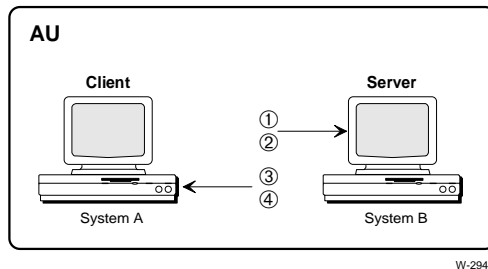
To provide *client-based protection* within an AU, all the nodes should have identical UDFs and all server nodes should have identical CDFs. The network administrator can maintain a Master UDF with verified users on one central master node, for

distribution to the other nodes in the AU. The administrator can also maintain a list of verified clients on the master node, which can be added to the CDFs on the AU's servers.

For client-based protection within an AU:

- The client system uses the UDF to validate the user
- The server system uses the CDF to validate the client system
- One system maintains a master copy of the UDF and CDF

Figure 2-3 illustrates the process of client-based security.



W-2944

1. User validation in UDF
2. User request transmitted
3. Client validation in CDF
4. Request granted

Figure 2-3. Client-based Protection Within an AU

A client-based security system requires these steps:

1. The user at System A is validated from the UDF at System A. This validation occurs when the user logs onto System A.
2. If step 1 is successful and the user attempts to access a file on System B, the client transmits the user's request, along with the client name and password for System A, to System B.
3. The server verifies the client name and password of System A using the CDF located at System B. If the client name and password of System A are valid, the server assumes that the user at System A is also valid.
4. If step 3 is successful, the user's request is processed.

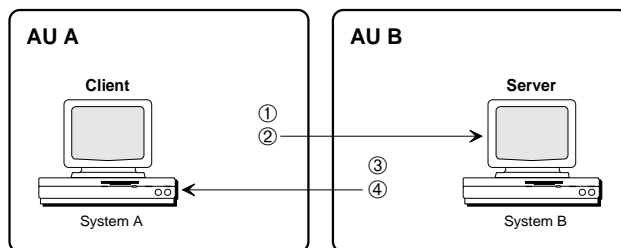
Server-based Protection

If System A's client name and password are not in the server's CDF, step 3 in the last section fails. The connection can still be made, however, if the server can determine that the user on System A is a verified user. This method of confirming users prior to granting remote file access is *server-based protection*, which offers the highest level of security.

For server-based protection:

- The user's name and password must be identical in both UDFs.
- The user ID and group ID may be different. A user named Jill could have ID 5 on one system and ID 20 on another. If the passwords on both systems are the same, Jill can access files between systems.
- The client system uses its UDF to validate the user.
- The server system uses its UDF to validate the user.

Figure 2-4 illustrates the process of server-based security.



W-2945

1. User validation in UDF
2. User request transmitted
3. User validation in UDF
4. Request granted

Figure 2-4. Server-based Protection Across AUs

A server-based security system requires these steps:

1. The user at System A is validated from the UDF at System A. This validation occurs when the user logs onto System A.
2. If step 1 is successful and the user attempts to access a file on System B, the client transmits the user's request, along with the client name and password for System A, to System B.

3. The server is not able to verify the client name and password of System A, using the CDF located at System B.
4. If step 3 fails, the server verifies the user's name and password, using the UDF located at System B.
5. If step 4 is successful, the user's request is processed.

For user validation to succeed, the user must be defined in both AUs. The UDFs on System A and System B must list identical user names and passwords, but the user IDs can be different.



Network Access Using iRMX-NET

This chapter describes how to set up iRMX-NET network access. This discussion focuses on setting up the network files to automatically log on to the network at boot time. However, there are many ways to set up and manage a system, and you may prefer to do some of these operations from the command line or through the programmatic interface.

See also: The Programmatic Interface, Chapter 10

The instructions and examples assume that the OS software has been installed in the default directories specified in the installation instructions. If your software installation is nonstandard, change the pathnames as needed.

Before you begin, make sure that:

- The iRMX OS is running.
- The NIC appropriate for this bus configuration has been installed.
- The NIC is connected to the Ethernet LAN hardware.

Because the network is part of the OS, some of the network setup is normally done during general installation or system configuration. The network configuration is set up in these places, depending on the OS:

- */net/data* file
- *:config:loadinfo* file for loadable network jobs
- Interactive Configuration Utility (ICU) for first-level network jobs
- *:config:rmx.ini* file for the DOSRMX and iRMX for PCs OSs
- *bps* file for the iRMX III OS on a Multibus II system

That level of configuration should already be complete by the time you begin the steps in this chapter.

See also: AU configuration and setup example, Chapter 5;
Installation, *Installation and Startup*;
Loading network jobs, *System Configuration and Administration*;
ICU User's Guide and Quick Reference

Figure 3-1 gives an overview of connecting to the network. The lighter boxes show the steps covered in this chapter. The steps in the darker boxes are discussed elsewhere in this or other manuals.

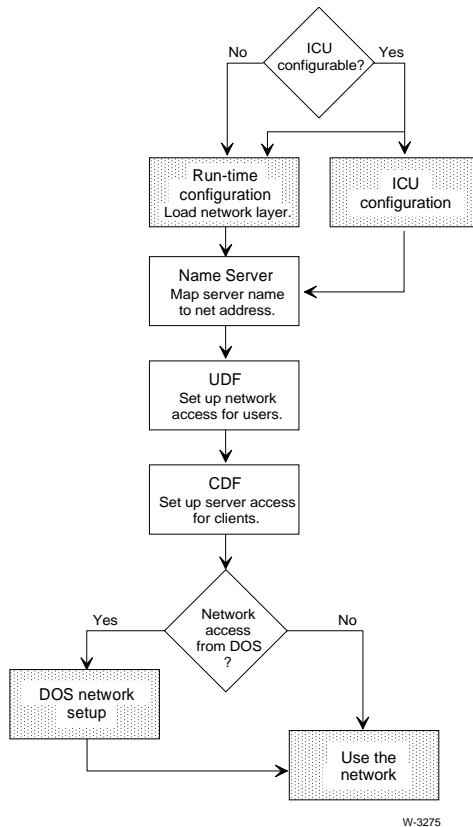


Figure 3-1. Network Setup

Overview

The process of obtaining network access through iRMX-NET varies, depending on how you will use the node and what version of the OS it runs. These sections provide the details of the various setup options, but here is a brief overview. You need to do one or more of these:

Client only or
client/server node

If the AU includes other nodes, define your user name in the Master UDF for distribution to all nodes. Add the client name to the CDF on servers within the AU.

Server only or
client/server node

Add a server name for this computer to the Name Server object table. If the AU includes other nodes, and you will use this computer as a client, define your user name in the Master UDF for distribution to all nodes. Add the other client names in the AU to the local CDF file.

Node
communicating
outside the AU

In addition to the previous steps, make sure this system has at least one user name and password in common with each node you will communicate with outside the AU.

DOSRMX node

In addition to the previous steps, make sure the OS is set up to load iRMX-NET at boot time. This gives you network access from the iRMX OS.

See also: *Using iRMX-NET in a DOS Environment, System Configuration and Administration*

Adding a Server to the Name Server Object Table

If the computer you are setting up will be used strictly as a client, you can skip this section.

To make a server visible to the entire network, you must catalog its node name, transport address, and the type of services it provides with the Name Server. Then any other node on the network can contact this server, simply by supplying its server name. This discussion explains the simplest case: adding the local node name to its own Name Server object table. However, there are many other things that can be done with the Name Server.

See also: Programming the Name Server, Chapter 11

Choosing a Server Name

The server name must be unique on the network, not just within the AU. The name is not case-sensitive; that is, the Name Server sees COMMSYS, Commsys and commsys as the same name. If you try to enter a duplicate name, the Name Server displays an error message.

Entering Information Into the Object Table

The Name Server usually gets information from the **loadname** command.

Loadname reads the node name and description from an input file and enters the information into the Name Server object table. By default the system is configured to do an automatic **loadname** at boot time.

The input file for the **loadname** command is a file named *data* in the *:sd:net* directory. Each line of the *:sd:net/data* file corresponds to one or two entries in the Name Server object table. The easiest way to create this file is to edit an example file, *data.ex*, that is provided with the OS software. To create a *data* file this way:

1. Move to the *:sd:net* directory and create a copy of the example file, by entering:

```
attachfile :sd:net
copy data.ex to data
```

2. Edit the *data* file. The first line of the file is the template for the local server name information:

```
local_name/nfs: TYPE=rmx: ADDRESS=;
```

Substitute the node name for `local_name` in the first line, and delete the remaining lines. You do not need to specify an address. The Name Server obtains that from the NIC.

3. Enter the contents of the file into the Name Server with this command:

```
loadname
```

You can also register a server name by executing a **setname** command on the local system from the command line.

See also: `:sd:net/data.ex` file, Chapter 11

loadname, **unloadname**, and **setname** commands, *Command Reference*

Defining Network Users in the UDF

On any node running the iRMX OS, the UDF contains the network user definitions. By default the UDF is in the `:sd:rmx386/config` directory. On systems with an ICU, this is configurable.

See also: Network configuration examples, Chapter 5

To make network administration easier, each AU has one Master UDF on the AU's master node. The Master UDF is distributed to all the nodes in the AU, so any user name in this file is known throughout the AU.

To add a new network user:

1. Add the new name to the UDF on the user's local computer. Because the UDF contains encrypted passwords, you must create and modify it with a special utility, rather than ordinary text editors. To do this, log on as Super and enter:

```
password
```

For the Super user, a menu of options appears. Choose `Add a user` from the menu and follow the prompts. When you are prompted, create directories for the user:

```
Do you want to create the user directories? y
```

2. Add the new name to the Master UDF. To modify the UDF, log on to the AU's master node as Super and enter:

```
password
```

When you are prompted, do not create user directories on the master node for this user.

3. Make sure that each of the other nodes on the network attach to the master node and copy the new version of the Master UDF.

See also: Adding users, *System Configuration and Administration*;
password command, *Command Reference*;
Copying the Master UDF, Chapter 4

Accessing Other AUs

To access files across AUs, a user must have a definition in the UDF files on the remote server and the local client. The user name and password must be identical in both UDFs.

Static users cannot access files across AUs, with one exception. If the World user with a password of <CR> (carriage return) is defined in the server's AU, then a World static user can access the files that are available to World on that server.

See also: Static users, *System Configuration and Administration*

Backing Up the Master UDF File

If this is not the master node in the AU, maintain a copy of the Master UDF file on this computer. This enables you to detach the local node from the network and continue to use it if the master node goes down. Of course, diskless nodes, which cannot store local copies of the system software, require the network at all times.

To copy the UDF file, attach the master node and use the remote file copy procedure discussed later in this book.

See also: Copying the Master UDF, Chapter 4

Adding a Client to the CDF

Every server has a Client Definition File (CDF) listing the names and passwords of client nodes in the AU. By default the CDF is in the `./sd:rmx386/config` directory. On systems with an ICU, this is configurable.

See also: Network configuration examples, Chapter 5

To access a server within the AU, add this node's client name and password to the server's CDF. If this node is not listed in a server's CDF, access is controlled through server-based protection.

You must modify the CDF with a special utility, rather than ordinary text editors. Log on to the server as Super and enter:

```
modcdf
```

Choose `add a client` from the menu. At the appropriate prompts, enter this node's client name and password, as specified during ICU configuration or in the `rmx.ini` file. (This is not necessarily the same as the node's server name, which was registered with the Name Server.) The client name must be unique within the CDF. The name and password are case sensitive, so nodes named `COMMSYS`, `Commsys` and `commsys` could all be listed in the same CDF.

Repeat the process on every server within the AU that this node will access. In AUs where all nodes will be used as both servers and clients, it is faster to set the CDF up on one node and then copy the file to the other nodes when they are connected to the network.

⇒ Note

To use `modcdf` to update the CDF, the `iRMX-NET` server job must be running on the system, either loaded with the `sysload` command or configured into the OS with the ICU.

See also: `modcdf` command, *Command Reference*;
Server-based Protection, Chapter 2;
Network configuration and `modcdf` examples, Chapter 5;
Copying the CDF File, Chapter 4

Diagnostics

iRMX-NET provides two ways to obtain more information during the network setup process. The **netinfo** command returns the Ethernet address and status of the communications board. The **inamon** utility provides a variety of information about the status of iNA 960.

See also: **netinfo** and **inamon** commands, *Command Reference*

What's Next?

The iRMX-NET server and client are now ready to use.

iRMX III and
iRMX for PCs
nodes

Local users can begin remote file access.

Remote users can access the public directories specified during ICU configuration. You can extend remote access to other local directories with the **offer** command.

DOSRMX nodes

Local users can begin remote file access from the iRMX screen.

Remote users can access the DOS file system as well as the iRMX file system through iRMX-NET. You can extend remote access to other DOS or iRMX directories with the **offer** command.

See also: Public directories, Chapter 4;
offer command, *Command Reference*;
Using iRMX-NET in a DOS Environment, *System Configuration and Administration*



Using the Network

4

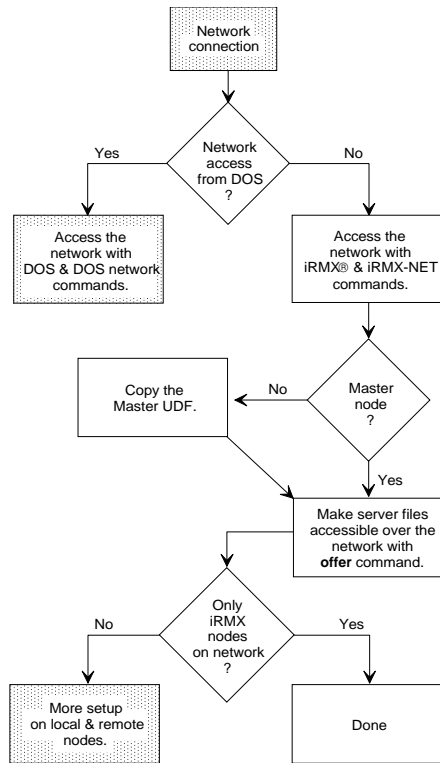
This chapter explains how to share files across the iRMX-NET network, including accessing files on other nodes (as a client) and providing access to local files for users on other nodes (as a server). You can use your node as a client, a server, or both simultaneously.

This discussion assumes that your local computer and any other computers you plan to access are already set up as nodes on the network.

See also: Network Access Using iRMX-NET, Chapter 3

Figure 4-1 gives an overview of remote file access from both the client and the server points of view. The lighter boxes show the steps covered in this chapter.

See also: Using iRMX-NET in a DOS Environment, *System Configuration and Administration*, to use DOS networking with DOSRMX



W-3277

Figure 4-1. Remote File Access

Accessing Remote Files

Accessing files on another computer involves making a connection to the computer and giving it a logical name. Then the remote files are manipulated in the same way as local files, except that the logical name in the remote file's pathname is the remote connection.

Connecting to a File Server

iRMX provides three commands for managing connections to remote servers: **attachdevice**, **detachdevice**, and **logicalnames**. These commands are also used for other purposes, which this discussion does not cover.

See also: **attachdevice**, **detachdevice** and **logicalnames** commands,
Command Reference

Attaching a Server

Establish a connection to the server with the **attachdevice** command by entering:

```
attachdevice server_name as logical_name remote
```

Where:

server_name

is the node's server name registered with the iRMX-NET Name Server

logical_name

specifies a logical name for the server

remote

indicates that the device being attached is a remote server

If more than one user logs into the local computer, you may want to invoke **attachdevice** as Super. That makes the connection available to all local users, but only Super can disconnect it.

⇒ Note

If the **attachdevice** command is not successful, and the *server_name* node is indeed configured as a server and is running, check these items:

- The iRMX-NET client (*remotefd.job*) and appropriate iNA 960 job (*i*.job*) are installed in the */rmx386/jobs* directory.
- A **sysload** command for these jobs is in the *:config:loadinfo* file and no semicolon precedes the command (assuming that the network jobs are loaded at runtime rather than linked with the ICU).

Detaching a Server

To detach from a server, log on with the user name that issued the **attachdevice** command. Enter:

```
detachdevice logical_name
```

Use the same logical name assigned to the server with the **attachdevice** command.

Listing Remote Connections

To list the remote servers connected to this computer, enter:

```
logicalnames l
```

This lists all the current logical names. Connections to remote servers are identified as *ldev* (logical device name) and *REM* (remote file driver). For example:

```
System Logical Names:
```

name	type	fdr	con	dev name	owner	pathname
M	ldev	REM	0	system_a	# 0	:M:
C	ldev	REM	0	system_c	# 65505	:C:

Using Remote Files

You can use almost any OS command or program to access files and devices on a remote server. The exceptions are commands and programs that physically manipulate the drives, such as **format**. Also, access to remote files is governed by access permissions established between the local and remote nodes.

See also: Making Local Files Accessible to Other Nodes, in this chapter

If you use a remote file's full pathname on a command line, the logical name of the server becomes the prefix to the pathname. For example, suppose `system_a` is attached as `:m:`. To list the files contained in the public directory `usr1` residing on `system_a`, use this command:

```
dir :m:usr1
```

Suppose that the `usr1` directory contains a file named `data1`. To display the contents of the file, use this command:

```
copy :m:usr1/data1
```

Copying Files Across the Network

In addition to accessing remote files, you can copy files from one computer to another across the network. For example, suppose you want a local copy of the *data1* file that you just looked at on *system_a*. Use this command to copy it to your local computer:

```
copy :m:usr1/data1 to usr2/data1
```

Copying the Master UDF

One of the first tasks when a node is connected to the network is to copy the Master UDF from the master node of the AU. There are two ways this can happen. ICU-configurable systems provide an automatic UDF copy option, set on the UPD line of the User Definition File screen. Or you can connect to the master node and do it yourself.

See also: Setting Up the Administrative Unit, Chapter 5;
 For ICU-configurable systems: UPD, *ICU User's Guide and Quick Reference*

The UDF file must always be in your local *:sd:rmx386/config* directory. To copy the file, you must be logged onto the client node, not the master node. For example, suppose the master node in the AU is named *pcmastr*. From your local computer, you could attach the master node and copy the file with commands similar to these:

```
attachdevice pcmastr as ms remote  
copy :ms:rmx386/config/udf over :config:udf
```

Notice that the logical name *:config:* is used instead of the longer *rmx386/config* pathname on the local computer. The exact pathname of the remote UDF file may vary, depending on how the master node's public directories are set up for network access.

See also: Logical names, **dir** and **copy** commands, *Command Reference*

Copying the CDF

The CDF is also kept in your local `:sd:rmx386/config` directory. Unlike the UDF, the CDF file can be copied between the master node and the client node while you are logged on to either computer. If you are logged on to a client node, use commands similar to these:

```
attachdevice pcmastr as ms remote
copy :ms:rmx386/config/cdf over :config:cdf
```

On the other hand, if you have a number of computers to update, you can log on to the master node and use commands similar to these:

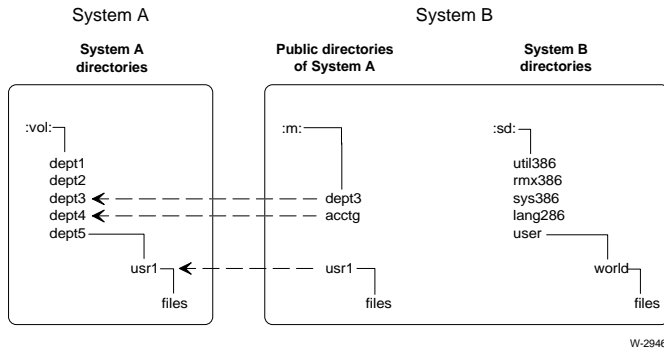
```
attachdevice pcbus as pc remote
copy :config:cdf over :pc:rmx386/config/cdf
attachdevice sys320 as s3 remote
copy :config:cdf over :s3:rmx386/config/cdf
attachdevice sys520 as s5 remote
copy :config:cdf over :s5:rmx386/config/cdf
```

Making Local Files Accessible to Other Nodes

Not all files on a file server are available for remote access. Only when directories are specifically made available can their contents be accessed by client nodes.

Directories that are available for access by remote users are called *public directories*. They are given public names, which can differ from their local directory names. A client sees only the public directories. These include not only the directories that were specified as public, but also each directory's *subtree*, which is all the data files and nested directories contained in the directory.

As an example, assume that a file server has a directory structure like System A in Figure 4-2. Three of its directories are public: `:vol:dept3`, `:vol:dept4`, and `:vol:dept5/usr1`.



System B attaches System A as logical name `:m:`. Users on System B see only the public directory names from System A.

Figure 4-2. Public Directories as Seen from a Client

Now look at the view from the client, System B. It has its own directory structure, starting from `:sd:`. In addition, a second directory structure, starting from `:m:`, is visible. The directory `:m:` was created when a user on System B attached System A, with a command line like this:

```
attachdevice system_a as m remote
```

The directory `:m:` is the root directory of the remote server, as seen from the client. This is called the *virtual root directory*. From the client, all server directories specified as public appear directly under the virtual root directory, regardless of where they exist in the server's directory structure. Thus `:vol:dept5/usr1` is visible simply as `:m:usr1` on System B.

Notice the public names. The *dept3* and *usr1* directories have retained their names, but *dept4* has been given the public name *acctg*.

Making a directory public gives remote users access to its entire subtree. In this example *:vol:dept5/usr1* is public, so *:vol:dept5/usr1/files* is also accessible. However, the directories above a public directory, like *:vol:* and *:vol:dept5*, are not visible to clients and cannot be accessed.

When a directory becomes public, access rights for local users do not change. Remote users cannot delete a public directory on another computer, or change its name. Otherwise, they have the same access rights as local users. To delete a file on another computer, remote users must have both append and update access to it. File and directory access rights are controlled by user name. For example, suppose that the user *World* is given the right to list the files in directory *:vol:dept5/usr1* on System A. Then user *World* on System B can list the files in directory *:m:usr1*.

If any nodes on the network are DOS clients, make sure the files they need are accessible to *World*.

See also: **attachdevice** and **permit** commands, *Command Reference*

Setting Up Public Directories

Several public directories, mostly *sd* and its sub-directories, are defined by default. On ICU-configurable systems you can change the default on the PDIR screen of the ICU.

You can change the list of public directories at run time, using the **iRMX-NET offer** and **remove** commands. You must log on to the server to do this. Going back to the example in Figure 4-2, suppose you want to give remote users access to all the files on System A. The easiest way is to make *:vol:* a public directory, with this command line:

```
offer :vol: as vol
```

Where:

:vol: is the pathname to the directory.

vol is the directory's public name.

This lets remote users move at will from *:vol:* down through its entire subtree.

Listing Public Directories

To find out what public directories are defined on your computer, use the **publicdir** command:

```
publicdir l
```

The *l* (*long*) parameter displays the full pathname of each directory and the device where it resides, as well as its public name. For example, now that you have made *:vol:* public on System A, the public directory list looks like this:

```
PUBLIC DIRECTORIES OF THE SERVER
```

OFFERED NAME	DEV NAME	PATHNAME
ACCTG	D_DOS	/dept4
VOL	D_DOS	/
DEPT3	D_DOS	/dept3
USR1	D_DOS	/dept5/usr1

Removing Public Directories

Now suppose you decide to remove the other public directories, because users can get to them through `:vol:`. Use this command line:

```
remove acctg, dept3, usr1
```

Any public directory can be removed, but you must be logged onto the server when you do it.

See also: **offer**, **publicdir**, **remove** and **permit** commands,
Command Reference

Protecting Files on a Server

When remote users access a server's file system, additional file protection is often needed. If a directory or file in a public directory's subtree does not need to be shared with remote users, consider moving it. Especially on nodes where everything under the `:sd:` directory is accessible, you might build a separate directory structure for private files on a different logical drive.

Protect directories and files that need to be shared by using the **permit** command to limit user access to them. For example, you could give World read-only access to files, while Super gets read/append/update access. This is equally effective whether the files are shared across the network or by a group of local users.

On DOSRMX servers, all users of a DOS file have the same access, because World is the only user supported by the DOS file system. If some users must have append or update access to a file, while others should have read-only access, put the file on an iRMX partition.

See also: **permit** command, *Command Reference*

What's Next?

If your computer is a client, you have connected it to servers and begun accessing remote files. If it is a server, you have provided local file access to remote users. If it is not the master node in the AU, you have made a local copy of the Master UDF.

What happens next depends on the nature of the computers with which this node communicates.

- | | |
|----------------|--|
| All iRMX nodes | If all the nodes are running the iRMX OS, nothing more is necessary. |
| DOS nodes | If some DOS nodes are included, no more setup is necessary, but you should be aware of certain restrictions.

See also: iRMX and DOS Interoperability, Chapter 6 |
| UNIX nodes | You must make changes to the iRMX UDF, and do setup on the UNIX nodes. If these are nodes on older versions of SV-OpenNET, they must be added to the iRMX-NET Name Server using a spokesman node.

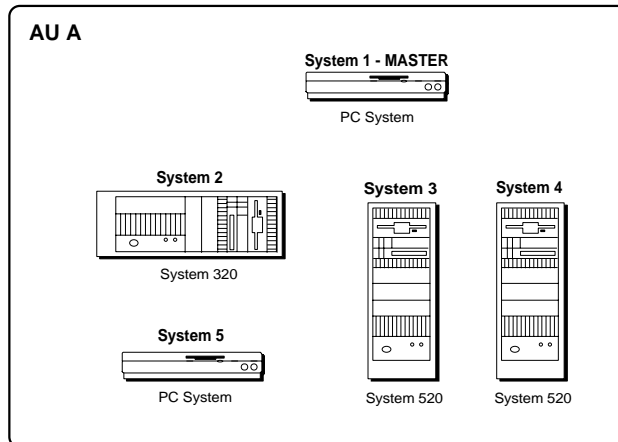
See also: iRMX and UNIX Interoperability, Chapter 6; Spokesman node, Chapter 11 |



Example: Configuring an Administrative Unit 5

This chapter shows how to set up multiple iRMX-NET nodes in the same Administrative Unit (AU). The example AU includes five nodes, as shown in Figure 5-1. System 1 is a PC that is the master node in the AU. System 2 is an Intel System 320 with Multibus I architecture. Systems 3 and 4 are Intel System 520s with Multibus II architecture. System 5 is another PC.

The examples show the configuration of network parameters, but not the basic configuration and generation of the OS.



W-2947

System 1 contains the master UDF for Systems 1 through 5.

Figure 5-1. Single Administrative Unit

Configuring the Systems

Two of the nodes in this chapter use a load-time configuration file; the others have an ICU. The ICU configuration examples assume the standard definition files for networking applications, provided with the iRMX III OS in the `/rmx386/icu;` directory. When you use the files, the backup version is restored to a definition file of the same name, but with an extension of `.def` instead of `.bck`.

See also: *System Configuration and Administration*;
ICU User's Guide and Quick Reference;
Standard definition files, *Installation and Startup*

Configuring the Master Node

By default any node is configured to be both a server and a client, but you can change that on systems with an ICU. Make sure the master node is configured as a file server, however, because it contains the Master UDF that the other nodes need to access. The only default iRMX-NET parameters you must change on the master node are the client name and password.

See also: The User Definition File, Chapter 2

System 1: iRMX for PCs Node

In this example the master node, named PCMASTR, is a PC platform running iRMX for PCs. To configure an iRMX for PCs system, edit the `:config:rmx.ini` load-time configuration file.

1. Put the client name and password in the appropriate lines of the `rmx.ini` file. For example:

```
CNN= 'PCMASTR' ;      Client Node Name
CNP= 'PCPASS' ;      Client Node Password
```

2. Generate the system as usual, including setting up the `:sd:net/data` file and booting the system.

See also: Adding a Server to the Name Server Object Table, Chapter 3;
Starting iRMX for PCs, *Installation and Startup*;
`rmx.ini` file, *System Configuration and Administration*

ICU-configurable Master Node

If you choose a master node with an ICU, configure it this way:

1. Specify the client name and password on the ICU's Client Definition File (CDF) screen. For example:

```
(CNN) Client Name          320MASTR
(CNP) Client Password      320PASS
```

2. Generate the system as usual, including setting up the `:sd:net/data` file and booting the system.

See also: Adding a Server to the Name Server Object Table, Chapter 3;
Starting iRMX III.2, *Installation and Startup*;
CDF screen, *ICU User's Guide and Quick Reference*

Configuring the Other Nodes

All nodes, except the master node, must be set up as clients. By default, they are configured to be both client and server, but you can change that on systems with an ICU. On client nodes you only need to change a few default iRMX-NET parameters, which are named in the sections that follow. In general, the setup involves these steps:

1. Specify a client name and password.
2. Set up nodes with an ICU to automatically copy the Master UDF file. This requires changing parameters on the User Definition File (UDF) and Logical Names (LOGN) screens. On nodes without an ICU, skip this step; the Master UDF file will be copied later.
3. Generate the system as usual.
4. Do not boot the computer yet. First the master node must be running, with a server name assigned and with the Master UDF and CDF in place. This happens later in the process.

See also: Setting Up the Administrative Unit, in this chapter

System 2: Multibus I, System 320

System 2, named SYS320, is an iRMX III System 320. To configure it, invoke the ICU using the *38620net.bck* file and make these changes:

1. On the Client Definition File (CDF) screen, specify these parameters:

```
(CNN) Client Name          SYS320
(CNP) Client Password      320PASS
```

2. On the User Definition File (UDF) screen, specify these parameters:

```
(MUL) Master UDF Location  REMOTE
(MLN) Master UDF Logical Name PCM
(MPN) Master UDF Path Name /RMX386/CONFIG/
(MD) Master UDF Device     PCMASTR
(LUL) Local UDF Location   NAMED
(LLN) Local UDF Logical Name SD
(LPN) Local UDF Path Name /RMX386/CONFIG/
(LD) Local UDF Device      w0
```

3. On the Logical Names (LOGN) screen, add this entry:

```
PCM, PCMASTR, REMOTE, 0H
```

System 3: Multibus II, System 520

System 3 is named SYS520. It is an iRMX III Multibus II System 520 with an SBC 486/133SE board. To configure it, invoke the ICU using the *486133.bck* file and make these changes:

1. On the Client Definition File (CDF) screen, specify these parameters:

```
(CNN) Client Name          SYS520
(CNP) Client Password      520PASS
```

2. On the User Definition File (UDF) screen, specify these parameters:

```
(MUL) Master UDF Location  REMOTE
(MLN) Master UDF Logical Name PCM
(MPN) Master UDF Path Name /RMX386/CONFIG/
(MD) Master UDF Device     PCMASTR
(LUL) Local UDF Location   NAMED
(LLN) Local UDF Logical Name SD
(LPN) Local UDF Path Name /RMX386/CONFIG/
(LD) Local UDF Device      SCW
```

3. On the Logical Names (LOGN) screen, add this entry:

```
PCM, PCMASTR, REMOTE, 0H
```

System 4: Multibus II, System 520

System 4 is a Multibus II system with multiple hosts. It has two CPU hosts and an I/O server: an SBC 386/258 with a CSM/002 in slot 0, an SBC 486/125 in slot 2, and an SBC 386/120 in slot 3. Any combination of CPU hosts and I/O servers would work, however. Each host has its own name, and you need to generate a separate configuration for each one.

Host 1: 520SRV

The first host, named 520SRV, is configured as both a client and a server. It accesses the local hard disk using the I/O server. To configure it, invoke the ICU using the *486125net.bck* file and make these changes:

1. On the Client Definition File (CDF) screen, specify these parameters:

```
(CNN) Client Name           520SRV
(CNP) Client Password       520SPASS
```

2. On the User Definition File (UDF) screen, specify these parameters:

```
(MUL) Master UDF Location   REMOTE
(MLN) Master UDF Logical Name  PCM
(MPN) Master UDF Path Name   /RMX386/CONFIG/
(MD)  Master UDF Device      PCMASTR
(LUL) Local UDF Location     NAMED
(LLN) Local UDF Logical Name  SD
(LPN) Local UDF Path Name    /RMX386/CONFIG/
(LD)  Local UDF Device       SCW
```

3. On the Logical Names (LOGN) screen, add this entry:

```
PCM, PCMASTR, REMOTE, 0H
```

Host 2: 520CLI

The second host, named 520CLI, is configured as a client only. It accesses the local hard disk through the iRMX-NET file server software running on the first host's CPU. To configure it, invoke the ICU using the *386120rsd.bck* file and make these changes:

1. On the Client Definition File (CDF) screen, specify these parameters:

```
(CNN) Client Name           520CLI
(CNP) Client Password       520CPASS
```

2. On the User Definition File (UDF) screen, specify these parameters:

(MUL) Master UDF Location	REMOTE
(MLN) Master UDF Logical Name	PCM
(MPN) Master UDF Path Name	/RMX386/CONFIG/
(MD) Master UDF Device	PCMASTR
(LUL) Local UDF Location	REMOTE
(LLN) Local UDF Logical Name	SD
(LPN) Local UDF Path Name	/RMX386/CONFIG/
(LD) Local UDF Device	520SRV

3. On the Logical Names (LOGN) screen, add this entry:

```
PCM, PCMASTR, REMOTE, 0H
```

System 5: PC Bus Platform

System 5, named PCBUS, is a PC platform running DOSRMX. To configure this node, the only default iRMX-NET parameters you need to change are the client name and password. (The extra parameters you changed on the client systems with ICUs were used to automatically copy the Master UDF, which you cannot do here.) On an DOSRMX system, change the client name and password by editing the *:config:rmx.ini* load-time configuration file.

1. Put the client name and password into the appropriate lines of the *rmx.ini* file. For example:

```
CNN= 'PCBUS' ;      Client Node Name
CNP= 'PCBPASS' ;   Client Node Password
```

2. Generate the system as usual, including setting up the *:sd:net/data* file and booting the system.

Setting Up the Administrative Unit

These sections describe the files you need to modify to set up an AU and the order in which the nodes should be booted.

System 1

To set up the master node:

1. Boot the system.
2. By default the system is configured to do an automatic **loadname** at boot time. To check whether this was done, enter:

```
getname
```

If necessary, use the **setname** command to assign the server name. For example, to set a server name identical to its client name:

```
setname PCMASTR
```

Server names are not case-sensitive.

See also: Adding a Server to the Name Server Object Table, Chapter 3

3. Enter a **publicdir** command and make sure the `:sd:rmx386` directory is included in the list of public directories, so the other nodes can copy the UDF and CDF files in `:sd:rmx386/config`. If necessary, **offer** the directory:

```
offer :sd:rmx386 as rmx386
```

4. Invoke the **iRMX password** command and enter the names and passwords of each user in the AU. Only these users are allowed to log on to any of the nodes within the AU.
5. Invoke the **iRMX-NET modcdf** command. For security, remove the default client name `rmx` from the CDF. Then enter the client names and passwords of each of the client nodes in the AU. Include the master node only if it operates as a client as well as a server. Unlike server names, both the client names and client passwords of **iRMX-NET** nodes are case-sensitive.

Modcdf Example

All client names and passwords in this example network were defined using uppercase characters. Text following a semicolon is a comment.

```
MODCDF      ; invoke the MODCDF command

D           ; delete a node
rmx        ; remove the default client name

A           ; add a node
PCMASTR    ; add the master node (optional entry)
PCPASS     ; add master password
PCPASS     ; repeat password

A           ; add a node
SYS320     ; add System 2 name
320PASS    ; add System 2 password
320PASS    ; repeat password

A           ; add a node
SYS520     ; add System 3 name
520PASS    ; add System 3 password
520PASS    ; repeat password

A           ; add a node
520SRV     ; add System 4 Host 1 name
520SPASS   ; add System 4 Host 1 password
520SPASS   ; repeat password

A           ; add a node
520CLI     ; add System 4 Host 2 name
520CPASS   ; add System 4 Host 2 password
520CPASS   ; repeat password

A           ; add a node
PCBUS      ; add System 5 name
PCBPASS    ; add System 5 password
PCBPASS    ; repeat password

E           ; update the CDF and exit
```

Systems 2 through 5

After the master node is running, with a server name assigned and with the Master UDF and CDF in place, complete the network setup for each of the other nodes within the AU. For each node, reset and reboot the newly generated system.

During initialization an ICU-configurable node attaches to the master node and assigns the logical name *:pcm:* to the remote device. Then it copies the Master UDF over its own local UDF. (In Figure 5-1, this happens on Systems 2 and 3 and on both hosts on System 4.)

At this point the other nodes can only connect to the master node. To enable connections between all the nodes, you need to perform these steps on Systems 2 through 5:

1. Log on as one of the users defined on System 1, the master node. If the local node will act as a server as well as a client, make sure its server name was set during iRMX-NET initialization. Enter:

```
getname
```

See also: Adding a Server to the Name Server Object Table, Chapter 3

2. Copy the CDF file on the master node to *:sd:rmx386/config/cdf* on each local node. Use these commands:

```
attachdevice PCMASTR as pcm remote  
copy :pcm:rmx386/config/cdf over :config:cdf
```

The *:sd:rmx386/config* directory is the default location, which was not changed during the configuration of the example nodes. On ICU-configurable nodes, however, you can specify another path name on the CDF screen.

See also: Copying the CDF, Chapter 4

3. On System 5, the DOSRMX node, copy the Master UDF file to the local *:sd:rmx386/config/udf* file. For example:

```
copy :pcm:rmx386/config/udf over :config:udf
```

See also: Copying the Master UDF, Chapter 4

4. Establish a connection to any server in the AU. For example, Host 1 of System 4 is configured as both a client and a server. To attach to the Host 1 server, use this command:

```
attachdevice 520SRV as 520 remote
```

The node name (520SRV) required by the **attachdevice** command is the server name registered with the Name Server, not the client name. In this example the two names are identical, but they do not have to be.

5. Delete the connection to a server with the **detachdevice** command. For example, to delete the connection created above, use this command:

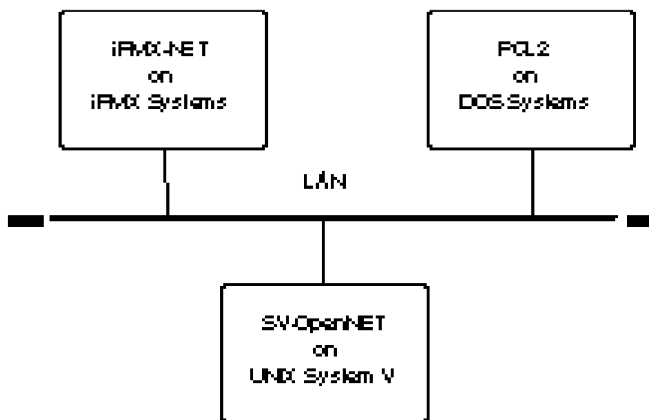
```
detachdevice :520:
```



Example: Configuring Multiple Operating Systems 6

This chapter explains how to configure and use the DOS and UNIX systems for maximum interoperability with the iRMX OS. After such configuration, these other systems can share files with iRMX nodes on iRMX-NET.

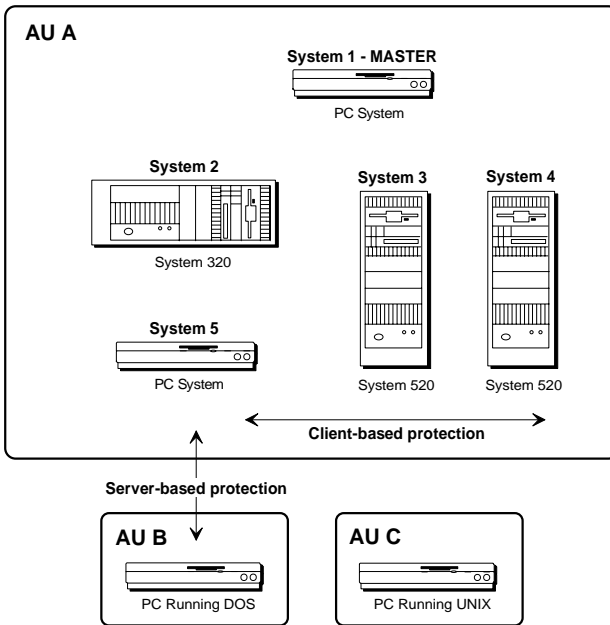
Each system runs with an OpenNET product that understands the Network File Access (NFA) protocols, as shown in Figure 6-1. In a few cases, the server's OS provides a capability that the client's does not, or vice versa. This chapter describes those capabilities that are not fully transparent over OpenNET.



W-2000

Figure 6-1. The OpenNET Network

Examples in this chapter add DOS and UNIX nodes to the iRMX network example in the last chapter. The AU configured in the previous chapter, with its five iRMX nodes, is referred to here as AU A. Two new AUs are configured in this chapter: AU B with one DOS node and AU C with one UNIX node. Figure 6-2 shows all three AUs.



System 1 contains the master UDF for Systems 1 through 5.

Figure 6-2. Multiple Operating System Network

The DOS System

A PC executing DOS and the PCL2 R3.0 (or later) software can access iRMX servers. The MS-Net software, which is shipped with the PCL2(A) NIC, or the IBM PC LAN software must be running on the PC.

See also: Software and hardware requirements for networking on a DOS-based PC, *OpenNET PCL2 for DOS Installation Guide*

For this example, the DOS system is named SYSDOS. The AU containing the DOS system is named AU B. Because DOS does not support multiple users, each DOS node forms a separate AU. Users are specified only when establishing access to a remote server.

⇒ Note

The system described here is a PC running DOS as its only OS. This section does not necessarily apply to DOSRMX systems, which can run both DOS and the iRMX OS.

See also: Using iRMX-NET in a DOS Environment, *System Configuration and Administration*

Connecting a DOS Client to an iRMX Server

To establish a connection between SYSDOS and any of the five iRMX nodes in AU A, complete these steps.

On the iRMX Server

1. Define an iRMX user name for the DOS system. Contact the system manager of AU A for help. The system manager uses the **password** command to define a user, as shown in the example below. (Text after a semicolon is a comment.)

```
password ; invoke the password command
a        ; add a user
pcuser   ; user name
PASWD    ; password must be in UPPERCASE
PASWD    ; password must be in UPPERCASE
876      ; user id
<Enter>  ; group id = default
<Enter>  ; comments = none
```

```

<Enter>    ; UNIX home directory = none
<Enter>    ; UNIX shell = none
y          ; add to the UDF?
y          ; create the user directories?
e          ; exit

```

The password must be defined in upper case characters, because DOS converts all entered passwords into upper case.

2. Add the new user on each of the iRMX systems. Either add it to each node separately, or add it to the master system, PCMASTR. From the master system it will be distributed to each of the iRMX systems when they are rebooted or updated.

See also: Copying the Master UDF, Chapter 4

3. Make sure that the server names of the iRMX servers are in the Name Server object table.

See also: Adding a Server to the Name Server Object Table, Chapter 3

On the DOS Client

1. Attach an iRMX server from the DOS system. In this example the iRMX server is PCMASTR.

```

net start rdr sysdos
net use e: \\pcmastr\pcuser PASWD

```

Where:

e: is an unused DOS drive, E:.. A space must follow the colon.

pcmastr

is the iRMX server name registered with the Name Server.

pcuser

is the name of the user logging on to the iRMX server. A space must follow the name if there is a password.

PASWD

is the optional user password. You can enter it in either upper or lower case; whatever you enter is converted to upper case.

Now you can use E: just as you would a local drive, like A: or B:.. The logical device E: corresponds to the virtual root directory of the iRMX server.

2. Access the iRMX server from the DOS system. For example:

```
e:
cd user\pcuser
dir
c:
```

3. To detach the iRMX server, enter:

```
net use e: /d
```

iRMX and DOS Interoperability

An iRMX client cannot access a PCL2 (DOS) server. The only possible network connection is from a DOS client to an iRMX server.

The PCL2 Name Server

PCL2 R3.0 (and later) provides a Name Server that can access the iRMX-NET Name Server to find the Transport Address of an iRMX node.

DOS Client Restrictions

When a DOS client accesses an iRMX server, these restrictions apply:

- The iRMX-NET software does not support DOS messaging and DOS locking. Applications that support these features cannot be used across the network.
- A DOS pathname is limited to 63 or 66 characters, so paths for remote iRMX files are likewise limited. This restriction depends on the version of DOS, and the character count includes 2 characters for a drive specifier (for example, C:).
- DOS filenames are limited to eight characters plus a three-character extension. iRMX filenames that do not conform to this limit are not recognized by the DOS system.
- A DOS application assumes that files can be created in the root directory, and uses that directory for temporary files. Because iRMX-NET does not allow files to be created at the virtual root, the first public directory defined in iRMX-NET is used instead. In the default configuration, this is the *work* directory. Files that a DOS client attempts to create or delete at the iRMX root are actually created and deleted from this *work* directory. All other commands must specify the real pathname, such as *:e:work/filename*, rather than just the virtual root pathname of *:e:filename*.

- A DOS client cannot access a file on an iRMX server if the filename contains a question mark character (?). This is because iRMX enables a question mark in a directory or filename, but DOS does not. DOS recognizes the question mark only as a wildcard character. For example, a DOS client cannot read the *:prog:r?logon* file of an iRMX server. Attempting to access an iRMX directory name with a ? from DOS causes an invalid path error message. Similarly, attempting to access an iRMX filename with a ? from DOS causes an `ERROR READING FILE` error message.

The UNIX System

The UNIX node in this sample network is a PC Bus system named *sysunx*. A UNIX system can be in an AU with other UNIX and iRMX systems, or alone in a separate AU. In this example *sysunx* is the only system in AU C. The example assumes that the system is configured with the appropriate System V OpenNET software:

- SV-OpenNET R3.2.3 or later
- SV4-OpenNET R2.0 or later

The sample iRMX server is 520SRV, and the iRMX client is 520CLI. Both iRMX systems are in AU A. Their setup and configuration was discussed earlier.

See also: AU configuration and setup example, Chapter 5

Connecting a UNIX Client to an iRMX Server

To establish a connection between *sysunx* and the iRMX server 520SRV, complete these steps.

On the iRMX Server

1. Make sure the iRMX node's server name is registered with the Name Server.
See also: Adding a Server to the Name Server Object Table, Chapter 3
2. Use the **modcdf** command to add the UNIX node's client name and password to the iRMX server's CDF file.

See also: **modcdf** example, Chapter 5

On the UNIX Client

1. Create the local node's client name and password, using the **modself** utility.
2. Check that the server name of the iRMX server you intend to access is registered with the Name Server. For example, for the iRMX server 520SRV, use this command:

```
nslocate 520SRV
```

3. Attach an iRMX server from the UNIX system. For 520SRV, use this command:

```
net use sys2 //520srv/world
```

This example defines *//sys2* as the name for the remote server 520SRV. Here the user name is *world*, but you can specify any user name and password defined in the UDF of the iRMX server.

4. Enter the password for the specified user when prompted.
5. To detach the iRMX server, enter:

```
net use sys2 /d
```

Connecting an iRMX Client to a UNIX Server

To establish a connection between *sysunx* and the iRMX client 520CLI, complete these steps.

On the UNIX Server

1. Use the **modcdf** command to add the iRMX node's client name and password to the UNIX server's CDF file.

You can find this information on the iRMX node, in the ICU's CDF screen or the *:config:rmx.ini* file.

On the iRMX Client

1. Check that the server name of the UNIX server you intend to access is registered with the Name Server. For example, for the UNIX server *sysunx*, use this command:

```
findname sysunx
```

2. Attach a UNIX server from the iRMX system:

```
attachdevice sysunx as unx remote
```

3. To access files in the UNIX system, use *:unx:* as the logical name.
4. To detach the UNIX system, enter:

```
detachdevice unx
```

Setting Up the Administrative Unit

This example puts the UNIX and iRMX systems in separate AUs, but you can also combine them in a single AU. If you are familiar with SV-OpenNET terminology, the iRMX-NET term Administrative Unit (AU) is equivalent to the SV-OpenNET term *subnet*.

iRMX and UNIX Nodes in Separate AUs

When UNIX and iRMX systems are in different AUs, complete these steps:

1. On one UNIX node within each AU (subnet), which contains a system being accessed by an iRMX-NET client, edit the UNIX system files */etc/passwd*, */etc/group*, and */etc/shadow* to define iRMX users who will be accessing the UNIX server.

Do not define user names whose only difference is capitalization. UNIX distinguishes between upper- and lower-case characters in user names, but the iRMX OS does not. Passwords are always case-sensitive, on both UNIX and iRMX systems.

See also: UNIX OS documentation for information on adding users

2. Copy the updated */etc/passwd*, */etc/shadow*, and */etc/group* files to all other UNIX nodes in the same AU (subnet).
3. On the iRMX-NET system containing the Master UDF, use the iRMX **password** command to define any UNIX users who will access an iRMX-NET server in the AU. Since iRMX systems do not support groups, add the UNIX groups to the Master UDF as iRMX users.
4. Copy the Master UDF over the local UDFs on all other iRMX systems in the AU.

iRMX and UNIX Nodes in the Same AU

Within an AU (subnet), clients perform all user validation, and the servers then validate the client. When UNIX and iRMX systems are in the same AU, complete these steps:

1. Choose a UNIX node to be the master node within the AU (subnet).
2. On the master UNIX node, edit the UNIX system files */etc/passwd*, */etc/group*, and */etc/shadow* to define the iRMX users that will access the UNIX server.

Do not define user names whose only difference is capitalization. UNIX distinguishes between upper- and lower-case characters in user names, but the iRMX OS does not. Passwords are always case-sensitive, on both UNIX and iRMX systems.

Make sure that UNIX group IDs do not conflict with the user IDs assigned. Where conflicts occur, change the group IDs. The UNIX **chgid** utility can be used to update the file system following such changes.

See also: UNIX OS documentation for information on adding users

3. Copy the updated */etc/passwd*, */etc/shadow*, and */etc/group* files to all other UNIX nodes in the same AU (subnet).
4. Configure the */etc/passwd* file as the iRMX-NET Master UDF.
5. Copy the Master UDF to any iRMX for PCs or DOSRMX nodes in the AU. The ICU-configurable iRMX nodes copy the Master UDF automatically when the iRMX system is booted.
6. On each server in the AU, define all clients in the AU using the UNIX **netadm** utility or the iRMX-NET **modcdf** command.

See also: **modcdf** example, Chapter 5

iRMX and UNIX Interoperability

Both UNIX and iRMX systems have capabilities that are not supported by the other OS. These differences affect anyone going between the two systems:

- UNIX and iRMX files have different line terminators. UNIX files use a line-feed, while iRMX files use a combination of carriage return and line-feed.
- iRMX-NET does not perform text format conversions, so file sharing between iRMX and UNIX systems requires a compatible set of tools. Intel tools combined with UNIX tools are often not a compatible set. For example, AEDIT and iC-386 are compatible, and UNIX **vi** and **cc** are compatible; however, AEDIT and **cc** are not compatible. You cannot compile a file edited with AEDIT with **cc** unless you change the line terminators.

These sections list other differences that are mainly concerns when going from iRMX to UNIX nodes, or from UNIX to iRMX nodes, but not both.

SV-OpenNET Server Features and Restrictions

When an iRMX client accesses remote UNIX files, these restrictions apply:

- iRMX users see UNIX filenames without consideration for case. For example, iRMX users cannot distinguish between the UNIX files *ABC* and *abc*. If the two files are in the same directory, the iRMX client can only address the first file.
- iRMX users cannot specify the iRMX carat (^) and leading slash (/) symbols for UNIX pathnames, but they can specify the UNIX . (dot) and .. (dot-dot) symbols instead. iRMX users cannot see the . and .. in UNIX directory listings, however.
- `E_LIMIT` errors pertain to UNIX server resources.

- The **rename** command cannot move a UNIX directory out of the parent directory. For example, this command, where *tmpdir* is a directory, succeeds:

```
rename :unx:tmp/tmpdir to :unx:tmp/newtmpdir
```

This command fails:

```
rename :unx:tmp/tmpdir to :unx:usr/newtmpdir
```

This restriction does not apply to UNIX data files.

- UNIX supports groups, but iRMX does not. An iRMX client considers all entries of an access list as accessors to be checked in the client's UDF. A UNIX server considers the first entry of an access list as the file's owner, and checks the entry in the server's UDF. The server considers the second entry of an access list to be the group, and checks the entry in the Group Definition File.
- UNIX treats the first ID of an access list as the file owner. The **permit** command can change rights associated with the first ID, but not the ID itself. The **permit** command can change both the second ID of the access list and the rights associated with it. UNIX always contains the World's rights in the third entry of the access list; the **permit** command cannot remove World from the access list, even if World has no rights. When an iRMX user lists remote UNIX directories, the iRMX client displays all three accessors (first ID, second ID, and World) even if they have no rights.
- iRMX and UNIX servers calculate file access rights differently. The iRMX servers grant the user the logical sum of the rights allowed if the user exists in the access list, plus any rights given to World. UNIX servers grant only the rights allowed to the first accessor in the access list whose ID matches that of the user ID. If the user ID matches the first access list entry, a UNIX server grants the rights allowed to the first accessor. If this check fails and if the group affiliated with the user matches the second access list entry, a UNIX server grants the rights for the second accessor. If both checks fail, a UNIX server grants the third accessor's rights, which are the World's rights.

An iRMX client attempts to compensate for this discrepancy by this technique: when the World user creates a remote UNIX file, the iRMX client places World with full access rights in all three access list entries. When an iRMX user grants rights to remote UNIX files for the user World, the iRMX client grants these same permissions to all three accessors. When an iRMX user denies rights to remote UNIX files, the iRMX client does not remove from any of the accessors the rights that the World accessor has.

iRMX Server Restrictions

When a UNIX client accesses remote iRMX files, these restrictions apply:

- Neither the owner nor the group of an iRMX file can be changed.
- Links cannot be used to create additional names for iRMX files. This restriction prevents the use of some UNIX utilities.
- File and record locking are not available for iRMX files.
- The STICKY, SETUID, and SETGID file attributes are not supported. Any attempt to set these bits to 1 is rejected, and any attempt to reset these bits to 0 is ignored.
- An iRMX file cannot be opened in append mode from a UNIX client.

Connecting to Nodes on Older Versions of SV-OpenNET

This example assumes that the UNIX nodes were on SV-OpenNET R3.2.3 or SV4-OpenNET R2.0 or later. These versions provide Name Server capability. You can also interoperate with older versions by creating some additional Name Server entries.

An iRMX client can communicate with an earlier UNIX server if you load the server's name and address into an iRMX-NET Name Server object table. The iRMX node whose local object table includes the UNIX server's information becomes a spokesman for the server.

See also: Programming the Name Server, Chapter 11

For a UNIX client without Name Server capability to access an iRMX-NET server, enter the name and Transport Address of the iRMX-NET server into the */net/data* file on the UNIX system. SV-OpenNET provides the **netadm** utility to manipulate the */net/data* file. The **netadm** utility assumes that the iRMX TSAP-ID is 1000H and the subnet is 1. If you use different values (typically, the subnet is 0), use **netadm** to add the address, and then manually edit the */net/data* file to change those values.

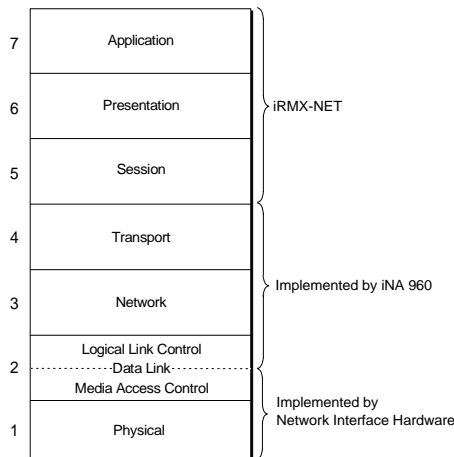
See also: SV-OpenNET documentation for information on the **netadm** utility



Network Software Implementation

7

The iRMX network jobs are part of the iRMX OS software. The basic network jobs are called iNA 960. Separate iRMX-NET jobs run on top of iNA 960 to provide transparent file access. Figure 7-1 illustrates how iNA 960 and iRMX-NET services fit into the International Standards Organization (ISO) Open Systems Interconnection (OSI) model. iNA 960 general-purpose network services include the Data Link, Network, and Transport layers defined in the OSI model. iNA 960 has no relationship to, and does not run on, the Intel i960[®] microprocessor.



OM04342

Figure 7-1. ISO OSI Model

iRMX-NET includes a command-line interface and file services, in addition to the programmatic network access provided by the underlying iNA 960 software.

Hardware Environments

The iNA 960 software is based on subnetworks that at the Data Link level use the IEEE 802.3 Ethernet specification. The software supports the 82586, 82596, and 82595TX Ethernet components, as well as the virtual Ethernet provided in the

Multibus II subnet. Unless otherwise specified, this manual uses the term 82586 to refer to all hardware Ethernet components.

iNA 960 supports PC Bus, Multibus I, and Multibus II systems.

Software COMMputer and MIP Environments

The iNA 960 software is provided by a variety of board-specific network jobs that can be configured with or loaded onto the iRMX OS. These are called iNA 960 COMMputer jobs or MIP jobs:

- A COMMputer job is a version of iNA 960 that executes on the same board as the OS and the application. The board can use either a hardware Ethernet connection or the virtual Ethernet connection provided in the Multibus II subnet. The Ethernet hardware can be built into the baseboard or can be a Network Interface Connector (NIC), such as a MIX 560 module, that works integrally with the baseboard.

See also: Multibus II subnet, Chapter 9

- MIP jobs support what is called a COMMengine environment, where the OS runs on one board and iNA 960 runs on a separate board. If this separate board is a standalone NIC, you set up the MIP job to download an iNA 960 file to the NIC. However, in a Multibus II system you also have the option of using a different COMMputer board as the NIC for a board that runs a MIP job. In this case, you do not download an iNA 960 file from the MIP job, because the separate COMMputer already includes iNA 960. In either case, the MIP job acts as an interface between iNA 960 on the other board and application programs on the board that runs the MIP job.

MIP jobs were formerly called Multibus Interprocessor Protocol jobs, but with the added support for PCs in recent releases of the OS, the name has changed to Message Interprocess Protocol.

See also: MIP details and error messages, Appendix B

The iNA 960 MIP and COMMputer jobs are referred to collectively as *i*.job*. This manual uses the term iNA 960 to refer to the capabilities of both MIP and COMMputer jobs. As far as your application is concerned, there is no difference between them. Both types of job provide the iNA 960 services described in this manual.

Some COMMputer jobs support multiple subnets so they can act as routers between subnets. These jobs are preconfigured to use specific NICs as the ports to subnets. Some of them can also use the Multibus II backplane as a virtual Ethernet interface.

See also: *Multibus II Subnets*, Chapter 9
i.job, System Configuration and Administration* for details about which jobs run in which hardware environments

Overview of iNA 960 Software

Table 7-1 shows the specific ISO services provided by the iNA 960 software and the ISO specifications used to implement those services.

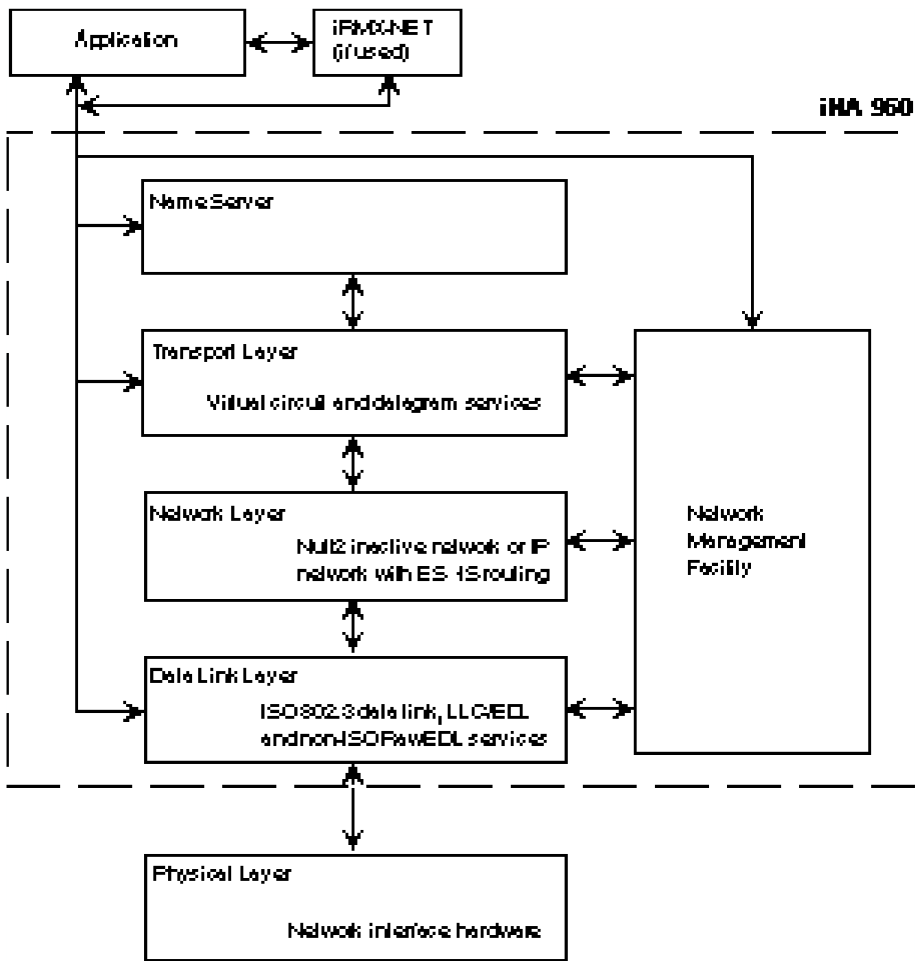
Table 7-1. iNA 960 Services and ISO Specifications

ISO Service Provided By iNA	ISO Specifications Used To Implement the Service
Transport Virtual Circuit IS 8072	IS 8073 Class 4
Transport Datagram IS 8072 Addendum 1	IS 8602
Connectionless Network Layer* IS 8348 Addendum 2	IS 8473
Internetwork Routing	IS 9542 or Static User Entries
Data Link IEEE 802.2	IEEE 802.2 Type 1 and IEEE 802.3 Ethernet

* There is no external interface to this service.

The iNA Layers

Figure 7-2 shows the layers of iNA 960 software.



CM004 943

Figure 7-2. iNA 960 Software Layers

The Name Server

The Name Server maps network addresses and other numeric values to more easily-remembered names. Your application can use the programmatic interface to the Name Server. If you run iRMX-NET in addition to iNA 960, iRMX-NET uses the Name Server services to provide user commands.

The Transport Layer

The Transport Layer provides transparent data transfer between processes. Two types of data transfer service are implemented: virtual circuit and datagram. The virtual circuit service provides point-to-point, error-free, guaranteed delivery of messages in the sequence they are sent. It also provides a high-priority message delivery mechanism, called expedited data. The datagram service attempts to deliver messages without guaranteeing delivery, order of delivery, or integrity of the message.

The Network Layer

The Network Layer performs message routing and relay. It delivers messages within a subnetwork and across interconnected subnetworks. The Network Layer also offers a datagram delivery service to higher layers. There is no direct application interface to the Network Layer.

For each system bus type there are two preconfigured versions of iNA 960 that differ at the Network Layer. The Null2 version does not provide internetwork routing. The ES-IS version includes an IP-protocol internetwork routing service. The routing tables can be built and updated statically by the application or dynamically using the ISO ES-IS routing protocol (IS 9542).

The Data Link Layer

The Data Link Layer transforms the raw transmitted and received data of the Physical Layer into a communication channel that appears error-free to the Network Layer. A Data Link connection is built upon one or more physical connections. This layer provides the functions and protocols used to establish, maintain and release Data Link connections. In addition, the Data Link Layer is responsible for framing packets and detecting errors. The Data Link Layer has two interfaces:

- The External Data Link (EDL) interface enables an application to bypass the Transport and Network layers and directly access the services of the Data Link layer (IEEE 802.2 LLC Type 1).

- The RawEDL interface lets an application access non-802.3 packets, so non-ISO protocol stacks can run on top of iNA 960. Thus the ISO Transport services can coexist with such non-ISO protocols as TCP/IP or NetWare, sharing the same NIC. The RawEDL services can also be used to capture and monitor non-ISO packets for network analysis.

The Network Management Facility

The Network Management Facility (NMF) provides functions for reading and setting database objects that are maintained internally by each of layer of iNA 960. By monitoring these objects, an application can gather network usage information such as peak activity, total packets sent, and CRC errors. The application can change the values of database objects to optimize network performance or manage internetwork routing tables.

The Programmatic Interface

An application requests iNA 960 services by using data structures called request blocks. All request blocks contain a common set of header fields and, depending on the function being requested, may have additional function-specific fields.

See also: Chapter 10 for the general request block interface and the system calls used to manipulate request blocks;
Chapters 11-16 for request block structures for each iNA 960 or Name Server function

Overview of iRMX-NET Software

The iRMX-NET software executes within the boundaries of the Session, Presentation, and Application Layers; however, the Presentation and Session Layers are not formally implemented. You can use one or both of these parts of iRMX-NET; they are jobs that you configure into the OS (with the ICU) or load separately (with a **sysload** command):

- Client, or File Consumer (*remotefd.job*) provides transparent file access to systems that run the iRMX-NET File Server.
- File Server (*rnetserv.job*) makes files on the local system available to remote systems that run the iRMX-NET Client.

The iRMX-NET Client and Server includes these parts:

Name Server	<p>iRMX-NET uses the iNA 960 Name Server to provide transparent file access. Most iRMX-NET user commands use the Name Server to access remote files.</p> <p>See also: Name Server, Chapter 2</p>
User Administration	<p>The User Administration (UA) module maintains the files that are used by a system administrator when making additions and deletions of users and systems in an iRMX-NET environment. A system administrator has the responsibility of overseeing the assignment of users and systems, and of maintaining general network security.</p>
Apex File Access	<p>The Apex File Access (AFA) module is the operating system-dependent part of the server. AFA receives requests from the File Server module. The AFA executes the necessary file operations that correspond to the user's requests.</p>
File Consumer	<p>The File Consumer module, with the Remote File Driver (RFD), provides the functions of the client system. The RFD passes the local user requests to the File Consumer, which then transmits the requests across the network. The File Consumer is independent of the OS; it does not make iRMX file system I/O calls. However, the RFD, as part of the BIOS, does make iRMX system calls.</p>
File Server	<p>Together, the File Server and AFA modules provide the server functions. The File Server performs transactions for users at remote nodes. When a remote user initiates a request, the File Server receives the request, interprets it, and passes the request to the AFA module for processing. The File Server is independent of the OS; it does not use iRMX file system I/O calls.</p>

Data Flow Through iRMX-NET and iNA 960 Software

Figures 7-3 and 7-4 illustrate the functions of the iRMX-NET modules for COMMPuter and COMMengine systems.

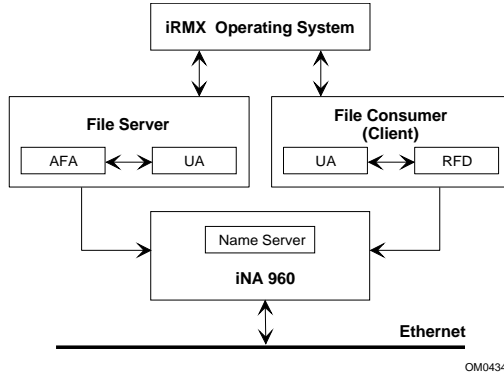


Figure 7-3. iRMX-NET Data Flow on COMMPuter Systems

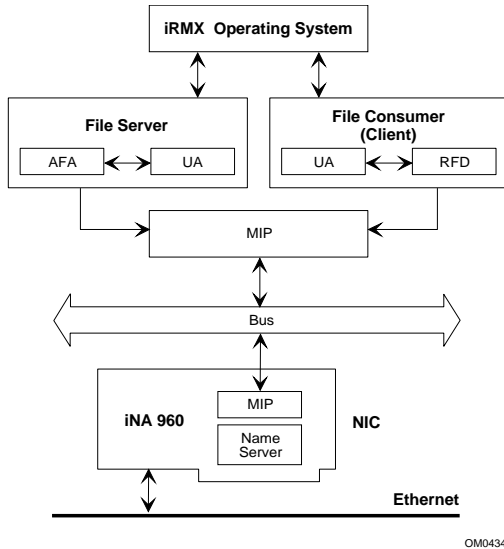


Figure 7-4. iRMX-NET Data Flow on COMMengine Systems

Configuring the MIP

In earlier releases of the OS you had to edit configuration files and use assembler macros to configure the MIP driver used with iNA 960 software in a COMMEngine environment. You now configure the MIP with the IMIPJ screen in the ICU.



This chapter introduces and defines the topology and addressing schemes used by the iNA Network Layer. *Topology* refers to how the network is physically or logically constructed. The topology of a network also plays an important role in determining how entities within the network are addressed.

The iNA 960 Network Topology

An iNA 960 network is one or more interconnected subnetworks, usually called subnets. A subnet is two or more connected end systems, as shown in Figure 8-1. An *end system* (ES) is a node that runs either the Null2 or the ES-IS Network Layer of iNA 960 software. Subnets are connected by intermediate systems or internetwork routers (also implementations of iNA 960 software). *Intermediate systems* (IS) handle relay and routing between end systems in one subnet and end systems in other subnets using the most efficient path. Depending on network topology, relay and routing may occur across two subnets or across many subnets.

Figure 8-1 illustrates a network consisting of a single subnet. Such a network includes only end systems. The end systems may use either a Null2 or ES-IS network job.

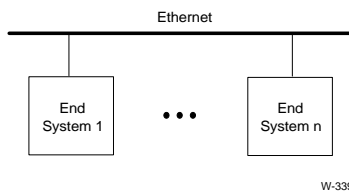
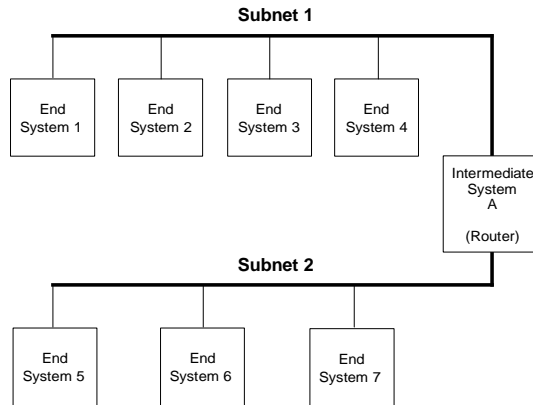


Figure 8-1. A Single Subnetwork

Figure 8-2 illustrates a network consisting of two subnets connected by an intermediate system (an internetwork router). The end systems typically run the ES-IS version of iNA 960, which enables either static or ES-IS routing.



W-2955

Figure 8-2. Two Interconnected Subnetworks

General Subnetwork Types

The basic building block for a network is the subnet. There are three generic types of subnets:

- Point-to-point subnet
- Broadcast subnet
- General topology subnet

A point-to-point subnet supports only two systems. The two systems can be either two End Systems or an End System and an Intermediate System.

A broadcast subnet supports an arbitrary number of end systems and intermediate systems. Any system in a broadcast subnet can transmit a single message to one, all, or some subset of the systems in the subnet. To transmit a message to all of the systems, the application uses a broadcast address defined for the particular subnet implementation. To transmit a message to a subset of the systems, the application uses a multicast address defined for that subnet implementation. An example of a broadcast subnet is one employing the IEEE 802.2 Type 1 LLC (Logical Link Control) and IEEE 802.3 MAC (Media Access Control).

Similar to a broadcast subnet, a general topology subnet supports an arbitrary number of end systems and intermediate systems. However, a general topology subnet may or may not support the broadcast and multicast transmission capability of a broadcast subnet. An example of a general topology subnet is one employing the IEEE 802.2 Type 1 LLC.

iNA 960 Subnetworks

iNA 960 subnet implementations are broadcast subnets which use the IEEE 802.2 Type 1 LLC and the IEEE 802.3 MAC.

Network Addressing

As defined in the OSI Reference Model, the layer entities on a node can communicate with their counterparts or peers on other nodes. The iNA 960 software implements a mechanism that enables peer entities to communicate over a network. The peers identify and locate each other with an identifier called an address.

There are one or more addresses for each layer entity; for example, there are transport (TSAP) addresses, network (NSAP) addresses, and data link or subnet addresses. This section discusses the iNA NSAP (Network Service Access Point) addresses and subnet addresses. TSAP (Transport Service Access Point) addresses are described in the Transport Layer chapter.

The NSAP address tells how to reach a user of the Network Layer services. In the iNA 960 software, the only user of the Network Layer services is the Transport Layer. An NSAP address tells iNA how to reach the Transport Layer services. Applications cannot directly access iNA Network Layer services.

Network Service Access Point (NSAP) Address

An iNA 960 NSAP is the equivalent of an IP address in TCP/IP protocols. However, unlike TCP/IP, iNA 960 does not make the NSAP address available to the user at the command line. You can set NSAP addresses in the */net/data* file and can access them programmatically.

See also: */net/data.ex* file, Chapter 11

The connection between the Network Layer and the Transport Layer is an NSAP, identified by an NSAP address. The NSAP address supplies the information needed by the Network Layer to identify the NSAP (and thereby the Transport Layer) at either a local or remote node. Because Network Layer users access services at NSAPs, the NSAP address is how a Network Layer user may be identified.

An NSAP address should not be confused with a data link or subnet (e.g., IEEE 802 MAC) address. Strictly speaking, an NSAP address is a logical address assigned by an addressing authority. For any network implementation, there may or may not be some syntactic relationship between an NSAP address and the subnet address that maps to it. A syntactic relationship is not required by ISO standards.

NSAP addresses are defined hierarchically. An NSAP address is assigned by an addressing authority. That authority may allocate a complete NSAP address or authorize a sub-authority to allocate addresses out of the authorizing authority's address space. The sub-authority may in turn authorize lesser authorities to allocate portions of its address space. Each authority administers a domain of the NSAP address space. The NSAP address structure mirrors this domain structure.

An NSAP address consists of two parts, an Initial Domain Part (IDP) and a Domain Specific Part (DSP). The IDP is further divided into two parts, an Authority and Format Identifier (AFI) and an Initial Domain Identifier (IDI). The AFI determines the format of the IDI and the syntax of the DSP. Given an AFI, the maximum length of an NSAP address is known. The AFI specifies the authority that allocates values of the IDI, and the IDI indicates which authorities allocate values for the DSP. The DSP may in turn have further structure as defined by the authority indicated by the IDI. Initial portions of an NSAP address are used for routing to a specific subnet while some latter portion in the DSP is used for determining a specific node in the subnet.

The least significant byte of the DSP is the NSAP selector. The selector indicates which one of the (possibly many) Transport Layer entities requesting service at an NSAP that the NSAP address refers to.

Subnet Address

In iNA 960, subnet addresses are much simpler than NSAP addresses. A subnet address identifies a Subnet Point of Attachment (SNPA). The SNPA is the conceptual point where a system is attached to a subnet. Like NSAPs, SNPAs are abstractions and their meaning in network implementations is up to the implementer. A subnet address is the addressing information that the Network Layer gives to the subnet service provider to indicate where the subnet service should send the message. iNA 960 subnet addresses are a concatenation of a node's 48-bit IEEE 802 Media Access Control (MAC) address and the Network Layer LSAP (Data Link Service Access Point) selector. By convention, the LSAP selector is FEH. The MAC address is more commonly called an Ethernet address and is typically assigned by the manufacturer of the network interface hardware, usually in PROM.

Internetwork Routing

The routing function maps an NSAP address supplied by the user to a subnet address that the subnet service understands. That subnet address may be the address of the message destination or the address of a router that will perform another mapping and relay the message to another router or to the message's destination. This routing function can occur in both end systems and intermediate systems.

See also: Internet routing, Chapter 16

iNA 960 Network Layer Addressing Schemes

The iNA 960 software supports two Network Layer addressing schemes. These addressing schemes recognize NSAP addresses that conform to the format described in IS 8348 Addendum 2:

- Null2, which is the inactive subset of the IS 8473 protocol, does not support internetwork routing
- ES-IS (end system to intermediate system) addressing supports two internetwork routing methods:
 - Static, which uses the MAP 2.1 routing scheme for mapping NSAP addresses to subnet addresses.
 - End system to intermediate system (ES-IS), for dynamic routing. This implements the protocol described in IS 9542.

The iNA 960 software is preconfigured with a Network Layer using either Null2 or ES-IS addressing. The ES-IS jobs support both static and dynamic (ES-IS) routing.

See also: *i*.job, System Configuration and Administration*

Null2 Network Addressing

iNA Network Layers configured for Null2 addressing recognize 11-byte NSAP addresses. When specified as a hexadecimal string, the Null2 address has this form:

```
4900xxYYYYYYYYYYYYFE00
```

Where:

49 The Authority and Format Identifier (AFI). By convention, the AFI for iNA NSAP addresses is 49H.

00xx A local subnet identification number. The second byte (xx) in a Null2 address has no meaning, but the value 0 is recommended.

YYYYYYYY The six-byte Ethernet address for the node.

FE The Data Link LSAP selector. By convention the iNA LSAP selector is 0FEH.

00 The NSAP selector. For Null2, this byte is optional. If not present, it is assumed to be 0. However, for compatibility with ES-IS routing, this byte must always be present.

Static Internetwork Addressing

The ES-IS configurations of iNA 960 support static internetwork addressing, where the application maintains static routing tables with NMF commands. Static routing recognizes two NSAP address formats. One format is the Null2 format described in the previous section. The other format is an extension of the Null2 format as shown in this:

```
49xxxxyyyyyyyyyyyyFE00
```

Where:

xxxx A two-byte subnet identification number.

yyyyyyyyyyyy
 The six-byte Ethernet address for the node.

The subnet ID specifies a particular subnet to which packets can be routed.

Except for the subnet ID, all other bytes in the address are the same as in a Null2 address. However, the NSAP selector (the last byte of the address) is not optional; it must be present. An NSAP selector of 0 specifies the Null 2 addressing scheme. For ES-IS addressing, the NSAP selector must not be 0.

ES-IS static addressing Network layers implement the MAP 2.1 static internetwork routing scheme. Nodes can be configured as end systems or intermediate systems (routers). Routing is determined by user-defined static tables located in intermediate systems.

See also: Routing tables, Chapter 16

End System to Intermediate System (ES-IS) Network Addressing

iNA Network Layers configured for ES-IS network addressing recognize the previously described Null2 and Static internetwork addresses.

⇒ **Note**

ES-IS addressing assumes that the last byte of an NSAP address is present and is the NSAP selector.

An NSAP address in an ES-IS environment is given meaning by routing tables located in End Systems and/or Intermediate Systems. These tables are dynamically updated based on configurable parameters. Systems in a network can periodically notify other systems of their existence and new systems can announce their presence at startup.

ES-IS configured Network layers implement the internetwork routing protocol described in IS 9542.

Choosing a Network Layer Configuration

The iNA 960 jobs are available in both Null2 and ES-IS versions. If you use the ES-IS version of a job, dynamic (ES-IS) routing is performed by default, unless you set up static routing tables with NMF commands. The routing algorithm checks both static and dynamic routing tables to see if a subnet can be reached. Preconfigured ES-IS versions of iNA 960 can store up to three static table entries (three intermediate system addresses).

Use these guidelines to help determine which Network Layer configuration is right for a particular subnet implementation.

A Null2 configuration may provide the best network performance under these conditions:

- The network has only one local subnet, with no internet router
- The node configurations are nearly homogeneous
- It is unlikely that nodes are to be added or removed often

A Static internetwork configuration performs similarly to an ES-IS configuration if both types of network remain stable. This is true whether the subnet configurations are homogeneous or nonhomogeneous. The key criterion is stability. A Static configuration is best where:

- The network includes multiple subnets whose membership is stable
- The internetwork router connections are fixed (nonexpanding)
- Systems on the network don't regularly go down

An ES-IS configuration may provide the best tradeoff between performance and ease of making changes, where:

- The network is large and subnet membership is constantly changing
- The network requires the flexibility to be easily changed
- The internetwork router connections change often



Configuring Networks with the Multibus II Subnet

Although iNA 960 has always supported routing in the ES-IS configurations, the software shipped with the iRMX OS did not always contain jobs preconfigured to support multiple NICs. Jobs without this support could not act as routers from one subnet to another.

The OS now includes iNA 960 jobs for Multibus II systems that support all possible combinations of NICs for those systems; these jobs act as routers between the subnets. In addition, some jobs support the *Multibus II subnet*, which is a virtual Ethernet interface across the backplane.

The Multibus II subnet enables all boards in a Multibus II system to act as independent network hosts. Each host has at least one network address and Ethernet address, regardless of whether a given board includes its own Ethernet controller. This allows communication between boards using iNA 960 (with or without iRMX-NET) or TCP/IP, using the Multibus backplane as the LAN medium. When the system includes at least one iNA 960 router between the Multibus II subnet and a hardware NIC, you have access to both this internal backplane network and any external Ethernet networks.

With the Multibus II subnet, an iNA 960 transport stack runs on every board in the Multibus II system. With a separate Ethernet address and iNA transport stack, each board can run TCP/IP independently. This eliminates the need for systems where:

- Boards without NIC hardware had to set up a COMMengine environment (MIP job) to share a single transport stack that ran on the board with NIC hardware.
- For boards that ran MIP jobs, GDT slots 4096-4767 were unavailable for use by the application.
- Only one Ethernet address (per NIC) applied to all boards in the system.
- Only one iNA 960 RawEDL client was allowed in the system, which meant that only the host board with a NIC could run TCP/IP.

This chapter describes mostly the configuration changes needed to use the Multibus II subnet, with or without routing to an external subnet. However, some iNA 960 jobs support multiple hardware NICs, and route between subnets without using the Multibus II subnet. The routing principles described here also apply to those jobs.

Routing Between Subnets

Any Multibus II board that has an Ethernet controller connecting to an external network and that also uses the Multibus II subnet must be set up as a router between the two subnets. If you use both iNA 960/iRMX-NET and TCP/IP on the external subnet you must configure the board to be both an ES-IS router for iNA 960 and an IP router for TCP/IP.

⇒ **Note**

The descriptions in this chapter apply only to configuring iNA 960 routers. For details about configuring TCP/IP, see *Configuring TCP/IP for the Multibus II Subnet, TCP/IP and NFS for the iRMX Operating System*

Definition of a Router

This manual refers to the hosts that transfer packets between subnets as routers, whether they use iNA 960 or TCP/IP networking. As used here, the term “router” is synonymous with the term “gateway” as commonly used in TCP/IP literature.

Some people define a gateway to mean a system that not only separates different segments of network, but also translates protocols as it passes packets between the networks. The term router used here does not mean that type of gateway.

The discussion of configuring routers applies only to iRMX systems used to route packets between subnets. It is beyond the scope of this manual to describe configuration of any independent routers you may use to separate interconnected subnets. However, use the principles described here when you do any such configuration. For example, if you support iNA 960 protocols across an independent router, you must assign unique iNA 960 subnet IDs to each subnet.

ES-IS vs. Null2 Jobs

There are two basic kinds of network jobs defined by iNA 960:

- ES-IS jobs are capable of routing network packets. Jobs with names that end in *e* are ES-IS, for example, *imix560e*.
- Null2 jobs are not capable of routing network packets. Jobs with names that end in *n* are Null2, for example, *imix560n*.

An ES-IS job is not necessarily a router, but it forms network addresses so that they can be routed. Only ES-IS jobs that support multiple subnets can act as routers.

ES-IS Routing

Each iNA 960 ES-IS job maintains tables of systems it can contact. The tables contain the names and network addresses of those other systems. There are two parts to these routing tables:

- Static routing tables specify all the other Intermediate Systems that can be contacted from the local host (be it ES or IS). You must set up the static routing information on each IS.
- Dynamic routing tables are built by each system based on *hellos*, or acknowledgments sent periodically between End Systems and Intermediate Systems. Each IS builds a list of all ES systems with which it has direct contact. Conversely, each ES builds a list of the IS(s) with which it has direct contact. You set up each board to be an ES or IS by specifying what types of hellos it sends and receives. Based on the hellos it receives, each system automatically builds its own dynamic routing tables.

With the static and dynamic routing tables in place, you can send a packet from an ES on one subnet to an ES on another subnet, without any direct knowledge of the network path required to get there. When you send the message, your ES puts the packet on your subnet. If the packet has a different subnet ID than your own, the IS forwards the packet to whatever IS in its routing tables is specified to handle that subnet. If the destination ES is on the subnet connected to the second IS, the packet is delivered there.

However, the ES you are attempting to contact may be separated from your subnet by at least one intermediate subnet, each connected by one or more ISs. If so, the second IS passes the packet along to whatever IS is specified in its (the second IS's) static routing tables to handle the destination subnet, and so on. Through this process, the packet is routed to the subnet for which it is intended, and is received by the destination ES.

If there are multiple ISs on the same subnet, each one needs information about which subnets the other ISs can reach. You add this information when you set up the static routing tables.

Later sections in this chapter describe the process of setting up iNA 960 routers.

See also: iNA 960 Topology and Addressing, Chapter 8
Internetwork Routing, Chapter 16

Ethernet Addresses in the Multibus II Subnet

An Ethernet address is a six-byte hexadecimal value used to identify a particular host on the network. The Ethernet address is also called a MAC (media access control) address. Typically, MAC addresses are coded into the firmware on an Ethernet network interface controller, or NIC.

Since there is no Ethernet hardware associated with the subnet on the backplane, the iNA 960 jobs for the Multibus II subnet assign MAC addresses to each board according to the slot number. In every system the base address is A2 A4 A6 A8 AA 00. This is the MAC address of the board in slot 0. The software sets the last byte of the address to the slot number. The board in slot 1 has address A2 A4 A6 A8 AA 01, and the addresses progress to A2 A4 A6 A8 AA 13 for a 20-slot system. This ensures that each board on the subnet has a unique MAC address.

Although every Multibus II subnet uses the same range of MAC addresses, the combination of a unique subnet ID with the MAC address provides a unique iNA 960 network address for each board. With TCP/IP software, you assign a unique IP address to each board.

Router boards that use the Multibus II subnet and also have one or more hardware NICs have multiple MAC addresses. The virtual NIC provided by the Multibus II subnet is assigned the MAC address described above. Each hardware NIC has its own MAC address embedded in the firmware.

Data Link Subsystem ID for the Multibus II Subnet

In programming calls to iNA 960 you specify a subsystem ID as part of the request block (RB) interface. The subsystem ID specifies the iNA 960 subsystem being called. For the Data Link layer, there are several subsystem IDs, depending on the type of subnet in use. To specify the Data Link for the Multibus II subnet, use the subsystem value 2FH.

See also: Data Link calls, Chapter 13

Name Server Search Domain

When you use the **attachdevice** command to connect to a remote system, the iNA 960 Name Server searches only subnet ID 1 by default. If your network includes multiple subnets, or even if it includes only one subnet but you have changed the subnet ID from the default, you must set the Name Server search domain to include all appropriate subnet IDs. There are two ways to do this: in the ICU configuration or with a **domain** command. The instructions in this chapter describe using both methods.

See also: **domain** command, *Command Reference*

You need to set the search domain only if you use iNA 960/RMX-NET across different subnets. For TCP/IP access only, the Name Server search domain is not needed.

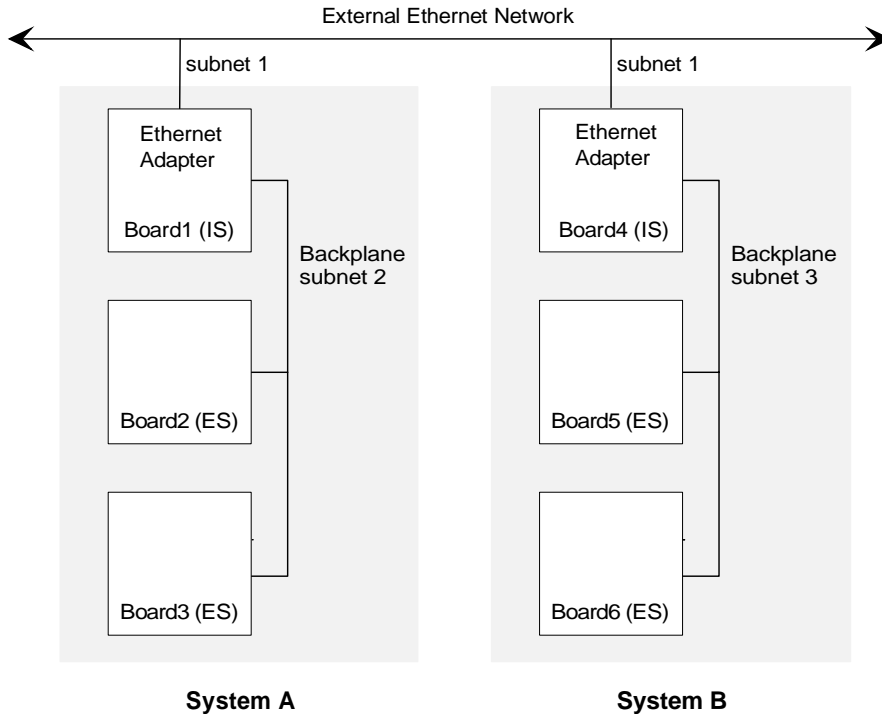
Overview of Setting up the Multibus II Subnet

These are the steps you will perform to make use of the Multibus II subnet. Some steps are optional, depending on your system and the kind of networks you want to use:

- Make a map of the total network to identify what iNA 960 subnet IDs you need to assign.
- Choose the correct iNA 960 job(s) for your hardware.
- Configure the iNA 960 job(s) into the OS with the ICU or set up a loadable version of the job. In either case, you may have to change the subnet IDs configured into the job, depending on your subnet map of the network.
- Optionally change subnet IDs on other systems in the network to match those set up in your Multibus II system(s).
- Use either the **inamon** utility or your application program to set up iNA 960 routing tables.
- If you plan to use TCP/IP on more than one board in the system, set up the TCP/IP configuration files for the Multibus II Subnet and (optionally) for routing.

Step 1: Mapping the Network

Before you begin configuring the network software, set up a map of your network and determine what iNA 960 subnet IDs to assign. For example, Figure 9-1 shows a simple network consisting of two Multibus II systems. Each is attached to an external network through an Ethernet controller. This can be any onboard Ethernet controller, such as on the SBC 486/166SE board.



OM03562

Figure 9-1. Mapping Subnets

The map of this network assigns subnet ID 1 to the Ethernet connections. The backplane of each system has a unique subnet ID, so that System A's backplane is subnet 2 and System B's is subnet 3.

Boards 1 and 4 are routers, called Intermediate Systems (ISs) in iNA 960 terminology, because they have two network connections and connect two subnets. In each case, one of the subnets is the Multibus II subnet, and the other subnet is the external Ethernet connection.

Boards 2, 3, 5, and 6 are End Systems (ESs), because they have a single network connection, in this case the Multibus II subnet. You typically configure Intermediate Systems to also be End Systems, but it is pointless to configure a system with only one network attachment to be an Intermediate System.

Although it is not mandatory to assign subnet ID 1 to the external Ethernet connection, this simplifies the work in configuring the systems. It also allows for compatibility with existing systems on the external network that do not route packets, since Null2 networking jobs are preconfigured to use subnet 1.

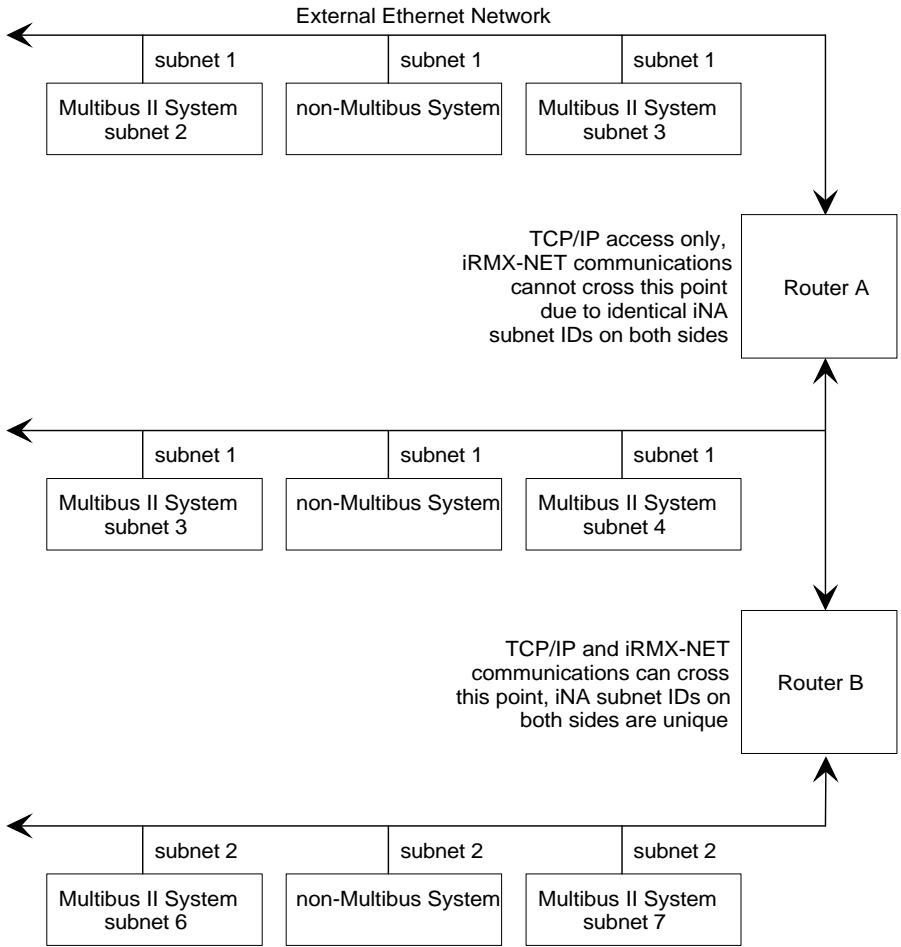
⇒ **Note**

If necessary, you can configure a Null2 job to use a different subnet ID, using the ICU. See Step 6, on page 91.

You could have a more complicated internetwork scheme with multiple external subnets connected by routers. If you use only TCP/IP to access interconnected external subnets, each external subnet can have the same iNA 960 subnet ID. For example, you could have the situation shown in the top and middle external subnets of Figure 9-2, where the two subnets connected by Router A are both subnet 1, and two of the Multibus systems on either side of the router are both subnet 3. You can use iNA 960/iRMX-NET on both subnets with ID 1, because each Multibus II subnet attached to each subnet 1 has a unique subnet ID. (The non-Multibus systems on these subnets use subnet ID 1, since there is no internal subnet.)

However, to use iNA 960 or iRMX-NET through a router, all subnets on either side of the router must have a unique iNA 960 subnet ID. This is shown in the middle and bottom external subnets of Figure 9-2, on either side of Router B. Not only are the external subnets unique (IDs 1 and 2), but each Multibus II subnet has a unique ID.

In the example in Figure 9-2, you would configure Router A as a TCP/IP router only. You would configure Router B to be both a TCP/IP router and an iNA 960 Intermediate System.



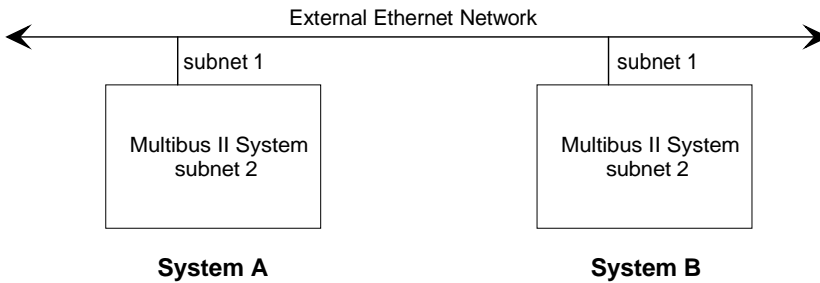
OM03563

Figure 9-2. Mapping Subnets with an Internetwork

Using Only TCP/IP Outside the Multibus II Subnet

You might want to use iNA 960 and/or iRMX-NET transport services within the Multibus II system, but not across the external network. In that case, all external subnets could have ID 1, and all Multibus II subnets could have ID 2 (or any other ID), as illustrated in Figure 9-3. In this example, boards within System A can communicate with each other using iNA 960/iRMX-NET, and so can boards in System B, but they must use TCP/IP to communicate between the systems.

In the example in Figure 9-3, you would not configure any iNA 960 Intermediate Systems. You would configure the boards connected to the external network as TCP/IP routers, or gateways.



OM03564

Figure 9-3. Mapping Subnets for TCP/IP Access, but no iNA 960 Access

Step 2: Choosing the iNA 960 Jobs

For every board in the system, choose one of the iNA 960 jobs listed in Table 9-1. Each job is preconfigured with one or more subnets. Table 9-1 indicates which subnet is used for an onboard Ethernet NIC and which is used as the Multibus II (backplane) subnet. The order is important when you assign the subnet IDs, as described in subsequent steps.

For boards that will act as routers, choose the correct multiple-subnet job for that board. For boards that do not act as routers, use the *impe* job to give access to the Multibus II subnet and the iNA 960 transport stack without using a hardware NIC. If the system does not contain a router, you can use the Null2 version of the Multibus II subnet job, *impn*, but boards that use this job cannot send or receive packets across a router to another subnet. Table 9-1 also lists jobs that do not include the Multibus II subnet, such as the *ihiexe* job. Boards that use such jobs cannot use the backplane as a network connection. These jobs support routing between multiple external subnets, but not to the Multibus II subnet.

All jobs are available either as loadable or linkable jobs. Choose the type of job according to your OS configuration for each board. For example, where you configure the application into ROM, or on a diskless application (**rsd.bck* definition file), you must use the ICU to include the linkable version of the job. Otherwise, you can use a loadable job.

Table 9-1. iNA 960 COMMputer Jobs for the Multibus II Subnet

Job	OS	Subnets	NICs (and Default Subnet IDs) for 1st, 2nd, 3rd, 4th Subnet
ihiexe	III	2	SBC 486/1xxSE (1), SBX 586 (2)
ihimpe	III	2	SBC 486/1xxSE (1), MB II backplane (2)
ihiexmpe	III	3	SBC 486/1xxSE (1), SBX 586 (2), MB II backplane (3)
imxmpe	III	2	1 MIX560 (1), MB II backplane (2)
i2mxe	III	2	2 MIX 560s (1, 2)
i2mxmpe	III	3	2 MIX 560s (1, 2), MB II backplane (3)
i3mxe	III	3	3 MIX 560s (1-3)
i3mxmpe	III	4	3 MIX 560s (1-3) MB II backplane (4)
ie1mpe	all	2	SBC 486SX/DXxx with EWENET (1), MB II backplane (2)
ie2mpe	all	2	SBC P5090 or P5120 (1), MB II backplane (2)
ie3mpe	all	2	SBC P5200(1), MB II backplane (2)
imp?	all	1	MB II backplane only (1)

? Specify N for Null2 (no routing capability) or E for ES-IS routing
 III/all iRMX III OS only, or any of iRMX for PCs, DOSRMX, or iRMX III OS

Table 9-1 shows the default subnet ID(s) associated with each subnet in a job. The default subnet IDs may not match the ones you have assigned in your map of the network. If so, change the subnet IDs either when you load the job (with SNIDx parameters in the **sysload** command) or by reconfiguring the job with the ICU. The process of using the ICU to generate either first-level or loadable iNA 960 jobs is described in the next steps.

⇒ **Note**

You can use Multibus II subnet jobs on some boards in the system along with non-Multibus II subnet jobs on other boards in the same system. In other words, you might still choose to use a MIP job on a board for which you do not need to use the Multibus II subnet.

However, do not use a MIP job on any board for which you choose a Multibus II subnet job. Also, do not use a MIP job on one board as an interface to a Multibus II subnet version of iNA 960 running on another board.

Table 9-1 does not list any of the MIP jobs or other iNA 960 jobs that do not relate to the Multibus II subnet or to routing.

See also: *i*.job, System Configuration and Administration* for a complete list of iNA 960 jobs

Step 3: Configuring Jobs in the ICU

If you use only the loadable iNA 960 jobs provided with the OS, ignore the instructions in this step and proceed to Step 5. However, you can use the instructions in this step and Step 4 to produce a loadable job that has different default subnet IDs than the loadable jobs supplied with the OS.

For each board where you use a linkable iNA 960 job or need to change the configuration of a loadable job, invoke the ICU and configure the job as follows:

- A. NET screen: Set the MIP parameter to No and the CMP parameter to Yes, to include an iNA 960 COMMputer job. (You may also choose to add the iRMX-NET server and client, and/or TCP/IP on this screen.)
- B. ICMPJ screen: Specify the iNA 960 job name in the OFN parameter. Use one of the names shown in Table 9-1 without a *.job* extension.
- C. ICMPJ screen: For the *imprn* job, leave the Network Layer, or NL parameter set to 1 for a Null2 network. For any other job, set NL to 3, for ES-IS.
- D. ICMPJ screen: Specify the appropriate subnet IDs in the SN1 through SN4 parameters. For example, Table 9-1 shows that for the *imxmpe* job, the first subnet applies to the MIX560 NIC and the second subnet is the backplane. Set the values according to your subnet ID map of the network. If your external network is subnet 3 and your internal network is subnet 7, set the SN1 parameter to 3 and the SN2 parameter to 7. Do not set any SN* parameters that do not apply; the *imxmpe* job contains only two subnets and you cannot add more.
- E. NSDOM screen: Specify all subnet IDs that you want the Name Server to search when you attach to a remote system. Use your map of the network and include all subnet IDs to which you want to connect with either iNA 960 or iRMX-NET. The maximum number of subnets to be searched is 80 (the ICU displays a new screen for each set of 20 IDs). You can specify subnet IDs not currently in use, for future expansion. However, adding more subnet IDs to the search domain slows down Name Server operations.
- F. If you plan to configure the iNA 960 job into the OS, make sure that on the ICMPJ screen the CLJ parameter is set to No. Then continue with any other necessary ICU configuration. Generate the system as usual and submit the *.csd* file produced by the ICU to build the OS image. However, if you want to produce a loadable job with the new configuration, do not generate the system now, but proceed with the instructions in Step 4.



Note

Some preliminary instructions for using the Multibus II subnet described these changes: On the FC screen for the iRMX-NET File Consumer, you were instructed to change the DDS parameter from 1488 to 1344. On the iRMX-NET File Server AFAU screen where the USS parameter is set to 70H, you were instructed to change the SBS parameter from 1488 to 1344. These instructions applied only for preliminary software shipped prior to release 2.2 of the OS. Do not change the default value of 1488 for these parameters. The value 1488 is required to work with all versions of iNA 960 jobs shipped with the OS.

Step 4: Creating a Loadable Network Job

To create a loadable job from a linkable job that you have configured, as in Step 3, follow this process:

- A. ICMPJ screen: While in the ICU, make any configuration changes you need, such as changing the subnet IDs for the job.
- B. ICMPJ screen: Specify the new network job name in the OFN parameter.
- C. ICMPJ screen: Set the CLJ (create loadable job) parameter to Yes.
- D. When done, use the ICU Generate command to generate the system, but don't submit the *.csd* file produced by the ICU.
- E. To create the loadable iNA 960 job, submit the *icmp.csd* file produced by the ICU. This generates the loadable job in the directory where you invoked the ICU, with the name of the linkable job and a *.job* extension.
- F. If you made configuration changes to the iRMX-NET server, create a loadable version of the job by submitting the *metsrv.csd* file produced by the ICU. This generates a new *metsrv.job* file in the directory where you invoked the ICU.
- G. If you made configuration changes to the iRMX-NET Consumer (the client and remote file driver job), create a loadable version of the job by submitting the *metcln.csd* file produced by the ICU. This generates a new *remotefd.job* file in the directory where you invoked the ICU.
- H. Add the new loadable job(s) to the *loadinfo* file as described in Step 5.

Step 5: Using Loadable Jobs

If you use only linkable iNA 960 jobs, ignore the instructions in this step and proceed to Step 6.

- A. For each board in the system on which you use loadable network jobs, choose the appropriate job from Table 9-1.
- B. If the job's default subnet IDs are not correct for your map of the network, configure new subnet IDs and create a loadable job from the linkable job as described in Steps 3 and 4.
- C. Edit the *rmx386/config/loadinfo* file to remove or comment out the **sysload** invocation of any current iNA 960 jobs. Add a line to load the new iNA 960 job, specifying the job's pathname and *.job* extension, for example:

```
sysload /rmx386/jobs/imxmpe.job
```

- D. If you did not set up the job's Name Server search domain in the ICU with the NSDOM screens, use the **domain** command to set the search domain of all subnets the Name Server will access. You can add the command to the *loadinfo* file following the **sysload** command that loads the iNA 960 job. The syntax is:

```
domain [-a ID[-range]] [-d ID[-range]]
```

Without any parameters, **domain** displays the current search domain. The *-a* parameter adds one or more subnet IDs. The *-d* parameter deletes one or more. With either parameter, **domain** displays the current search domain after the addition or deletion. Specify either a single subnet ID or a range of IDs, separated with a dash (-) and no spaces. The ID must be a four-digit hexadecimal number followed by an H.

Example 1: To add subnet 4 to the current search domain, enter:

```
domain -a 0004H
```

Example 2: Assume that your external subnet ID is 3, with Multibus II subnets 4, 5, and 6. Your external subnet is connected through a router to subnet 1, which contains Multibus II subnets 10 through 26 (0AH through 1AH). To enable searching of all subnets from 1 to 1AH, enter:

```
domain -a 0001H-001AH
```

The maximum number of subnets to be searched is 80. You can specify subnet IDs not currently in use. However, adding more subnet IDs to the search domain slows down Name Server operations.

Step 6: Changing Subnet IDs on Other Systems

You may have non-Multibus II systems on your network that use either ES-IS or Null2 versions of iNA 960 jobs. The Null2 jobs (*i*n* jobs) do not implement routing. However, both types of jobs will work properly on a network with an IS as long as their subnet IDs match the one you assign to their subnet.

iNA 960 jobs with a single subnet are preconfigured with subnet ID 1. If you set up the external network to be subnet 1, or if you don't have any non-Multibus II systems using iNA 960 jobs on the network, ignore this step.

If you assign the external network any subnet ID other than 1, you must change the subnet ID for both Null2 and ES-IS network jobs used by any systems on that network:

- A. Follow the instructions in Steps 3 and 4 to produce either linkable or loadable versions of the jobs.
- B. For loadable jobs, make a backup copy of the original iNA 960 job shipped with the OS, then copy the new version of the job from your ICU generation directory to the `\mx386\jobs` directory on the target machine.
- C. For loadable jobs, install the job as described in Step 5.

Step 7: Modifying the net/data File

For every diskless board (**rsd.bck* files), you must change the network address for the remote boot client in the */net/data* file on the file server. This applies to any system that boots remotely, not only to remote-boot clients on Multibus II subnets.

iRMX-NET file servers that support remotely booted nodes have an entry in the */net/data* file that contains the Ethernet (MAC) address of the client with the Multibus II slot ID appended. The slot ID field is 00 for Multibus I systems and PCs. For example, in previous releases of the OS, the */net/data* entry for a client in slot 2 was like the one below, where the address is the MAC address used by the client followed by the slot number:

```
client_name:  TYPE=PT0005: ADDRESS=00AA00021E2702;
```

OM03570

This address has been expanded by two bytes to also include the subnet ID, as shown below:

```
client_name:  TYPE=PT0005: ADDRESS=0005A2A4A6A8AA0202;
```

OM03571

- A. Modify the */net/data* entry for every remote boot client to use the second form shown above. Substitute the client name, subnet ID, MAC address and slot number in your entries. Note that the MAC address in the example above is the one imposed by the Multibus II subnet, where the last byte of the MAC address is also the slot number.

See also: */net/data.ex* file, Chapter 11

Step 8 - 10 Overview: Configuring iNA 960 Routing

To set up routing on an iNA 960 job that supports multiple subnets, you need to change a variety of network objects in tables that control routing and interaction at the network level between ESs and ISs. To change network objects, you can:

- Write an application program that modifies the objects directly or that accepts user input regarding addresses and system names the router needs to contact.
- Use the **inamon** utility to modify the objects directly from the command line.
- Modify and run a submit file supplied with the OS that invokes **inamon** to supply the necessary information.

The following discussion and Steps 8 through 10 describe an example of the last method: using the *iset.csd* submit file to invoke **inamon** and change the appropriate network objects for routing. If you want to set up routing in another way, use this discussion as an example of what objects to examine and modify.

See also: List of network objects, Appendix C
Chapter 16 for information about programmatically changing objects
inamon, Command Reference

Using Inamon to Configure Routing

The OS installs a submit file, */net/iset.csd*, that you can use to set iNA 960 ES-IS routing objects. The submit file invokes the **inamon** utility in batch mode and changes network objects according to the way you edit the file. You can invoke it from the *:config:loadinfo* file to automatically set up the network for each board, using this syntax:

```
submit :sd:net/iset
```

For each board, make a copy of the file you will submit and edit the file according to these guidelines (described in the following steps):

- On every board that uses the Multibus II subnet, set four flags that determine whether the system is an ES, IS, or both.
- On IS boards only, configure the static routing tables, which specify all other IS routers that can be contacted from this IS.

The format of the *iset.csd* file is shown in Figure 9-4 on page 94; the line numbers are not part of the file. Edit the file in a three-step process as described in the following sections.

Line #

```
1. inamon batch
2. set 3912 ff
3. set 3913 ff
4. set 3915 ff
5. set 3916 ff
6. ; 3 set ST 49 00 03 00 aa 00 03 24 9f fe 00 ROUTER3 SBX586 00 aa 00
   03 17 a3 fe
7. ; exit
8. ; 3 set N ROUTER3 49 00 03
9. ; exit
10. exit
```

Figure 9-4. Example iset.csd File

Step 8: Establishing ES and IS Hellos

Table 9-2 lists the network objects that determine whether a board is an ES, an IS, or both. In this step you will set up the appropriate objects to make each board operate in the ES-IS protocol.

Table 9-2. Configuring ES and IS Hellos

Object #	Object Name	ES only	IS only	ES and IS
3912H	Send ES Hellos	True (FF)	False (00)	True (FF)
3913H	Send IS Hellos	False (00)	True (FF)	True (FF)
3915H	Receive ES Hellos	False (00)	True (FF)	True (FF)
3916H	Receive IS Hellos	True (FF)	False (00)	True (FF)

- A. For every board in the system, make a copy of *iset.csd*. Edit lines 2-5 of each file to specify whether the board is an ES, IS, or both. Set the final value in each line to 00 (false) or FF (true) according to Table 9-2. In most cases, you should configure every IS to also be an ES. The only reason you would not do this is when the IS is used purely as a router, not as a workstation.

As Table 9-2 indicates, an End System is one that sends ES hellos and receives IS hellos; it does not receive and process hellos from other End Systems. An Intermediate System is one that sends IS hellos and receives ES hellos. An Intermediate System does not receive and process hellos from other Intermediate Systems unless it is also an End System.

- B. Leave lines 6-9 of *iset.csd* commented out with a semicolon for this step. **Inamon** does not recognize the semicolon as a comment, but it does not perform any commands that begin with a semicolon. Instead it issues an error message. When you submit *iset.csd* later in this step, **inamon** will display this message for each commented line:

```
Invalid command
```

You can ignore this error message.
- C. Add the command `submit <pathname>iset` to the `:config:loadinfo` file, after the invocation of any loadable network jobs and any **domain** commands. If you use different files to load different sets of jobs for remotely-booted boards, add the appropriate submit line to the file for each board.
- D. Assuming you have done all the other configuration steps to this point, reboot the system. Do this for all Multibus II systems on the network.
- E. With the systems running the Multibus II subnet jobs, and each board in its ES-IS configuration, remotely-booted boards should boot properly. This indicates that the Multibus II subnet on each system is configured correctly. If remotely-booted boards do not boot, review the configuration steps to this point.

The next step is to use **inamon** to get the routing information needed to complete the editing of *iset.csd*. This is described in the following section.

⇒ **Note**

If you use only TCP/IP outside the Multibus II subnet, you do not need to configure the iNA 960 static routing tables on IS boards. Remove lines 6-9 of the *iset.csd* file. At this point in the configuration, you can ignore Steps 9 and 10, and proceed to set up TCP/IP.

See also: *Configuring TCP/IP for the Multibus II subnet, TCP/IP and NFS for the iRMX Operating System*

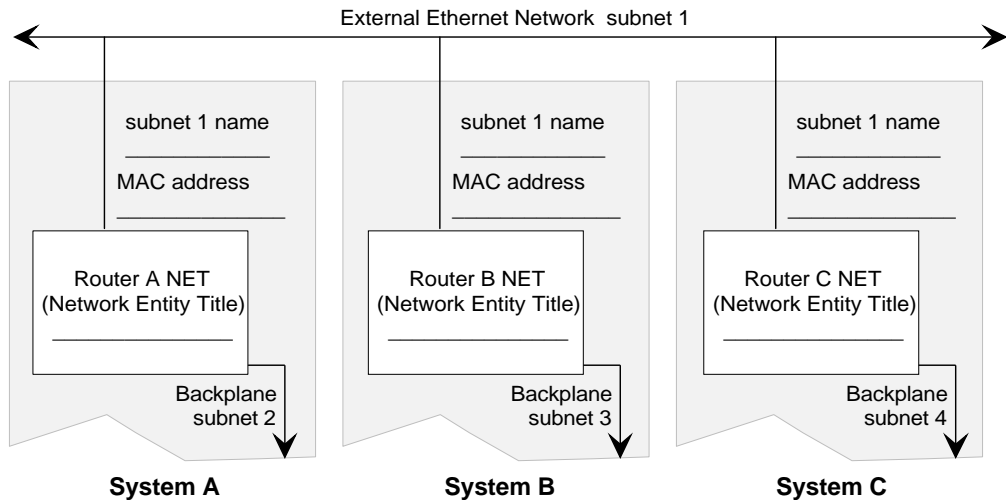
If you use iNA 960 and/or iRMX-NET outside of the Multibus II subnet (in other words, on external subnets), continue with Step 9.

Step 9: Getting the NET and Subnet Information

On every Intermediate System there are three items of information you need to obtain:

- Network Entity Title (NET), which is the primary NSAP address used by iNA 960 for this IS (an NSAP address is the iNA 960 term for a network address, or Network Service Access Point).
- Subnet name(s) used by this IS for any subnet(s) that lead to any other IS
- MAC address(es) corresponding to the subnet(s) that lead to any other IS

Use the map of your system to record this information so it is available for Step 10. For example, assume your network looks like the one in Figure 9-5. As you perform this step, fill in the information shown in the blanks in this figure.



OM03565

Figure 9-5. Routing Information on a Single External Network

⇒ Note

Each board may have a different name for the subnet it is attached to, depending on the iNA 960 job running on that board. For example, in Figure 9-5, there is not a single name for subnet 1. If the three IS systems run different iNA 960 jobs, each may have a different name for subnet 1.

- A. Invoke **inamon** on each IS.

- B. Type 3 to get the Router Management menu.
- C. Examine object 391E, which displays the subnet table, similar to the following. The items you're interested in are the subnet names and IDs:

```
[ 391E]   SUBNET TABLE

I82596      / 08H / FEH / 49 00 01 1 000000FE
MPSN       / 08H / FFH / 49 00 02 0 000000FE
```

OM03572

- D. Refer to your map and record the name(s) of any subnet connections that lead to other ISs. In the example listing above, if there were other ISs on both subnets 1 and 2, you would record I82596 as this board's name for subnet 1 and MPSN as this board's name for subnet 2.
- E. Now examine object 3919, which displays NSAP addresses for this host, similar to the following. An ES would have only one NSAP address, but each IS has multiple addresses. The first address in the list is the NET, which is the primary NSAP address used for this host. Embedded in each NSAP address is a subnet ID and MAC address.

```
[ 3919]   LOCAL NSAP ADDRESSES

49 00 01 00 AA 00 03 17 E3 FE 00 ← NET
49 00 02 A2 A4 A6 A8 AA 02 FE 00
```

OM03573

- F. Record the NET for this IS.
- G. Refer to your map and record the MAC address(es) of any subnet connections that lead to other ISs. In the example listing above, if there were other ISs on both subnets 1 and 2, you would record MAC address 00 AA 00 03 17 E3 as the connection to subnet 1 and A2 A4 A6 A8 AA 02 as the connection to subnet 2.
- H. Exit **inamon** by typing E twice.

Step 10: Setting Up the iNA 960 Static Routing Tables

As shown in Figure 9-4 on page 94, lines 6 and 8 of *iset.csd* are examples of the two pieces of information you need to establish on each IS.

Line 6 defines a path to another IS that has access to a remote subnet. It adds an entry to the static routing table on this IS. The format is:

```
3 set ST 49 00 03 00 AA 00 02 1E 27 FE 00 ROUTER3 SBX586 00 AA 00 02 1E 27 FE
```

3	set	ST	49	00	03	00	AA	00	02	1E	27	FE	00	ROUTER3	SBX586	00	AA	00	02	1E	27	FE
Sets a static route	NET of destination router			Arbitrary name for destination router			Source router's name for subnet leading to destination router			MAC address of port on the destination router that is attached to this subnet			Always FE, standard LSAP (link service access point)									

Invokes menu 3, Router Management in inamon

OM03574

Line 8 tells iNA 960 to use that IS for access to a specific subnet. It adds an entry to the NSAP reachable table on this IS. The format is:

```
3 set N ROUTER3 49 00 05
```

3	set	N	ROUTER3	49	00	05
Subnet to be reached through this router	Always 49, standard Authority and Format Identifier (AFI)			Name specified for destination router in previous line		
Sets this value in the NSAP reachable table			Invokes menu 3, Router Management in inamon			

OM03575

- A. For each subnet to be reached, add a pair of such lines to *iset.csd*. For example, if all routers are attached to a single external network as shown in Figure 9-6 (page 100), every router points to all the other routers on the network. Router A uses two lines in *iset.csd* to point to Router B and two lines to point to Router C. Router B points to A and C, while Router C points to A and B.
1. In the first line (similar to line 6 of *iset.csd*), substitute:
 - NET of the destination router
 - An arbitrary name for that router
 - Name of the subnet leading to the destination router, as defined by this (the source) router
 - MAC address of the destination router on the subnet that connects the two routers
 2. In the second line (similar to line 8 of *iset.csd*), substitute:
 - The name of the destination router that you used in the first line
 - Subnet ID of the subnet that can be reached through the destination router

For example, consider the possible routing information shown in Figure 9-6 on page 100 and Figure 9-7 on page 102.

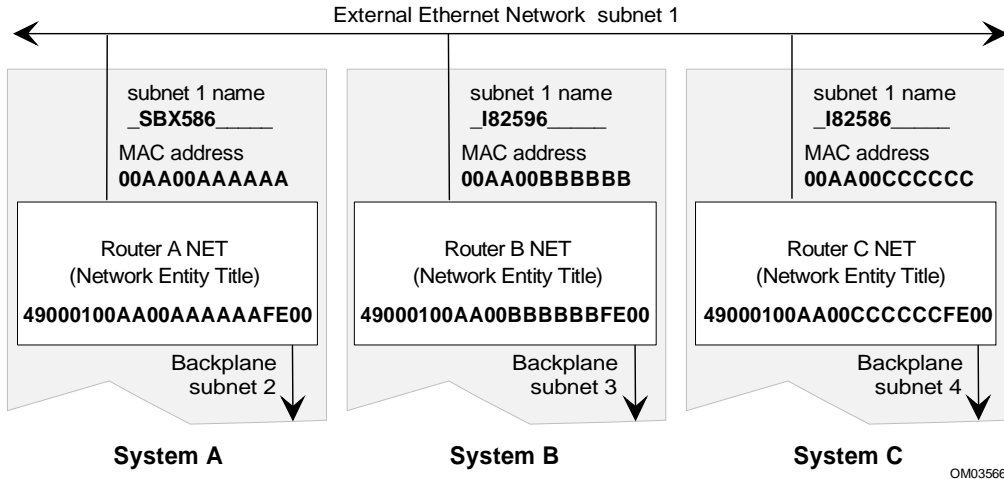


Figure 9-6. Example Routing Information on a Single External Network

Assume that Routers A, B, and C have the NET values, MAC addresses, and local names for subnet 1 that are shown in Figure 9-6. You would add the lines shown below to the *iset.csd* files for the three routers (in place of lines 6-9 as shown in Figure 9-4). You could substitute your own names for Router2, Router3, and Router4 in these lines.

Router A

```
3 set ST 49 00 01 00 AA 00 BB BB BB FE 00 ROUTER3 SBX586 00 AA 00 BB BB BB FE
exit
3 set N ROUTER3 49 00 03
exit
3 set ST 49 00 01 00 AA 00 CC CC CC FE 00 ROUTER4 SBX586 00 AA 00 CC CC CC FE
exit
3 set N ROUTER4 49 00 04
exit
```

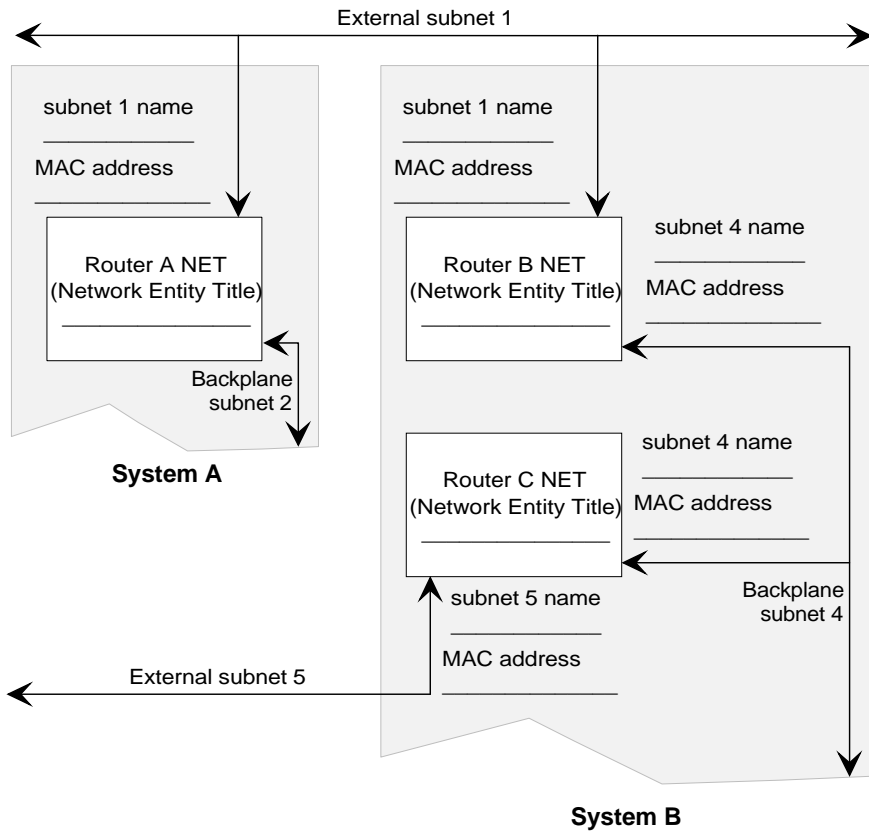
Router B

```
3 set ST 49 00 01 00 AA 00 AA AA AA FE 00 ROUTER2 I82596 00 AA 00 AA AA AA FE
exit
3 set N ROUTER2 49 00 02
exit
3 set ST 49 00 01 00 AA 00 CC CC CC FE 00 ROUTER4 I82596 00 AA 00 CC CC CC FE
exit
3 set N ROUTER4 49 00 04
exit
```

Router C

```
3 set ST 49 00 01 00 AA 00 AA AA AA FE 00 ROUTER2 I82586 00 AA 00 AA AA AA FE
exit
3 set N ROUTER2 49 00 02
exit
3 set ST 49 00 01 00 AA 00 BB BB BB FE 00 ROUTER3 I82586 00 AA 00 BB BB BB FE
exit
3 set N ROUTER3 49 00 03
exit
```

To reach subnets that are separated by more than one IS, the first IS points to the next IS, which in turn points to the next. For example, Figure 9-7 shows two Multibus II systems. System A contains one IS, Router A. System B has two ISs, Routers B and C. For boards in System A to send messages on external subnet 5, Router A must send packets through Router B, which forwards the packets to Router C.



OM03567

Figure 9-7. Routing Information on Multiple External Networks

- B. For multiple router hops as shown in Figure 9-7, include lines like the following in the *iset.csd* files for the routers. Make sure to add `exit` statements after each line, as show in Figure 9-4 on page 94.

Router A

```
3 set ST <NET_of_IS_B> ROUTER4 <IS_A_name_of_subnet_1> <IS_B_MAC_on_subnet_1>FE
3 set N ROUTER4 49 00 04
3 set N ROUTER4 49 00 05
```

The first line above defines the name Router4 (it could be any name) for Intermediate System B. The second line says to use Router4 for access to subnet 4. The third line says to also use Router4 for access to subnet 5.

Router B

```
3 set ST <NET_of_IS_A> ROUTER2 <IS_B_name_of_subnet_1> <IS_A_MAC_on_subnet_1>FE
3 set N ROUTER2 49 00 02
3 set ST <NET_of_IS_C> ROUTER5 <IS_B_name_of_subnet_4> <IS_C_MAC_on_subnet_4>FE
3 set N ROUTER5 49 00 05
```

The first two lines above establish the name Router2 for Intermediate System A and specify that Router2 is used for access to subnet 2. The last two lines establish the name Router5 for Intermediate System C and specify that Router5 is used for access to subnet 5. Note that the third line specifies Router C's MAC address on Multibus II subnet 4, since that is the route from Router B.

Router C

```
3 set ST <NET_of_IS_B> ROUTER1 <IS_C_name_of_subnet_4> <IS_B_MAC_on_subnet_4>FE
3 set N ROUTER1 49 00 01
3 set N ROUTER1 49 00 02
```

The first line above defines the name Router1 for Intermediate System B. The second line says to use Router1 for access to subnet 1. The third line says to also use Router1 for access to subnet 2.

- C. After editing the *iset.csd* file on every IS in the overall network, reboot all the systems. You should now be able to connect from a board on one subnet to a board on another, using `iNA 960 transport` or `iRMX-NET` commands.

Step 11: TCP/IP Configuration

After you have iNA 960 networking operable on the Multibus II subnet, you can set up TCP/IP to work as well. When using the Multibus II subnet, you can run TCP/IP on more than one board in the system. However, there are configuration changes necessary when some boards that run TCP/IP do not have a local hard disk and must boot remotely.

See also: Configuring TCP/IP for the Multibus II subnet, *TCP/IP and NFS for the iRMX Operating System*

Increasing Performance for Remotely-Booted Boards

Once you have the configuration working, there is a performance enhancement step for remotely-booted boards. These boards attach to the file server using the name of the server from the `/net/data` file. By default, that means they use the external MAC address for the server instead of the server's Multibus II MAC address. To increase the performance of file access, you can make a new file server name available, with the NSAP address that includes the Multibus II subnet MAC address and subnet ID.

A. Specify server names for individual subnets in one of the following ways:

- Use the `local_name/nfsx` mechanism in the `/net/data` file to specify individual server names, as illustrated in the `/net/data.ex` file.

See also: `/net/data.ex` file, Chapter 11

- Use the `SNIDx` parameter in the `setname` command to specify server names for individual subnets.

See also: `setname`, *Command Reference*

B. Set the new file server name as a BPS (bootstrap parameter string) value for the boot clients. To do this, edit the `/msa/config/bps` file. Find the sections that designate the boot client boards. For example, the BPS parameters for the board in slot 2 follow the line `[bl_host_id = 2]`. In every section of the file that applies to a boot client, add a line like this:

```
rq_sd = newname;
```

where `newname` is the new file server name you specified in the `setname` command.

Use the syntax and spacing shown above. All lines except the last in each section of the BPS file must end with a semicolon.

See also: BPS Parameters, *MSA for the iRMX Operating System*



Application programs request iNA 960 network services through data structures called *request blocks*. Request blocks exchange control information and response information between iNA 960 and the network application. A request block may also contain data to be sent or received between applications, or may point to separate buffers holding such data.

To perform a network function, the application first allocates and fills any data buffers to be sent with the request block, and formats the request block. Then it makes a `cq_comm_rb` system call to deliver the request block to iNA 960. When iNA 960 has executed the function specified in the request block, it returns the request block and data (if any) to the application.

See also: `cq_comm_rb` in this chapter for the general request block format

The application must implement resource management system calls to reclaim request blocks and any associated buffers that remain after the process that owns them terminates or is aborted. The system calls described in this chapter ensure that iNA 960 network services (e.g., open virtual circuits at the Transport Layer) are closed when the application process terminates or is aborted.

Referencing Data Buffers in Request Blocks

Request blocks can only accommodate small amounts of data. Larger amounts of data are held in application memory segments or buffers and referenced in the request block. For example, an application that sends a data packet does not place the data into the request block. Instead, it formats a request block that contains a field pointing to a separate data buffer. In the same fashion, an application that expects to receive a data packet formats a request block pointing to a data buffer where iNA 960 will write the received data.

You cannot use pointers to data buffers in request blocks sent to iNA 960 and the Name Server. Instead you must specify the absolute address of the buffer. Although your code may use pointers, you must translate the pointers to addresses before sending the request blocks.

Using Addresses in iNA 960 Request Blocks

For iNA 960, a data buffer reference must be a 32-bit value that is meaningful on the board where iNA 960 operates. As described earlier, iNA 960 may be operating on a separate NIC (using a MIP job in a COMMEngine environment) or on the same CPU as the application (a COMMPuter job). The application does not know whether iNA 960 is on the same board or a different one.

However, since iNA 960 may be on a different board, you cannot use pointers in iNA 960 request blocks. Your application must translate pointers to absolute addresses before sending the request block.

Translating Pointers

In a protected-mode environment, an application cannot easily translate pointers to physical (absolute) addresses. The iNA 960 interface libraries include the translation system call **cq_comm_ptr_to_dword** to translate pointers to double word (32-bit) physical addresses.

The application must use the **cq_comm_ptr_to_dword** call to convert any pointer fields in the request block, before sending the request block. Each application must keep track of its own pointers. When a request block is returned, the application accesses the data buffers by mapping the physical address in the request block to the corresponding pointer.



Note

Beginning with release 2.2 of the OS, you must also perform the pointer conversion described above in request blocks sent to the Name Server. Earlier versions of iNA 960 used pointer fields in the Name Server request blocks.

Limitations on Buffer Size

Typically, the application formats a request block as the first data structure in a memory segment, with data buffers following the request block in the same segment. There is no limit on the size of such a segment, or on the location of data buffers within the segment. The data buffers need not be contiguous with the request block. However, data buffers cannot be larger than 64K bytes.

Interface Libraries and Link Sequences

The general iRMX system call libraries provide the interface to the `cq_` system calls described in this chapter.

See also: Interface Libraries, *System Call Reference*

⇒ **Caution**

Prior to release 2.2 of the OS, applications that called the `cq_` system calls used the `cqc.lib`, `cql.lib`, and `cqc32.lib` libraries in the `/rmx386/rmxnet/lib` directory. These libraries are no longer provided. You must link your existing applications with the appropriate general OS interface libraries.

Bind the appropriate library with applications that call `cq_` system calls. For an example of the bind (link) sequence to use with your application, see the network example programs under the `\rmx386\demo` directory.

See also: Bind sequences, *Intel386 Family Utilities*

Include Files

Include the appropriate files listed below in your code. The include files provide external declarations for system calls, and define constants and data types for request blocks and other data structures used in this manual. The include files for C are in the `/intel/include` directory; PL/M files are in the `/rmx386/inc16` directory.

C	PL/M	Description
<code>cqcomm.h</code>	<code>cqcomm.ext</code>	External declarations of <code>cq_</code> system calls
<code>cqcommon.h</code>	<code>cqcommon.lit</code>	Definition of the common request block header and other literal values common to all layers
<code>cqname.h</code>	<code>cqnam.lit</code>	Literals for the Name Server layer
<code>cqtransp.h</code>	<code>cqtransp.lit</code>	Literals for the Transport layer
<code>cqdatal.h</code>	<code>cqdatal.lit</code>	Literals for the Data Link layer
<code>cqnmf.h</code>	<code>cqnmf.lit</code>	Literals for the NMF layer
<code>cqroute.h</code>	<code>cqroute.lit</code>	Literals for routing structures

Two other PL/M files are in the `/rmx386/inc16` directory strictly for backward compatibility with older PL/M applications: `cqrb.ext` and `cqname.lit`. You should use `cqcomm.ext` and `cqnam.lit`, listed in the table above, instead of these older files.

Programming with Structures

This manual displays request blocks and data buffers as structures, using C syntax. When you write a program that uses the structures shown in this manual, these considerations apply:

- All structures shown as typedefs are defined in the appropriate header file, listed in the previous section. In your program you may use these structure types without defining them yourself.
- All structures must be packed; each field shown in the structure must be exactly the length shown. Many C compilers pad structure fields with bytes of 0 so that each field is a multiple of the compiler word size. If this padding is performed by default, you must specifically disable the padding for iRMX and iNA 960 structures. In the iC-386 compiler, you disable padding with a `#pragma noalign` statement. Structures defined as types in the C header files have the padding disabled with such a statement. If you use these structure types, you do not have to disable padding in your code. The PL/M compiler does not pad structures.
- Many structures in this manual include array fields whose lengths can vary. Such arrays are shown with a length of 1, because the array length must be specified to define the structure as a type. Array fields are typically preceded by a length field, as shown in the Name Server structure below:

```
typedef struct name_buffer {
    unsigned char      name_length;
    unsigned char      name[1];
} NAME_BUFFER;
```

When you use a structure with such an array, set the array length to the correct value for your code. For example, in the structure above, you could specify a value for `name_length`, then set the `name` array to that length. Other alternatives are to specify the length of a given array to its maximum allowable size, or to a size that you consistently use in your code.

Using the `cq_` System Calls

Invoke the system calls in this order:

1. Create a user for the application with the `cq_create_comm_user` call. This call also ensures that network resources are released if the application is terminated or aborted.
2. Create a message mailbox with the `rq_create_mailbox` system call. The mailbox will be used to receive request block segment tokens that are returned by iNA 960.

See also: `rq_create_mailbox`, *System Call Reference*

3. Format a request block and any associated data buffers.
4. Convert pointers to data buffers into 32-bit absolute addresses, using the `cq_comm_ptr_to_dword` system call. Place the addresses in pointer fields of the request block.
5. Send the request block to iNA 960 using the `cq_comm_rb` call.
6. Check the `except_ptr` field of `cq_comm_rb` for exception codes.
7. Wait at the mailbox with a `rq_receive_message` system call for the request block segment token to be returned.

See also: `rq_receive_message`, *System Call Reference*

8. Check the `response` field of the returned request block for exceptions.
9. Continue processing with the results from the request block.
10. Return to step 3. You need not repeat step 4 if the application uses the same data buffers and keeps track of the pointers to them.
11. When the application is done with a particular user session, use the `cq_delete_comm_user` call to release network resources. The application should also release its other resources, such as mailboxes and request blocks, using the appropriate resource management system calls.

If you use the same data buffers in subsequent calls to iNA 960, you need not repeat the `cq_comm_ptr_to_dword` conversion. Use the same values in the request block buffer fields after converting them to absolute addresses, then separately keep track of the pointer values for these addresses.

The maximum number of response mailboxes in use by applications calling **cq_comm_rb** is limited by the number of external mailboxes in MIP jobs. The default value is 10. For an application that calls **cq_comm_rb** and uses a MIP job, configure the MIP job increase the number to at least the number of mailboxes created by the application. Use the NEM parameter on the appropriate MIP1, MIP2, or MIPAT screen of the ICU. If the number of external mailboxes exceeds the maximum configured value, an E_MBX_LIMIT (0FFF6H) exception is returned as a response code in the request block.

In a COMMengine environment, if an application makes a **cq_create_comm_user** or **cq_comm_rb** call and the iNA 960 COMMputer or MIP is not running, the system will hang. To prevent this, look up the object INARDY in the root directory, with the **lookup_object** system call. This object is cataloged when iNA 960 has been loaded and is functioning. If this object does not exist, do not make **cq_** calls.

Exception Handling

If you develop an iNA 960 application that sets up its own exception handler, you must bind the application so that the local exception handler resides in the lower 64K of the code segment. Otherwise, the internal **cq_** routines that do a **get_exception_handler** call followed by a **set_exception_handler** call will fail, returning a code of 8003H.

System Calls to iNA 960

Table 10-1 lists the system calls you use to communicate with iNA 960 and the Name Server. The following sections describe each iNA 960 system call. The descriptions contain the calling syntax for both PL/M and C; the PL/M syntax is listed first.

Table 10-1. System Calls for Access to iNA 960 and the Name Server

Call	Description
cq_comm_multi_status	Returns NIC and iNA 960 status information from a specified NIC
cq_comm_ptr_to_dword	Converts a pointer to the corresponding 32-bit absolute address
cq_comm_rb	Delivers a request block to iNA 960 or to the Name Server for processing
cq_comm_status	Returns NIC and iNA 960 status information
cq_create_comm_user	Creates a user ID for programmatic access to iNA 960
cq_create_multi_comm_user	Creates a unique user ID for programmatic access to a specified NIC and iNA 960 job
cq_delete_comm_user	Releases all resources and returns all request blocks held on behalf of a specified user ID

cq_comm_multi_status

Returns NIC and iNA 960 software status information for a specific NIC. This routine is not applicable in environments where the application and the iNA 960 software run on the same processor.

Syntax, PL/M and C

```
call cq$comm$multi$status (instance, name_ptr, host_id_ptr,  
    nic_status_ptr, except_ptr);
```

```
cq_comm_multi_status (instance, name_ptr, host_id_ptr,  
    nic_status_ptr, except_ptr);
```

Parameter	PL/M Data Type	C Data Type
instance	WORD_16	unsigned short
name_ptr	POINTER	unsigned char *
host_id_ptr	POINTER	unsigned char *
nic_status_ptr	POINTER	unsigned short far *
except_ptr	POINTER	unsigned short far *

Parameters

instance

A value between 0 and 19 that specifies the NIC board that is returning the software status information.

name_ptr

A pointer to a string containing the name, in ASCII, of the NIC.

host_id_ptr

A pointer to a 6-byte array containing the Ethernet address of the NIC.

nic_status_ptr

A pointer to the test number that failed. The high byte is the failed test number and the low byte identifies the type of Multibus II test performed:

Low Byte	Meaning
10H	Microcontroller initialization check
11H	Processor initialization check
12H	Built-In Self Test (BIST)

except_ptr

A pointer to a variable declared by the application where the call returns a condition code.

Condition Codes

0000H	NIC has not yet been initialized.
0001H	NIC is in the run state.
0003H	NIC has been reset.
0004H	NIC failed to respond to a command (timeout).
0005H	There are no NIC boards in the system.
0006H	The specified NIC board is not in the system
00FFH	NIC did not respond to a boot command.
0FFFEH	Multi-NIC calls not supported by this system.

cq_comm_ptr_to_dword

Converts a pointer to the corresponding 32-bit absolute address.

Syntax, PL/M and C

```
dw = cq$comm$ptr$to$dword (ptr, except_ptr);
```

```
dw = cq_comm_ptr_to_dword (ptr, except_ptr);
```

Parameter	PL/M Data Type	C Data Type
dw	WORD_32	unsigned long
ptr	POINTER	void far *
except_ptr	POINTER	unsigned short far *

Return Value

dw The returned absolute address.

Parameters

ptr The pointer to convert.

except_ptr

A pointer to a variable declared by the application where the call returns a condition code.

Additional Information

For request blocks to be sent to iNA 960, make this call to convert each pointer before filling in pointer fields of the request block. iNA 960 request blocks that reference data buffers must contain absolute physical addresses rather than pointers to the buffers.

The size of data buffers referenced in the request block must not be larger than 64K bytes.

Condition Codes

0000H No exceptional conditions.

0008H This function call is not part of the present configuration.

cq_comm_rb

Delivers a request block to iNA 960 or to the Name Server for processing.

Syntax, PL/M and C

```
call cq$comm$rb (rb_token, except_ptr);
```

```
cq_comm_rb (rb_token, except_ptr);
```

Parameter	PL/M Data Type	C Data Type
rb_token	SELECTOR	selector
except_ptr	POINTER	unsigned short far *

Parameters

rb_token

The iRMX token for a segment containing a request block.

except_ptr

A pointer to a variable declared by the application where the call returns a condition code. If a MIP exception occurs, the call returns an E_MIP_ERROR exception. This means the actual error is indicated in the `response` field of the returned request block, rather than in this field.

Additional Information

To request the services of iNA 960, an application first formats a request block of parameters, then sends the request block with the **cq_comm_rb** system call. The system call returns without waiting for the request block to be processed. The iNA 960 software receives the request block, executes the command, and writes values into the request block. Then it returns the token for the request block's segment to a mailbox specified in the request block itself. The application waits at the mailbox for this token with a **rq_receive_message** system call. The application does not need to catalog the return mailbox in any object directory.

The general format of a request block is shown below. Each iNA 960 command is specified by an opcode and a subsystem value. The first nine fields are common to all request blocks; the application must set these fields before calling **cq_comm_rb**.

Different iNA 960 commands have varying lengths of request block arguments following the `response` field. The argument fields are described in the individual commands in this manual. Initialize all reserved and unused fields to 0 before sending a request block.

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;
    selector          user_id;
    unsigned char     resp_port;
    selector          resp_mbox;
    selector          rb_seg_tok;
    unsigned char     subsystem;
    unsigned char     opcode;
    unsigned short    response;
} RB_COMMON;

struct rb {
    RB_COMMON          header;
    unsigned char     args[];
};
```

reserved
Set to 0.

length The total number of bytes in the request block, which is 16 plus the length of the arguments fields.

user_id
An identifier specifying the user issuing the command. This is the value returned from the **cq_create_comm_user** or **cq_create_multi_comm** system calls.

resp_port
Specify 0FFH as the response port for an iRMX application.

resp_mbox
An iRMX token for the mailbox that receives a message when the request block is returned.

rb_seg_tok
The iRMX token for the segment holding this request block.

subsystem

A value specifying an iNA 960 subsystem, as shown below:

Value Subsystem

Data Link for:

20H	Boards with 82586 component, including first MIX560 board in the system
21H	SBX 586 board, EWENET module, or EtherExpress™ 16
22H	Second MIX560 board in the system
23H	Third MIX560 board in the system
24H	82595TX component, EtherExpress™ PRO/10, SBC P5090 and P5120 PC-compatible boards, all versions
25H	DEC 21143 component, SBC P5200 PC-compatible boards, all versions
2FH	Multibus II subnet
40H	Transport Virtual Circuit
41H	Transport Datagram
50H	Name Server
80H	Network Management Facility (NMF)
81H	NMF boot server commands: SUPPLY_BUFFER and TAKEBACK_BUFFER

opcode

A code that specifies a particular iNA 960 command.

See also: Following chapters for each command's opcode

response

Initialize to 0 before calling **cq_comm_rb**. If iNA 960 receives the request block, it fills in the response code before returning the request block. The response code indicates the success or failure of the command. Response codes applicable to each command are listed in the individual command descriptions in this manual.

The MIP job can also return response codes in the request block, if an error occurs while sending the request block to iNA 960. The existence of these response codes is indicated by an **E_MIP_ERROR** in the **except_ptr** parameter of the **cq_comm_rb** system call. Any other MIP errors will print to the screen and delete the network job.

0FFE8H	The iRMX-NET software is not running yet. A problem occurred at initialization that prevented the network job from coming up. Reboot the system and look for error messages at initialization.
0FFECH	All internal tables are currently full. Try the command again after a request block is returned.
0FFF0H	The MIP has encountered an unexpected iRMX error. Verify the OS configuration.

- 0FFF6H The limit for the number of available user mailboxes has been reached. Change the limit with one of these methods:
- In an ICU-configurable system, increase the NEM parameter (number of external mailboxes) in MIP configuration, and regenerate the MIP job.
 - Wait before sending the request to iNA until after a posted request block has been returned.
- 0FFF8H A fatal, unrecoverable error has occurred in the MIP driver that communicates with iNA, for one of these reasons:
- The request block was incorrectly formatted
 - iNA 960 is not responding
 - For Multibus I systems, the MIP request queues have been overwritten
 - There is a hardware failure
- Verify that the request block is formatted correctly and that the iNA transport software is functional. If the problem continues, reboot the OS and try again.
- 0FFFAH iNA 960 is out of resources. Try the request later, after some posted request blocks have been returned.
- FFEEH (Multibus II only). MIP Driver Internal Buffer Management error. The buffer size must be an even number of bytes.
- See also: MIP Error Codes, Appendix E

Condition Codes

- 0000H No exceptional conditions.
- 00FFH The MIP driver is unable to deliver the request block to the iNA 960 software. See the request block response field for the error code returned by MIP.

cq_comm_status

Returns NIC and iNA 960 software status information. This routine is not applicable in environments where the application and the iNA 960 software run on the same processor.

Syntax, PL/M and C

```
call cq$comm$status (name_ptr, host_id_ptr, nic_status_ptr,
                    except_ptr);
```

```
cq_comm_status (name_ptr, host_id_ptr, nic_status_ptr,
                except_ptr);
```

Parameter	PL/M Data Type	C Data Type
name_ptr	POINTER	unsigned char *
host_id_ptr	POINTER	unsigned char *
nic_status_ptr	POINTER	unsigned short far *
except_ptr	POINTER	unsigned short far *

Parameters

name_ptr

A pointer to an iRMX OS string (RMX_STRING data type) containing the name, in ASCII, of the NIC.

host_id_ptr

A pointer to a 6-byte array containing the Ethernet address of the NIC.

nic_status_ptr

A pointer to the test number that failed. If the low byte is less than 10H, then the board is Multibus I and the byte value is the failed test number. If the low byte is greater than or equal to 10H, then the high byte is the failed test number and the low byte identifies the type of Multibus II test performed:

Low Byte	Meaning
10H	Microcontroller initialization check
11H	Processor initialization check
12H	Built-In Self Test (BIST)

except_ptr

A pointer to a variable declared by the application where the call returns a condition code.

Condition Codes

- 0000H NIC has not yet been initialized.
- 0001H NIC is in the run state.
- 0003H NIC has been reset.
- 0004H NIC failed to respond to a command (timeout).
- 00FFH NIC did not respond to a boot command.

cq_create_comm_user

Creates a user ID for programmatic access to iNA 960.

Syntax, PL/M and C

```
comm$user = cq$create$comm$user (except_ptr);
```

```
comm_user = cq_create_comm_user (except_ptr);
```

Parameter	PL/M Data Type	C Data Type
comm_user	WORD_16	selector
except_ptr	POINTER	unsigned short far *

Return Value

comm_user

A unique value representing the created user. Use this value in the `user_id` field of request blocks.

Parameter

except_ptr

A pointer to a variable declared by the application where the call returns a condition code.

Additional Information

Call **cq_create_comm_user** once before making any other **cq_** calls. In all subsequent calls to iNA 960 from this application, specify the value returned from **cq_create_comm_user** in the `user_id` field of the request block.

This system call helps ensure that communication between iNA 960 and an application job is gracefully released when the application terminates, for example, when the user types <Ctrl-C>. The clean-up mechanism frees resources such as virtual circuits. It also prevents iNA 960 from returning request blocks or delivering data into memory that is no longer allocated to a terminated job.

Condition Codes

0000H No exceptional conditions.

cq_create_multi_comm_user

Creates a user ID for programmatic access to iNA 960 associated with a specified NIC.

Syntax, PL/M and C

```
comm$user = cq$create$multi$comm$user (instance, except_ptr);
```

```
comm_user = cq_create_multi_comm_user (instance, except_ptr);
```

Parameter	PL/M Data Type	C Data Type
comm_user	WORD_16	selector
instance	WORD_16	unsigned short
except_ptr	POINTER	unsigned short far *

Return Value

comm_user

A unique value representing the created user. Use this value in the `user_id` field of request blocks.

Parameters

instance

A value between 0 and 19 that specifies the NIC board for which the user ID is obtained.

except_ptr

A pointer to a variable declared by the application where the call returns a condition code.

Additional Information

This system call helps ensure that communication between iNA 960 and an application job is gracefully released when the application job terminates, for example, when the user types <Ctrl-C>. The clean-up mechanism frees resources, such as virtual circuits. It also prevents iNA 960 from returning request blocks or delivering data into memory that is no longer allocated to a terminated job.

The mechanism works like this:

1. The application job calls **cq_create_multi_comm_user** to obtain a unique user ID token. This call should be made before any request blocks are sent to iNA 960.
2. The application includes the user ID token obtained from **cq_create_multi_comm_user** in the `user_id` field of all request blocks.
3. When the application job is terminated (normally or abnormally) the OS invokes an MIP clean-up mechanism that will free up all iNA 960 resources held on behalf of the job's user ID. This includes virtual circuits and unreturned request blocks.

Condition Codes

- | | |
|--------|---|
| 0000H | No exceptional conditions. |
| 0FFF4H | NIC is off-line. |
| 0FFFEH | Multi-NIC calls not supported by this system. |

cq_delete_comm_user

Invokes the clean-up mechanism described under **cq_create_comm_user** and **cq_create_multi_comm_user**, causing iNA 960 to release all resources (such as virtual circuits) and return all request blocks held on behalf of a specified user ID.

Syntax, PL/M and C

```
call cq$delete$comm$user (rb_token, except_ptr);
```

```
cq_delete_comm_user (rb_token, except_ptr);
```

Parameter	PL/M Data Type	C Data Type
rb_token	SELECTOR	selector
except_ptr	POINTER	unsigned short far *

Parameters

rb_token

The iRMX token for the delete request block. The contents of a delete request block are described below. A delete request block causes the iNA 960 software to delete all pending requests, and then return all request blocks associated with the user specified in the `comm_user_id` field of the delete request block. The request blocks are returned to the iRMX mailbox specified in the `resp_mbox` field in the delete request block.

except_ptr

A pointer to a variable declared by the application where the call returns a condition code. If the condition code is 0FFH, check the response field of the delete request block for the condition code.

Additional Information

A delete request block has this format:

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;
    selector          user_id;
    unsigned char     resp_port;
    selector          resp_mbox;
    selector          rb_seg_tok;
    unsigned char     subsystem;
    unsigned char     opcode;
    unsigned short    response;
} RB_COMMON;

struct delete_rb {
    RB_COMMON          header;
    selector           comm_user_id
};
```

reserved For the use of and filled in by the iNA 960 software. Set to 0.

length The length in bytes of the entire request block, including the reserved and length fields. This field must be filled in by the user.

user_id A unique value associated with the resource management task making the delete request. This is the value returned from a call to the **cq_create_comm_user** or **cq_create_multi_comm_user** function in the resource management task (not the `user_id` value associated with the aborted application job). The field value must be filled in by the user.

resp_port
Specify 0FFH as the response port for an iRMX application.

resp_mbox
An iRMX token for the mailbox that will receive a message when the request block is returned.

rb_seg_tok
The iRMX token for the segment holding this request block.

subsystem
Zero must be filled in by the user.

opcode
Zero must be filled in by the user.

response

A value returned by the iNA 960 software that indicates the result of the delete request. A value in the range 0H through 80H and FFFEH identify iNA 960 errors while a value in the FFxxH range (except FFFEH), identify a MIP error.

0FFF6H The limit for the number of available user mailboxes has been reached. Change the limit with one of these methods:

- In an ICU-configurable system, increase the NEM parameter (number of external mailboxes) in MIP configuration, and regenerate the MIP job.
- Wait before sending the request to iNA until after a posted request block has been returned.

0FFF8H A fatal, unrecoverable error has occurred in the MIP driver that communicates with iNA, for one of these reasons:

- The request block was incorrectly formatted
- iNA 960 is not responding
- For Multibus I systems, the MIP request queues have been overwritten
- There is a hardware failure

Verify that the request block is formatted correctly and that the iNA transport software is functional. If the problem continues, reboot the OS and try again.

0FFFAH iNA 960 is out of resources. Try the request later, after some posted request blocks have been returned.

0FFFCH The port specified in the response_port field of the request block cannot be used. Specify a different port and try again.

FFEEH (Multibus II only). MIP Driver Internal Buffer Management error. The buffer size must be an even number of bytes.

See also: MIP Error Codes, Appendix E

comm_user_id

The `user_id` value associated with the aborted application job. This is the value returned from a call to the `cq_create_comm_user` or `cq_create_multi_comm_user` function in the application job (not the `user_id` value associated with the resource management task). The field value must be filled in by the user.

Application programs loaded dynamically under the iRMX HI rarely use this call, because the clean-up mechanism is invoked automatically at the time the application job is terminated. However, some applications, particularly if they are run as background jobs, may find this call useful, such as when resetting themselves after a catastrophic error.

The routine passes a special `delete_request_block` to the iNA 960 software. This special delete request block releases the iNA 960 request blocks and internal iNA 960 resources associated with the specified application process.

After `cq_delete_comm_user` is called, the User ID token obtained from `cq_create_comm_user` or `cq_create_multi_comm_user` is still valid. This means that multiple `cq_delete_comm_user` calls can be made on this token.

Condition Codes

- | | |
|-------|---|
| 0000H | No exceptional conditions. |
| 00FFH | The MIP driver is unable to deliver the request block to the iNA 960 software. See the request block response field for the error code returned by MIP. |



This chapter covers the Name Server, which manipulates a distributed database of network objects. The primary purpose of the Name Server is to correlate an OS-defined or user-defined name with a numeric value, such as the network address of a server system. This discussion assumes that you have read the introduction to the Name Server earlier in this book.

See also: iRMX-NET Overview, Chapter 2

The procedures in this chapter include several iRMX-NET commands used to manipulate the Name Server. The full syntax and explanation of these commands is not covered here.

See also: Individual commands, *Command Reference*

Through the Name Server programming interfaces, an application can dynamically add, delete, and inspect the network objects contained in the Name Server object table. The application sends and receives request blocks using the `cq_comm_rb` call to program the Name Server.

See also: Using the `cq_` System Calls, Chapter 10

The Name Server Object Table

The Name Server's main function is to dynamically map the names of network objects to their addresses. OpenNET networks are made up of many kinds of objects: AUs, servers, clients, devices, and users. The network objects the Name Server deals with most often are servers.

The Name Server operates as a distributed database. Each node on the network maintains its own Name Server object table, where it lists information about network objects. Several entries in the object table are placed there automatically when iNA960/iRMX-NET software is initialized on a computer. These fixed entries include the Ethernet address and details about the network implementation and the computer architecture. You can add other local objects for resources on the local node available to remote users, such as a file server or a virtual terminal server.

If the network contains nodes that do not have Name Server capability, you can catalog the node's server name and address in the local object table, making your computer a spokesman for these objects. The only OpenNET nodes that require a separate spokesman are UNIX systems earlier than SV-OpenNET R3.2.3.

Figure 11-1 shows the object table for an example node as displayed by the **listname** command. Each entry in the Name Server object table contains the name, property type (Property column), and property value (Value column) of a network object. The first two sections in the figure are fixed entries. The last two sections are entries for the iRMX-NET file server and client. These are not fixed entries and may appear in the opposite order, depending on whether the server or client job loads first.

Name	Property	Unique	PV_Type	Value
FSTSAP	00000H	NO	SIMPLE	10 00H
FCTSAP	00001H	NO	SIMPLE	11 00H
INARELNUM	00004H	NO	SIMPLE	03H
INANLNUM	00004H	NO	SIMPLE	01H
NSCOMMENGINE	00004H	NO	SIMPLE	FFH
TLCOMMENGINE	00004H	NO	SIMPLE	00H

The following entries depend on the number of subnets in the iNA 960 job. For example, there can be up to 4 MYHOSTID entries, 1 for each subnet, where xx varies from 01 to 04.

MYHOSTID	00004H	NO	SIMPLE	00 AA 00 02 57 86H
MYHOSTIDxx	00004H	NO	SIMPLE	00 AA 00 02 57 86H
INASUBNET	00004H	NO	SIMPLE	00 01H
INASUBNETxx	00004H	NO	SIMPLE	00 01H

The following entries are added by file servers from the */net/data* file. The BSMB2 entry is for the server in slot 0 and the BSSLOT2 entry is needed by the client in slot 2 (note that the last two digits of addresses in the Value column are the slot number for these entries).

RNETSRV	00004H	NO	SIMPLE	52 4E 45 54 53 52 56 00 AA 00 02 57 86H
BSMB2	00003H	YES	SIMPLE	0B 49 00 00 00 AA 00 02 57 86 FEH 00 02 10 00H
BSMB2	00005H	YES	SIMPLE	00 AA 00 02 57 86 00H
BSMB2	00006H	YES	SIMPLE	0B 49 00 00 00 AA 00 02 57 86 FEH 00 02 30 00H
BSSLOT2	00005H	YES	SIMPLE	00 01 00 AA 00 02 57 86 02H
NSDONE	00004H	NO	SIMPLE	52 4D 58 00 AA 00 02 57 86H

The following entry is for the client (file consumer), taken from the */net/data* file.

MYNAME00	00002H	NO	SIMPLE	72 6D 78H
----------	--------	----	--------	-----------

Figure 11-1. The Name Server Object Table

In the object table, `Name` means the name of the object, such as the server name `BSMB2` in this example.

The `Property` column lists the property type, a numeric code that tells what kind of information is represented by the property value in the last column. Table 11-1 lists the possible property types:

Table 11-1. Property Types for the Name Server

Property Type	Meaning
0000H	File Server TSAP ID
0001H	File Consumer (client) TSAP ID
0002H	Name of the client
0003H	File Server Transport Address
0004H	Configuration objects
0005H	Host-unique ID
0006H	Remote Launch Server Transport Address
0007H	Reserved
0008H	VT Server Transport Address
0009H - 7FFFH	Reserved
8000H - FFFFH	Available for applications

Property types 0, 1, 2, and 4 are for fixed entries, automatically added by iNA 960. Objects that you or iRMX-NET add to the table generally have property types 3, 5, 6 and 8.

`Unique` indicates whether this combination of object name and property type are unique on the network. The fixed entries are not unique; the object table on every node in the network includes these objects. Other non-unique objects can be added to the object table through the programmatic interface. Non-unique objects are, in effect, local objects. Each computer can read the value of the object in its own object table, but it cannot access the object with that name on a remote node. The Name Server guarantees the uniqueness of any object entered through the Human Interface. Before it accepts a new object, it checks all the other object tables on the network for objects with the same name and property type.

`SIMPLE` in the `PV_Type` column means that the property value in the last column is a simple string, rather than a complex structure in which each element is an object, such as a mail list made up of network users. Structured property types are not supported in iNA 960/iRMX-NET.

The `Value` column is the property value, a field containing specific information about this object, usually based on the network address. For objects of property types 3, 6 and 8, the `Value` column contains the server's transport address. For objects of property type 5, that column contains the host-unique ID, combining the Ethernet address and a slot ID.

Adding an Object to the Name Server Object Table

You can make entries in the object table by:

- Performing an automatic **loadname** command during system initialization. By default, the iRMX-NET file server job is configured to do this.
- Invoking the **loadname** command on any iRMX-NET node.
- Invoking the **setname** command on any iRMX-NET server node.
- Using the ICU to configure the iRMX OS to perform an automatic **setname** during system initialization, by setting the ICU's FSN parameter on the FS screen to the name of the server.
- Adding entries programmatically with a Name Server ADD_NAME command.

Enter information for the **setname** command on the command line. For example, the following command adds the name of the file server with the Ethernet address for the second subnet configured into the iNA 960 job:

```
setname bsmb2_2 SNID2
```

Or, for a computer that is used strictly as a client node:

```
setname labsys2 HID
```

The **loadname** command, on the other hand, loads information from an input file, typically */net/data*. For convenience, **loadname** is most often used.

See also: **setname** and **loadname** commands, *Command Reference*; ADD_NAME command, in this chapter

⇒ **Note**

In a Multibus II system with boards that boot dependently, it is important that you have a */net/data* file with valid entries. The client boards need the address of the server, and the existence of this file enables an initialization sequence between the file server and clients. See the NSDONE and RNETSRV parameters on page 145 for details.

Loading Objects from the `:sd:net/data` File

The `:sd:net/data` file is the input file for the **loadname** command. Typically, the only entry you need in the file is the local file server (nfs) name. (See the first line in Figure 11-2). You may need to add entries with the name and address of systems that need a network spokesman or entries for a particular use by your application. For example, you might want to store an iNA 960 transport address, which is a different format than Name Server addresses. You can copy an existing `net/data` file containing all required servers from another iRMX or UNIX system. If none is available, copy and edit the example file, `:sd:net/data.ex`, or create the file yourself.

See also: Adding a Server to the Name Server Object Table, Chapter 3,
 Transport addresses, Chapter 12

After you create the `:sd:net/data` file, invoke **loadname** to read the names and property values of objects from the file and enter the information into the Name Server object table. Each line in the file becomes one or two entries in the object table. The next time you reboot the system, an automatic **loadname** loads the information; you need not repeat the **loadname** at the command line.

Editing the `:sd:net/data.ex` File

The example file, `:sd:net/data.ex`, is provided with the OS. Each line of the file is a sample entry for a different network object. Figure 11-2 lists the `data.ex` file. The fields in *italic* (including the `#` characters) are variables to be replaced with the specific values for the object being entered. Each entry's first variable is based on the type of object the template is intended for, including the OS and architecture of the computer where the object resides.

Using `data.ex` as an example, add all the required servers to the `\net\data` file and delete any unused example lines. Then invoke the **loadname** command.

```
local_name1/nfs:    TYPE=rmx:    ADDRESS=;
local_name2/nfs2:  TYPE=rmx:    ADDRESS=;
local_name3/nfs3:  TYPE=rmx:    ADDRESS=;
local_name4/nfs4:  TYPE=rmx:    ADDRESS=;
slot2:             TYPE=PT0005: ADDRESS=ssss#####02;
slot3:             TYPE=PT0005: ADDRESS=ssss#####03;
slot4:             TYPE=PT0005: ADDRESS=ssss#####04;
slot5:             TYPE=PT0005: ADDRESS=ssss#####05;
slot6:             TYPE=PT0005: ADDRESS=ssss#####06;

rmx_mb1_rsd:       TYPE=PT0005: ADDRESS=ssss#####00;
rmx_mb2_rsd:       TYPE=PT0005: ADDRESS=ssss#####$$;
rmx_mb2_msd:       TYPE=PT0005: ADDRESS=ssssA2A4A6A8AA$$$$;

xnx_srv_i1/nfs:    TYPE=xenix:  ADDRESS=0X80000A00000001#####000000;
xnx_srv_i2/nfs:    TYPE=xenix:  ADDRESS=0X80000A00000001#####000000;
xnx_vts_i2/vts:    TYPE=xenix:  ADDRESS=0X40000A00000001#####000000;

ndx_srv_i1/nfs:    TYPE=indx:   ADDRESS=0X80000A00000001#####000000;

vms_srv_i1/nfs:    TYPE=vms:    ADDRESS=0X80000A00000001#####000000;
vms_vts_i1/vts:    TYPE=vms:    ADDRESS=0X40000A00000001#####000000;

unx_srv_i1/nfs:    TYPE=unix:   ADDRESS=0X80000A00000001#####000000;
unx_srv1_i3/nfs:   TYPE=unix:   ADDRESS=0X80000A00000001#####000000;
unx_vts_i3/vts:    TYPE=unix:   ADDRESS=0X40000A00000001#####000000;
unx_srv2_i3:       TYPE=PT0003: ADDRESS=0X80000A00000001#####000000;
unx_mb2_srv/nfs:   TYPE=unix:   ADDRESS=0X80000A00000001#####000000;

rmx_vts_mb1/vts:   TYPE=rmx:    ADDRESS=0X40000A00000001#####000000;
rmx_vts_sl2/vts:   TYPE=rmx:    ADDRESS=0X40020A00000001#####000000;

any_object:        TYPE=PT####: ADDRESS=####...#####;
```

Figure 11-2. The `:sd:net/data.ex` File

Make substitutions in the *data.ex* file to create a */net/data* file as follows:

local_name entries

For each subnet in a job, make an entry for the local server. Each name must be unique. These entries do not require an Ethernet address because that value is already stored in the local *myhostidx* object. These entries set the local server name for the file server, VT server, and remote load server.

The entry followed by */nfs* (or */nfs1*, which has the same effect) applies to the first subnet in a job. The entry followed by */nfs2* applies to the second subnet, etc. Use only the entries that apply to your iNA 960 job. Any entries that do not apply revert to the first subnet. For example, if you include *local_name* entries for */nfs3* and */nfs4*, but use a job with only two subnets, the names for the */nfs3* and */nfs4* objects are set to the Ethernet address of the first subnet.

See also: Chapter 9 for iNA 960 jobs with multiple subnets

slot entries

On a Multibus II system where other host boards share the local Ethernet address, include entries for the other hosts, such as *slot2-slot6*. The entries are not limited to six slots; specify the appropriate names for hosts in slots on your system. Include entries like these examples only when you do not use the Multibus II subnet, where every board on the subnet has its own Ethernet address assigned.

In the *ssss* part of the address, substitute the subnet ID that applies. In the *#####* part, substitute the Ethernet (MAC) address, followed by the slot number.

rmx_mb1_rsd and *rmx_mb2_rsd* entries

On a system where remote boot clients use the local hard disk of the boot server, include entries for the boot client hosts. The entries are not limited to clients in the same system. Specify the appropriate names for the boot clients.

In the *ssss* part of the address, substitute the subnet ID that applies. In the *#####* part, substitute the Ethernet (MAC) address, followed by the slot number (in place of *\$\$*) for boot clients in a Multibus II slot. For boot clients in PCs or Multibus I systems, specify 00 for this last byte.

rmx_mb2_msd entries

On a system where remote boot clients use the Multibus II subnet to connect to the boot server in the same system, include an entry like this for each boot client.

In the *ssss* part of the address, substitute the subnet ID that applies. In the first *\$\$*, substitute the slot ID of the client as the last byte of the special Multibus II MAC address. In the second *\$\$*, also substitute the slot ID of the client.

xnx_srv_i1 through *rmx_vts_sl2* entries

These are examples of how to specify file servers (*/nfs*) and remote login VT servers (*/vts*) on other OSs. For example, you would replace *xnx_mb2_srv* with the server name of a UNIX server on Multibus II and replace ##### with the Ethernet address.

any_object entry

This is a general-purpose example of how to specify an object in the */net/data* file. These entries require a 4-digit property type; choose one from Table 11-1 on page 133. Instead of the Ethernet address alone, specify the entire 34-digit transport address.

See also: Transport addresses, Chapter 12

Syntax of the *:sd:net/data* File

The general syntax of lines in the *:sd:net/data* file is:

```
name/object_type:TYPE=system:ADDRESS=net_address;
```

Where:

name The name of the network object being accessed on the system. The name must follow the Name Server object-naming conventions.

object_type

An optional field representing the type of network object. If you specify object type *nfs* or *nfs1* through *nfs4* (network file server), the **loadname** command generates two entries; one for property type 3 and one for property type 5. If you specify object type *vts* (virtual terminal server), there is one entry for property type 8. The **loadname** command ignores other object types.

system

Identifies the system or property type of an object. The maximum length of this field is six characters. If the object type is *nfs* or *vts*, **loadname** ignores this field, but you can use the field to specify one of the supported OSs *unix* or *rmx* (entered in either upper or lower case characters).

If the object type is not *nfs* or *vts*, then you must specify the property type in the system field as follows:

```
TYPE = PT property
```

Where PT indicates that the characters to follow represent the property type, and:

property

A string of four hexadecimal digits representing the numeric property type.

net_address

The value for the specified property type. The syntax example in the *data.ex* file for other OSs such as Unix (*unx_mb2_srv/nfs* example) is a transport address as follows (but without the separating spaces):

```
addr_id tsap 0A subnet E_net 000000;
```

Where:

addr_id

A two-character field indicating the format of the subsequent address to be entered into the Name Server object table. The two characters can be one of these two values:

- | | |
|----|--|
| 0X | Indicates that the subsequent data is used to generate an iNA 960 transport address that corresponds to the version of iNA 960 used in the local system. |
| i1 | Indicates that the subsequent address is to be entered in iNA R1.3 format, irrespective of the iNA 960 release running on the NIC. This is necessary for communicating from an iNA R1.3 client to an iNA R3.0 server, because the iNA R1.3 client cannot recognize the iNA R3.0 server address format. |

tsap The iNA 960 TSAP-ID for the server, from this list:

- | | |
|-------|---|
| 8000H | for UNIX |
| 1000H | for iRMX Multibus I or PC Bus |
| 10xxH | for iRMX Multibus II, where <i>xx</i> represents the slot ID of the server's host CPU |
| 3000 | for the <i>r1s</i> type |
| 4000 | for the <i>vts</i> type |

0A A constant for this form of address.

subnet A constant that identifies the subnet. For this form of address, the value must always be 00000001 for iRMX and UNIX

servers using an iNA 960 Null2 network layer. You cannot substitute one of the subnet IDs used by the Multibus II subnet versions of iNA 960.

E_net The 12-digit (6-byte) hexadecimal Ethernet address of the NIC. This is the string shown as ##### in Figure 11-2. To find the Ethernet address of a UNIX system, use the SV-OpenNET **enetinfo** command.

000000 An unused field.

For example, if the address is for a VT server running on a Multibus II board in slot 3, with Ethernet address 00AA00025A70, the *net_address* would be:

```
0X40030A0000000100AA00025A70000000
```

You can omit the *net_address* field when specifying a local name. The syntax for a local system name is:

```
name/nfs:TYPE=rmx:ADDRESS=;
```

The syntax of the address for property type 5 is:

```
subnet E_net slot
```

Where:

subnet The 4-digit (2-byte) hexadecimal subnet ID.

E_net The 12-digit hexadecimal address uniquely specifying an Ethernet NIC.

slot The 2-digit slot ID of the host CPU in Multibus II systems. For Multibus I and PC Bus systems, this value is 00.

Other Name Server Operations

iRMX-NET commands can also remove objects from the object table, display local Name Server information, and obtain information about other nodes on the network.

See also: *Command Reference* for details on commands described here

Deleting an Object from the Name Server Object Table

Objects remain in the object table until the system is rebooted or until they are removed by the user that entered them. You can use **deletename** to remove an object that was entered into the object table with the **setname** or **loadname** command. For example:

```
deletename bsmb2
```

To remove all the objects in your `:sd:net/data` file from the object table, use the **unloadname** command. This works like **loadname**, in reverse.

Obtaining Local Name Server Information

To see the Name Server object table on your computer, use the **listname** command. Sample output of this command is shown in Figure 11-1.

Use the **getaddr** command to retrieve the Ethernet address of the local system. This returns the value for the local object named `myhostid`.

If you use an iNA 960 job with multiple subnets, use the **netinfo** command to get the Ethernet address and other information about all subnets in the job.

Obtaining Remote Name Server Information

Two commands allow you to get information about remote nodes. If you know the server name, use the **findname** command with the `L` switch to find the Ethernet address. For example:

```
findname unixsl P=0008 L
```

The `P=0008` is the property value for the virtual terminal server. If you do not specify a property value, the command defaults to property type 5, the file server.

If you know the Ethernet address, use the **getname** command to return the name of the server. This is the object name of property type 5 in that computer's object table. For example:

```
getname A=00AA00025A70
```

If you do not include an Ethernet address, **getname** returns the local server name.

Object Table Entries at Initialization

When an iRMX network system is loaded, the system is initialized with certain entries in the Name Server object table. These fixed entries are not configurable. All of the entries are used internally by the iRMX-NET subsystems; therefore, you must not modify or delete the values. However, network applications can look up their values, such as the Ethernet address of the system.

The initial fixed objects are non-unique; every Name Server object table on the network contains them. Table 11-2 gives a definition of the objects, and each one is explained in detail in the following pages. For the format of the object table entries, see Figure 11-1 earlier in this chapter.

The `server_name` and `MYNAMExx` entries in Table 11-2 are not fixed. They are configurable and are added to the object table by iRMX-NET when it initializes.

Table 11-2. Object Table Entries

Name	Property Type	Field Meaning
FSTSAP	0	The File Server TSAP-ID.
FCTSAP	1	The File Consumer TSAP-ID.
INARELNUM	4	The release number of the iNA 960 Software.
INANLNUM	4	Code for the Network Layer that is configured in the iNA 960 Transport Software.
NSCOMMENGINE	4	Indicates if the Name Server runs on the communications or host board.
TLCOMMENGINE	4	Indicates if iRMX-NET runs in a COMMengine or COMMputer environment.
MYHOSTIDxx	4	Ethernet address(es), where xx represents the first through fourth subnet in the job.
INASUBNETxx	4	Subnet ID(s), where xx represents the first through fourth subnet in the job.
server_name	3,5,6	File Server names.
MYNAMExx	4	The iRMX-NET client (File Consumer) name, where xx represents the slot ID in a Multibus II system.
RNETSRV	4	The iRMX-NET server catalogs this object during its initialization. RSD clients (boards that boot remotely) wait for the NSDONE object below only if RNETSRV is cataloged.
NSDONE	4	The iRMX-NET server catalogs this object after loading entries from <code>/net/data</code> so client systems can synchronize initialization.

FSTSAP This object indicates the File Server TSAP-ID. Although two bytes are stored in the Name Server object table for the TSAP-ID, only the first byte is valid. For an iRMX file server, it is 10H. The second byte is always stored as a 0. iRMX-NET leaves the second byte set to 0 for Multibus I and PC Bus systems. In Multibus II systems, iRMX-NET overwrites the second byte with the slot ID of the host.

See also: TSAP addresses, Chapter 12

FCTSAP This object name represents the File Consumer (client) TSAP-ID. As with FSTSAP, only the first byte is valid. For an iRMX client, it is 11H. The second byte is always stored as a 0. iRMX-NET leaves the second byte set to 0 for Multibus I and PC Bus systems. In Multibus II systems, iRMX-NET overwrites the second byte with the slot ID of the host.

INARELNUM

This is the release number of the iNA 960 Transport software used with iRMX-NET. The value of this entry is:

- 1 for iNA 960 Release 1.X
- 2 for iNA 960 Release 2.X
- 3 for iNA 960 Release 3.X

iRMX-NET only supports iNA 960 Release 3.X.

INANLNUM

This object represents the network layer configuration of the iNA 960 Transport Software. In ICU-configurable systems you can specify this value in the NL parameter of the ICMPJ or IMIPJ screen. The value of this entry is:

- 0 for the Null1 Network Layer
- 1 for the Null2 Network Layer
- 3 for the ES-IS Network Layer

The iNA 960 software supplied with the OS includes either the Null2 or the ES-IS Network Layer.

See also: iNA 960 Network Layer Addressing Schemes, Chapter 8

NSCOMMENGINE

This entry is a flag to indicate whether the Name Server runs on the host board or the NIC. If this value is 0, then the Name Server runs on the host board. If this value is 0FFH, the Name Server runs on the NIC. On COMMengine systems the Name Server runs on the NIC, even though this value is 0.

See also: Network Software Implementation, Chapter 7

TLCOMMENGINE

This object indicates whether iRMX-NET operates in the COMMengine environment or in the COMMputer environment. If the value is 0, the COMMengine environment is used. Otherwise, a COMMputer is used.

MYHOSTIDxx

These entries contain the system's Ethernet addresses. If the job has more than one subnet, there are multiple entries with the Ethernet addresses for each subnet. The Ethernet address is obtained from the iNA 960 Transport Software NMF layer, and entered by the Name Server when the system is booted. This entry can be used by an application to locate the Ethernet address of the system.

This object should not be confused with the property type 5 objects that contain a host's unique ID. This object contains only the Ethernet address. The property type 5 object contains a combination of Ethernet address and slot-ID for Multibus II systems, and Ethernet address and 00 for Multibus I and PC Bus systems.

If an application uses a configuration of iNA 960 that does not contain NMF, or if a non-Ethernet subnet is used, the value for this object can be provided by the application. The Name Server makes a call to a procedure called **ns_get_host_id** (which you can supply) if the call to iNA 960's NMF layer fails.

INASUBNETxx

These entries contain the iNA 960 subnet IDs used in iNA 960 transport addresses in the Name Server protocol packets. If the job has more than one subnet, there are multiple INASUBNETxx entries with the IDs for each subnet. The subnet ID is also used for transport addresses that are loaded into the object table of the system. In ICU-configurable systems, set the subnet ID(s) in the SN1 through SN4 parameters of the ICMPJ screen.

server_name

These entries represent the name of the iRMX-NET File Server. On iRMX for PCs or DOSRMX systems, set the server name in the *rmx.ini* file. On ICU-configurable systems, set the name in the FSN parameter of the FS screen. To add more names for servers, use the */net/data* file.

MYNAMEslot-ID

This entry represents the name of the File Consumer (client) used by iRMX-NET when making a connection with a remote file server. On iRMX for PCs or DOSRMX systems, set the client name in the *rmx.ini* file. On ICU-configurable systems, set the name in the CNN parameter of the CDF screen. iRMX-NET adds the slot ID as the last byte in the name in Multibus II systems. In Multibus I and PC systems, the slot ID is replaced with a 0.

RNETSRV This entry is added by the file server when its initialization begins. This value is used together with NSDONE below during a dependent boot sequence.

NSDONE This entry is added by the file server once it has finished adding all the entries from the */net/data* file. In a Multibus II system where some diskless boards boot dependently, iRMX-NET clients in any other slot than the file server use this value to synchronize their initialization with the file server. These clients attempt to synchronize using NSDONE only if the RNETSRV object is cataloged by the file server.



CAUTION

The NSDONE initialization mechanism occurs automatically as long as there is a */net/data* file with any entries in it. If you do not have such a file or if it is empty, the dependent boot sequence fails.

Location of the Name Server

The Name Server always runs with the iNA 960 software:

- In COMMengine systems, the Name Server runs on the NIC along with the iNA 960 transport software. The other iRMX-NET modules run on the host CPU board along with the iRMX OS. In these systems, regardless of the OS, the Name Server is preconfigured along with the iNA 960 download file.

Running the Name Server on Multibus II COMMengine systems facilitates the presence of multiple hosts in the same chassis. One Name Server provides services and acts as the spokesman for all hosts within the Multibus II system.

- In COMMputer systems all the network software, including the Name Server, runs on the same CPU board as the iRMX OS. In DOSRMX and iRMX for PCs, the Name Server is preconfigured into the iNA 960 job. In ICU-configurable systems, you can configure Name Server values on the NS screen of the ICU.

See also: Name Server preconfigured values, Appendix A;
 Overview of iRMX-NET Software, Chapter 7

Request Block Arguments

The Name Server commands listed in this chapter all use the same argument structure following the common header fields in the request block. However, not all of the commands use every field in the request block arguments. Each command description lists which fields are input and output arguments. Initialize reserved fields and unused fields to 0. The argument fields have this structure:

```
typedef struct name_server_rb {
    RB_COMMON          header;
    unsigned char      reserved[6];
    unsigned long      name_buffer_addr;
    unsigned char      unique_name_flag;
    unsigned short     property_type;
    unsigned char      property_value_type;
    unsigned long      pv_buffer_addr;
    unsigned long      extra_buffer_addr;
    unsigned short     extra_buffer_length;
} NAME_SERVER_RB;
```

Where:

`name_buffer_addr`

An address that points to a buffer containing the name of the object.
The name has a maximum length of 16 characters.

`unique_name_flag`

A flag that indicates if the object is unique.

`property_type`

The property type of the object. This code specifies what type of property value is stored in this object.

See also: Defined property types, ADD_NAME command

`property_value_type`

Indicates whether the property value is simple (00H) or structured (01H). Structured property values are not supported; set this to 0.

`pv_buffer_addr`

An address that points to a buffer containing the property value. This buffer can be up to 256 bytes long. The property value is the numeric data associated with this object's name, typically a network address.

`extra_buffer_addr`

An address that points to a buffer where the Name Server returns additional information for some commands. This buffer can be up to 4096 bytes long.

`extra_buffer_length`

The size of the extra buffer, in bytes.

⇒ **Note**

For the addresses in the structure above, you must use the **cq_comm_ptr_to_dword** call to convert pointers to addresses before sending the request block in Name Server commands.

See also: Using Addresses in iNA 960 Request Blocks, Chapter 10

Example Software

The OS software includes an example application using the Name Server. See the network example files under the */rmx386/demo* directory.

Name Server Commands

Table 11-3 lists the opcodes and command names for the Name Server functions. Use the `subsystem` and `opcode` fields in the request block header (`rb_common`) to specify the Name Server command. The command names are declared as literal values in the include files for the Name Server; you can specify these command names as opcodes.

See also: Include Files, Chapter 10,
 Programming with Structures, Chapter 10

Table 11-3. Name Server Commands

Opcode	Literal	Description
0H	ADD_NAME	Adds a new object to the local object table.
08H	ADD_SEARCH_DOMAIN	Specifies subnet IDs the Name Server will search.
03H	CHANGE_VALUE	Changes the property value of an object in the local object table.
01H	DELETE_NAME	Deletes all properties of an object from the local object table.
04H	DELETE_PROPERTY	Deletes the property value of an object in the local object table.
09H	DELETE_SEARCH_DOMAIN	Removes subnet IDs from the Name Server search.
05H	GET_NAME	Returns the object name, given its property type and value.
0AH	GET_SEARCH_DOMAIN	Returns subnet IDs currently enabled to search.
06H	GET_SPOKESMAN	Returns the Ethernet address for the local system.
02H	GET_VALUE	Returns the property value of an object.
07H	LIST_TABLE	Lists all objects in the local object table.

Table 11-3 lists response codes that can be returned in Name Server request blocks.

Table 11-4. Name Server Response Codes

Code	Literal	Meaning
01H	OK_RESPONSE	The operation was successful.
02H	E_NAME_EXIST	The object name with the specified property already exists in the network.
04H	E_NAME_NOT_EXIST	The object name with the specified property does not exist in the network.
06H	E_BAD_NAME	The object name in the request block is not valid (for example, it is longer than 16 characters), or the name buffer pointer does not point to a valid buffer.
08H	E_PT_EXIST	The property type specified for the object already exists in the network.
0AH	E_PT_NEXIST	The property type specified for the object does not exist in the network.
0CH	E_BAD_PVT	The specified property value type is invalid.
0EH	E_BAD_PV	The length of the specified property value exceeds the configured maximum, or pv_buffer_ptr is not valid.
10H	E_NSPACE	The local object table is full; no new objects can be added unless the size of the table is increased.
12H	E_BUFF_SPACE	The buffer supplied for returning parameters is too small.
14H	E_NAME_OPCODE	The opcode specified in the request block is invalid.
16H	E_MAX_RESP	The number of responses received by the Name Server for the query exceeds the configured limit.
18H	E_BAD_BUF_PTR	The extra buffer pointer is not valid.
1AH	E_NO_MEMORY	No internal buffers are available. Try the function after some outstanding Name Server RBs are returned.
1CH	E_NO_DELETION	Deletion is not allowed on this object; entries needed by iRMX-NET cannot be deleted by applications.
1EH	E_RB_FORMAT_BAD	The request block is not formatted correctly.

ADD_NAME

ADD_NAME adds a new object to the local Name Server object table and broadcasts names over the network. The new object is composed of a name, a property type, and a property value, which you supply as input parameters to the function.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user
*/
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;      /* 0H or ADD_NAME */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON            header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr; /* input */
    unsigned char       unique_name_flag; /* input */
    unsigned short      property_type;   /* input */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;  /* input */
    unsigned long       extra_buffer_addr; /* not used */
    unsigned short      extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

name_buffer_addr

An address that points to a buffer containing the name of the object, with this structure. The maximum length is 16 bytes.

```
typedef struct name_buffer {
    unsigned char       name_length;
    unsigned char       name[1];      /* set to length */
} NAME_BUFFER;
```

`unique_name_flag`

Specifies whether the new object is unique (OFFH) or non-unique (0).

`property_type`

Specifies the type of property value stored in the new object, from these values:

Value	Type
0000H	File Server TSAP ID (used by iRMX-NET)
0001H	File Consumer TSAP ID (iRMX-NET)
0002H	Name of the consumer (iRMX-NET)
0003H	File Server Transport Address (iRMX-NET)
0004H	Configuration objects
0005H	Host-unique ID
0006H	Remote Launch Server Transport Address
0007H	Reserved
0008H	VT Server Transport Address (iRMX Virtual Terminal)
0009H - 7FFFH	Reserved
8000H - FFFFH	Available for applications

`property_value_type`

Set to 0.

`pv_buffer_addr`

An address that points to a buffer containing the property value for the new object, with this structure:

```
typedef struct value_buffer {
    unsigned short    length;
    unsigned char     value[1];    /* set to length */
} VALUE_BUFFER;
```

Responses

Output Arguments

None

Response Codes

See Table 11-4 on page 149.

Additional Information

The Name Server responds with an `E_BAD_NAME` response code if the name length is greater than 16 characters. The maximum length of the property value is a Name Server configuration parameter. The Name Server responds with an `E_BAD_PV` response code if the property value length exceeds the configured maximum length.

If the object already exists, either locally or remotely, the Name Server returns an error. If you specify the new object as unique, the Name Server checks to see if the same object is used elsewhere in the network; if so, it returns an error. For non-unique objects, the Name Server checks to make sure the same object is not entered as unique elsewhere in the network; if so, it returns an error.

ADD_SEARCH_DOMAIN

ADD_SEARCH_DOMAIN adds new subnet IDs to the list of subnets the Name Server searches to resolve a name into an address.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of name_server_rb */
    selector          user_id;     /* cq_create_comm_user
*/
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 50H or NAME_SERVER */
    unsigned char     opcode;     /* 8H or
                                ADD_SEARCH_DOMAIN */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON         header;
    unsigned char     reserved[6];
    unsigned long     name_buffer_addr;    /* not used */
    unsigned char     unique_name_flag;   /* not used */
    unsigned short    property_type;     /* not used */
    unsigned char     property_value_type; /* set to 0 */
    unsigned long     pv_buffer_addr;    /* input */
    unsigned long     extra_buffer_addr; /* not used */
    unsigned short    extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

property_value_type

Set to 0.

pv_buffer_addr

The address pointing to a buffer that holds the list of subnet IDs, with this structure:

```
struct domain_list_struct {
    unsigned short  length;                /* overall */
    unsigned char   count;
    unsigned short  search_domain_list[1]; /* set to count */
} DOMAIN_LIST_STRUCT;
```

Where:

length The total number of bytes in length, count and the array of subnet IDs, with a maximum value of 163.

count The number of entries in the list, with a maximum of 80.

search_domain_list

An array of subnet IDs to add to the search domain.

Responses

Response Codes

See Table 11-4 on page 149.

Additional Information

You can add up to 80 subnet IDs. The Name Server will search all the specified subnets when you make a request to attach to a remote device. If your network has iNA 960 subnets that are not on the Name Server search domain, the Name Server will not make requests of those subnets. You can add subnet IDs that are not currently in use, for future expansion, but searching unused subnet IDs slows down the Name Server operations.

CHANGE_VALUE

CHANGE_VALUE overwrites the existing property value of an object in the local Name Server object table.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;  /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;     /* 3H or CHANGE_VALUE */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON            header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* input */
    unsigned char       unique_name_flag;    /* not used */
    unsigned short      property_type;       /* input */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;     /* input */
    unsigned long       extra_buffer_addr;   /* not used */
    unsigned short      extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

`name_buffer_addr`

An address that points to a buffer containing the object name whose property value is to change. The buffer has this structure:

```
typedef struct name_buffer {
    unsigned char    name_length;
    unsigned char    name[1];        /* set to length */
} NAME_BUFFER;
```

`property_type`

The type of property value whose value is to change.

See also: Property types, ADD_NAME command

`property_value_type`

Set to 0.

`pv_buffer_addr`

An address that points to a buffer containing the new value for the property, with this structure:

```
typedef struct value_buffer {
    unsigned short   length;
    unsigned char    value[1];      /* set to length */
} VALUE_BUFFER;
```

Responses

Output Arguments

None

Response Codes

See Table 11-4 on page 149.

Additional Information

You could accomplish the CHANGE_VALUE function by using the DELETE_PROPERTY and ADD_NAME commands. However, ADD_NAME has the additional overhead of broadcasting names over the network, which is not necessary just to change the value of the property.

DELETE_NAME

DELETE_NAME deletes the specified object and all properties associated with it, from the local Name Server object table. To delete only one property of an object, use DELETE_PROPERTY.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;      /* 1H or DELETE_NAME */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON           header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* input */
    unsigned char       unique_name_flag;   /* not used */
    unsigned short     property_type;       /* not used */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;     /* not used */
    unsigned long       extra_buffer_addr;  /* not used */
    unsigned short     extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

name_buffer_addr

An address that points to a buffer containing the name of the object to delete, with this structure. The maximum length is 16 bytes.

```
typedef struct name_buffer {
    unsigned char       name_length;
    unsigned char       name[1];      /* set to length */
} NAME_BUFFER;
```

property_value_type

Set to 0.

Responses**Output Arguments**

None

Response Codes

See Table 11-4 on page 149.

DELETE_PROPERTY

DELETE_PROPERTY deletes a property from the specified object. To delete all the object's properties, use the DELETE_NAME command.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of name_server_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 50H or NAME_SERVER */
    unsigned char     opcode;      /* 4H or DELETE_PROPERTY */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON          header;
    unsigned char     reserved[6];
    unsigned long     name_buffer_addr;    /* input */
    unsigned char     unique_name_flag;   /* not used */
    unsigned short    property_type;      /* input */
    unsigned char     property_value_type; /* set to 0 */
    unsigned long     pv_buffer_addr;     /* not used */
    unsigned long     extra_buffer_addr;  /* not used */
    unsigned short    extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

`name_buffer_addr`

An address that points to a buffer containing the name of the object, with this structure. The maximum length is 16 bytes.

```
typedef struct name_buffer {
    unsigned char    name_length;
    unsigned char    name[1];      /* set to length */
} NAME_BUFFER;
```

`property_type`

Specifies the type of property value to delete.

See also: Property types, ADD_NAME command

`property_value_type`

Set to 0.

Responses

Output Arguments

None

Response Codes

See Table 11-4 on page 149.

DELETE_SEARCH_DOMAIN

DELETE_SEARCH_DOMAIN removes subnet IDs from the list of subnets the Name Server searches to resolve a name into an address.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;     /* 9H or
                                     DELETE_SEARCH_DOMAIN */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON           header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* not used */
    unsigned char       unique_name_flag;   /* not used */
    unsigned short      property_type;      /* not used */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;     /* input */
    unsigned long       extra_buffer_addr;  /* not used */
    unsigned short      extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

property_value_type

Set to 0.

pv_buffer_addr

The address pointing to a buffer that holds the list of subnet IDs, with this structure:

```
struct domain_list_struct {
    unsigned short length;           /* overall */
    unsigned char count;
    unsigned short search_domain_list[1]; /* set to count */
} DOMAIN_LIST_STRUCT;
```

Where:

length The total number of bytes in length, count and the array of subnet IDs, with a maximum value of 163.

count The number of entries in the list, with a maximum of 80.

search_domain_list

An array of subnet IDs to remove from the search domain.

Responses

Response Codes

See Table 11-4 on page 149.

GET_NAME

GET_NAME returns the name(s) of the object that has the given property type and value.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of name_server_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;  /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 50H or NAME_SERVER */
    unsigned char     opcode;      /* 5H or GET_NAME */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON         header;
    unsigned char     reserved[6];
    unsigned long     name_buffer_addr;    /* not used */
    unsigned char     unique_name_flag;    /* not used */
    unsigned short    property_type;       /* input */
    unsigned char     property_value_type; /* set to 0 */
    unsigned long     pv_buffer_addr;      /* input */
    unsigned long     extra_buffer_addr;   /* in/out */
    unsigned short    extra_buffer_length; /* input */
} NAME_SERVER_RB;
```

Input Arguments

`property_type`

The type of property value for the object.

See also: Property types, ADD_NAME command

`property_value_type`

Set to 0.

pv_buffer_addr

An address that points to a buffer containing the property value, with this structure:

```
typedef struct value_buffer {
    unsigned short    length;
    unsigned char    value[1];    /* set to length */
} VALUE_BUFFER;
```

extra_buffer_addr

An address that points to a buffer where the Name Server will return a list of names having the given property type and value.

extra_buffer_length

The size of the extra buffer in bytes.

Responses

Output Arguments

extra_buffer_addr

An address that points to the buffer where the Name Server returns a list of names, with this structure:

```
typedef struct each_name {
    unsigned char    entry_length;
    unsigned char    entry_name[1];
} EACH_NAME;

struct extra_buffer {
    unsigned short    length;    /* overall */
    unsigned char    count;    /* of names */
    EACH_NAME        name_list[1] /* set to count */
};
```

Where:

count The number of entries in the list

name_list The list of returned names. The first byte of each entry contains the length of the name that follows.

Response Codes

See Table 11-4 on page 149.

Additional Information

Since the Name Server enables you to enter different names for objects having the same property type and value, more than one name may be returned for this function. The request is retransmitted over the network and object names are collected and returned by the Name Server.

The Name Server is preconfigured with values for the number of times the request is retransmitted, the time interval between each retransmission, and the maximum number of responses that can be handled by the Name Server. In ICU-configurable systems you can configure these values with the RET, NR and RSP parameters on the NS screen of the ICU.

If the number of responses is more than the configured maximum, the Name Server returns an E_MAX_RESP response code. An E_BUFF_SPACE response code is returned if the extra buffer length is too short to hold all names. However, in both cases, the list of names will be valid. If E_BUFF_SPACE is returned, try the function again with a larger buffer.

See also: Name Server configuration values, Appendix A

GET_SEARCH_DOMAIN

GET_SEARCH_DOMAIN returns the list of subnet IDs that the Name Server is enabled to search.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;      /* 0AH or
                                     GET_SEARCH_DOMAIN */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON           header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* not used */
    unsigned char       unique_name_flag;    /* not used */
    unsigned short      property_type;       /* not used */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;      /* in/out */
    unsigned long       extra_buffer_addr;   /* not used */
    unsigned short      extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

`property_value_type`

Set to 0.

`pv_buffer_addr`

An address pointing to an application buffer where the Name Server will return the list of subnet IDs. The buffer must be large enough to hold all possible subnet IDs; its maximum size is 163 bytes.

Responses

Output Arguments

`pv_buffer_addr`

An address pointing to the application buffer where the Name Server returns the list of subnet IDs, using this structure:

```
struct domain_list_struct {
    unsigned short  length;           /* overall */
    unsigned char   count;
    unsigned short  search_domain_list[1]; /* set to count */
} DOMAIN_LIST_STRUCT;
```

Where:

`length` The total number of bytes in `length`, `count` and the array of subnet IDs, with a maximum value of 163.

`count` The number of entries in the list, with a maximum of 80.

`search_domain_list`

An array of subnet IDs currently in the search domain.

Response Codes

See Table 11-4 on page 149.

GET_SPOKESMAN

GET_SPOKESMAN finds the spokesman ID (Ethernet address) of the system whose Name Server has cataloged the specified object.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;      /* 6H or GET_SPOKESMAN */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON           header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* input */
    unsigned char       unique_name_flag;    /* not used */
    unsigned short      property_type;       /* input */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;      /* not used */
    unsigned long       extra_buffer_addr;   /* in/out */
    unsigned short      extra_buffer_length; /* in/out */
} NAME_SERVER_RB;
```

Input Arguments

`name_buffer_addr`

An address that points to a buffer containing the name of the object for which the spokesman ID is required. The buffer has this structure.

```
typedef struct name_buffer {
    unsigned char       name_length;
    unsigned char       name[1];        /* set to length */
} NAME_BUFFER;
```

`property_type`

The property type of the object.

See also: Property types, `ADD_NAME` command

`property_value_type`

Set to 0.

`extra_buffer_addr`

An address that points to a buffer where the Name Server will return the Ethernet address of the system where the object is cataloged.

`extra_buffer_length`

The size of the extra buffer in bytes. An Ethernet address is six bytes long.

Responses

Output Arguments

`extra_buffer_addr`

An address that points to a buffer holding the returned Ethernet address.

`extra_buffer_length`

The size of the extra buffer in bytes. The Name Server changes the length of the extra buffer to the actual length of the returned value.

Response Codes

See Table 11-4 on page 149.

Additional Information

The `GET_SPOKESMAN` function finds the Name Server system where a given object is entered. This information is particularly helpful for administering and maintaining a network.

GET_VALUE

GET_VALUE returns the property value of an object, which can be in either the local Name Server object table or on another system.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;      /* 2H or GET_VALUE */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON           header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* input */
    unsigned char       unique_name_flag;    /* input */
    unsigned short      property_type;       /* input */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;      /* in/out */
    unsigned long       extra_buffer_addr;   /* not used */
    unsigned short      extra_buffer_length; /* not used */
} NAME_SERVER_RB;
```

Input Arguments

`name_buffer_addr`

An address that points to a buffer containing the name of the object, with this structure. The maximum length is 16 bytes.

```
typedef struct name_buffer {
    unsigned char       name_length;
    unsigned char       name[1];      /* set to length */
} NAME_BUFFER;
```

`unique_name_flag`

A flag indicating whether to return a unique value for the object: 00H for non-unique and 0FFH for unique. You may specify unique even if the object is stored as non-unique; the command will return the value for only the local non-unique object. For example, if you want the value for the local MYHOSTID object of type 4H, specify unique in this field even though the MYHOSTID object is non-unique.

`property_type`

Specifies the type of property value to return.

See also: Property types, ADD_NAME command

`property_value_type`

Set to 0.

`pv_buffer_addr`

An address that points to a buffer where the Name Server returns the property value. The structure of the value buffer depends on the value of `unique_name_flag`, but in either case, set the length of the value buffer large enough to hold returned values before issuing the request block.

```
typedef struct value_buffer {
    unsigned short    length;
    unsigned char     value[1];    /* set to length */
} VALUE_BUFFER;
```

When `unique_name_flag` is set to unique, the buffer has this structure:

```
typedef struct value_buffer {
    unsigned short    length;
    unsigned char     value[1];
} VALUE_BUFFER;
```

When `unique_name_flag` is set to non-unique, the buffer has this structure:

```
typedef struct each_value {
    unsigned short    value_length;
    unsigned char     value[1];
} EACH_VALUE;

typedef struct nu_value_buffer {
    unsigned short    length;    /* overall */
    unsigned char     count;
    EACH_VALUE        value_list[1];
} NU_VALUE_BUFFER;
```

Responses

Output Arguments

`pv_buffer_addr`

An address that points to the value buffer returned by the Name Server (see the structures shown above). The `count` and `value` fields are filled in by the Name Server.

Response Codes

See Table 11-4 on page 149.

Additional Information

The GET_VALUE function can retrieve the property value of a unique or non-unique object. For a unique object, the returned value buffer contains just one property value. For a non-unique object, the returned buffer contains a list of property values. The Name Server obtains the list of values by retransmitting the request over the network up to a configured maximum number of times, and collecting the responses from various systems. The list of returned values for a non-unique object is shown in the `nu_value_buffer` structure under the `pv_buffer_ptr` parameter above.

The Name Server is preconfigured with values for the number of times the request is retransmitted, the time interval between each retransmission, and the maximum number of responses that can be handled by the Name Server. In ICU-configurable systems you can configure these values with the RET, NR and RSP parameters on the NS screen of the ICU.

If a response is not received after retransmitting the request the maximum number of times, the Name Server returns an `E_NAME_NOT_EXIST` response code. If the number of responses received is more than the configured maximum, an `E_MAX_RESP` response code is returned. However, in this case, property values returned in the list are valid. A response code of `E_BUFF_SPACE` is returned if the value buffer length is not sufficient to hold the property values.

See also: Name Server configuration values, Appendix A

LIST_TABLE

LIST_TABLE lists all objects cataloged in the local Name Server object table.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of name_server_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 50H or NAME_SERVER */
    unsigned char       opcode;      /* 7H or LIST_TABLE */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct name_server_rb {
    RB_COMMON            header;
    unsigned char       reserved[6];
    unsigned long       name_buffer_addr;    /* not used */
    unsigned char       unique_name_flag;    /* not used */
    unsigned short      property_type;       /* not used */
    unsigned char       property_value_type; /* set to 0 */
    unsigned long       pv_buffer_addr;      /* not used */
    unsigned long       extra_buffer_addr;   /* in/out */
    unsigned short      extra_buffer_length; /* input */
} NAME_SERVER_RB;
```

Input Arguments

property_value_type

Set to 0.

extra_buffer_addr

An address pointing to a buffer that will hold the list of objects returned by the Name Server.

extra_buffer_length

The size of the extra buffer in bytes.

Responses

Output Arguments

`extra_buffer_addr`

An address that points to the buffer where the Name Server returns the list of objects, in this structure:

```
typedef struct each_object {
    unsigned char    name_length;
    unsigned char    name[1];        /* name_length */
    unsigned char    unique_name_flag;
    unsigned short   property_type;
    unsigned char    pv_type;
    unsigned short   pv_length;
    unsigned char    property_value[1]; /* pv_length */
} EACH_OBJECT;

struct list_buffer {
    unsigned short   length;          /* overall */
    unsigned char    count;
    EACH_OBJECT      object_list[1]; /* set to count */
} LIST_BUFFER;
```

Where:

`count` The number of entries in the list.

`object_list`
 An array of returned objects. For a description of the fields in the `each_object` structure, see the request block fields in the `ADD_NAME` command.

Response Codes

See Table 11-4 on page 149.

Additional Information

The `LIST_TABLE` function returns only the objects defined in the local object table. It does not list objects cataloged on remote systems. The application must provide a buffer large enough to hold all the returned names, property types and values. An `E_BUFF_SPACE` response code indicates the buffer is too small; however, the list of returned objects is valid.



Programming the Transport Layer 12

The iNA 960 Transport Layer provides message delivery or transport services between application processes running on network end systems. The applications send and receive request blocks using the Transport Layer, using the `cq_comm_rb` call.

See also: Using the `cq_` System Calls, Chapter 10

The applications are identified by transport service access point (TSAP) addresses, consisting of a network service access point (NSAP) address and TSAP selectors. A TSAP selector identifies the access point between the client process and the Transport service.

See also: NSAP (Network Service Access Point), Chapter 8

⇒ **Note**

This chapter refers to the application as a client, meaning it is a client of the Transport service. This has no meaning in terms of the client/server relationship between applications. Both client and server applications are clients of the Transport service.

Transport Services

The Transport service provides these two types of message delivery services:

- A reliable connection-oriented virtual circuit (VC) message delivery service between two TSAP addresses.
- A non-guaranteed connectionless datagram delivery service between one TSAP address and another (point-to-point delivery), or between one TSAP address and several other TSAP addresses (multicast delivery).

Table 12-1. Transport Layer Commands

Opcode	Literal	Description
4H	ACCEPT_CONNECT_REQUEST	Accepts the connection request after the AWAIT_CONNECT_REQUEST_CLIENT command
0DH	AWAIT_CLOSE	Notifies the client when a connection is terminated
3H	AWAIT_CONNECT_REQUEST_CLIENT	Enables the application to directly control whether a connection is accepted
2H	AWAIT_CONNECT_REQUEST_TRAN	Permits the Transport service to independently accept a connection request
0CH	CLOSE	Closes a connection or indicates that an incoming connection request is being refused.
0H	OPEN	Sets up a CDB (connection database) to manage a specific VC (virtual circuit) connection between the service client and the external process
16H	RECEIVE_ANY	Receives data by VC service using a buffer that may be used by any CDB
7H	RECEIVE_DATA	Receives normal data by VC service for a specific CDB
12H	RECEIVE_DATAGRAM	Receives datagrams
0AH	RECEIVE_EXPEDITED_DATA	Receives expedited data from the remote node for a specific CDB
1H	SEND_CONNECT_REQUEST	Attempts to establish an active connection to the remote node
5H	SEND_DATA	Sends data using normal VC service
11H	SEND_DATAGRAM	Sends data using datagram service
6H	SEND_EOM_DATA	Sends data using normal VC service and marks EOM (end of message)
9H	SEND_EXPEDITED_DATA	Sends up to 16 bytes of data using expedited VC delivery service

continued

Table 12-1. Transport Layer Commands (continued)

Opcode	Literal	Description
0EH	STATUS	Provides status information for VC services from the Transport Layer and for a specific CDB
13H	WITHDRAW_DATAGRAM_RECEIVE_BUFFER	Withdraws one or more datagram receive buffers to reclaim resources
0BH	WITHDRAW_EXPEDITED_BUFFER	Withdraws one or more expedited VC receive buffers to reclaim resources
8H	WITHDRAW_RECEIVE_BUFFER	Withdraws one or more normal VC receive buffers to reclaim resources

Virtual Circuit Service

The VC Transport service uses the ISO 8073 standard Class 4 transport protocol which provides these services:

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

The Transport service provides these services by means of a connection or VC. A pair of applications set up the connection (connection establishment phase), transfer data (data transfer phase) and terminate or disconnect the connection (connection termination phase) between themselves.

Example Software

The OS software includes an example application using the Transport VC services. See the files under the */rmx386/demo/network* directory.

Datagram Service

The datagram Transport service uses the ISO 8602 Connectionless Transport protocol to transfer data between application processes without setting up a connection. This service gives no guarantee of data delivery. Data may be lost or misordered. In addition, data may be multicast at one time to a single destination or to several destinations.

Buffers

Use of the Transport service requires passing address information and data back and forth between the Transport service client and the Transport service. The application loads address information and data into a buffer memory area, and then uses a 32-bit addresses to specify the location of the buffers.

There are three types of buffers used by the Transport service:

- The TSAP Address Buffer
- Contiguous Buffers
- Noncontiguous Buffers

Buffer Addressing

If the Transport service client uses pointers to reference data buffers, the pointers must be translated to physical addresses for use in request blocks. The client is responsible for both the forward and reverse translation.

See also: Translating pointers, Chapter 10

TSAP Address Buffer

The TSAP address buffer holds both the local and remote address information to establish a Transport service connection or transfer. The TSAP address buffer consists of local NSAP and TSAP selectors, a remote NSAP address, and a remote TSAP selector. The local TSAP selector describes the location of the local Transport service client. The remote NSAP address identifies the remote node. This can be either the remote end of the VC connection or the remote destination of a datagram. A remote TSAP selector must be specified for an active connection to a remote client.

The local network service access point ID (NSAP selector) specifies the access point through which the Transport service gains the services of the Network Layer. The local Transport service most often uses a default local NSAP selector, or optionally, the application can specify one.

The formats for an NSAP address are specified in ISO 8348 Addendum 2. The NSAP selector is usually 1 byte long and is the last byte of the NSAP address. The length of an NSAP address or NSAP selector depends on the underlying network service provided to the Transport service. For an iNA 960 Network Layer, the application should always specify the NSAP selector value and an NSAP selector length of 1 byte (do not allow either to default). The content of the NSAP address is transparent to the Transport service, although the length must be known in order to allocate memory that will be used to store the address.

The TSAP selector length has not been standardized (by ISO) since it depends on the conventions of the Transport service client. For the iNA 960 Transport service the TSAP selector length is 2 bytes. The maximum length TSAP selector in iNA 960 is 32 bytes.

The Transport service enables for variable length NSAP addresses, NSAP selectors, and TSAP selectors. The Transport service client stores these values in a client-defined buffer and gives the address of this buffer to the Transport service.

This structure logically illustrates the contents of a TSAP address buffer allocated by a Transport service client. For the actual structure see Figure 12-1 or the *cqtransp.h* or *cqtransp.lit* include files. The local NSAP selector is usually defaulted, with `loc_nsap_sel_len` set to zero, and the `loc_nsap_sel` field does not appear in the typedef for the structure `ta_buffer`. In this case, the Transport service depends on the Network Layer to maintain the default local NSAP.

```

struct TSAP_address_buffer {
    unsigned char    loc_nsap_sel_len;
    unsigned char    loc_nsap_sel [loc_nsap_sel_len];
    unsigned char    loc_tsap_sel_len;
    unsigned char    loc_tsap_sel [loc_tsap_sel_len];
    unsigned char    rem_nsap_addr_len;
    unsigned char    rem_nsap_addr [rem_nsap_addr_len];
    unsigned char    rem_tsap_sel_len;
    unsigned char    rem_tsap_sel [rem_tsap_sel_len];
};

```

The client must be aware of the lengths and formats of the NSAP addresses used to specify the remote ends of a connection or datagram transfer. The client loads the length of the NSAP address into the remote NSAP address length field and allocates enough space in the TSAP address buffer to load the address.

Application processes communicating using the Transport service must agree on formats and lengths of NSAP/TSAP selectors. The lengths are loaded into the appropriate TSAP address buffer length fields where enough buffer space has been allocated to accommodate them. Limits on the maximum lengths of the NSAP address and TSAP selectors are configured into the iNA 960 software.

Use of NSAP or TSAP selectors in an address is optional. A null NSAP or TSAP selector is one with a length of 0. In this case, there is no corresponding content field in the TSAP address buffer. On transmission, a null local NSAP selector relies on the underlying Network Layer to provide a default NSAP selector for transmitting the transport protocol data units (TPDUs).

The default remote NSAP selector for iNA Network layers in the `rem_nsap_sel` field in `remote_nsap_address` is a 1-byte NSAP selector set to 0. An NSAP selector of 0 specifies the Null2 addressing scheme. A nonzero NSAP selector specifies the IP addressing scheme. Null local or remote TSAP selectors cause the Transport service to use a default TSAP selector value. This value is determined by the Transport Layer configuration. Incoming connect requests or connect confirm TPDUs that don't have TSAP selectors in them also cause the Transport service to use a default TSAP selector value. A connection with a null TSAP selector listens for default incoming TSAP selectors.

See also: iNA Network Layer Addressing Schemes, Chapter 8,
 Configuration values, Appendix A

Figure 12-1 shows a sample transport address as it might appear in the iRMX `:sd:net/data` file. Your application could use the structures in Figure 12-1 to fill a TSAP address buffer with such an address. The numbers above elements of the address correspond to the numbers at the end of lines in the structure declarations. This is an example of a fully specified TSAP address.

See also: `:sd:net/data` file, Chapter 11,
 LSAP Identifiers, Chapter 13

```

          1  2  3    4  5  6    7                8  9  10 11
node_1:address=0x 00 02 4141 0B 49 0003 00AA000003A2 FE 00 02 4141;

struct nsap_addr {
unsigned char    afi;                /* 5 */
unsigned short  subnet;             /* 6 */
unsigned char    host_id[6];        /* 7 */
unsigned char    lsap_sel;          /* 8 */
unsigned char    nsap_sel;          /* 9 */
} NSAP_ADDR;

struct ta_buffer {
    unsigned char loc_nsap_sel_len; /* 1, set to zero */
    unsigned char loc_tsap_sel_len; /* 2 */
    unsigned char loc_tsap_sel [loc_tsap_sel_len]; /* 3 */
    unsigned char rem_nsap_addr_len; /* 4 */
    NSAP_ADDR     rem_nsap_addr;
    unsigned char rem_tsap_sel_len; /* 10 */
    unsigned char rem_tsap_sel [rem_tsap_sel_len]; /* 11 */
};

```

Figure 12-1. TSAP Address Format

Table 12-2 shows valid and invalid values for the fields in the TSAP address buffer structure, `ta_buffer`, for active and passive connections.

Table 12-2. TSAP Address Buffer Field Values

Address Buffer Field	Active Connection	Passive Connection
loc_nsap_sel_len = 0 = 1	Valid ¹ Valid	Valid ¹ Valid
loc_nsap_sel = 0 ≠ 0	Valid ² Valid ²	Valid ² Valid ²
loc_tsap_sel_len = 0 = 2	Valid ¹ Valid ³	Valid ¹ Valid ³
loc_tsap_sel = 0 ≠ 0	Invalid Valid	Invalid Valid
rem_nsap_addr_len = 0 ≠ 0	Invalid Valid ⁴	Invalid Valid
rem_nsap_addr = 0 ≠ 0	Invalid Valid	Valid ⁵ Valid
rem_tsap_sel_len = 0 = 2	Valid ¹ Valid ⁶	Valid ¹ Valid
rem_tsap_sel = 0 ≠ 0	Invalid Valid	Valid ⁷ Valid

¹ Use configuration default

² Depends on Transport Layer configuration

³ Only if `loc_tsap_sel` ≠ 0

⁴ Only if `rem_nsap_addr` ≠ 0

⁵ NSAP address is unspecified and TSAP address is unspecified or partially specified depending on value of `rem_nsap_addr`

⁶ Only if `rem_tsap_sel` ≠ 0

⁷ Only if NSAP address is unspecified (`rem_nsap_addr` = 0)

TSAP addresses are either specified, partially specified, or unspecified depending on the contents of the TSAP address buffer.

Fully specified TSAP address Required for active connections and has specified the `loc_tsap_sel`, `rem_nsap_addr`, and `rem_tsap_sel` fields with these values:

```
loc_nsap_sel_len = 0 or 1
loc_tsap_sel_len = 0 or 2
loc_tsap_sel ≠ 0
rem_nsap_addr_len ≠ 0
rem_nsap_addr ≠ 0
rem_tsap_sel_len ≠ 0
rem_tsap_sel ≠ 0
```

Partially specified TSAP address Works only with passive connections and has these values:

```
loc_nsap_sel_len = 0 or 1
loc_tsap_sel_len = 0 or 2
loc_tsap_sel ≠ 0
rem_nsap_addr_len ≠ 0
rem_nsap_addr = 0
rem_tsap_sel_len = 0 or 2
rem_tsap_sel ≠ 0
```

Unspecified TSAP address Works only with passive connections and has these values:

```
loc_nsap_sel_len = 0 or 1
loc_tsap_sel_len = 0 or 2
loc_tsap_sel ≠ 0
rem_nsap_addr_len ≠ 0
rem_nsap_addr = 0
rem_tsap_sel_len = 2
rem_tsap_sel = 0
```

Use these guidelines for specifying addresses:

- Unspecified means that the address or selector value, not the length, is zero.
- Unspecified local TSAP selectors are invalid.
- Using fully specified NSAP addresses with unspecified remote TSAP selectors is invalid.

Contiguous Buffers

With the VC service, the transport protocol enables the optional transfer of a small amount of data during the connection establishment and termination phases. Up to 32 bytes of data can be transferred during the connection establishment phase (SEND_CONNECT_REQUEST and AWAIT_CONNECT_REQUEST commands) and up to 64 bytes of data can be transferred during the connection termination phase (CLOSE command). To transfer data during these phases, the Transport service interface requires that a single contiguous buffer block be allocated in application memory to send or receive the data.

The iNA 960 software follows certain rules regarding the transfer of data during the connection establishment and termination phases, depending on the values of buffer length (`client_data_len`) and address (`client_data_buf_addr`) fields specified in the iNA 960 command request block. These rules are used by the iNA 960 software:

- If `client_data_len = 0`, no data is sent. Data can be received if `client_data_buf_addr ≠ 0`.
- If `client_data_len ≠ 0`, data of specified length is sent.
- If `client_data_buf_addr = 0`, no data is sent or received.
- If `client_data_buf_addr ≠ 0`, data may or may not be sent depending on the value of `client_data_len`.

Regardless of the value of `client_data_len` for data to be sent, data may be received and can be up to 64 bytes in length. Therefore, `client_data_buf_addr` should always point to the start of a contiguous 64 byte block.

Noncontiguous Buffers

For transferring data through VCs or datagrams, the application may allocate multiblock noncontiguous buffers to hold the data. The buffers are set up as an array of structures of this type within the request block. The address of each buffer in the array must be converted from a pointer to a 32-bit physical address before sending the request block to iNA 960.

```
typedef struct data_block {
    unsigned long    address;
    unsigned short   length;
} DATA_BLOCK;
```

ISO Reason Codes

Table 12-3 lists the ISO reason codes that can be returned in a request block when a transport connection is disconnected. The reasons are defined as constants in the include files.

Table 12-3. ISO Reason Codes

Code	(Decimal)	Reason
0		NOT_SPECIFIED
1		CONGESTION_AT_TSAP
2		CLIENT_NOT_ATTACHED_TO_TSAP
3		ADDRESS_UNKNOWN
80H	128	CLIENT_DISCONNECT (Normal disconnect started by client)
81H	129	CONNECT_REQ_TRANSPORT_CONGESTION (Remote transport service congestion at connect request time)
82H	130	CONNECT_NEGOT_FAILURE (Connection negotiation failed; for example, the proposed classes are not supported)
83H	131	DUPLICATE_CONNECTION
84H	132	MISMATCHED_REFS (connection references)
85H	133	PROTOCOL_ERROR
87H	135	REFERENCE_OVERFLOW
88H	136	CONNECTION_REQ_REFUSED (On this network connection)
8AH	138	INVALID_LENGTH (of header or parameter)

Virtual Circuit Commands

Establishing and using a VC connection encompasses three different operational phases:

1. Connection Establishment Phase
2. Data Transfer Phase
3. Connection Termination Phase

There are Transport commands for each phase that enable the Transport client to establish a connection to a remote node, transfer data, and then terminate the connection. The Transport service also provides a command to query the status of a connection.

Commands to Establish a Connection

The connection establishment commands enable a Transport service client to open and maintain a connection to a remote node. They are:

```
OPEN
SEND_CONNECT_REQUEST
AWAIT_CONNECT_REQUEST_TRAN
AWAIT_CONNECT_REQUEST_CLIENT
ACCEPT_CONNECT_REQUEST
```

The OPEN command sets up an internal memory resource to manage a VC connection between the Transport service client and the external process. A memory segment called a connection database (CDB) is allocated by the OPEN command to help manage the connection. The CDB contains information that reflects the current state of an open connection.

Once the CDB is allocated, the Transport service client may attempt to establish an active connection to the remote node using the SEND_CONNECT_REQUEST command. Such a request is referred to as an active open.

To use the VC to passively listen for an incoming connection request (as the remote node would have to), the application can permit the Transport service to independently accept a connection request by using the AWAIT_CONNECT_REQUEST_TRAN command.

To directly control whether the connection is accepted, the application can instead use the AWAIT_CONNECT_REQUEST_CLIENT command. To accept the connection request after completion of an AWAIT_CONNECT_REQUEST_CLIENT command, the application uses an ACCEPT_CONNECT_REQUEST command. This establishes an active connection.

Commands for the Data Transfer Phase

These data transfer commands enable a Transport service client and a remote process to exchange expedited or normal data after the successful completion of the Connection Establishment Phase.

```
SEND_DATA
SEND_EOM_DATA
RECEIVE_DATA
WITHDRAW_RECEIVE_BUFFER
SEND_EXPEDITED_DATA
RECEIVE_EXPEDITED_DATA
WITHDRAW_EXPEDITED_BUFFER
RECEIVE_ANY
```

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

Expedited data takes the form of short urgent messages that have a higher transmission priority than normal data. The normal flow control mechanisms of the Transport service are bypassed in order to transmit expedited data ahead of any normal data. There is a configured limit of 16 bytes for the amount of expedited data that may be transferred or received.

Posting Receive Buffers for Virtual Circuits

Some data transfer commands post receive buffers, which the Transport service uses to store data received over a specific VC connection.

The Transport service relies on the application to post all receive buffers for data received for a VC. Use these guidelines to manage receive buffers:

- Data received on one connection cannot use a buffer specifically posted for another connection.
- Multiple buffers can be posted (except with the `RECEIVE_ANY` command) in a single command using the `num_blks` argument, but they will all belong to the same TSDU.
- The `RECEIVE_ANY` command can be used to post a buffer that is not referenced to a specific connection, but is referenced to a list of up to 20 connections. If data is received on a connection for which there is no specific buffer posted, and the connection is in the reference list for a buffer posted by `RECEIVE_ANY`, the data is placed in that buffer.
- Only one buffer can be posted per `RECEIVE_ANY` command. Multiple buffers must be posted with multiple `RECEIVE_ANY` commands.

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

Commands to Terminate a Connection

Connection termination commands enable a Transport service client to terminate an open or active VC connection. Terminating an active VC also terminates any activity on the connection during the data transfer phase. The commands are:

CLOSE
AWAIT_CLOSE

⇒ **Note**

Connection termination is not graceful; data in transit may be lost.

The CLOSE command closes a connection or may serve as an indication that an incoming connection request is being refused by the Transport service client, after an AWAIT_CONNECT_REQUEST_CLIENT command.

The AWAIT_CLOSE command is used by a client to ensure that the client is notified when a connection is terminated.

Datagram Commands

The datagram commands enable a local Transport service client and a remote user process to exchange datagram messages. These are the datagram commands:

SEND_DATAGRAM
RECEIVE_DATAGRAM
WITHDRAW_DATAGRAM_BUFFER

Transmission of datagrams using these commands does not guarantee that the datagram messages will arrive at their destination in the order that they were sent, nor do they guarantee that the datagrams will arrive at all. The receiver of a datagram transmission must post the necessary receive buffers.

Posting Receive Buffers for Datagrams

The Transport service relies on the application to post all receive buffers for data received from a remote Transport service using the datagram service. The posted buffer is available only to a specific TSAP. Only datagrams addressed to that TSAP can use that buffer for passing data.

The data in each datagram sent by the Transport service is a self-contained entity. If the total datagram buffer space available for a TSAP is less than the length of data received in the datagram, the datagram is discarded with no data buffered. Otherwise, the data is buffered. Data from one datagram can be buffered in one or more posted receive buffers. The request block for the buffer containing the last byte of data in the received datagram has a response code indicating that this buffer is the end-of-message (EOM). The EOM buffer is returned when the last data of the datagram is buffered, even if space remains in the buffer. Thus, a returned buffer can contain data from no more than one received datagram.

Datagram receive buffers are posted and maintained separately from VC receive buffers.

Transport Service Commands

The Transport service commands in this chapter are specified by the `subsystem` and `opcode` fields in the request block header, `rb_common`. The commands use similar argument structures following the common header fields. Each command description lists which fields are input and output arguments. Initialize reserved fields and unused fields to 0. The structures are provided as typedefs in the include files for the Transport Layer.

See also: Include Files, Chapter 10,
 Programming with Structures, Chapter 10

ACCEPT_CONNECT_REQUEST

ACCEPT_CONNECT_REQUEST indicates the application's acceptance of the Transport connection specified by the reference. This is the positive response to a previously returned AWAIT_CONNECT_REQUEST_CLIENT request block. The application can return an optional buffer of client data with the connection confirmation in a contiguous buffer. This command is only used by an application that has received a response to an AWAIT_CONNECT_REQUEST_CLIENT command. The AWAIT_CONNECT_REQUEST_TRAN command does not provide the option of accepting or rejecting a connection.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of conn_req_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 4H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct conn_req_rb {
    RB_COMMON          header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[4];
    unsigned short    ack_delay_estimate; /* output */
    unsigned long     ta_buffer_addr;    /* output */
    unsigned short    persistence_count; /* unused */
    unsigned short    abort_timeout;    /* unused */
    unsigned short    reference;        /* input */
    unsigned char     qos;              /* unused */
    unsigned short    negot_options;    /* output */
    unsigned long     client_data_buf_addr; /* in/out */
    unsigned char     client_data_len;  /* in/out */
} CONN_REQ_RB;
```

Input Arguments

`reference`

Identifies the CDB this request applies to.

`client_data_buf_addr`

An address descriptor that identifies a contiguous 64-byte buffer. If the address is zero, no buffer is allocated and there is no client data sent with the request. Also, no data will be received. To send client data with the request, the buffer address and length must be nonzero and the data (0 to 32 bytes) must be loaded in the buffer.

`client_data_len`

The number of bytes of data to send.

Responses

Output Arguments

`iso_reason_code`

The ISO disconnect reason code, if the connection was aborted by the remote Transport service during the connection establishment phase. Otherwise, the value of this argument is 0.

See also: Table 12-3 on page 185

`ack_delay_estimate`

0 is always returned.

`ta_buffer_addr`

An address pointing to a TSAP address buffer that identifies the local and remote end nodes of a VC connection.

See also: TSAP address buffer structure, page 180

`negot_options`

The agreed-upon negotiation options.

See also: `negot_options`, `AWAIT_CONNECT_REQUEST_CLIENT` and `AWAIT_CONNECT_REQUEST_TRAN` commands

`client_data_buf_addr`

If the connection attempt was successful and a buffer was allocated, the request block will return in the buffer any data (at most 32 bytes) contained in the connection confirmation received from the remote Transport service. If the connection attempt was rejected by the remote Transport service and the local Transport service gives up, the request block will return in the buffer up to 64 bytes of any data contained in the disconnect request from the remote Transport service. The received data overwrites any data in the buffer that was sent in the request block.

`client_data_len`

The length of any data received.

Response Codes

<code>OK_RESPONSE</code>	1H	The connection just became established on completion of the three-way handshake.
<code>UNKNOWN_REFERENCE</code>	6H	The CDB corresponding to this reference is not allocated.
<code>OK_CLOSED_RESP</code>	7H	The local client aborted the connection before completion of the three-way handshake.
<code>BUFFER_TOO_LONG</code>	0AH	More than 32 bytes of client data were sent. The connection maintains its current state awaiting another local client response.
<code>REM_ABORT</code>	0EH	The remote Transport service aborted the connection before completion of the three-way handshake.
<code>LOC_TIMEOUT</code>	10H	The local Transport service timed out before completion of the three-way handshake. The connection was aborted.
<code>DUP_REQ</code>	14H	This is a duplicate connection response. The connection is already established. This error can occur if this call is made for a connection for which a connection request has not previously been returned to the client.

AWAIT_CLOSE

AWAIT_CLOSE requests notification that a specified connection has terminated. An ISO reason code is returned to indicate the cause of the disconnection.

Request Block

```

typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of vc_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 40H */
    unsigned char       opcode;      /* 0DH */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long       address;      /* in/out */
    unsigned short      length;      /* in/out */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON           header;
    unsigned char       iso_reason_code; /* output */
    unsigned char       reserved[15];
    unsigned short      reference;     /* in/out */
    unsigned char       qos;           /* unused */
    unsigned short      buf_len;       /* output */
    unsigned char       num_blks;      /* input */
    DATA_BLOCK         data_blk_list[1]; /* [num_blks] */
} VC_RB;

```

Input Arguments

`reference`

Identifies the CDB this request applies to.

`num_blks`

The number of separate buffers to be received. Each buffer is a block of contiguous memory defined by the `data_blk_list[i].address` and `data_blk_list[i].length` arguments. Set to 0 if no buffer is allocated. In this case, client data received in a disconnect request is ignored.

`data_blk_list[i].address`

The address descriptor for the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the i^{th} buffer. The total length of data in all blocks cannot exceed 64 bytes.

Responses

Output Arguments

`iso_reason_code`

The ISO reason code received in the disconnect request.

See also: Table 12-3 on page 185

`reference`

The reference for the connection that was deleted.

`buf_len`

The total length of the data received in the buffers posted by this command.

`data_blk_list[i].address`

The address for the start of the i^{th} buffer.

`data_blk_list[i].length`

This value is the length of the data in the last posted buffer to receive data. It is only meaningful for that buffer.

Response Codes

UNKNOWN_REFERENCE	6H	The reference does not correspond to an allocated CDB.
OK_CLOSED_RESP	7H	The local client aborted the connection or the connection was already closed when this command was requested.
REM_ABORT	0EH	A disconnect request was received from the remote Transport service on the specified connection.

LOC_TIMEOUT	10H	The local Transport service timed out.
DUP_REQ	14H	Another AWAIT_CLOSE command was posted previously on this connection.

Additional Information

If a buffer is allocated in this command, the remote application that sends the disconnect request can send data that may explain the cause of the disconnection. The longest disconnect message permitted by the ISO standard is 64 bytes. If the allocated buffer is smaller than the received data length, only the data that fits in the buffer is returned. The remainder is lost. If no buffer is allocated, any received disconnect data is discarded.

AWAIT_CONNECT_REQUEST_TRAN **AWAIT_CONNECT_REQUEST_CLIENT**

There are two versions of the `AWAIT_CONNECT_REQUEST_` command, with different opcode values. Both commands wait for an incoming connection request to the local TSAP address. The application specifies criteria for the type of request, including addresses to listen for, negotiation options, and whether data sent with the connection request will be received. The differences between the commands are:

- With the `AWAIT_CONNECT_REQUEST_TRAN` command, the transport service determines whether to make the connection without further input from the application. This is called a passive open.
- With the `AWAIT_CONNECT_REQUEST_CLIENT` command, the transport service passes the connection request to the application for further consideration. Transport then waits for a reply as to whether the connection is accepted or rejected. This is called an active open. The application accepts the request with an `ACCEPT_CONNECT_REQUEST` command or rejects it with a `CLOSE` command.

⇒ **Note**

The description of these commands refers to the application as a client, meaning the client of the Transport service. The application that calls these commands is actually a server in the context of a client/server network relationship, because it is the server application that waits for connection requests.

The `AWAIT_CONNECT_REQUEST_CLIENT` command is useful for a server application that restricts access to itself based on criteria that can be passed in a connection request. For example, the server might inspect a user login name against a password list or restricted user list before accepting the connection.

Request Block

```

typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;        /* of conn_req_rb */
    selector          user_id;       /* cq_create_comm_user */
    unsigned char     resp_port;    /* 0FFH */
    selector          resp_mbox;    /* mailbox token */
    selector          rb_seg_tok;   /* segment token */
    unsigned char     subsystem;    /* 40H */
    unsigned char     opcode;       /* 2H for _TRAN
                                     3H for _CLIENT */
    unsigned short    response;     /* initialize to 0 */
} RB_COMMON;

typedef struct conn_req_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[4];
    unsigned short    ack_delay_estimate; /* output */
    unsigned long     ta_buffer_addr;    /* in/out */
    unsigned short    persistence_count; /* input */
    unsigned short    abort_timeout;    /* input */
    unsigned short    reference;        /* input */
    unsigned char     qos;              /* input */
    unsigned short    negot_options;    /* in/out */
    unsigned long     client_data_buf_addr; /* in/out */
    unsigned char     client_data_len;  /* in/out */
} CONN_REQ_RB;

```

Input Arguments

`ta_buffer_addr`

An address pointing to a TSAP address buffer that specifies the local and remote end nodes of a VC connection. For the `AWAIT_CONNECT_REQUEST_CLIENT` command, the TSAP address must be fully specified. For the `AWAIT_CONNECT_REQUEST_TRAN` command, the remote TSAP address may be fully specified, partially specified, or unspecified. Specified TSAP selectors and NSAP addresses must have a nonzero length.

The contents field is filled with zeros for unspecified TSAP selectors or NSAP address. The lengths of the remote net address and local or remote TSAP selectors must not exceed the limits specified in the system configuration, otherwise an addressing error will occur. The local TSAP selector must either be null or specified (nonzero). The local NSAP selector may be specified (nonzero), unspecified (zero), or null (zero length). Multiple connections to or from a single TSAP address can be requested.

See also: TSAP address buffer structure, page 180

`persistence_count`

The number of times to retry an active connection attempt upon connection refusal, before giving up. Connection refusal means that the remote system refuses the connection, not that it failed to respond to the connection attempt. A connection refusal typically occurs when the remote system is not listening (it hasn't executed a passive open). Values may be:

Value	Meaning
0	The configured value will be used
0FFFFH	Retry forever
1 to 0FFFEH	This value will be used as the persistence count

`abort_timeout`

The retransmission timeout period before aborting the connection, in 51-millisecond units. Possible values are:

Value	Meaning
0	Use <code>abort_timeout</code> configuration value (not 0)
0FFFFH	Never time out
1-0FFFEH	Use this number of 51-millisecond time units

This is how long the Transport service will continue to transmit without receiving a response. This applies to both the connection establishment and data transfer phases. During the connection establishment phase, this value controls how long a connection request will be retransmitted when there is no response. During the data transfer phase, this value controls how long data is retransmitted when there is no ACK. The timeout period does not apply to the connection termination phase; the timeout for connection termination is a Transport service configuration parameter.

`reference`

Identifies the CDB this request applies to.

`qos` Quality of service: the only possible parameter is the transmit priority for underlying subnetworks that support it. This is a value in the range 0 to 15; 0 is the highest priority. For iNA 960 802.3 subnets, set `qos` to 0; priority is not supported.

negot_options

Specifies various classes of service and additional options requested for negotiation on this connection. If `negot_options` is zero, default options are used, as specified by the `def_negot_options` configuration parameter. Otherwise, break the value into four nibbles and specify options, where nibble 1 is least significant:

Nibble	Value	Meaning
1	0	use 7-bit sequence numbers
	2	use 31-bit sequence numbers
2	4	class four service
	0	no expedited service, do checksums
3	1	expedited service, do checksums
	2	no expedited service, no checksums
	3	expedited service, no checksums
4	8	client specified (nondefault) negotiation options

These are valid values for this argument:

Value	Meaning
8342H	Expedited data; no checksum; transport class 4; 31-bit sequence numbers
8340H	Expedited data; no checksum; transport class 4; 7-bit sequence numbers
8242H	No expedited data; no checksum; transport class 4; 31-bit sequence numbers
8240H	No expedited data; no checksum; transport class 4; 7-bit sequence numbers
8142H	Expedited data; checksum; transport class 4; 31-bit sequence numbers
8140H	Expedited data; checksum; transport class 4; 7-bit sequence numbers
8042H	No expedited data; checksum; transport class 4; 31-bit sequence numbers
8040H	No expedited data; checksum; transport class 4; 7-bit sequence numbers

client_data_buf_addr

An address descriptor that identifies a contiguous 64-byte buffer. If the address is zero, no buffer is allocated and there is no client data sent with the request. Also, no data will be received. To receive any data that may be returned with the request, but not send any data, specify a nonzero address and set `client_data_len` to zero. If the address is nonzero, the Transport service assumes that a 64-byte buffer is allocated. To send client data with the request, the buffer address and length must be nonzero and the data (1 to 32 bytes) must be loaded in the buffer.

client_data_len

The number of bytes of data to send with this request.

Responses

Output Arguments

`iso_reason_code`

The ISO disconnect reason code, if the connection was aborted by the remote Transport service during the connection establishment phase. Otherwise, the value of this argument is 0.

See also: Table 12-3 on page 185

`ack_delay_estimate`

0 is always returned.

`ta_buffer_addr`

The buffer contains returned values that identify the local and remote end nodes of the VC connection. The returned remote address is fully specified.

`negot_options`

The agreed-upon negotiation options using the encoding defined in the input argument description.

`client_data_buf_addr`

If a buffer was allocated, the request block returns in the buffer any data (at most 32 bytes) received from the connection request. The received data overwrites any data in the buffer that was sent in the original request block.

`client_data_len`

The length of any data received in the buffer.

Response Codes

<code>OK_RESPONSE</code>	1H	This is only returned for the <code>AWAIT_CONNECT_REQUEST_TRAN</code> command. The request was accepted. The connection is now established and in the data transfer phase.
<code>OK_DECIDE_REQ_RESP</code>	5H	Returned only for the <code>AWAIT_CONNECT_REQUEST_CLIENT</code> command. The request is acceptable based on addressing, negotiation options and data buffer availability. The request block is returned so that the application can decide whether to accept the connection. The Transport service awaits the application's response.

UNKNOWN_REFERENCE	6H	The CDB corresponding to this reference is not allocated.
OK_CLOSED_RESP	7H	The local client withdrew its willingness to listen for remote connection requests.
ILLEGAL_REQ	0CH	The client specified invalid negotiation options. The connection attempt was aborted.
OK_CONN_REQ_RESP	0DH	The local client withdrew its willingness to listen for remote connection requests. Instead, it is actively requesting a connection with a remote Transport service using this connection.
REM_ABORT	0EH	The connection request was accepted by Transport but the remote Transport service aborted the connection during the connection establishment phase.
LOC_TIMEOUT	10H	This is only returned for the AWAIT_CONNECT_REQUEST_TRAN command. The request was accepted but Transport timed out before completion of the three-way handshake. The connection is aborted.
DUP_REQ	14H	This is a duplicate connection request: Transport is already awaiting a remote request or the connection is already established.
ILLEGAL_ADDRESS	1AH	The client specified invalid TSAP address options or the local TSAP selector or remote TSAP address length exceeds the configuration limits.
NETWORK_ERROR	1CH	A Network layer error was reported at the transport/network interface.

Additional Information

These commands indicate that the Transport service client is willing to consider incoming connection requests from a remote transport service. It is assumed that a local CDB was allocated and a reference was returned to the client as a result of a previous OPEN command. The client specifies that reference in the current command.

A SEND_CONNECT_REQUEST command for a connection will override an existing AWAIT_CONNECT_REQUEST_ command for the same connection, as long as the connection handshake has not begun under the AWAIT_CONNECT_REQUEST_. When this occurs, the AWAIT_CONNECT_REQUEST_ command request block is returned with 0DH (OK_CONN_REQ_RESP) in the response field.

Using a series of these commands, a client can await connection requests. For an incoming connection request, the Transport service scans the CDBs listening for requests using these commands. A request is considered matched to a CDB if the request passes these tests:

- Address match tests
- Negotiation option tests
- Client data buffer availability test

Address Match Test

For the address match test, the remote address specified in AWAIT_CONNECT_REQUEST_ commands may be fully specified, partially specified, or unspecified, with these results:

Fully Specified	Only incoming connection requests from the exact remote TSAP address will be considered. This application is waiting for a request from a specific type of application on a specific node.
Partially Specified	A connection request from only one specific TSAP selector at any remote NSAP address, with an address length not exceeding that specified in the command, will be considered. This application is waiting for a request from a specific type of application on any node.
Unspecified	A connection request from any remote TSAP address will be considered. However, the lengths of the TSAP selector and NSAP address from the remote node cannot exceed the lengths specified in this command. This application is waiting for a request from any application anywhere that knows of the existence of this application.

For each CDB listening for connection requests, the address matching is done with this precedence:

1. A fully specified remote TSAP address
2. A partially specified remote TSAP address
3. An unspecified remote TSAP address

If all three address match attempts fail for one CDB, then that CDB is skipped and the incoming connection request is checked against other CDBs.

A connection request passes the address match test only if all of these conditions are true:

- The `AWAIT_CONNECT_REQUEST_` command is issued prior to receipt of a connection request whose TSAP address satisfies the remaining requirements.
- The lengths of the source NSAP address and TSAP selector in the incoming request do not exceed the corresponding lengths for the remote address in this command's address buffer. If fully specified, the lengths must be equal.
- The source TSAP address in the incoming request matches the remote TSAP address specified in this command's address buffer. If this command's remote TSAP address is unspecified, any incoming NSAP address and TSAP selector pair will match. If this command's remote TSAP address is partially specified, any incoming NSAP address will match, but the incoming source TSAP selector must be an exact match. If this command's remote TSAP address is fully specified, the incoming NSAP address and TSAP selector must be exact matches.
- The length of the destination TSAP selector in the incoming request equals the length of the local TSAP selector in this command's TSAP address buffer.
- The destination TSAP selector in the incoming request matches the local TSAP selector defined in this command's TSAP address buffer.

Negotiation Options Test

For the first CDB that passes the address match test, a check is made for compatible negotiation options as defined by the ISO standard. For incompatible options, the connection request is checked against other CDBs awaiting requests, starting again with the address match test.

Client Data Buffer Availability Test

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

If the incoming request has no data, it is matched to the first CDB that passes the address match and negotiation options tests.

Differences Between the `_CLIENT` and `_TRAN` Commands

If no awaiting CDB is found that matches the incoming connection request, the Transport service rejects the connection request. No request blocks are returned to `ACCEPT_CONNECT_REQUEST_` commands. This permits the Transport service to await further connection requests that may be valid.

Both versions of the `AWAIT_CONNECT_REQUEST_` command operate identically up to the point where a connection request is matched to a CDB.

For a CDB specified by the `AWAIT_CONNECT_REQUEST_TRAN` command, the Transport service itself accepts the request, based only on addressing, negotiation options, and client data buffer availability. The client thus pre-establishes its connection-acceptance criteria with this command. If the incoming request matches a CDB, the connection is immediately accepted for the request. The request block is not returned until completion of the three-way handshake that establishes the connection. Thus, the return of this request block serves as a confirmation of connection establishment. Any client data received with the request is returned in the request block. Delivery of client data is a best-effort attempt and is not guaranteed.

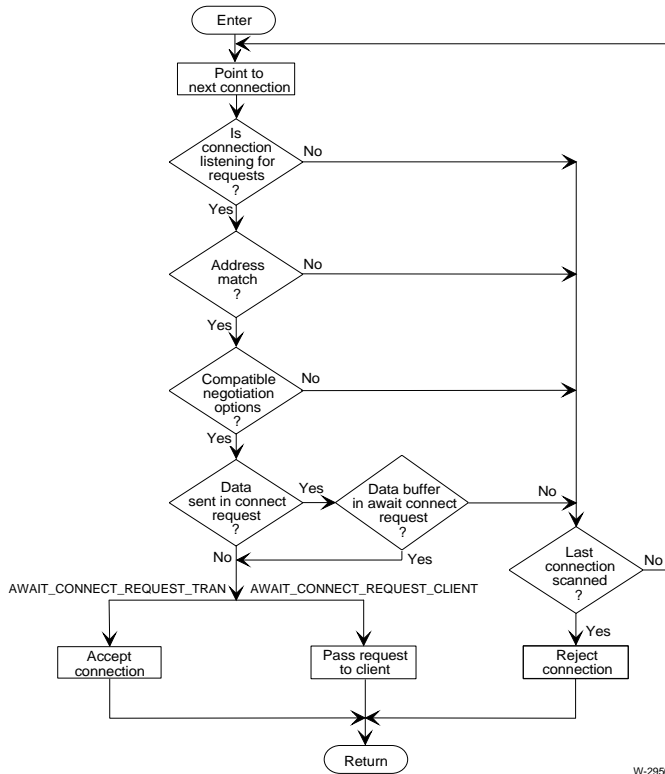
For a CDB specified by the `AWAIT_CONNECT_REQUEST_CLIENT` command, when a connection request matches the CDB, the Transport service returns the request block for further consideration by the application. The request block contains the remote address, negotiation options, and any client data. The Transport service waits for a reply from the application, which is either:

- An `ACCEPT_CONNECT_REQUEST` command to accept the connection
- A `CLOSE` command to reject the connection

Figure 12-2 summarizes the process used by the Transport service to accept connection requests.

AWAIT_CONNECT_REQUEST_TRAN AWAIT_CONNECT_REQUEST_CLIENT

For either version of the command, the application may rescind its willingness to listen for connection requests by issuing a CLOSE command, with a reference specifying the connection. This deletes the CDB and ends the use of the reference.



W-2956

Figure 12-2. Connection Request Consideration Policy

CLOSE

CLOSE requests termination of an existing connection or rejects an incoming connection request.

Request Block

```

typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of vc_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 40H */
    unsigned char       opcode;      /* 0CH */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long       address;     /* input */
    unsigned short      length;     /* input */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON           header;
    unsigned char       iso_reason_code; /* input */
    unsigned char       reserved[15];
    unsigned short      reference;     /* input */
    unsigned char       qos;           /* unused */
    unsigned short      buf_len;      /* unused */
    unsigned char       num_blks;     /* input */
    DATA_BLOCK         data_blk_list[1]; /* [num_blks] */
} VC_RB;

```

Input Arguments

`iso_reason_code`

The encoded ISO standard reason for the close operation. To reject a connect request received from `AWAIT_CONNNECT_REQUEST_CLIENT`, set to 88H.

See also: Table 12-3 on page 185

reference

Identifies the CDB this request applies to.

num_blks

The number of separate buffers that contain optional data to send with the disconnect request. Set to 0 if there is no data to transmit. Each buffer is a block of contiguous memory that is defined by the `data_blk_list[i].address` and `data_blk_list[i].length` arguments.

`data_blk_list[i].address`

The address descriptor for the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the i^{th} buffer. The total length of data in all blocks cannot exceed 64K bytes

See also: Table 12-4 on page 231

Responses

Output Arguments

None

Response Codes

UNKNOWN_REFERENCE	6H	The reference does not correspond to an allocated CDB.
OK_CLOSED_RESP	7H	Confirms disconnection, disconnect collision, or already closing or closed.
BUFFER_TOO_LONG	0AH	A client data length greater than 64 bytes was specified.
OK_REJECT_CONN_RESP	0BH	Successful rejection of a connection request.
LOC_TIMEOUT	10H	Transport service timed out without receiving a confirmation of its disconnect request.

Additional Information

If the connection is already established, this call initiates the ISO transport connection termination procedure. Any normal or expedited data queued for sending will not be sent. The application may send up to 64 bytes of data with the disconnect request.

If the receiver has previously issued an `AWAIT_CLOSE` command, the ISO reason code and any data received with the disconnect request (along with the ISO reason code) will be passed to the buffer allocated with the command. Otherwise, the disconnect request data may be discarded.

The `CLOSE` command is also used to reject a connection request received from an `AWAIT_CONNECT_REQUEST_CLIENT` command. Data passed with the `CLOSE` can be sent to the remote Transport service to explain the reason for the rejection.

A `CLOSE` issued in response to a connection request or issued to abort an already established connection deletes the CDB. Any posted receive buffers (normal or expedited) or queued send requests (normal or expedited) will be returned to the application. An `AWAIT_CLOSE` command will also be returned. The `CLOSE` request block is always the final request block returned.

If the connection is aborted by a remote Transport service, any posted receive buffers, queued send requests, or `AWAIT_CLOSE` request blocks are returned to the local application and the CDB is deleted. If there are no queued request blocks to report the remote abort, the CDB is not deleted, but is marked closed. The next time the application tries to issue a request block to that CDB, the request block is returned with a `REM_ABORT` response (code `0EH`) and the CDB is deleted. Any further requests on that connection generate an `UNKNOWN_REFERENCE` response.

OPEN

OPEN allocates a connection database (CDB) as the first step in establishing a VC. The returned value identifies the VC in subsequent commands.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of open_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 40H */
    unsigned char       opcode;      /* 0H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct open_rb {
    RB_COMMON            header;
    unsigned short      reference;   /* output */
} OPEN_RB;
```

Input Arguments

None

Responses

Output Arguments

reference

A value identifying the CDB allocated by this command. Store this value for use in other Transport service commands.

Response Codes

OK_RESPONSE	1H	The CDB was allocated and the reference returned.
NO_RESOURCES	4H	Could not allocate any more CDBs. The reference is returned as 0.

Additional Information

This is the first command to issue whenever you open a new VC (connection); the VC requires memory for the connection database (CDB). All CDBs reside in memory on the same board that contains the communications software. There is a preconfigured maximum number of CDBs, and therefore a maximum number of VCs.

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

The Transport service client uses the connection by referencing the CDB through a 16-bit number called a connection reference. The reference is returned to the Transport service client when the CDB is allocated by this command. The Transport service returns the reference to the client in the reference field of the open request block. The application then refers to the connection in other Transport service commands (e.g., data transfers) by supplying the connection reference number as an input argument.

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

RECEIVE_ANY

RECEIVE_ANY is similar to the RECEIVE_DATA command; this command posts a buffer to store data received using the transport normal delivery service. Unlike the RECEIVE_DATA command, this command posts a receive buffer that can be used by any CDB in a list of connection references. A RECEIVE_ANY buffer stores received data for which there is no normal RECEIVE_DATA buffer posted. This service is governed by the regular transport flow control mechanisms.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_ext_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 16H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;      /* input */
    unsigned short    length;      /* input */
} DATA_BLOCK;

typedef struct vc_ext_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[15];
    unsigned short    reference;     /* output */
    unsigned char     qos;           /* input */
    unsigned short    buf_len;       /* output */
    unsigned char     num_blks;      /* input */
    DATA_BLOCK       block[1];
    unsigned short    ref_list_count; /* input */
    unsigned long     ref_list_ptr;  /* input */
}; VC_RB
```

Input Arguments

qos Set to zero.

`num_blks`

Only one buffer is posted with this command, so the value of this argument must be 1.

`block[0].address`

The address for the start of the buffer.

`block[0].length`

The length of the buffer, which cannot exceed 64K bytes.

`ref_list_count`

The number of connection references in the reference list, up to 20. If the local Transport service receives data in a CDB for which no specific normal RECEIVE_DATA buffer has been posted, the reference list for the first available RECEIVE_ANY buffer is checked for a reference matching the reference for the received data. If a matching reference is found, the received data is placed in that RECEIVE_ANY buffer. If no matching reference is found, the reference list for the next RECEIVE_ANY buffer in the queue is checked.

`ref_list_ptr`

The address for the start of the reference list, which is an array of unsigned short values.

Responses

Output Arguments

`iso_reason_code`

The ISO disconnect reason code, if the connection was aborted by the remote Transport service during the connection establishment phase. Otherwise, the value of this argument is 0.

See also: Table 12-3 on page 185

`reference`

A value identifying the CDB that used the buffer.

`buf_len`

The total length of the data received in the buffer posted by this command.

`block[0].address`

The address descriptor for the start of the buffer.

`block[0].length`

The length of the data in the buffer.

Response Codes

OK_RESPONSE

1H

All the buffers pointed to by the request block are filled with data and no EOM was signaled.

OK_EOM_RESP	3H	Transport signaled an EOM. The data in the buffer constitutes the end of a TSDU.
NO_RESOURCES	4H	The CDB normal receive queue is full. No more normal receive buffers can be posted until some already posted are returned.
UNKNOWN_REFERENCE	6H	The reference does not correspond to any allocated CDB.
OK_CLOSED_RESP	7H	The local client aborted the connection or the connection was closing at the client's request when the buffer was posted.
REM_ABORT	0EH	The remote Transport service aborted the connection.
LOC_TIMEOUT	10H	The buffer was returned due to a connection timeout abort.
BAD_REF_COUNT	24H	Zero was specified for the ref_list_count field.

Additional Information

The maximum length of a RECEIVE_ANY buffer is 64K bytes. Depending on Transport Layer configuration, buffers may be posted prior to establishment of the connection.

RECEIVE_DATA

RECEIVE_DATA posts one or more receive buffers for a specific connection. The buffers store data received from the Transport normal delivery service. This service is governed by the regular transport flow control mechanisms.

Request Block

```

typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of vc_rb */
    unsigned char       user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 40H */
    unsigned char       opcode;      /* 7H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long       address;     /* in/out */
    unsigned short      length;     /* in/out */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON           header;
    unsigned char       iso_reason_code; /* output */
    unsigned char       reserved[15];
    unsigned short      reference;     /* in/out */
    unsigned char       qos;           /* unused */
    unsigned short      buf_len;      /* output */
    unsigned char       num_blks;     /* input */
    DATA_BLOCK         data_blk_list[1]; /* [num_blks] */
} VC_RB;

```

Input Arguments

`reference`

Identifies the CDB for which the receive buffer is being posted.

`num_blks`

The number of separate buffers to receive data. Each buffer is a block of contiguous memory that is defined by the `data_blk_list[i].address` and `data_blk_list[i].length` arguments.

`data_blk_list[i].address`

The address pointing to the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the i^{th} buffer. The total length of data in all blocks cannot exceed 64K bytes.

Responses

Output Arguments

`iso_reason_code`

The ISO disconnect reason code, if the connection was aborted by the remote Transport service during the connection establishment phase. Otherwise, the value of this argument is 0.

See also: Table 12-3 on page 185

`reference`

Identifies the CDB that used the buffer.

`buf_len`

The total length of the data received in the buffer(s) posted by this command.

`data_blk_list[i].address`

The address for the start of the i^{th} buffer.

`data_blk_list[i].length`

This is meaningful only for the last posted buffer that received data. This value is the length of the data in that buffer.

Response Codes

OK_RESPONSE	1H	All the buffers pointed to by the request block are filled with data and no EOM was signaled.
OK_EOM_RESP	3H	Transport signaled an EOM. The data in the buffers constitutes the end of a TSDU.
NO_RESOURCES	4H	The CDB normal receive queue is full. No more normal receive buffers can be posted until some already posted are returned.
UNKNOWN_REFERENCE	6H	The reference does not correspond to any allocated CDB.
OK_CLOSED_RESP	7H	The local client aborted the connection or the connection was closing on client request when the buffer was posted.
OK_WITHDRAW_RESP	9H	Zero or more normal receive buffers were withdrawn from Transport service.
REM_ABORT	0EH	The remote Transport service aborted the connection.
LOC_TIMEOUT	10H	The buffer was returned due to a connection timeout abort.

Additional Information

The total length of all receive buffers pointed to by a single RECEIVE_DATA request block must not exceed 64K bytes. Depending on Transport Layer configuration, buffers may be posted prior to establishment of the connection.

RECEIVE_DATAGRAM

RECEIVE_DATAGRAM posts a receive buffer on behalf of a TSAP to receive data from a transport datagram. The datagram buffer queues are maintained separately from the VC buffer queues.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of datagram_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 41H */
    unsigned char       opcode;      /* 12H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long       address;      /* in/out */
    unsigned short      length;      /* in/out */
} DATA_BLOCK;

typedef struct datagram_rb {
    RB_COMMON           header;
    unsigned char       reserved[4];
    unsigned long       ta_buffer_addr; /* in/out */
    unsigned char       qos;          /* input */
    unsigned short      buf_len;      /* output */
    unsigned char       num_blks;     /* input */
    DATA_BLOCK         data_blk_list[1]; /* [num_blks] */
} DATAGRAM_RB;
```

Input Arguments

`ta_buffer_addr`

An address pointing to a TSAP address buffer that specifies the local and remote end nodes for a datagram transfer. The local TSAP selector must be loaded into the buffer. The length of the local TSAP selector and NSAP selector must not exceed the limit specified in the system configuration, otherwise an addressing error occurs. The local TSAP selector value (which must be nonzero) specifies the TSAP that posts the buffer. The buffer is placed in a queue reserved only for that TSAP selector. Any datagrams received with a destination TSAP selector matching the TSAP selector of the queue can pass its data to the buffer.

The remote TSAP is irrelevant and is ignored. The remote address of the data link entity associated with the remote Transport service, and the remote NSAP selector and TSAP selector fields are not input parameters. However, the fields must be reserved to the proper length to buffer the source TSAP address of a received datagram.

See also: TSAP address buffer structure, page 180

`qos` If the low order bit of the high order nibble is set, the Transport service verifies the checksum (if present) of the incoming datagram.

`num_blks`

The number of separate buffers to be received. Each buffer is a block of contiguous memory defined by the `data_blk_list[i].address` and `data_blk_list[i].length` arguments.

`data_blk_list[i].address`

The address pointing to the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the i^{th} buffer, which must be nonzero. The total length of data in all blocks cannot exceed the maximum NSDU size minus a small overhead for the transport datagram header.

Responses

Output Arguments

`ta_buffer_addr`

An address pointing to the returned buffer containing the remote NSAP address and TSAP selector that specify the remote address of the received datagram.

See also: TSAP address buffer structure, page 180

`buf_len`

The total length of the data received in the buffers posted by this command.

`data_blk_list[i].address`

The address descriptor for the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the data in the last posted buffer to receive data. This value is only meaningful for that buffer.

Response Codes

<code>OK_RESPONSE</code>	1H	The buffers pointed to by the request block are completely filled with data.
<code>OK_EOM_RESP</code>	3H	The buffer contains data to the end of the datagram. A request block can return data from no more than one transport datagram.
<code>NO_RESOURCES</code>	4H	There are no more resources to manage the buffers posted for the TSAP.
<code>ILLEGAL_ADDRESS</code>	1AH	An addressing error was detected.

RECEIVE_EXPEDITED_DATA

RECEIVE_EXPEDITED_DATA posts expedited receive buffers for a specific connection. The buffers store data received from the transport expedited data delivery service. Expedited data bypasses the normal transport flow control mechanisms.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 0AH */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;     /* in/out */
    unsigned short    length;     /* in/out */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[15];
    unsigned short    reference;     /* in/out */
    unsigned char     qos;           /* unused */
    unsigned short    buf_len;      /* unused */
    unsigned char     num_blks;     /* input */
    DATA_BLOCK       data_blk_list[1];
} VC_RB;
```

Input Arguments

reference

Identifies the CDB for which the expedited receive buffer is being posted.

num_blks

Must be set to 1 because the expedited data will be sent in a single TPDU.

data_blk_list[0].address

The address descriptor for the start of the buffer.

data_blk_list[0].length

Set to 16: the maximum amount of expedited data that can be sent is 16 bytes.

Responses**Output Arguments**

iso_reason_code

Set to 14 if the request block was returned due to a remote abort; otherwise set to 0.

reference

Identifies the CDB that used the buffer.

data_blk_list[0].address

The address for the start of the buffer containing the received expedited data.

data_blk_list[0].length

The length of the received data.

Response Codes

OK_EOM_RESP	3H	The buffer is returned with data from a single expedited TPDU.
NO_RESOURCES	4H	The CDB expedited receive queue is full. No more expedited receiver buffers can be posted until some that are posted are returned.
UNKNOWN_REFERENCE	6H	The reference does not correspond to an allocated CDB.
OK_CLOSED_RESP	7H	The local client aborted the connection.
BUFFER_TOO_SHORT	8H	The length of the first buffer block posted with the request is less than 16.
OK_WITHDRAW_RESP	9H	Zero or more expedited receive buffers were withdrawn from Transport service.
ILLEGAL_REQ	0CH	Expedited service not available.

REM_ABORT	0EH	The remote Transport service aborted the connection.
LOC_TIMEOUT	10H	The connection terminated on a timeout.

Additional Information

Each receive buffer can hold data from only one expedited TPDU. Data from two or more such TPDU's are not combined into one buffer even if the data would fit. The buffers for each request must be at least 16 bytes long to accommodate the longest expedited data TPDU that can be received.

Depending on Transport Layer configuration, expedited receive buffers may be posted prior to establishment of the connection.

The queues of expedited receive buffers are maintained separately from the queues of normal receive buffers. More than one expedited data buffer may be posted at a time, but only one may be sent or received at a time.

SEND_CONNECT_REQUEST

SEND_CONNECT_REQUEST requests a connection to a fully specified remote TSAP address. This performs an active open of the VC. The CDB must already be allocated with an OPEN command. Data may be sent and received in the client data buffer.

Request Block

```

typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of conn_req_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 40H */
    unsigned char       opcode;      /* 1H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct conn_req_rb {
    RB_COMMON           header;
    unsigned char       iso_reason_code; /* output */
    unsigned char       reserved[4];
    unsigned short      ack_delay_estimate; /* output */
    unsigned long       ta_buffer_addr;    /* input */
    unsigned short      persistence_count; /* input */
    unsigned short      abort_timeout;     /* input */
    unsigned short      reference;         /* input */
    unsigned char       qos;               /* input */
    unsigned short      negot_options;     /* input */
    unsigned long       client_data_buf_addr; /* in/out */
    unsigned char       client_data_len;   /* in/out */
} CONN_REQ_RB;

```

Input Arguments

`ta_buffer_addr`

An address pointing to a TSAP address buffer that specifies the local and remote end nodes of a VC connection. The local TSAP selector must be fully specified, i.e., must have either zero length or nonzero length and nonzero value. The remote NSAP address must be fully specified. The remote TSAP selector must be fully specified. The length of the remote net address and local or remote TSAP selectors must not exceed the limits specified in the system configuration, otherwise an addressing error will occur. Multiple connections to, or from, a single TSAP address can be requested.

See also: TSAP address buffer structure, page 180

`persistence_count`

The number of times to retry an active connection attempt upon connection refusal, before giving up. Connection refusal means that the remote system refuses the connection, not that it failed to respond to the connection attempt. A connection refusal typically occurs when the remote system is not listening (it hasn't executed a passive open). Values may be:

Value	Meaning
0	The configured value will be used
0FFFFH	Retry forever
1 to 0FFFEH	This value will be used as the persistence count

`abort_timeout`

The retransmission timeout before aborting the connection, in 51-millisecond time units. Values may be the same as for `persistence_count`.

This specifies how long the Transport service will continue to transmit without receiving a response. This applies to both the connection establishment and data transfer phases. During the connection establishment phase, this controls how long a connection request will be retransmitted when there is no response. During the data transfer phase, this controls how long data is retransmitted when there is no ACK. This does not apply to the connection termination phase; the timeout for connection termination is a Transport service configuration parameter.

`reference`

Identifies the CDB this request applies to.

`qos` Quality of service: the only possible parameter is the transmit priority, for underlying subnetworks that support it. This is a value in the range 0 to 15, where 0 is the highest priority. For iNA 960 Data Link 802.3 subnets, set `qos` to zero; transmit priority is not supported.

`negot_options`

Specifies various classes of service and additional options requested for negotiation on this connection. If `negot_options` is zero, default options are used, as specified by the `def_negot_options` configuration parameter. Otherwise, break the value into four nibbles and specify options, where nibble 1 is the least significant:

Nibble	Value	Meaning
1	0	use 7-bit sequence numbers
	2	use 31-bit sequence numbers
2	4	class four service
3	0	no expedited service, do checksums
	1	expedited service, do checksums
	2	no expedited service, no checksums
4	3	expedited service, no checksums
	8	client specified (nondefault) negotiation options

These are valid values for this argument:

Value	Meaning
8342H	Expedited data; no checksum; transport class 4; 31-bit sequence numbers
8340H	Expedited data; no checksum; transport class 4; 7-bit sequence numbers
8242H	No expedited data; no checksum; transport class 4; 31-bit sequence numbers
8240H	No expedited data; no checksum; transport class 4; 7-bit sequence numbers
8142H	Expedited data; checksum; transport class 4; 31-bit sequence numbers
8140H	Expedited data; checksum; transport class 4; 7-bit sequence numbers
8042H	No expedited data; checksum; transport class 4; 31-bit sequence numbers
8040H	No expedited data; checksum; transport class 4; 7-bit sequence numbers

`client_data_buf_addr`

An address descriptor that identifies a contiguous 64-byte buffer. If this value is zero, no buffer is allocated and there is no client data sent with the request. Also, no data will be received. To receive any data that may be returned with the response but not send any data, specify an address and set `client_data_len` to zero. If the address is not 0, a 64-byte buffer is assumed to be allocated. To send client data with the request, the data (0 to 32 bytes) must be loaded in the buffer.

`client_data_len`

The length of the client data in the client data buffer. The range of valid values is 0 to 32. If the length is 0, no data will be sent.

Responses

Output Arguments

`iso_reason_code`

82H if the connection negotiation failed. This indicates the request was accepted by the remote Transport service, but the local Transport service aborted the connection because the options the remote Transport service negotiated were unacceptable.

If the connection was rejected by the remote Transport service, `iso_reason_code` indicates the reason for the rejection. Otherwise, `iso_reason_code` is 0.

See also: Table 12-3 on page 185

`ack_delay_estimate`

0 is always returned.

`client_data_buf_addr`

If the connection attempt was successful and a buffer was allocated, the request block will return in the referenced buffer any data (at most 32 bytes) contained in the connection confirmation received from the remote Transport service. If the connection attempt was rejected by the remote Transport service and the local Transport service gives up, the request block returns up to 64 bytes of any data contained in the disconnect request from the remote Transport service. The received data overwrites any data in the buffer that was sent in the original connection request.

`client_data_len`

The length of any data received in response to the connection request.

Response Codes

OK_RESPONSE	1H	The request was accepted by the remote Transport service and the connection is now established in the data transfer phase.
UNKNOWN_REFERENCE	6H	The client-specified reference does not correspond to an allocated CDB.
OK_CLOSED_RESP	7H	The local client aborted the connection while the connection request was outstanding.
BUFFER_TOO_LONG	0AH	The <code>client_data_len</code> field was greater than 32 bytes. The connection attempt was aborted.

ILLEGAL_REQ	0CH	Invalid negotiation options were specified. The connection attempt was aborted.
LOC_TIMEOUT	10H	The request was unanswered and the retransmission timer timed out which aborted the connection attempt.
DUP_REQ	14H	This is a duplicate connection request; a request was already in progress for this reference or the connection was already established.
CONN_REJECT	16H	The connection attempt was rejected by the remote Transport service and the local Transport service gave up after the persistence count expired.
NEGOT_FAILED	18H	The request was accepted by the remote Transport service but the local Transport service aborted the connection because of a negotiation failure at the local end.
ILLEGAL_ADDRESS	1AH	Invalid local or remote TSAP address specified, or the NSAP address length exceeds max_net_addr_len, or the local or remote TSAP selector exceeds the max_tsap_id_len parameter that was specified at system configuration.
NETWORK_ERROR	1CH	A Network layer error at the transport/network interface. This usually means that the specified NSAP address is unreachable.

Additional Information

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

The SEND_CONNECT_REQUEST command can actively request a connection either immediately after a previous OPEN was issued, or after one of the connection listening commands, AWAIT_CONNECT_REQUEST_TRAN or AWAIT_CONNECT_REQUEST_CLIENT, is issued. In the latter case, the current command is valid only if the connection is still listening for a connection request and has not started the connection handshake with the remote Transport service. When a SEND_CONNECT_REQUEST command overrides one of the AWAIT_CONNECT_REQUEST_ commands, the request block for the overridden command is returned to the client with a response code of 0DH.

The client may ask that the Transport service request the connection a specified number of times in spite of a rejection by the remote transport service. This retry count is the `persistence_count` specified in the request block by the client. When the number of retries exceeds the count, the local Transport service gives up and indicates connection rejection to the client. Persistence is invoked only if the ISO reason code returned in the remote Transport service's rejection TPDU is one of these:

Value	Meaning
0	Unspecified
2	No one listening at remote TSAP selector
81H	TSAP congestion
88H	Connection refused

Persistence is not applied in the case where the local client decides to close the connection while transport is requesting the connection.

`Abort_timeout` is used if there is no reply at all to the connection request.

This request block is returned to the client either upon detection of an error, upon connection establishment, or upon rejection when local Transport service gives up. Thus, the receipt of this request block by the client serves as a connection confirmation or failure indication to the client.

SEND_DATA/SEND_EOM_DATA

SEND_DATA and SEND_EOM_DATA request transmission of the data in the buffers using the normal delivery service of the specified VC connection. The normal delivery service uses the regular flow control mechanisms. The SEND_EOM_DATA command specifies that the end of data in the buffers marks the end of the transport service data unit (TSDU).

Request Block

```

typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 5H SEND_DATA
                                   6H SEND_EOM_DATA */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;      /* input */
    unsigned short    length;      /* input */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[15];
    unsigned short    reference;     /* input */
    unsigned char     qos;           /* unused */
    unsigned short    buf_len;      /* unused */
    unsigned char     num_blks;     /* input */
    DATA_BLOCK       data_blk_list[1]; /* [num_blks] */
} VC_RB;

```

Input Arguments

`reference`

Identifies the CDB this request applies to.

`num_blks`

The number of separate buffers, where each buffer is a block of contiguous memory containing data to send. Each buffer is defined by the

`data_blk_list[i].address` and `data_blk_list[i].length` arguments.

`data_blk_list[i].address`

The address pointing to the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the data in the i^{th} buffer. The total length of data in all blocks cannot exceed 64K bytes.

See also: Table 12-4 on page 231

Responses

Output Arguments

`iso_reason_code`

The ISO disconnect reason code, if the connection was aborted by the remote Transport service during the connection establishment phase. Otherwise, the value of this argument is 0.

See also: Table 12-3 on page 185

Response Codes

<code>OK_RESPONSE</code>	1H	All the buffers in the request have been successfully transmitted and acknowledged by the remote Transport service.
<code>NO_RESOURCES</code>	4H	The CDB send queue is full. No more request blocks can be queued until some already queued SEND request blocks are returned.
<code>UNKNOWN_REFERENCE</code>	6H	The CDB corresponding to this reference is not allocated.
<code>OK_CLOSED_RESP</code>	7H	The local client aborted the connection and the queued request block is returned without transmitting its data.
<code>ILLEGAL_REQ</code>	0CH	The connection was closing or was already closed.

REM_ABORT	0EH	The remote Transport service aborted the connection.
LOC_TIMEOUT	10H	The local Transport service timed out waiting for a PDU acknowledgement. If this error occurs, the local Transport service disconnects the connection.

Additional Information

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

The sum of the lengths of all buffers pointed to by a single SEND_DATA or SEND_EOM_DATA request block is limited to a maximum value that depends on the sequence number format, maximum TPDU size, and maximum NSDU size. The maximum NSDU size is determined only at run time. Table 12-4 shows the maximum total buffer length for various maximum TPDU sizes that can be negotiated. The `max_tpd_size` value is a power of 2 ($2^7 = 128$ bytes).

Table 12-4. Maximum Total Buffer Lengths

Negotiated <code>max_tpd_size</code>	Sequence Number Format	
	7-Bit	31-Bit
7 (128 bytes)	15K*	64K
8 (256 bytes)	62K*	64K
9 (512 bytes)	64K*	64K
10 (1024 bytes)	64K	64K
11 (2048 bytes)	64K	64K

* For maximum TPDU size values of 7, 8, or 9 with 7-bit sequence numbering, the values shown in Table 12-4 are approximate. The actual values depend on the maximum NSDU size determined by the Transport Layer at run time.

An application can make a SEND_DATA request (depending on the Transport Layer configuration) anytime after the initial OPEN command is issued. The Transport service accepts SEND_DATA requests even if the connection has not yet entered the *established* state. When a connection is established, the Transport service transmits the corresponding transmit buffers in the order in which they are queued. Since the Transport service always attempts to send full TPDU's, it copies information from transmit buffers into the TPDU without concern for the buffer or block boundaries. It never copies information from more than one request block into the same TPDU. In addition, the Transport service guarantees that an EOM not only indicates the end of a message, but also the end of a TPDU.

The remote receive buffer sizes need not match the transmit buffer sizes; data is delivered as long as there is any receive buffer space at the remote node.

SEND_DATAGRAM

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

Request Block

```

typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of datagram_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 41H */
    unsigned char       opcode;      /* 11H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long       address;     /* input */
    unsigned short      length;     /* input */
} DATA_BLOCK;

typedef struct datagram_rb {
    RB_COMMON           header;
    unsigned char       reserved[4];
    unsigned long       ta_buffer_addr; /* input */
    unsigned char       qos;          /* input */
    unsigned short      buf_len;     /* unused */
    unsigned char       num_blks;    /* input */
    DATA_BLOCK         data_blk_list[1]; /* [num_blks] */
} DATAGRAM_RB;

```

Input Arguments

`ta_buffer_addr`

An address pointing to a TSAP address buffer that specifies the local and remote end nodes for a datagram transfer. This buffer must be loaded with addressing information specifying the local (source) TSAP selector, the remote NSAP address, and remote (destination) TSAP selector of the datagram. The TSAP address must be fully specified. The lengths of the remote net address and local or remote TSAP selectors must not exceed the limits specified in the system configuration, otherwise an addressing error occurs.

See also: TSAP address buffer structure, page 180

`qos` The low order nibble specifies the priority class used by underlying subnets that support it, (IEEE 802.4 token bus). The range is 0-15, with 0 being the highest priority. For iNA 960 Data Link 802.3 subnets, set this nibble to zero; transmit priority is not supported.

The low order bit of the high order nibble specifies whether to do a checksum on the datagram (1 = do checksum). The next higher bit of the high order nibble specifies whether to query the Network service for the maximum NSDU size or to use the default value (1 = query).

`num_blks`

The number of separate buffers to send. Each buffer is a block of contiguous memory that is defined by the `data_blk_list[i].address` and `data_blk_list[i].length` arguments.

`data_blk_list[i].address`

The address descriptor for the start of the i^{th} buffer.

`data_blk_list[i].length`

The length of the i^{th} buffer. The length must be nonzero. The total length of data in all blocks cannot exceed the maximum NSDU size minus a small overhead for the transport datagram header.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	1H	The data has been queued for transmission by the Network Layer.
BUFFER_TOO_SHORT	8H	The buffer length (<code>data_blk_list[i].length</code>) was set to 0.
BUFFER_TOO_LONG	0AH	The data length exceeds the maximum NSDU size.
ILLEGAL_ADDRESS	1AH	The local TSAP selector or remote TSAP address exceeds the configuration limits, or an address error was detected by the underlying Network Layer.

Additional Information

Transport datagram service does not provide a fragmentation/reassembly capability. Therefore, the length of the data cannot exceed the maximum network service data unit (NSDU) size provided by the underlying service. If the Network Layer does not provide a segmentation/reassembly service, the NSDU size is bounded by subnet data length restrictions. If the Network Layer does provide segmentation/reassembly capabilities, the NSDU size may be larger than the size imposed by subnet data length restrictions. In any implementation, the maximum NSDU size is determined by the Network Layer configuration.

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

SEND_EXPEDITED_DATA

SEND_EXPEDITED_DATA requests transmission of up to 16 bytes of data in the buffer using the expedited delivery service of the specified connection. More than one expedited data buffer may be posted at a time, but only one may be sent at a time.

Request Block

```

typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 9H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;     /* input */
    unsigned short    length;     /* input */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[15];
    unsigned short    reference;     /* input */
    unsigned char     qos;           /* unused */
    unsigned short    buf_len;      /* unused */
    unsigned char     num_blks;     /* input */
    DATA_BLOCK       data_blk_list[1]; /* [num_blks] */
} VC_RB;

```

Input Arguments

`reference`

Identifies the CDB this request applies to.

`num_blks`

Set to 1; only one buffer can be sent with this command.

`data_blk_list[0].address`

The address descriptor for the start of the buffer.

`data_blk_list[0].length`

The length of the data in the buffer. The length must be greater than 0 and less than or equal to 16.

Responses

Output Arguments

`iso_reason_code`

14 if the request block was returned due to a remote abort; otherwise set to 0.

Response Codes

OK_RESPONSE	1H	The expedited data in the buffer was acknowledged.
NO_RESOURCES	4H	The CDB expedited send queue is full. No more expedited send request blocks can be queued at this time until some already queued expedited send request blocks are returned.
UNKNOWN_REFERENCE	6H	The specified reference does not correspond to an allocated CDB.
OK_CLOSED_RESP	7H	The local client aborted the connection.
BUFFER_TOO_SHORT	8H	The buffer is empty. Either <code>num_blks</code> is 0 or the block length is 0.
BUFFER_TOO_LONG	0AH	A data length greater than 16 bytes was specified, or <code>num_blks</code> is greater than 1. The transmission was aborted.
ILLEGAL_REQ	0CH	Either the service to transmit the expedited data is not available for this connection, or the connection was closing or was already closed.

REM_ABORT	0EH	The remote Transport service aborted the connection.
LOC_TIMEOUT	10H	Transport timed out without receiving expedited acknowledgement of the data.

Additional Information

With this service, the expedited data is transmitted immediately and is guaranteed to arrive before any data currently in the process of being transmitted. Expedited data transmission is not subject to flow control; it jumps the flow control queue.

In preconfigured versions of iNA 960, buffers may be posted prior to establishment of the connection. This is dependent on Transport Layer configuration.

STATUS

STATUS queries for information about the VC services provided by the Transport Layer, and if requested, for information pertaining to a specific VC connection. The status information is returned immediately.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;     /* 0EH */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;     /* in/out */
    unsigned short    length;     /* in/out */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* output */
    unsigned char     reserved[15];
    unsigned short    reference;     /* input */
    unsigned char     qos;           /* unused */
    unsigned short    buf_len;      /* output */
    unsigned char     num_blks;     /* input */
    DATA_BLOCK       data_blk_list[1];
} VC_RB;
```

Input Arguments

`reference`

Identifies the CDB for which status is being requested. If this value is zero, only connection-independent status information is returned. If this value is nonzero, both connection-independent status and information pertaining to the specified connection is returned.

`num_blks`

Set to 1; only one buffer is posted with this command.

`data_blk_list[0].address`

The address descriptor for the start of the status buffer.

`data_blk_list[0].length`

The length of the buffer, which must be enough to hold all the returned information.

For connection-independent status (`reference = 0`), the length is 48 bytes plus the configured size of a transport address (`ta_buffer_size` is a Transport Layer configuration parameter).

For complete status (`reference` is nonzero), the length is 144 bytes plus the value of `ta_buffer_size`.

Responses

Output Arguments

`iso_reason_code`

The ISO reason code received in the disconnect request.

See also: Table 12-3 on page 185

`buf_len`

The length of the status data being returned.

`data_blk_list[0].address`

The address descriptor for the start of the status buffer. The buffer contains connection-independent parameters at the start of the buffer. If connection-dependent status was requested, the connection-dependent fields follow the connection-independent ones. In multi-byte fields, the least significant byte appears first in the buffer. The parameter fields are defined later in this description.

`data_blk_list[0].length`

The length of data returned in the buffer.

Response Codes

OK_RESPONSE	1H	The buffer contains the requested status information or no buffer was posted.
UNKNOWN_REFERENCE	6H	The specified (nonzero) reference does not correspond to an allocated CDB.
OK_CLOSED_RESP	7H	The referenced connection was closed.
BUFFER_TOO_SHORT	8H	The allocated buffer was too short for the requested status information.
REM_ABORT	0EH	The connection was terminated by a remote disconnect request.
LOC_TIMEOUT	10H	The connection terminated on a timeout.

Additional Information

If no buffer is posted to receive the status information (`num_blks = 0`, `data_blk_list[0].address = 0`, or `data_blk_list[0].length = 0`) and the referenced connection (if any) is not closed, the command request block is returned with an `OK_RESPONSE`. If the referenced connection is closed, the command request block is returned with an `OK_CLOSED_RESP` and the disconnect reason code.

The information returned by the command is shown below, divided into structures of connection-independent and connection-dependent fields. Some of the returned values are the same as Transport Layer objects that can be read or set with NMF commands.

See also: `READ_OBJECT` and `SET_OBJECT` commands, Chapter 14, Object definitions, Appendix C

Connection-Independent Status Parameters

```
struct con_independent_status {
    unsigned short    cur_max_cdb;
    unsigned short    num_cdb;
    unsigned short    def_persist;
    unsigned short    def_abort_to;
    unsigned short    def_negot_options;
    unsigned char     max_tpdu_size;
    unsigned char     reserved;
    unsigned short    closing_abort_to;
    unsigned long     def_retran_to;
    unsigned long     min_retrans_time;
    unsigned short    max_window_size_n;
    unsigned short    max_window_size_e;
    unsigned short    min_credit;
    unsigned char     reserved[20];
};
```

`cur_max_cdb`

The total number of connection databases (CDBs) configured into the iNA 960 Transport Layer (same as object 4003H).

`num_cdb`

The number of CDBs currently allocated (same as object 4006H).

`def_persist`

Default `persistence_count` used if the application specifies a persistence value of 0 in a `SEND_CONNECT_REQUEST` request block.

`def_abort_to`

Default abort timeout value in 51-millisecond units, used if the application specifies an `abort_timeout` value of 0 in a `SEND_CONNECT_REQUEST` or `AWAIT_CONNECT_REQUEST_TRAN` or `AWAIT_CONNECT_REQUEST_CLIENT` request block.

`def_negot_options`

Default negotiation options requested if the application specifies a `negot_options` value of 0 in the request block of a `SEND_CONNECT_REQUEST`, `AWAIT_CONNECT_REQUEST_TRAN`, or `AWAIT_CONNECT_REQUEST_CLIENT` command. These values are valid:

Value	Meaning
8342H	Expedited data; no checksum; transport class 4; 31-bit sequence
8340H	Expedited data; no checksum; transport class 4; 7-bit sequence
8242H	No expedited data; no checksum; transport class 4; 31-bit sequence
8240H	No expedited data; no checksum; transport class 4; 7-bit sequence
8142H	Expedited data; checksum; transport class 4; 31-bit sequence
8140H	Expedited data; checksum; transport class 4; 7-bit sequence
8042H	No expedited data; checksum; transport class 4; 31-bit sequence
8040H	No expedited data; checksum; transport class 4; 7-bit sequence

`max_tpdu_size`

Maximum TPDU size requested at the local node, specified as the exponent in a power of 2 (range of 2 to 13).

`closing_abort_to`

The abort timeout in 51-millisecond units, used when closing a connection.

`def_retran_to`

The initial value of the default retransmit timeout in 51-millisecond units, used during the connection establishment phase.

`min_retrans_time`

The minimum retransmission timeout in 51-millisecond units. This is a lower bound on the retransmission timeout used during the data transfer and connection termination phases.

`max_window_size_n`

The maximum flow control window size that can be reported to a remote node if normal (7-bit) sequence numbers are used for a connection.

`max_window_size_e`

The maximum flow control window size that can be reported to a remote node if extended (31-bit) sequence numbers are used for a connection.

`min_credit`

The minimum flow control credit that can be reported to a remote node. If 0, the window may be closed. If nonzero, the window cannot close.

Connection-Dependent Status Parameters

```
struct con_dependent_status {
    unsigned char    state;
    unsigned char    reserved;
    unsigned short   loc_ref;
    unsigned short   rem_ref;
    unsigned short   persist;
    unsigned short   abort_to_hi;
    unsigned long    retran_to_dw;
    unsigned short   reserved;
    unsigned short   pending_rec_data;
    unsigned short   rcv_buf_rej_cnt;
    unsigned char    cbtq_buf_cnt;
    unsigned char    pcbq_buf_cnt;
    unsigned char    exp_cbtq_buf_cnt;
    unsigned char    exp_pcbq_buf_cnt;
    unsigned char    close_buf_cnt;
    unsigned char    reserved;
    unsigned short   loc_credit;
    unsigned short   rem_credit;
    unsigned long    loc_ack_no;
    unsigned long    rem_ack_no;
    unsigned long    next_transmit;
    unsigned long    highest_sent;
    unsigned long    loc_exp_ack_no;
    unsigned long    rem_exp_ack_no;
    unsigned short   loc_subseq_no;
    unsigned short   rem_subseq_no;
    unsigned short   client_options;
    unsigned char    class_options;
    unsigned char    options;
    unsigned char    conn_max_tpdu_size;
    unsigned char    qos;
    unsigned short   max_tpdu_data_len;
    unsigned short   reserved;
    unsigned short   max_nsdu_size;
    unsigned char    reserved[26];
    unsigned char    cdb_ta_buffer[ta_buffer_len];
};
```

`state`

The current state of the connection. Only the low order nibble is significant, with these possible values:

Value	State	Description
0	Listen	The local node is waiting for an incoming connection request from a remote node.
1	CrSent	The local node transmitted a connection request and is waiting for connection confirmation from the remote node.
2	AckWait	A listening node received a connection request, sent its confirmation, and is now awaiting the completion of the 3-way handshake.
3	Estab	The connection is established and in the data transfer phase.
4	Closing	The application initiated a disconnect of the VC; the node is awaiting confirmation of the disconnect request from the remote node.
5	Closed	The CDB is closed to communication, but has not been released, pending an application request block to give notification of the format status of the connection.
6	Open	The CDB was allocated using an OPEN command, but no subsequent requests have been made on this connection.
7	Calling	A transient internal state that should be ignored.
8	CrRcvd	A listening node received a connection request and sent an indication to the client by returning an <code>AWAIT_CONNECT_REQUEST_CLIENT</code> request block; the connection is awaiting the response.
9	RefWait	The connection is closed, but the reference is being timed out before the CDB is deleted.

`loc_ref`

The connection reference maintained by the local station. This is the value specified in the `reference` field of request blocks associated with the connection.

`rem_ref`

The connection reference maintained by the remote station. During the lifetime of the connection, the two connected nodes identify the connection for each other using this and the `loc_ref` value.

`persist`

The connection refusal `persistence_count` used by the connection.

`abort_to_hi`

The high-order word of the 32-bit abort timeout value used by the connection.

`retran_to_dw`

The current retransmission timeout value, in 100 millisecond units.

pending_rec_data

The number of undelivered bytes in the last received TPDU.

rcv_buf_rej_cnt

The number of times a TPDU was discarded due to a lack of receive buffers.

cbtq_buf_cnt

The number of SEND_EOM_DATA request blocks currently posted to this connection.

pcbq_buf_cnt

The number of RECEIVE_DATA request blocks currently posted to this connection.

exp_cbtq_buf_cnt

The number of SEND_EXPEDITED_DATA request blocks currently posted to this connection.

exp_pcbq_buf_cnt

The number of RECEIVE_EXPEDITED_DATA request blocks currently posted to this connection.

close_buf_cnt

The number of CLOSE or AWAIT_CLOSE request blocks currently posted to this connection.

loc_credit

The current flow control credit the local station can report to the remote station. If $\text{min_credit} = 0$ and $\text{loc_credit} = 0$, there are no normal receive buffers locally posted and the local station's window is closed. If $\text{min_credit} < 0$ and $\text{loc_credit} = 0$, there are no normal receive buffers locally posted and the local station's window cannot be closed, so received packets are lost.

rem_credit

The current flow control credit the remote node has reported to the local node. If it is 0, the remote node has closed the window and no normal data can be transmitted at this time.

loc_ack_no

The next normal-data TPDU sequence number the local node expects from the remote node.

rem_ack_no

The (highest sequence number + 1) of a TPDU the local node transmitted that was acknowledged by the remote node.

next_transmit

The sequence number of the next TPDU to be sent.

highest_sent

The (highest sequence number + 1) of a transmitted TPDU. The TPDU may not yet be acknowledged.

`loc_exp_ack_no`

The sequence number of the last expedited TPDU the local node received that it acknowledged.

`rem_exp_ack_no`

The sequence number of the next expedited data TPDU that can be transmitted by the local node.

`loc_subseq_no`

The next sub-sequence number the local node will transmit in an acknowledgement TPDU.

`rem_subseq_no`

The last sub-sequence number received in an acknowledgement from the remote node.

`client_options`

The negotiation options requested by the client, with the same values described earlier for the `def_negot_options` field.

`class_options`

The class of services (should be 4) and sequence number format negotiated for the connection. These are nibbles 1 and 2 of the options in the `SEND_CONNECT_REQUEST` command:

Nibble	Value	Meaning
1	0	Use 7-bit sequence numbers
	2	Use 31-bit sequence numbers
2	4	Class four service

`options`

The ISO expedited services and checksum options negotiated on the connection:

Value	Meaning
0	No expedited service, do checksums
1	Expedited service, do checksums
2	No expedited service, no checksums
3	Expedited service, no checksums

`conn_max_tpdu_size`

The maximum TPDU size finally negotiated for the connection. This is an exponent of a power of 2 (range 7 to 13).

`qos`

The low order nibble defines the network-transparent priority class that may be used by the underlying subnet. The range is 0-15, with 0 being the highest priority. For iNA 960 Data Link 802.3 subnets, transmit priority is not supported.

`max_tpdu_data_len`

The maximum length of application data that can be in a data TPDU of the maximum negotiated TPDU size and maximum configured NSDU size.

`max_nsd_size`

The maximum size of an NSDU that the connection can pass to the underlying local Network Layer.

`cdb_ta_buffer[ta_buffer_len]`

A copy of the transport address (TA) buffer maintained by the connection. The length is variable and depends on the address field length defined in the buffer.

See also: TSAP address buffer structure, page 180

WITHDRAW_DATAGRAM_RECEIVE_BUFFER

WITHDRAW_DATAGRAM_RECEIVE_BUFFER requests that datagram receive buffers posted for a local TSAP selector be withdrawn and returned to the client. Only buffers that have not yet received data are withdrawn. Buffers with data received by Transport service are returned with the data intact, as described for the RECEIVE_DATAGRAM command.

This command may be used to withdraw datagram receive buffers at any time. The request block is returned after the receive buffers are returned.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;        /* of datagram_rb */
    selector          user_id;       /* cq_create_comm_user */
    unsigned char     resp_port;     /* 0FFH */
    selector          resp_mbox;     /* mailbox token */
    selector          rb_seg_tok;    /* segment token */
    unsigned char     subsystem;     /* 41H */
    unsigned char     opcode;        /* 13H */
    unsigned short    response;      /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;        /* unused */
    unsigned short    length;        /* unused */
} DATA_BLOCK;

typedef struct datagram_rb {
    RB_COMMON         header;
    unsigned char     reserved[4];
    unsigned long     ta_buffer_addr; /* input */
    unsigned char     qos;           /* input */
    unsigned short    buf_len;       /* in/out */
    unsigned char     num_blks;      /* unused */
    DATA_BLOCK       data_blk_list[1]; /* unused */
} DATAGRAM_RB;
```

Input Arguments

`ta_buffer_addr`

An address pointing to a TSAP address buffer. The local TSAP selector must be loaded into the buffer. This command withdraws one or more receive datagram buffers (if any) posted for that local TSAP selector. If no buffers are posted for the local TSAP selector, the request block is returned indicating that no buffers were withdrawn (`buf_len = 0`). The remote NSAP address and TSAP selectors are not input parameters.

See also: TSAP address buffer structure, page 180

`qos` Set to 0.

`buf_len`

The total number of bytes to withdraw. Enough posted receive datagram buffers for the TSAP are withdrawn to satisfy the byte specification. If the last buffer withdrawn has more bytes than required to satisfy the specification, the entire buffer is withdrawn. Thus, the original buffers posted are returned intact and more bytes may be returned than specified. Any buffers posted after the byte specification is satisfied will remain posted.

If `buf_len = 0`, or there are no buffers posted, this command is a null operation; zero bytes are returned. If there are fewer bytes posted than specified, all posted buffers are returned. If `buf_len = 0FFFFH`, all buffers for the connection will be withdrawn.

Responses

Output Arguments

`buf_len`

The exact number of bytes in all withdrawn buffers returned to the application. If the value of this argument is `0FFFFH`, all buffers for the connection were returned.

Response Codes

<code>OK_WITHDRAW_RESP</code>	9H	Zero or more datagram receive buffers were withdrawn from Transport service.
<code>ILLEGAL_ADDRESS</code>	1AH	An addressing error was detected.

WITHDRAW_EXPEDITED_BUFFER

WITHDRAW_EXPEDITED_BUFFER requests that expedited receive buffers posted for a specific connection be withdrawn and returned to the client. Only buffers that have not yet received data are withdrawn. Buffers with data received by the Transport service are returned to the client with the data intact as described for the RECEIVE_EXPEDITED_DATA command.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 0BH */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;      /* unused */
    unsigned short    length;      /* unused */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* unused */
    unsigned char     reserved[15];
    unsigned short    reference;     /* input */
    unsigned char     qos;           /* unused */
    unsigned short    buf_len;       /* in/out */
    unsigned char     num_blks;      /* unused */
    DATA_BLOCK       data_blk_list[1]; /* unused */
} VC_RB;
```

Input Arguments

reference

Identifies the CDB for which the expedited data buffer is being withdrawn. If no buffers are posted for the connection, the request block is returned indicating that no buffers were withdrawn (`buf_len = 0`).

buf_len

Specifies the total number of bytes to withdraw. Enough buffers posted for the connection are withdrawn to satisfy the byte specification. If the last buffer withdrawn has more bytes posted than required to satisfy the specification, the entire buffer is withdrawn. Thus, the original buffers are returned intact, and more bytes may be returned than specified. Any buffers posted after the byte specification is satisfied will remain posted. If `buf_len = 0`, or there are no buffers posted, this command is a null operation. If there are fewer bytes in posted buffers than specified here, all posted buffers are returned.

If `buf_len = 0FFFFH`, all expedited buffers for the connection will be withdrawn.

Responses

Output Arguments

buf_len

The sum of the lengths of all buffers that were withdrawn and returned to the client. The value `0FFFFH` indicates that all buffers for the connection were withdrawn.

Response Codes

UNKNOWN_REFERENCE	6H	The reference does not correspond to any allocated CDB.
OK_WITHDRAW_RESP	9H	Zero or more expedited receive buffers were withdrawn from Transport service.

Additional Information

This command may be used to withdraw expedited receive buffers at any time. It is especially useful to reclaim resources that are no longer needed. This request block is returned following the return of all expedited receive buffers to the client.

WITHDRAW_RECEIVE_BUFFER

WITHDRAW_RECEIVE_BUFFER requests that normal receive buffers previously posted for a specific connection be withdrawn and returned to the client. Only buffers that have not yet received data are withdrawn. Buffers with data received by the Transport service are returned to the client with the data intact as described for the RECEIVE_DATA command.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of vc_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 40H */
    unsigned char     opcode;      /* 8H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct data_block {
    unsigned long     address;      /* input */
    unsigned short    length;      /* input */
} DATA_BLOCK;

typedef struct vc_rb {
    RB_COMMON         header;
    unsigned char     iso_reason_code; /* unused */
    unsigned char     reserved[15];
    unsigned short    reference;     /* input */
    unsigned char     qos;           /* unused */
    unsigned short    buf_len;       /* in/out */
    unsigned char     num_blks;      /* unused */
    DATA_BLOCK       data_blk_list[1]; /* unused */
} VC_RB;
```

Input Arguments

reference

Identifies the CDB for which the normal receive buffers are being withdrawn. If no buffers are posted for the connection, the request block is returned indicating that no buffers were withdrawn (`buf_len = 0`).

buf_len

Specifies the total number of bytes to withdraw. Enough posted buffers are withdrawn to satisfy the byte specification. If the last buffer withdrawn has more bytes posted than required to satisfy the specification, the entire buffer is withdrawn. Thus, the original buffers posted are returned intact, and more bytes may be returned than specified. Any buffers posted after the byte specification is satisfied will remain posted. If `buf_len = 0`, or there are no buffers posted, this command is a null operation. If there are fewer bytes posted than specified, all posted buffers are returned. If the value of this argument is `0FFFFH`, all buffers for the connection will be withdrawn.

Responses

Output Arguments

buf_len

The sum of the lengths of all buffers that were withdrawn and returned. The value `0FFFFH` indicates that all buffers for the connection were withdrawn.

Response Codes

UNKNOWN_REFERENCE	6H	The reference does not correspond to any allocated CDB.
OK_WITHDRAW_RESP	9H	Zero or more normal receive buffers were withdrawn from Transport service.

Additional Information

This command may be used to withdraw normal buffers at any time. It is especially useful when the client wishes to reclaim resources that are no longer needed.

Depending on the size and availability of posted buffers for this connection, this command may withdraw fewer, more, or the same number of bytes specified in the `buf_len` input parameter. Buffers are withdrawn until either no more buffers remain posted, or until the length specification is met.



Programming the Data Link Layer **13**

This chapter describes the facilities of the iNA 960 Data Link Layer and the subnets supported by it.

Overview of the Data Link Layer

The Data Link Layer includes two application interfaces: the External Data Link (EDL) directly accesses the ISO data link Control Layer, and the RawEDL is a non-ISO interface for lower-level functions.

The Data Link Layer transforms the raw transmission and reception facility of the subnet-dependent Physical Layer into a communications channel that appears error-free to the Network Layer. The Data Link Layer accomplishes this by assembling raw data packets taken from the Physical Layer into frames that are transmitted sequentially to the Network Layer. Conversely, the Data Link Layer takes frames from the Network Layer and disassembles them into raw data packets for transmission to the destination node. In addition, the Data Link Layer performs CRC checks on packets received from the Physical Layer.

The Data Link Layer provides a datagram service that does not ensure accurate reception of data. Reliable communications over the network is provided by the Transport Layer VC service.

A subnet is a collection of equipment and physical media comprising a homogeneous environment where end systems (nodes) are interconnected for communications purposes. A subnet usually has a common physical interconnection technology, data link protocol, and data link address mechanisms. This is especially true for Local Area Networks (LANs) where the nodes have these characteristics in common:

- The physical medium and methodology to access the medium
- The addressing format
- A well-understood data link message (or frame) format

Subnets may be individually interconnected by internetwork routers designed to resolve the incompatibilities between networks so that messages can be exchanged between nodes residing on separate subnets.

Individual subnets interconnected by one or more routers collectively define what is referred to as a Network. The Data Link Layer and the protocol used to control its functionality reside at the lowest level of the Network end node addressing hierarchy.

See also: Addressing and network topology, Chapter 8

The Data Link Layer maintains statistics that monitor the performance of a network node and record data error rates. The information is stored as objects that can be manipulated by the iNA 960 Network Management Facility.

See also: NMF commands, Chapter 14

The External Data Link (EDL) Interface

The Data Link Layer control software implements Class 1 of the Logical Link Control (LLC) sublayer described in the IEEE 802.2 standard, and the Media Access Control sublayer described in the IEEE 802.3 standard. The IEEE 802.3 standard supports the Carrier-Sense Multiple Access with Collision Detection (CSMA/CD) media access method.

The EDL commands are an interface to the Data Link control software. EDL commands circumvent the Transport and Network Layers and access the services of the Data Link Layer directly. The application that uses EDL commands must sufficiently duplicate the routing and data integrity functions of the Transport and Network Layers to ensure successful transmission and reception of data using the Data Link Layer.

The RawEDL Interface

In addition to the EDL commands, iNA 960 provides an extended set of EDL routines called RawEDL. This non-ISO interface enables an application to send and receive Ethernet packets directly to the network, bypassing the normal 802.3 interface provided by iNA 960. The interface is co-resident with the normal EDL routines and enables full use of iNA 960's EDL, Network, and Transport interfaces.

The RawEDL interface addresses two classes of applications. The first is an application that implements a non-ISO protocol stack, such as TCP/IP, XNS, Novell or DECnet, while retaining the functionality of iNA 960's ISO Transport. Examples of this are a multiple-protocol gateway, or an application that lets Intel OpenNET networking software co-exist with a TCP/IP stack.

The second use is for an application that needs low-level access to traffic on the Ethernet without having to program a low-level driver for the 82586 chip. Examples of this are a network monitor or bridge application. Such an application needs to work in promiscuous mode, receiving all network traffic regardless of destination address. The standard iNA 960 EDL interface would not allow such access.

An application that does protocol analysis also could use RawEDL commands. Such an application is normally interested in frames from only a few stations. RawEDL can filter out all unwanted frames, so the host CPU need not be bothered with unwanted traffic. Applications that want to receive all traffic can still do so.

Standard EDL commands return receive buffers as soon as a frame is received. RawEDL can accumulate received frames in the buffers and return them only when full. This further minimizes interaction with the main CPU, particularly when large buffers are used. A header is inserted in front of each received frame to allow the host to separate them again.

For protocol analysis purposes, it may not be necessary to record the entire frame since only the protocol header will suffice. The RawEDL receive function can automatically truncate frames longer than a specified maximum length.

iNA 960-Supported Hardware Subnets and Protocols

The iNA 960 software supports the IEEE 802.3 (Ethernet) specification. Preconfigured LAN subnets of this type are available for this Intel hardware:

- 82586/82596 LAN Coprocessor (LP 486, SBC 486/133SE, SBC 486/166SE)
- 82595TX (SBC P5090 for Multibus II, EtherExpress PRO/10 card)
- DEC 211A3 (SBCP5200 for Multibus II, various PC1 cards)
- SBC 186/530 Multibus II module
- MIX 386/560 Multibus II module
- SBC 552A module
- SBX 586 module
- MIX 560 Multibus II Ethernet COMMputer
- PCL2 and PCL2A Ethernet cards
- EtherExpress 16 Ethernet card
- EWENET module
- Multibus II subnet

The subsystem code in Data Link request blocks specifies the subnet type.

LSAP Identifiers

EDL applications communicate using link service access points (LSAPs). An LSAP selector identifies the LSAP at which a specific task or client process requests or receives Data Link services. Each receiving client is identified by a destination LSAP (DLSAP) selector, and each sending client is identified by a source LSAP (SLSAP) selector.

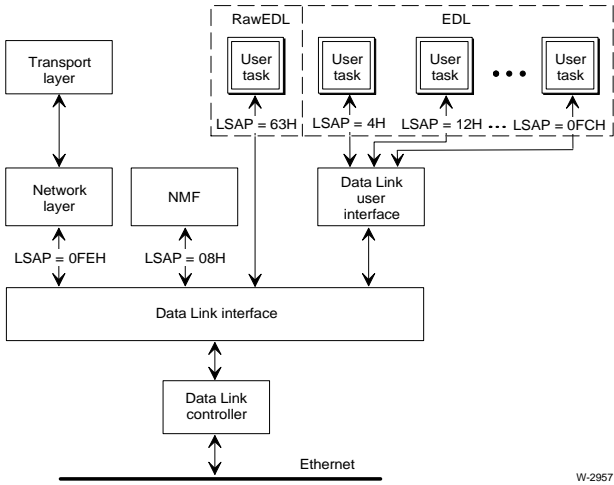
When a packet is sent, the destination process is identified by the DLSAP selector field in the first data buffer. Before a destination process can receive a packet, its DLSAP selector must be included in a list of the active LSAP selectors for the destination data link. The Data Link Layer only receives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

The EDL CONNECT command adds an LSAP selector to the active list. Any incoming packet containing that LSAP selector is routed to that process. The DISCONNECT command removes an LSAP selector from the active list when it is no longer needed. The maximum number of LSAP selectors that may be active at one time is specified in the subnet configuration.

EDL applications arbitrarily choose LSAP selectors that are a multiple of 4, in the range 4 through 0FCH. Applications cannot use LSAP selector 8; it is reserved for the NMF. LSAP selector 0FEH is used by the iNA 960 Network Layer.

The RawEDL interface uses LSAP selector 63H; an application uses this selector to specify a RawEDL command.

Figure 13-1 illustrates how LSAPs identify applications (user tasks) and iNA 960 subsystems to the Data Link Layer.



W-2957

Figure 13-1. Data Link Interface

Data Link Commands

Table 13-1 lists the EDL and RawEDL commands. The commands in this chapter are specified by the `subsystem` and `opcode` fields in the request block header, `rb_common`. Where the commands have the same name, the opcode is the same for both interfaces; specify the RawEDL interface with LSAP selector 63H. Detailed descriptions of each command follow this section. Each command description lists which fields are input and output arguments. Initialize reserved fields and unused fields to 0. The structures are provided as typedefs in the Data Link layer's include files.

See also: Using the `cq_` System Calls, Chapter 10;
 Include Files, Chapter 10;
 Programming with Structures, Chapter 10

Use EDL commands to perform these functions in your application:

- To establish and terminate the connection between the application and an LSAP, use `CONNECT` and `DISCONNECT`.
- To send and receive data packets through the Data Link and over the network, use `TRANSMIT` and `POST_RPD`.
- To dynamically change some configuration of the Data Link and the 82586 controller, use `CONFIGURE`, `IA_SETUP`, `MC_ADD`, and `MC_REMOVE`.

Use RawEDL commands to perform these functions:

- To establish and terminate the connection between the application and RawEDL, use `CONNECT` and `DISCONNECT`.
- To send and receive data packets, use `RAW_TRANSMIT`, `RAW_POST_RECEIVE`, and `FLUSH`.
- To dynamically change some configuration of the Data Link and the 82586 controller, use `CONFIGURE`, `MC_ADD`, and `MC_REMOVE`.
- To get timing information, use `READ_CLOCK`.

Table 13-1. Data Link Commands

EDL	RawEDL	Opcode	Description
CONFIGURE	CONFIGURE	88H	Sends configuration information to the data link controller
CONNECT	CONNECT	82H	Establishes a connection between a process and an LSAP, assigning a specific LSAP selector
DISCONNECT	DISCONNECT	83H	Terminates the connection for a specified LSAP selector
	FLUSH	81H	Returns RawEDL receive buffers that have collected any data
IA_SETUP		89H	Sets the Ethernet address of the local node
MC_ADD	MC_ADD	87H	Adds a multicast address to the list of addresses the controller listens to
MC_REMOVE	MC_REMOVE	8AH	Removes a multicast address from the controller's list
POST_RPD		85H	Posts one or more receive buffers to collect incoming data
	RAW_POST_RECEIVE	7FH	Posts a receive buffer to collect incoming data
	RAW_TRANSMIT	7EH	Transmits data
	READ_CLOCK	80H	Returns the current value of the local network timer
TRANSMIT		84H	Transmits a data packet

Table 13-2 lists the subsystem IDs you can use at the Data Link layer. Specify the appropriate value in the `subsystem` field of Data Link request blocks.

Table 13-2. Data Link Subsystem IDs

Table	Subsystem
	Data Link for:
20H	Boards with 82586 component, including first MIX560 board in the system
21H	SBX 586 board, EWENET module, or EtherExpress 16
22H	Second MIX560 board in the system
23H	Third MIX560 board in the system
24H	82595TX component, EtherExpress PRO/10, SBC P5090 PC-compatible board
25H	DEC 21143 component, SBC P5200 PC-compatible boards, all versions
2FH	Multibus II subnet

Table 13-3 lists all response codes that can be returned from Data Link commands in an IEEE 802.3 subnet. The command descriptions list response codes appropriate to the individual command.

Table 13-3. IEEE 802.3 Response Codes

Literal	Code	Description
E_ERROR	00H	Failure: reason not specified or unknown
OK_RESPONSE	01H	Execution with no errors; also implies that the end of the packet has not been returned
E_CONFIG_COUNT	02H	Number of configuration information bytes exceeds the maximum for the subnet
OK_EOP_RESPONSE	03H	OK end_of_packet response, implies error free execution and the return of the end of the packet
E_INSUFF_RCV_BUF	04H	Insufficient receive buffers
E_TX_SIZE_EXCEEDED	06H	Size of the transmit packet exceeds the configured maximum
E_OPCODE	08H	Invalid opcode value for Data Link commands
E_LSAP_NOT_EXIST	0AH	Connect/Disconnect error: LSAP does not exist
E_SUBSYSTEM	0CH	Incorrect subsystem code
E_ADDR_COUNT	0EH	Number of address bytes exceeds the maximum of six
E_NOT_OK	10H	The 82586 reports that command execution is not OK
E_MC_NOT_EXIST	12H	The multicast address to be removed does not exist
E_BUFFER_COUNT	14H	Buffer count exceeds the maximum of 4
E_NO_RESOURCES	16H	Out of resources

E_ZERO_LSAP	18H	LSAPS with value zero are not allowed
-------------	-----	---------------------------------------

CONFIGURE

The CONFIGURE command configures the 82586 data link controller. The configuration information is contained in a segment of memory that may be up to 12 bytes long. The actual configuration data is part of the request block.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of configure_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;      /* 88H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct configure_rb {
    RB_COMMON         header;
    unsigned short    reserved;
    unsigned short    count;       /* input */
    unsigned char     configure[12]; /* input */
} CONFIGURE_RB;
```

Input Arguments

count The size in bytes of the configuration information. This can be up to 12 bytes for the IEEE 802.3 subnet.

configure

An array of configuration data. These 12 bytes are defined by the argument field of the 82586 LAN coprocessor CONFIGURE command. The first byte of this array is the one that contains the Byte Count field.

See also: 82586 and 82596, Intel *Microcommunications* data book

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	Successful execution of the command.
E_CONFIG_COUNT	02H	The count argument value exceeds the maximum of 12 for IEEE 802.3 or 30 for IEEE 802.4.
E_SUBSYSTEM	0CH	Incorrect subsystem code.

Additional Information

These restrictions on the configuration data applies to boards based on the 82586 (including the SBX 586):

- The address allocation bit is always reset.
- The save bad packet option is always OFF. If turned ON in a command, it will be reset.
- The Ethernet address or data link address length must always be 6 bytes long.
- The Data Link performs packet padding operations. The application must not alter the minimum packet length parameter.

CONNECT

As an EDL command, CONNECT provides a connection between an application and the LSAP identified by the specified LSAP selector.

To specify the RawEDL command, use LSAP selector 63H. This does not conflict with concurrent use of EDL by other iNA 960 processes since 63H is not a multiple of 4. Although the RawEDL receive process does not filter on LSAP selectors, the CONNECT command is still required for house-keeping purposes.

For either version of the command, a connection must be made before any receive request blocks are posted for the LSAP.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of connect_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* Table 13-2 */
    unsigned char       opcode;      /* 82H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct connect_rb {
    RB_COMMON           header;
    unsigned char       lsap_sel;    /* input */
    unsigned char       reserved;    /* input for RawEDL */
    unsigned char       port;        /* input */
} CONNECT_RB;
```

Input Arguments

`lsap_sel`

For EDL, this is an arbitrary LSAP selector for the connection. Use a multiple of 4; 0 is invalid. Do not use an LSAP selector value bound in the subnet configuration to one of the other iNA 960 layers: LSAPs 0FEH or 08H. For RawEDL, set `lsap_sel` to 63H.

reserved

This field is meaningful only in the RawEDL CONNECT command. In that context it specifies the filter option to use. The RawEDL receive process uses these values to determine which data to receive, rather than filtering on an LSAP selector:

Value	Meaning
0	Filter on source address: the source address must match one of the specified multicast addresses
1	Filter on destination address: the destination address must match one of the specified addresses or it must be an Ethernet multicast or broadcast address
2	Logical AND of 0 and 1: both source and destination must match
3	Logical OR of 0 and 1: either source or destination must match
4	No special filtering other than the 82586 hardware: this would be used in a non-monitor application such as a bridge or application-implemented protocol stack

port Firmware-dependent. For Intel hardware this value must always be 0FFH.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	Successful execution of the command.
E_LSAP_NOT_EXIST	0AH	The specified LSAP does not exist.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_ZERO_LSAP	18H	A null LSAP selector was specified.
E_NO_RESOURCES	16H	The Data Link is out of resources.

Additional Information

If an LSAP has been established with the `CONNECT` command and there are receive buffers posted for the LSAP (using the `POST_RPD` command), any received packet that identifies this LSAP is placed in the buffers and the buffers are returned to their owners. Only nonzero LSAP selectors are accepted for processing. If the LSAP specified in this command is already active, a new association replaces the old one.

Only a configurable maximum number of LSAPs may be active at one time. The `CONNECT` command is ignored if the maximum number of connections are currently established.

DISCONNECT

DISCONNECT terminates the specified connection. If the connection does not exist, the command is ignored and an error is returned.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of disconnect_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;      /* 83H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct disconnect_rb {
    RB_COMMON          header;
    unsigned char     lsap_sel;    /* input */
    unsigned char     reserved;
} DISCONNECT_RB;
```

Input Arguments

lsap_sel

The LSAP selector identifying the connection to terminate. Use a multiple of 4; 0 is invalid. Do not use an LSAP selector value bound in the subnet configuration to one of the other iNA 960 layers: LSAP 0FEH or 08H. To specify the RawEDL DISCONNECT command, set lsap_sel to 63H.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	Successful execution of the command.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_ZERO_LSAP	18H	A null LSAP selector was specified.
E_LSAP_NOT_EXIST	0AH	The specified LSAP selector identifies a nonexistent LSAP.

Additional Information

Once a connection is disconnected, all receive buffers and receive request blocks posted with the LSAP being disconnected are returned to the application owning them. The application must ensure that the number of buffers posted does not exceed four times the number of receive request blocks.

The RawEDL application should issue a DISCONNECT before terminating. The RAWEDL command operates the same as the EDL command; it returns empty and partially filled receive buffers. The response code on these request blocks will be indeterminate.

FLUSH

The FLUSH command returns the current receive request block (issued with a RAW_POST_RECEIVE) if it has captured any data. This enables the application to examine data by polling for partially-filled buffers. Only the opcode and subsystem fields are relevant; all other request block fields are ignored.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of flush_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* Table 13-2 */
    unsigned char       opcode;      /* 81H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct flush_rb {
    RB_COMMON           header;
} FLUSH_RB;
```

Responses

Output Arguments

None

Response Code

OK_RESPONSE	01H	Successful execution of the command.
-------------	-----	--------------------------------------

IA_SETUP

The IA_SETUP (Individual Address Setup) command sets the Ethernet address for a node, overriding the Ethernet address set up by the hardware at system initialization.

⇒ Note

IA_SETUP is not supported by all subnets. For instance, the 595 Subnet driver returns an E_OPCODE error in response to an IA_SETUP command.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char      length;      /* of ia_setup_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char      resp_port;   /* 0FFH */
    selector            resp_mbox;  /* mailbox token */
    selector            rb_seg_tok; /* segment token */
    unsigned char      subsystem;   /* Table 13-2 */
    unsigned char      opcode;     /* 89H */
    unsigned short     response;    /* initialize to 0 */
} RB_COMMON;

typedef struct ia_setup_rb {
    RB_COMMON          header;
    unsigned short     reserved;
    unsigned short     count;      /* input */
    unsigned char      address[6]; /* input */
} IA_SETUP_RB;
```

Input Arguments

count The size in bytes of the Ethernet address; this value must be 6.

address

The new six-byte Ethernet address.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	Successful execution of the command.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_ADDR_COUNT	0EH	The number of bytes in the Ethernet address exceeds the maximum of 6.

MC_ADD

The MC_ADD command adds a multicast address to the data link multicast address list. These are addresses for which the controller will receive incoming data packets, in addition to broadcast packets and packets addressed to the address of this node. In a network-monitor or bridge application, this command can be used to specify a station to listen to.

An address of FFFFFFFF puts this node in promiscuous mode (receiving all network data packets) at the RawEDL level. The 82586 controller must also be programmed as promiscuous, using the CONFIGURE command.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;        /* of mc_add_rb */
    selector          user_id;       /* cq_create_comm_user */
    unsigned char     resp_port;     /* 0FFH */
    selector          resp_mbox;     /* mailbox token */
    selector          rb_seg_tok;    /* segment token */
    unsigned char     subsystem;     /* Table 13-2 */
    unsigned char     opcode;        /* 87H */
    unsigned short    response;      /* initialize to 0 */
} RB_COMMON;

typedef struct mc_add_rb {
    RB_COMMON          header;
    unsigned short     reserved;
    unsigned short     count;        /* input */
    unsigned char      mc_address[6]; /* input */
} MC_ADD_RB;
```

Input Arguments

count

The size in bytes of the multicast address; this number must be 6.

mc_address

The six-byte multicast address. The least significant bit of the first (most significant) byte must be 1, to specify the address is multicast.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	The address is successfully added to the multicast address list.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_ADDR_COUNT	0EH	The number of bytes in the Ethernet address exceeds the maximum of 6.
E_NO_RESOURCES	16H	The Data Link is out of resources.

Additional Information

Each address must be added with a separate command. The iNA 960 Data Link performs perfect multicast filtering, whereas the 82586 controller performs imperfect multicast filtering. The maximum number of multicast addresses that can be active at one time is determined by the subnet configuration.

See also: Broadcast and multicast addresses, *82596 User's Manual* or *32-Bit LAN Component User's Manual*

MC_REMOVE

The MC_REMOVE command removes a single multicast address from the list of active multicast addresses for a given data link. Each address must be 6 bytes long. Removing the address FFFFFFFFHH results in the station being non-promiscuous (not listening to all messages) at the RawEDL level. An address of all zeroes clears the multicast list.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of mc_remove_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;     /* 8AH */
    unsigned short    response;   /* initialize to 0 */
} RB_COMMON;

typedef struct mc_remove_rb {
    RB_COMMON          header;
    unsigned short    reserved;
    unsigned short    count;      /* input */
    unsigned char     mc_address[6]; /* input */
} MC_REMOVE_RB;
```

Input Arguments

`count`

The size in bytes of a multicast address; this number must be 6.

`mc_address`

The six-byte multicast address. If the address is all zeroes, the multicast list is cleared.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	The address is successfully removed from the active multicast address list.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_ADDR_COUNT	0EH	The number of bytes in the Ethernet address exceeds the maximum of 6.
E_MC_NOT_EXIST	12H	The specified multicast address was never added.

POST_RPD

POST_RPD posts a single receive request block together with up to four buffers. This receive request block and any associated buffers are kept by the data link software until the request block buffers are filled with incoming packets intended for the process that posted the buffers. An LSAP must be established with the CONNECT command before receive request blocks and their associated buffers may be posted for that LSAP.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of post_rpd_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;      /* 85H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct post_rpd_rb {
    RB_COMMON          header;
    unsigned char     lsap_selector; /* input */
    unsigned char     reserved;
    unsigned short    buf_count;     /* input */
    unsigned short    return_count;  /* output */
    unsigned short    byte_count[4]; /* in/out */
    unsigned long     buf_loc[4];    /* in/out */
} POST_RPD_RB;
```

Input Arguments

`lsap_selector`

The LSAP selector used to identify the connection. This parameter must match a value returned in a previous CONNECT command. Only data packets destined for this LSAP selector will be passed to the buffers posted for it by an application. Once the buffers are full, the receive request block and the associated buffers are returned to the application owning them.

`buf_count`

The number of buffers associated with the receive request block. The value may range from 0 to 4.

`byte_count`

An array of four values where `byte_count[i]` is the size in bytes of the buffer specified by `buf_loc[i]`. For the first buffer, `byte_count` must be at least 17

See also: The returned output below

`buf_loc`

An array of four addresses where `buf_loc[i]` points to the start of buffer *i*.

Responses

Output Arguments

`return_count`

The size in bytes of the information returned in this request block (up to four buffers), less the length of header information. If the returned packet fits in the buffers of this request block, or if this request block contains the beginning of a packet split between multiple request blocks, then `return_count` is 14 less than the total number of bytes returned in this request block. The `byte_count` values contain the actual length. The 14 bytes is the Media Access Control (MAC) header at the beginning of the first buffer.

When an incoming packet is split between more than one request block, `return_count` in the second and subsequent request blocks is the accurate number of bytes of data returned in the buffers of that request block.

`byte_count`

An array of four values where `byte_count[i]` is the size in bytes of information returned in the buffer specified by `buf_loc[i]`. In the first request block returned, this contains the full size of the first returned buffer, including the 14 bytes of the MAC header (destination and source addresses, ISO control information, and application data). This argument field is not updated for subsequent buffers if any were posted by this command; the `return_count` accurately specifies the size of subsequent receive buffers.

`buf_loc`

The addresses of the returned data buffers.

Response Codes

OK_RESPONSE	01H	Successful execution of the command; end of packet not returned.
OK_EOP_RESPONSE	03H	Successful execution and the end of packet is returned.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_LSAP_NOT_EXIST	0AH	The specified LSAP selector identifies a nonexistent LSAP.
E_BUFFER_COUNT	14H	The buf_count field exceeds the maximum of 4.

Additional Information

Whenever a packet is received by EDL, its LSAP selector associates it with a receive request block and any buffers that were posted by an application for the LSAP identified by that destination LSAP selector.

A maximum of one IEEE 802 receive packet may be passed to an application for every receive request block (and application buffers) posted. If the received packet is larger than the buffer space available in one receive request block, more than one request block must be posted. If the packet is larger than the total space available for all receive request blocks currently posted, the packet is discarded. There must be sufficient buffer space available (through one or more request blocks) to hold at least one packet of the size expected.

If a returned packet is split between more than one request block, the packet header information is only added to the beginning of the first buffer in the first request block. The first buffer of subsequent request blocks for that packet contain only application data.

Application Data Buffers

The first buffer returned with a receive request block contains destination and source addresses, ISO control information, and data. It must be at least 17 bytes long, plus the length of received data. The second and all subsequent buffers (to a maximum of 4) contain only data. The last buffer returned may contain fewer data bytes than the buffer is capable of holding. The format of the buffers is shown below:

```
typedef struct first_receive_buffer {
    unsigned char    destination_addr[6];
    unsigned char    source_addr[6];
    unsigned short   information_len;
    unsigned char    destination_lsap_selector;
    unsigned char    source_lsap_selector;
    unsigned char    iso_cmd;
    unsigned char    data[1];
} FIRST_RECEIVE_BUFFER ;

typedef struct next_receive_buffer {
    unsigned char    data[1];
} NEXT_RECEIVE_BUFFER ;
```

Where:

`destination_addr`

The Ethernet address of the node that received the packet.

`source_addr`

The Ethernet address of the node that sent the packet.

`information_len`

The length in bytes of the information in the packet (excluding header) received from the data link. The value is identical to the `return_count` field specified in the request block, if the packet's data fits into the buffers of a single request block. This value is the number of bytes received following the `information_len` field.

`destination_lsap_selector`

The LSAP selector for the Data Link entity that received the packet.

`source_lsap_selector`

The LSAP selector for the Data Link entity that sent the packet.

`iso_cmd` 03H for the 82586 component and 82586-based boards.

`data` An array of bytes that contains the actual data.

RAW_POST_RECEIVE

The RAW_POST_RECEIVE command acts as a garbage collector at the Data Link level. All packets not otherwise claimed by the OSI Network Layer or other EDL applications are sent to this interface.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* raw_post_receive_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;      /* 7FH */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct raw_post_receive_rb {
    RB_COMMON          header;
    unsigned char      lsap_selector; /* input */
    unsigned char      reserved;
    unsigned short     num_blks;      /* input */
    unsigned short     filled_length; /* output */
    unsigned short     buffer_length; /* input */
    unsigned short     max_copy_len;  /* input */
    unsigned short     max_frames;    /* input */
    unsigned short     actual_frames; /* output */
    unsigned long      buffer_ptr;    /* input */
} RAW_POST_RECEIVE_RB;
```

Input Arguments

`lsap_selector`

Set to 63H.

`num_blks`

The number of buffers; set to 1.

`buffer_length`

The length of the buffer specified by the `buffer_ptr` field. For an application such as a network monitor, it may be best to use a very long buffer.

`max_copy_len`

The length at which to truncate received frames. This length includes the header inserted before each frame. The minimum value of 22 accommodates the frame header plus the 802.3 header. Specify FFFFH to record the full length of each frame. Values smaller than 22 default to FFFFH.

`max_frames`

The maximum number of frames to receive. If 0, the request block is not returned until the buffer is full. If not 0, the request block is returned when the buffer is full or when the specified number of frames has been received. A monitor application could put a 0 value here (receive only full buffers). More interactive applications may specify 1 to have frames returned immediately.

`buffer_ptr`

The address pointing to the buffer where frames are received.

Responses

Output Arguments

`filled_length`

The actual number of bytes received in the buffer.

`actual_frames`

The total number of frames contained in the buffer. If the last frame in the buffer is incomplete (to be continued in the next buffer), it is not included in this count. If the first frame in the buffer is a continuation, it is not included in this count.

buffer_ptr

The indicated buffer contains returned frames. A 22-byte header is inserted before each frame:

```
typedef struct frame_header {
    unsigned short    record_length;
    unsigned long     time_stamp;
    unsigned short    lost_count;
    unsigned char     dest_address[6];
    unsigned char     src_address[6];
    unsigned short    len_or_type;
    unsigned char     frame_data[1];
} FRAME_HEADER;
```

record_length

The length of the frame plus header, including this field. If the frame has been truncated, this is the truncated length.

time_stamp

The time this frame was received, in clock ticks since iNA 960 began execution. The granularity of the time stamp depends on the configured clock rate. For standard iNA 960 configurations, the clock is configured as 2000; for 8 Mhz boards this gives a granularity of approximately 25 milliseconds. To find the true granularity, use the READ_CLOCK command.

lost_count

Number of frames lost since the last frame due to lack of buffer space.

dest_addr Destination Ethernet address of the packet.

src_addr Source Ethernet address of the packet.

len_or_type

Unlike the 802.3 header, the Ethernet header contains a Type field instead of a Length field. The application can determine if it is a Type, since all currently used Types are illegal lengths for 802.3 networks. If this is a Length field, it specifies the number of bytes in the frame_data field.

frame_data

The received data.

Response Codes

OK_RESPONSE	01H	Successful execution of the command. If the last frame in the buffer is incomplete; the frame will be continued in the next buffer. The first 22 bytes are never split across buffers.
OK_EOP_RESPONSE	03H	The last frame in the buffer is complete.

Additional Information

A frame is eligible for reception by RAW_POST_RECEIVE if it is a non-ISO packet, has a destination address other than the currently configured 82586 address, or is an ISO packet for which no other process is waiting (for example, when an ISO packet with a DLSAP not currently connected is received by the hardware). It is the responsibility of the application to do any demultiplexing that may be required, such as separating TCP/IP packets from XNS packets.

RAW_TRANSMIT

RAW_TRANSMIT transmits a packet of data. Unlike the EDL TRANSMIT command, RawEDL does not fragment data; this command only takes a single data buffer.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* raw_transmit_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* Table 13-2 */
    unsigned char       opcode;      /* 7EH */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct raw_transmit_rb {
    RB_COMMON            header;
    unsigned short      reserved;
    unsigned short      len_or_type; /* input */
    unsigned char       src_addr[6]; /* input */
    unsigned short      buf_cnt;     /* input */
    unsigned long       buffer_ptr;  /* input */
    unsigned long       dst_addr_ptr; /* input */
} RAW_TRANSMIT_RB;
```

Input Arguments

`len_or_type`

The first word to send after the source and destination addresses in an Ethernet packet; typically used as a type field in non-ISO networks.

`src_addr`

Currently, this array must be set to 0.



CAUTION

If this field contains any value other than 0, the 82586 is programmed to that value as a source address. No further packets sent to this node at the former (true) address will be received.

buf_cnt

The number of bytes of data in the buffer to send.

buffer_ptr

The address pointing to the data buffer.

dest_addr_ptr

The address pointing to a six-byte destination Ethernet address.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	Successful execution of the command.
E_TX_SIZE_EXCEEDED	06H	The size of the transmit packet exceeds the maximum configured for the Data Link Layer.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_NO_RESOURCES	16H	The Data Link is out of resources.

READ_CLOCK

The READ_CLOCK command returns the value of the internal network job timer as a 32-bit value. With this command, the application can calibrate the time stamps received from the RAW_POST_RECEIVE command. For example, the application could issue the command twice at a one-second interval and derive the number of network clock ticks per second. For the typical preconfigured iNA file, a clock tick is approximately 20 milliseconds (iTP4) to 25 milliseconds (iNA 960). With the READ_CLOCK command, the application can determine the value accurately.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of read_clock_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;      /* 80H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct read_clock_rb {
    RB_COMMON          header;
    unsigned long      timer_val;  /* output */
} READ_CLOCK_RB;
```

Responses

Output Arguments

timer_val

The number of internal clock ticks since the network job started. The length of the network clock tick is determined by a configuration parameter.

Response Code

OK_RESPONSE	01H	Successful execution of the command.
-------------	-----	--------------------------------------

TRANSMIT

TRANSMIT transmits a packet consisting of from 1 to 4 buffers. This command can be used without there being an established connection to the destination LSAP.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of transmit_rb /
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* Table 13-2 */
    unsigned char     opcode;      /* 84H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct transmit_rb {
    RB_COMMON         header;
    unsigned short    reserved;
    unsigned short    buf_count;   /* input */
    unsigned short    byte_count[4]; /* input */
    unsigned long     buf_loc[4];  /* input */
    unsigned long     dest_addr_ptr; /* input */
} TRANSMIT_RB;
```

Input Arguments

`buf_count`

The number of buffers specified by the TRANSMIT command, ranging from 0 to 4.

`byte_count`

An array of four values where `byte_count[i]` is the size in bytes of the buffer specified by `buf_loc[i]`. For the first buffer, the value of `byte_count` includes `destination_lsap_selector`, `source_lsap_selector`, and `iso_cmd`, in addition to data (see the `buf_loc` parameter).

buf_loc

An array of four addresses where `buf_loc[i]` points to the start of buffer *i*. The first buffer contains ISO control information in addition to data. Any subsequent buffers contain only data. The buffers have the format shown below:

```
typedef struct first_transmit_buffer {
    unsigned char    destination_lsap_selector;
    unsigned char    source_lsap_selector;
    unsigned char    iso_cmd;
    unsigned char    data[1];
} FIRST_TRANSMIT_BUFFER ;

typedef struct next_transmit_buffer {
    unsigned char    data[1];
} NEXT_TRANSMIT_BUFFER;
```

Where:

`destination_lsap_selector`

The LSAP identifying the destination entity to which the packet is forwarded.

`source_lsap_selector`

The LSAP for the source entity that sends the packet.

`iso_cmd` 03H for the 82586 component and 82586-based boards.

`data` An array of bytes that contains the actual data to transmit.

`dest_addr_ptr`

An address pointing to an array of 6 bytes where the destination Ethernet address is stored.

Responses

Output Arguments

None

Response Codes

OK_RESPONSE	01H	Successful execution of the command.
E_TX_SIZE_EXCEEDED	06H	The size of the transmit packet exceeds the maximum configured for the Data Link Layer.
E_SUBSYSTEM	0CH	Incorrect subsystem code.
E_BUFFER_COUNT	14H	The buf_count field exceeds the maximum of 4.
E_NO_RESOURCES	16H	The Data Link is out of resources.

Additional Information

If the total number of bytes transmitted is less than the minimum packet size, the Data Link pads the packet to the minimum size. The padding is transparent to the requesting application.

□ □ □

Using the Network Management Facility 14

In addition to the three layer services, iNA 960 provides a set of tools called the Network Management Facility (NMF). Using NMF commands, an application (typically for a network administrator) can gather information from the layer databases. The application uses this information to monitor, debug, and tune network performance.

Similar to the iNA 960 layer services, the NMF commands are based on a request block interface. The application sends and receives request blocks using the **cq_comm_rb** call to access NMF services. This chapter describes request blocks for the NMF commands.

See also: Using the **cq_** System Calls, Chapter 10;
Chapter 16 for the structure of buffers used in NMF commands to perform network routing

Each of the iNA 960 layer services maintains a database of objects that may be accessed by the NMF. These objects are layer-dependent parameters and variables that control and log layer activity.

The NMF provides commands for accessing host memory at a node. The NMF also provides remote download capability that makes it possible for one or more nodes to provide boot service to other nodes on the network.

The **inamon** (iNA Monitor) utility provides an interactive human interface to NMF commands.

See also **inamon**, *Command Reference*

NMF Services

The NMF provides these services:

- Attach Operations
- Layer Management
- Event Notification Operation
- Debugging Operations
- Maintenance Operations
- Remote Load Operations

The Attach Operations service can establish communications with all nodes on a network. This service requires the services of the iNA 960 Transport Layer to establish a connection, and will operate through internet routers.

The Layer Management service provides the capability to examine and modify iNA 960 database objects. The iNA 960 database objects are layer-dependent parameters and variables that control and log layer activity. This service requires the services of the iNA 960 Transport Layer to operate remotely, and will operate through internet routers. The Transport Layer is not needed for local functions.

The Event Notification service monitors event objects in the iNA 960 database. Event notifications are unsolicited messages typically associated with abnormal occurrences in the Transport and Network layers. Because event notification is unsolicited, a specific `AWAIT_EVENT` command is provided for an application to receive those messages.

The Debugging Operations service provides the capability to read or alter memory at a node. This requires the services of the iNA 960 Transport Layer, and will operate through internet routers.

The Maintenance Operations service provides two functions, Dumping and Echo Testing.

- Dumping is initiated by a requesting node and directed towards a target node. The target receives the dump command and responds by transmitting the dump data to the requester. This function is essentially a read-memory operation that uses the iNA 960 Data Link services of both nodes to send the dump data to the requester.
- Echo Testing enables one node to determine if another node is present on the network by testing the lowest-level communication path to that node. The testing node uses the iNA 960 Data Link services to transmit data to the target node and then listen for the target to return the transmitted data.

The Remote Load Operations service provides the capability to download software from one node to another. This service is typically used to download OS and network communications software and to initiate their execution. Nodes providing this service are referred to as *boot servers* and nodes receiving the service are referred to as *boot clients*.

Data transmitted between nodes during Dumping, Echo Testing, and Remote Loading uses the lowest-level services of the network. Those services are provided by the iNA 960 Data Link Control Layer and will not operate through internet routers.

NMF Operation

The services of the Network Management Facility are provided by specific NMF commands. Where (on what nodes) the NMF commands can be executed depends on how iNA 960 is configured for each NMF node.

Specific configurations provide nodes with the ability to issue and/or execute different types of NMF operations. A node configured as a *net manager* can issue NMF commands, while a node configured as a *net agent* can only execute and respond to NMF commands. You may configure a node as both a net manager and a net agent. These nodes may be anywhere on the network as long as they are configured appropriately.

Managers and Agents

NMF interactions within a network are between one net manager and one net agent. The net manager node typically contains high-level human interface software, not provided by iNA 960, that permits an application (like the Network Administrator) to perform various network management functions. The human interface software converts the application requests into NMF requests. The NMF requests are then sent to the iNA 960 NMF that is configured as a net manager. The net manager controls all NMF interactions with the target net agent.

The net agent contains NMF services that are configured to receive net manager requests, execute the requested operations, and pass responses back to the requesting net manager.

The iNA 960 NMF may be configured as one of these:

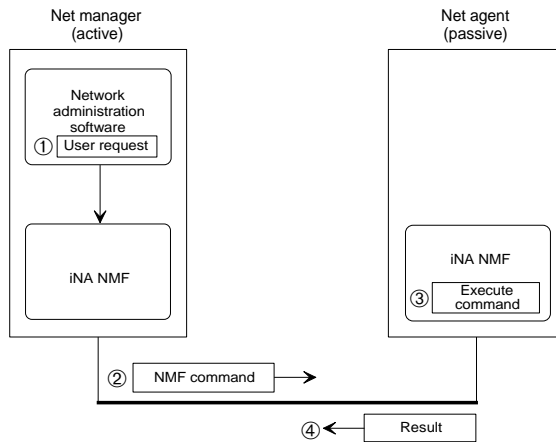
- For local operations only
- As a net agent
- As a net manager
- As both a net manager and a net agent

⇒ **Note**

Except in extreme memory-restricted situations, all iNA 960 nodes should at least be configured as net agents. That makes all iNA 960 layer databases available to all net manager nodes.

A node configured for local operations has only the Layer Management and Debugging Operations services available. The node's iNA 960 layer databases are not accessible (except through the Dump and Echo Test functions) by iNA 960 net manager nodes on the network.

An example of the type of interactions that occur between a net manager and a net agent is where an application requests a net manager to read or set the value of an iNA 960 layer object in one of the layer databases at a net agent. Figure 14-1 shows the steps taken to execute such an operation.



W-2958

Figure 14-1. A Typical Net Manager/Net Agent Interaction

The interaction steps shown in the figure are:

1. The application sends an NMF request to the iNA 960 NMF, which must be configured as a net manager.
2. The NMF net manager opens a connection to the net agent and passes the application request to the net agent as an NMF command.
3. The net agent executes the command by reading or setting the object in the specified iNA 960 layer database.
4. The net agent passes the result of the operation back to the requesting net manager.
5. The net manager closes the connection to the net agent.

Local Versus Remote NMF Operation

This section briefly reviews the overall structure of a simple network as viewed by an iNA 960 NMF node. Figure 14-2 illustrates this network; to simplify the discussion, the network structure shown does not contain any internet routers.

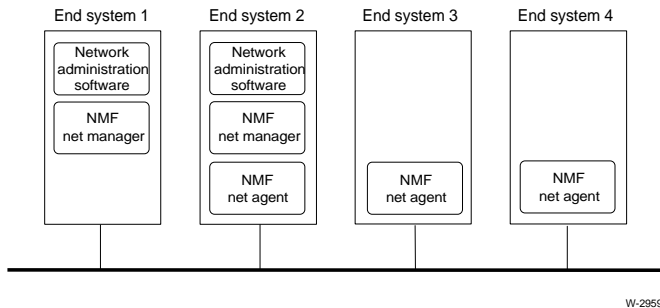


Figure 14-2. A Typical iNA 960 Network

Local Operation

Interactions between a net manager and a net agent that occur at the same node illustrate the local operation of NMF services (see Node 2 in Figure 14-2). In the example shown in the figure, no special data communications protocol is required beyond the internal communications capabilities of the node.

The net management requests are sent by the Network Administration Software such as iRMX-NET, (not part of iNA 960) to the iNA 960 NMF to be processed. The results are returned to the Network Administration Software. For example, requests may be made to query the local iNA 960 databases to monitor the communications performance of the local system.

Remote Operation

Remote NMF operations, on the other hand, require the network communication services provided by the iNA 960 layer services to transmit the net manager requests and net agent responses over the network.

A net manager and a net agent on physically separate network nodes may interact using the remote operation of NMF services. An example of a remote NMF operation would be to use a single net manager to monitor the functions and performance of all other nodes on the global network. In Figure 14-2, Nodes 2, 3, and 4 can be accessed and monitored remotely by Node 1. This is possible because the NMF on Node 1 is configured as a net manager and the NMFs on Nodes 2, 3, and 4 are configured as net agents. Node 2 can also access and monitor Nodes 3 and 4, but cannot access Node 1 because its NMF is not configured as a net agent, only as a net manager.

Excluding net agency from a node's NMF configuration protects that node's iNA 960 databases from remote access by other net managers.

The NMF commands that rely on Transport Layer services will operate through internet routers. NMF commands that rely on Data Link Control Layer services will not operate through internet routers.

NMF Communications Services

The iNA 960 NMF accesses remote nodes using the communications services provided by the iNA 960 layers. Most of the NMF services use the iNA 960 Transport Layer to provide reliable communications between a net manager and a remote net agent. Some of the NMF services use the lower-level services of the iNA 960 Data Link Control Layer for communications between a net manager and a remote net agent.

The address used by the NMF to locate the remote net agent is specified either as a Transport Layer address buffer (if the Transport Layer is used) or as a Data Link address buffer (if the Data Link Control Layer is used).

If the Transport Layer communications services are needed for a remote command, a connection must be explicitly established before the command is issued.

For example, assume that a net manager wants to issue certain net management requests to a remote node. The net manager must first make an explicit connection to the target net agent node using the NMF command `ATTACH_AGENT`. The established connection defines the path to the remote net agent node.

Once the net manager has completed the desired remote commands, it must explicitly remove the connection between itself and the remote net agent using the NMF command `DETACH_AGENT`.

The NMF supports only one open connection at a time. Multiple simultaneous open connections are not supported. Only net managers can establish connections and a single net manager can open only one connection at a time. A net agent can have only one connection from one net manager. Another net manager wanting to connect to that net agent must wait until the current connection is closed.

If the Data Link Control Layer communications services are used for a remote command, an explicit connection between a net manager and a net agent is not necessary.

Using NMF Commands

These sections describe how to use the NMF commands.

Net Agent Connection Commands

The `ATTACH_AGENT` and `DETACH_AGENT` commands are issued by a net manager that wants to establish or remove a connection to a local or remote net agent.

For local connections, the `ATTACH_AGENT` and `DETACH_AGENT` commands are optional. A `DETACH_AGENT` at the end of a communications exchange will return all unused `AWAIT_EVENT` buffers posted that were not returned through event reporting.

For remote connections, the `ATTACH_AGENT` command must be used to establish a connection to the target net agent. The connection must be established before the net manager is permitted to issue any commands that require the services of the iNA 960 Transport Layer. The `DETACH_AGENT` command ends an established connection and returns all unused `AWAIT_EVENT` buffers.

A number of the NMF commands must be preceded by an `ATTACH_AGENT` command before they are issued to a remote net agent. They are:

- `READ_OBJECT` and `SET_OBJECT` commands
- `READ_AND_CLEAR_OBJECT` command
- `READ_MEMORY` and `SET_MEMORY` commands

A typical sequence of NMF commands requested by a network administration application might be:

1. `ATTACH_AGENT` Open a connection to a net agent.
2. `READ_OBJECT` Read the value of layer objects at a target net agent.
3. `SET_OBJECT` Change the value of layer objects at target net agent.
4. (Any other NMF commands)
5. `DETACH_AGENT` Remove the connection to the target net agent.

For remote connections, the structure of the Transport Address Buffer is important because it contains the address used to locate the remote net agent. The structure is described in the `ATTACH_AGENT` command.

Layer Management Commands

The Layer Management service provided by NMF enables a network administration application to examine or modify the internal layer databases maintained by each iNA 960 subsystem.

Each iNA 960 layer (Transport, Network, Data Link) maintains a database of network management data structures called objects that represent various layer parameters. The objects contain configuration parameters or counters that indicate how the network is performing.

See also: iNA Network Objects, Appendix C

NMF Object IDs

NMF objects are identified by a two-byte ID code. The ID code appears in the form *wxyz*.

Where:

wx The *w* specifies the OSI layer that the object belongs to and the *x* specifies the layer or subsystem that the object belongs to.

yz Identifies the object.

The iNA 960 layers and subsystems have these *wx* values:

20yzH	82586 Data Link, including first MIX560 board in a Multibus II system
21yzH	Data Link for SBX 586, EWENET, or EtherExpress 16
22yzH	Data Link for second MIX560 board in a Multibus II system
23yzH	Data Link for third MIX560 board in a Multibus II system
24yzH	Data Link for 82595TX, EtherExpress PRO/10, or SBC P5090
2FyzH	Message passing Data Link for Multibus II subnet
31yzH	Network Layer
38yzH	Static Routing IP Network
39yzH	ES-IS Routing IP Network
40yzH	Transport Virtual Circuit
	4000H - 4020H Connection independent
	4040H - 405AH MAP 2.1
	4081H - 4093H Connection dependent
41yzH	Transport Datagram
80yzH	iNA 960 NMF
81yzH	iNA 960 NMF Boot Server

Values 38 and 39 are not valid in Null2 configurations.

Using Layer Management Commands

These three Layer Management commands enable a network manager to read or set the value of layer objects:

READ_OBJECT

Returns the value of the specified object or event.

SET_OBJECT

Sets the selected object with a specified value.

READ_AND_CLEAR_OBJECT

Returns the value of the specified object, then sets the value to 0.

Except for the local-only configuration of the NMF, the Layer Management commands rely on the Transport Layer to provide communication services between the net manager and a net agent. Therefore, an application that uses these commands must issue an ATTACH_AGENT command before issuing a Layer Management command and a DETACH_AGENT command after receiving the command response.

The connection reference number returned from the ATTACH_AGENT command is a required field in Layer Management command request blocks. If the command is only for the local net agent, an ATTACH_AGENT command is not needed; set the connection reference number to 0 in the request block.

The Layer Management commands can access one or more iNA 960 objects in a single command; however, all of the objects must be in the same layer. For example, a Transport Layer object and a Network Layer object cannot be read by a single invocation of the READ_OBJECT command.

Event Notification

An NMF net manager can receive notification of the occurrence of some layer event. Event notification is possible only from local net agents. The net manager uses the AWAIT_EVENT command to post a buffer that will record an event. When an event occurs, it is recorded in the buffer and the buffer is returned to the net manager that posted it.

Since event notification is a local-only function, the ATTACH_AGENT command is not necessary and the connection reference is always 0. The DETACH_AGENT command returns all unused AWAIT_EVENT buffers to the net manager.

See also: Layer Events, Appendix C

NMF Events

Only Network and Transport Layer events are managed by the iNA 960 NMF.

NMF events are identified by a two-byte ID code scheme similar to the one used for the NMF objects. These high byte (wx) values are used in event ID codes:

Value	Meaning
31yzH	Network Layer event
40yzH	Transport Layer event

Debugging Commands

As an aid to debugging operations, the NMF provides commands to read or set the host memory of any net agent on the network. The commands are READ_MEMORY and SET_MEMORY. These commands rely on the Transport Layer to provide communication services between the net manager and a net agent. Therefore, an application that uses these commands must issue an ATTACH_AGENT command before issuing a command and a DETACH_AGENT command after receiving the command response.

The READ_MEMORY and SET_MEMORY commands utilize the connection reference number returned by the ATTACH_AGENT command to identify the target net agent for the command.

Maintenance Commands

NMF Maintenance commands enable the net manager to dump the host memory of a local or remote net agent, or to determine if a remote net agent is present on the network. The DUMP command obtains a snapshot of the host memory of a net agent. The ECHO command verifies the existence of a net agent on the network.

The Maintenance commands use the Data Link Control Layer for communication between net managers and net agents. Since these commands do not use the Transport Layer for communication between nodes, they will not operate through an internet router.

For these commands, do not first use an ATTACH_AGENT command. Instead, specify a subnet address to locate the target local or remote net agent.

Remote Load Operations

The NMF provides a service for downloading OS and network communication software to remote network nodes. This service can be used to download software to and boot diskless remote nodes on the network. Alternately, the service can be used to download software to and boot a set of network nodes with the same version of software.

A remote loading operation requires the cooperation of two nodes. The node to be loaded is called the *Boot Client*. The node that does the downloading is called the *Boot Server*. The Boot Server is supplied as a separate job that uses the iNA 960 NMF; the Boot Client service is not. These are discussed together in another chapter. Two commands used for remote loading, `SUPPLY_BUFFER` and `TAKEBACK_BUFFER`, appear among the NMF commands at the end of this chapter.

See also: Remote Booting, Chapter 15

The NMF Commands

The NMF commands in this chapter are specified by the `subsystem` and `opcode` fields in the request block header, `rb_common`. The `subsystem` field must have one of these values:

- 80H for all NMF services except Remote Load Operations
- 81H for the Remote Load Operations

The commands use similar argument structures, following the common header fields. Each command description lists which fields are input and output arguments. Initialize reserved fields and unused fields to 0. The structures are provided as typedefs in the NMF include files.

See also: Include Files, Chapter 10;
Programming with Structures, Chapter 10

Table 14-1 briefly describes each NMF command. Detailed descriptions of each command follow.

Table 14-1. Network Management Facility Commands

NMF COMMAND	Opcode	Layer	Location	Description
Connection Operations				
ATTACH_AGENT	0BH	Transport	Remote/Local	Establish connection to net agent
DETACH_AGENT	0CH	Transport	Remote/Local	Break connection to net agent
Layer Management				
READ_OBJECT	0H	Transport	Remote/Local	Query or change layer database objects
SET_OBJECT	2H			
READ_AND_CLEAR_OBJECT	1H			
Event Notification				
AWAIT_EVENT	0AH	None	Local	Supply buffer to receive event notification
Debugging Operations				
READ_MEMORY	03H	Transport	Remote/Local	Read or Set memory of target agent host
SET_MEMORY	04H			
Maintenance Operations				
ECHO	06H	Data Link	Remote	Test network path to target agent
DUMP	05H	Data Link	Remote	Read memory of target agent host
Remote Boot Loading				
SUPPLY_BUFFER	8H	Data Link	Local	Supply or take back buffer to receive load data
TAKEBACK_BUFFER	9H			

ATTACH_AGENT

The ATTACH_AGENT command is used by a network manager to establish a connection to a local or remote network agent. This command is optional for setting up a local connection, but is mandatory for setting up a remote connection.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* attach_agent_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 80H */
    unsigned char       opcode;      /* 0BH */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct attach_agent_rb {
    RB_COMMON            header;
    unsigned short      reference;    /* output */
    unsigned long       address_buf_ptr; /* input */
} ATTACH_AGENT_RB ;
```

Input Arguments

address_buf_ptr

A pointer to (absolute address of) the buffer where the transport address of the remote net agent is stored. For connections to a local net agent, this value must be 0.

See also: Chapter 10 for information concerning the use of pointers and absolute addresses in iNA 960 request blocks

The format of a remote transport address is shown below.

```
struct address_buffer
    unsigned char    local_nsap_sel_len;
    unsigned char    local_nsap_sel [local_nsap_sel_len];
    unsigned char    local_tsap_sel_len;
    unsigned char    local_tsap_sel [local_tsap_sel_len];
    unsigned char    remote_nsap_addr_len;
    unsigned char    remote_nsap_addr [remote_net_addr_len];
    unsigned char    remote_tsap_sel_len;
    unsigned char    remote_tsap_sel [remote_tsap_sel_len];
};
```

Where:

`local_nsap_sel_len`

The length of the local NSAP selector. The value for iNA 960 configurations is 1.

`local_nsap_sel`

The local NSAP selector. The value for iNA 960 configurations is 0.

`local_tsap_sel_len`

The length of the local TSAP selector. The value for iNA 960 configurations is 2.

`local_tsap_sel`

The configured value for the local NMF TSAP selector. The value for iNA 960 configurations is 0300H.

`remote_nsap_addr_len`

The length of the remote target agent's NSAP address (including an NSAP selector of 0, which is the last byte of the address).

`remote_nsap_addr`

The target agent's NSAP address.

See also: NSAP addresses, Chapter 8

`remote_tsap_sel_len`

Must be set to 2.

`remote_tsap_sel`

The configured value for the remote NMF TSAP selector. The value for iNA 960 configurations is 0300H.

The addressing conventions presented here are valid even in cases where the target net agent node is reached through a network router.

Output Arguments

reference

A unique 16-bit number returned by the ATTACH_AGENT command that identifies the connection to the net agent. If `address_buf_ptr` was specified as 0 (local operation only), a 0 is returned in this field.

Response Codes

OK_RESPONSE	1H	Operation completed successfully.
E_NO_RESPONSE	2H	No response from remote net agent.
E_CONNECTION	8H	An error occurred while attempting to establish a connection with a remote net agent.
NOT_CONFIGURED	0AH	The requested command is not configured for the net manager.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_MAX_CONN	16H	The maximum number of connections permitted for the net manager are currently open. Currently, only one open connection is permitted for each net manager.
E_NO_NMF	0FFEH	No NMF available.

AWAIT_EVENT

The AWAIT_EVENT command posts a buffer that is filled and returned by iNA 960 when a layer event occurs. This command only records events from local net agents. Only one event is recorded for each AWAIT_EVENT buffer posted. Post multiple buffers to record multiple events.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of await_event_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 80H */
    unsigned char       opcode;      /* 0AH */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct await_event_rb {
    RB_COMMON           header;
    unsigned short      reference;    /* input */
    unsigned short      filled_length; /* output */
    unsigned long       event_buf_ptr; /* in/out */
    unsigned short      event_buf_length; /* input */
} AWAIT_EVENT_RB;
```

Input Arguments

reference

Specify 0, because event notification does not use a VC connection.

event_buf_ptr

A pointer to (absolute address of) the AWAIT_EVENT buffer.

event_buf_length

The length, in bytes, of the AWAIT_EVENT buffer.

Output Arguments

`filled_length`

The length, in bytes, of the data recorded in the AWAIT_EVENT buffer. The NMF updates the field once the AWAIT_EVENT command is executed.

`event_buf_ptr`

The event information is returned in the buffer, with this structure:

```
typedef struct await_event_buffer {
    unsigned short    event_len;
    unsigned short    event_id;
    unsigned char     event_time[17];
    unsigned short    reset_time_ctr;
} AWAIT_EVENT_BUFFER;
```

Where:

`event_len` The length, in bytes, of the data recorded in the buffer.

`event_id` The ID number of the NMF event, specifying the iNA 960 layer and the event number.

See also: Event IDs, Appendix C

`event_time`

The time the event occurred, specified as an ASCII string that shows Greenwich Mean Time (GMT). The format of the string is:

```
YYMMDDhhmmss[+/-]hhmm
```

Where YY = year (0-99), MM = month, DD = day, hh = hours, mm = minutes, and ss = seconds. The [+/-] hhmm is an offset, in hours and minutes, from GMT to local time.

`reset_time_ctr`

A counter that indicates how many times the net agent's system time has been set.

Response Codes

OK_RESPONSE	1H	Operation completed successfully.
E_OK_COMMAND	5H	Command completed. The request block was returned without a posted event.
E_CONNECTION	8H	Connection error.
NOT_CONFIGURED	0AH	No local net manager.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_REFERENCE	14H	Incorrect reference number.
E_NO_NMF	0FFE H	No local NMF.

DETACH_AGENT

The DETACH_AGENT command is used by a net manager to remove an established connection to a remote or local net agent. If the connection is to a remote agent, the DETACH_AGENT command ends the connection and returns all unused AWAIT_EVENT buffers to the Network Administration application software. If the connection is to a local agent, the DETACH_AGENT command simply returns unused AWAIT_EVENT buffers to the local net manager.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* detach_agent_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;  /* 0FFH */
    selector            resp_mbox;  /* mailbox token */
    selector            rb_seg_tok; /* segment token */
    unsigned char       subsystem;  /* 80H */
    unsigned char       opcode;     /* 0CH */
    unsigned short      response;   /* initialize to 0 */
} RB_COMMON;

typedef struct detach_agent_rb {
    RB_COMMON           header;
    unsigned short      reference;  /* input */
} DETACH_AGENT_RB;
```

Input Arguments

reference

The connection reference code returned by the ATTACH_AGENT command.

Response Codes

OK_RESPONSE	1H	Operation completed successfully.
NOT_CONFIGURED	0AH	The requested command is not configured for the net manager.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_REFERENCE	14H	Unrecognized reference number.
E_NO_NMF	0FFEH	No local NMF.

DUMP

The DUMP command requests a snapshot of host memory from a remote net agent, beginning at a specified address. If the net agent does not respond, the net manager tries twice again before assuming that the net agent is not responding. The maximum size of the memory image that can be returned by the command is limited by the maximum packet size of the subnet linking the net manager to the net agent. Before using the DUMP command, the application must set aside a buffer to receive the host memory image. The buffer must be at least as large as the largest requested memory image.

Request Block

```
typedef struct rb_common {
    unsigned short      reserved[2];
    unsigned char       length;      /* of dump_rb */
    selector            user_id;     /* cq_create_comm_user */
    unsigned char       resp_port;   /* 0FFH */
    selector            resp_mbox;   /* mailbox token */
    selector            rb_seg_tok;  /* segment token */
    unsigned char       subsystem;   /* 80H */
    unsigned char       opcode;      /* 05H */
    unsigned short      response;    /* initialize to 0 */
} RB_COMMON;

typedef struct dump_rb {
    RB_COMMON           header;
    unsigned long       subnet_addr_ptr; /* input */
    unsigned short      filled_length;   /* output */
    unsigned long       buffer_ptr;      /* input */
    unsigned short      buffer_length;   /* input */
    unsigned long       start_address;   /* input */
} DUMP_RB;
```

Input Arguments

`subnet_addr_ptr`

A pointer to a buffer containing the subnet address of the target remote net agent. The structure of the subnet address is:

```
typedef struct subnet_address {
    unsigned char    host_id[6];
    unsigned char    nmf_lsap;
} SUBNET_ADDRESS;
```

Where:

`host_id` The Ethernet address of the target net agent.

`nmf_lsap` The link service access point (LSAP) of the target net agent's NMF. This parameter specifies the address of the target net agent's Data Link Layer. The LSAP for the iNA 960 NMF is 08H.

`buffer_ptr`

A pointer to a buffer that will be used by the net manager to store the memory image dumped from the target net agent.

`buffer_length`

The length, in bytes, of the buffer indicated by the `buffer_ptr` parameter.

`start_address`

The starting address in the host memory of the target net agent where the DUMP operation will be performed. The NMF expects a 32-bit value that is meaningful on the host system.

Output Arguments

`filled_length`

The size, in bytes, of the data supplied by the net agent. This field is updated by the NMF after the net agent executes the DUMP command.

Response Codes

OK_RESPONSE	1H	Operation completed successfully.
E_NO_RESPONSE	2H	No response from remote net agent.
E_PACKET_LENGTH	4H	The packet of data received from the net agent has an incorrect packet length field.
NOT_CONFIGURED	0AH	No local net manager.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_NO_NMF	0FFEH	No local NMF.

Additional Information

The DUMP command is similar to the READ_MEMORY command. The main difference between them is the mechanism they use to establish connections to net agents. The READ_MEMORY command uses the Transport Layer, which requires a prior ATTACH_AGENT command to establish a connection and a DETACH_AGENT command to remove the connection. The DUMP command uses the Data Link Layer. This does not require the ATTACH_AGENT command, but does require a subnet address.

ECHO

The ECHO command determines whether a given remote net agent is present on the network and, if it is present, tests the communications path to the agent. The application specifies a count of random data bytes to transmit. If the net agent does not respond, the net manager tries twice again before assuming that the net agent is not responding. If the net agent responds, the net manager checks that the returned block of data exactly matches the transmitted block.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of echo_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 80H */
    unsigned char     opcode;      /* 06H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct echo_rb {
    RB_COMMON          header;
    unsigned long      subnet_addr_ptr; /* input */
    unsigned short    transmit_data_cnt; /* input */
    unsigned short    received_data_cnt; /* output */
} ECHO_RB;

subnet_addr_ptr
```

A pointer to a buffer containing the subnet address of the target remote net agent. The structure of the subnet address is:

```
typedef struct subnet_address {
    unsigned char     host_id[6];
    unsigned char     nmf_lsap;
} SUBNET_ADDRESS;
```

Where:

host_id The Ethernet address of the target net agent.

nmf_lsap The link service access point (LSAP) of the target net agent's NMF. This parameter specifies the address of the target net agent's Data Link Layer. The LSAP for the iNA 960 NMF is 08H.

transmit_data_cnt

The number of random bytes to transmit.

Output Argument

received_data_cnt

The number of bytes present in the request block returned by the target net agent.

Response Codes

OK_RESPONSE	1H	Operation completed successfully.
E_NO_RESPONSE	2H	No response from remote net agent.
E_DATA_MATCH	6H	Transmitted data and received data do not match.
NOT_CONFIGURED	0AH	No local net manager.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_NO_NMF	0FFE H	No local NMF.

Additional Information

The ECHO command uses the Data Link Layer rather than the Transport Layer. This means the command does not require a previous ATTACH_AGENT command, but does require a subnet address.

READ_AND_CLEAR_OBJECT

The `READ_AND_CLEAR_OBJECT` command returns the value of one or more iNA 960 objects. The request block and buffers for this command are the same as those for the `READ_OBJECT` command.

See also: `READ_OBJECT`, in this chapter

READ_MEMORY/SET_MEMORY

This command description applies to both the READ_MEMORY and SET_MEMORY commands. The READ_MEMORY command reads host memory from the net agent specified by the `reference` parameter. The SET_MEMORY command writes to the host memory.

The application must set aside a buffer in host memory before this command can be issued. For the READ_MEMORY command, the NMF writes the memory image to the buffer. For the SET_MEMORY command, the application writes the memory image to the buffer.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* of read_or_set_mem_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 80H */
    unsigned char     opcode;      /* 3H = READ_MEMORY
                                   4H = SET_MEMORY */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct read_or_set_mem_rb {
    RB_COMMON          header;
    unsigned short    reference;    /* input */
    unsigned short    filled_length; /* output */
    unsigned long     buffer_ptr;   /* in/out */
    unsigned short    num_bytes;    /* input */
    unsigned long     start_addr;   /* input */
} READ_OR_SET_MEM_RB;
```

Input Arguments

`reference`

The connection reference number returned by the ATTACH_AGENT command. For a local agent, the ATTACH_AGENT command is not necessary; specify 0.

`buffer_ptr`

A pointer to the buffer holding the memory image to write (for SET_MEMORY) or where the NMF writes the memory image (for READ_MEMORY).

`num_bytes`

The number of data bytes to read or write. The buffer pointed to by the `buffer_ptr` parameter must be at least as long as the `num_bytes` value.

`start_addr`

The starting address in the host memory of the target net agent where the operation will be performed. The NMF expects a 32-bit value meaningful to the host's message delivery mechanism (e.g., MIP).

Output Arguments

`filled_length`

For the READ_MEMORY command, this is the number of bytes of data stored in the response buffer, filled in by the NMF after executing the command.

`buffer_ptr`

For the READ_MEMORY command, the buffer holds returned data.

Response Codes

<code>OK_RESPONSE</code>	1H	Operation completed successfully.
<code>E_NO_RESPONSE</code>	2H	No response from remote net agent.
<code>E_CONNECTION</code>	8H	Connection error.
<code>NOT_CONFIGURED</code>	0AH	No local net manager.
<code>E_NMF_OPCODE</code>	0CH	The specified opcode field is not a valid NMF command.
<code>E_INSUFF_RESP_BUF</code>	0EH	Response buffer is too small.
<code>E_REFERENCE</code>	14H	Unknown reference.
<code>E_PROTOCOL_ERR</code>	18H	Protocol error in the reply from the target net agent.
<code>E_NO_NMF</code>	0FFEh	No local NMF.

READ_OBJECT/SET_OBJECT READ_AND_CLEAR_OBJECT

The READ_OBJECT command returns the value of one or more iNA 960 objects or events. Since the request block and buffers for this command are the same as those for the READ_AND_CLEAR_OBJECT and SET_OBJECT commands, this description applies to all three commands. READ_AND_CLEAR_OBJECT returns the current object value and sets the value to 0. SET_OBJECT changes the value of the specified objects. The command buffer referenced in the request block describes which object to act upon. iNA 960 fills in the response buffer with values read from the objects. For the SET_OBJECT command, the response buffer contains the new value of the object.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;        /* of nmf_object_rb */
    selector          user_id;       /* cq_create_comm_user */
    unsigned char     resp_port;    /* 0FFH */
    selector          resp_mbox;    /* mailbox token */
    selector          rb_seg_tok;   /* segment token */
    unsigned char     subsystem;    /* 80H */
    unsigned char     opcode;       /* 0H = READ_OBJECT
                                     1H = READ_AND_CLEAR
                                     2H = SET_OBJECT */
    unsigned short    response;     /* initialize to 0 */
} RB_COMMON;

typedef struct nmf_object_rb {
    RB_COMMON         header;
    unsigned short    reference;    /* input */
    unsigned short    filled_length; /* output */
    unsigned long     resp_buf_ptr; /* in/out */
    unsigned short    resp_buf_length; /* input */
    unsigned long     cmd_buf_ptr;  /* input */
    unsigned short    cmd_buf_length; /* input */
} NMF_OBJECT_RB;
```

Input Arguments

reference

The connection reference number returned by the ATTACH_AGENT command. For a local agent, specify 0.

resp_buf_ptr

A pointer or an absolute address to the response buffer.

resp_buf_length

The length of the response buffer.

cmd_buf_ptr

A pointer or an absolute address to the command buffer.

See also: Buffer format, later in this description

cmd_buf_length

The length of the command buffer.

Output Arguments

filled_length

The number of bytes of data stored in the response buffer. This value is filled in by the NMF after it executes the command.

resp_buf_ptr

The buffer contains the value returned by this command. For a SET_OBJECT command, the new value is returned.

See also: Buffer format, later in this description

Response Codes

OK_RESPONSE	1H	Operation completed successfully.
E_CONNECTION	8H	Connection error. This usually means that the remote address was incorrect.
NOT_CONFIGURED	0AH	No local net manager.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_INSUFF_RESP_BUF	0EH	Response buffer is too small.
E_LAYER_NOT_SUPP	10H	Layer not supported.
E_OBJ_MIX	12H	Bad mix of objects. Each request block can contain objects from only one layer.

E_REFERENCE	14H	Unknown connection reference in the reference parameter.
E_PROTOCOL_ERR	18H	Protocol error in the reply from the target net agent.
E_INTERNAL_BUF	1AH	Fatal Error, connection removed. The application must reattach the target agent.
E_NO_NMF	0FFEH	No local NMF.

Additional Information

The application must set aside a command buffer and a response buffer in host memory before invoking one of these commands. The command buffer specifies the list of objects that are subject to the command and, in the case of the SET_OBJECT command, contains the new object values. Create and fill in the command buffer before issuing the command.

The response buffer is used by the NMF to return object values and command execution status. For each object in the response buffer, there is a status code field. If the status code is 0 (successful completion), the value of the object is returned in the value field of the structure. If the status code is not 0, the value may or may not have been returned, depending upon the status code. The length field of the structure indicates whether or not a value was returned.

If the response code returned in the request block is equal to 1, the command was successfully executed for each object specified in the command buffer.

Usually, if the response code is not equal to 1, the net agent could not execute the command and nothing is returned in the response buffer. For example, if the net agent does not respond to a command, the net manager times out, a connection error occurs, and the response code will not be equal to 1.

If multiple objects specified in the command buffer are not in the same layer, the NMF returns the response code 12H. This error indicates that the net agent could not execute the command on all of the objects requested, but has executed the command up to the point where a different layer's object was encountered. The response buffer will contain the objects that were acted on prior to the error.

Command Buffer Format

```
typedef struct obj_cmd_info {
    unsigned short    object;
    unsigned short    modifier;
    unsigned short    length;
    unsigned char     value[1];    /* set to length */
} OBJ_CMD_INFO;

typedef struct command_buffer {
    unsigned char     num_obj;
    OBJ_CMD_INFO      obj_info[1]; /* set to num_obj */
} COMMAND_BUFFER;
```

Where:

- num_obj** The number of objects included in the buffer.
- object** The ID code for an object. All objects in the command buffer must be in the same layer.

See also: ID codes for iNA 960 objects, Appendix C
- modifier** For virtual circuit connection-dependent objects, specify the connection reference or router table entry whose values are to be accessed. For any other object, specify 0. Virtual circuit connection IDs can be found by reading Transport Layer object 4001H, which returns an array containing the connection references for all established VCs.
- length** The length of the `value` field, in bytes. For the `READ_OBJECT` and `READ_AND_CLEAR_OBJECT` commands, set `length` to 0.
- value** For a `SET_OBJECT` command, specify the new value of the object. This field is ignored for the `READ_OBJECT` and `READ_AND_CLEAR_OBJECT` commands.

See also: Object values, Appendix C

Response Buffer Format

```

typedef struct obj_resp_info {
    unsigned short    object;
    unsigned short    modifier;
    unsigned char     status;
    unsigned short    length;
    unsigned char     value[1]; /* set to length */
} OBJ_RESP_INFO;

struct response_buffer {
    unsigned char     num_obj;
    OBJ_RESP_INFO     obj_info[1]; /* set to num_obj */
} RESPONSE_BUFFER;

```

`num_obj` The number of objects included in the buffer.

`object` The ID code for an object.

`modifier` For virtual circuit connection-dependent objects, this is the connection reference or router table entry whose values are to be accessed. For any other object, this field has no meaning.

`status` One of these status codes that indicates the success or failure of the requested operation:

Status	Description
0	Successful completion.
1H - 5H	Reserved.
6H	The object ID does not exist or is not supported.
7H	Incorrect access operation (e.g., an attempt was made to set a read-only object). See also: Appendix C for a listing of the access permissions for each object.
8H	Reserved.
9H	Incorrect parameter value. The object value specified is out of range.
0AH - 0DH	Reserved.
80H	Incorrect modifier.
81H	The response buffer is too small.
82H	End of routing table. The specified table index (modifier field) is past the end of the table.
83H	Routing table entry empty. The specified table index (modifier field) corresponds to an empty table entry.
84H	No free routing table entry. The requested table entry cannot be created because the table is full.

`length` The length of the `value` field, in bytes. If the operation was not successful (`status` not equal to 0), `length` may or may not be 0.

`value` The value of an object. For the `READ_AND_CLEAR_OBJECT` command, this is the value before the object is cleared. For the `READ_OBJECT` and `SET_OBJECT` commands this is the value after the object is read or set.

See also: Object values, Appendix C

SET_MEMORY

The SET_MEMORY command writes to the host memory on the net agent specified in the request block. This command uses the same structure as the READ_MEMORY command.

See also: READ_MEMORY, in this chapter

SET_OBJECT

The SET_OBJECT command changes the value of one or more iNA 960 objects. The request block and buffers for this command are the same as those for the READ_OBJECT commands.

See also: READ_OBJECT, in this chapter

SUPPLY_BUFFER

The SUPPLY_BUFFER command supplies a buffer to the NMF to receive a data packet destined for the application rather than the NMF. When the NMF receives a Data Link packet with the NMF LSAP, the NMF checks the command field. If the NMF does not recognize the command and a buffer has been supplied, the NMF places the packet in the buffer and returns the SUPPLY_BUFFER request block. A remote-load application can use this mechanism to communicate.

See also: Remote Load Operations, in this chapter,
Remote Booting, Chapter 15

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;        /* of supply_buf_rb */
    selector          user_id;      /* cq_create_comm_user */
    unsigned char     resp_port;    /* 0FFH */
    selector          resp_mbox;    /* mailbox token */
    selector          rb_seg_tok;   /* segment token */
    unsigned char     subsystem;    /* 81H */
    unsigned char     opcode;       /* 8H */
    unsigned short    response;     /* initialize to 0 */
} RB_COMMON;

typedef struct supply_buf_rb {
    RB_COMMON         header;
    unsigned short    filled_length; /* output */
    unsigned long     buffer_ptr;    /* in/out */
    unsigned short    buffer_length; /* input */
} SUPPLY_BUF_RB;
```

Input Arguments

`buffer_ptr`

A pointer to (absolute address of) the supplied buffer.

`buffer_length`

The length, in bytes, of the supplied buffer.

Output Arguments

`filled_length`

The amount, in bytes, of the supplied buffer actually taken up by the message packet.

buffer_ptr

The buffer contains a packet of information, with the format shown below. The packet header is put in the buffer so that the application will know the source of the message. The header is in the same format as an IEEE 802.2 Type 1 subnet packet header, regardless of what type of Data Link service delivers the message.

```
typedef struct supply_buff {
    unsigned char    destination_addr[6];
    unsigned char    source_addr[6];
    unsigned short   length;
    unsigned char    dest_lsap_sel;
    unsigned char    src_lsap_sel;
    unsigned char    control;
    unsigned char    data[1];
} SUPPLY_BUFFER;
```

Where:

destination_addr

The Ethernet address of the message destination.

source_addr

The Ethernet address of the message source.

length

The total length of the remaining fields in the buffer (dest_lsap_sel through data).

dest_lsap_sel

The NMF LSAP selector at the destination, which is 8H.

src_lsap_sel

The NMF LSAP selector at the source, which is 8H.

control

This field is set to 3H.

data

The data sent in the message packet.

Response Codes

OK_RESPONSE	1H	Command completed successfully; this buffer contains a returned packet.
E_OK_BUFF_RELEASE	3H	Rather than being used, the buffer has been released with a TAKEBACK_BUFFER command.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.
E_INSUFF_RESP_BUF	0EH	The buffer is too small. This occurs when the buffer is too small to even hold the packet header. The buffer must be at least 18 bytes long.

Additional Information

The supplied buffer must be at least the length of the largest packet expected. If the buffer is shorter than the length of a received packet, one of two things can happen:

- If the buffer is too short to hold the packet header, the entire packet is discarded; no error is returned to the source.
- If the buffer is long enough to hold the packet header but too short to hold the data, the header is put in the buffer and the rest of the packet is discarded; no error is returned to the source.

TAKEBACK_BUFFER

The TAKEBACK_BUFFER command releases a buffer previously supplied to the NMF with the SUPPLY_BUFFER command.

Request Block

```
typedef struct rb_common {
    unsigned short    reserved[2];
    unsigned char     length;      /* takeback_buf_rb */
    selector          user_id;     /* cq_create_comm_user */
    unsigned char     resp_port;   /* 0FFH */
    selector          resp_mbox;   /* mailbox token */
    selector          rb_seg_tok;  /* segment token */
    unsigned char     subsystem;   /* 81H */
    unsigned char     opcode;      /* 9H */
    unsigned short    response;    /* initialize to 0 */
} RB_COMMON;

typedef struct takeback_buf_rb {
    RB_COMMON          header;
} TAKEBACK_BUF_RB;
```

Response Codes

OK_RESPONSE	1H	Command completed successfully.
E_NMF_OPCODE	0CH	The specified opcode field is not a valid NMF command.

□□□

This chapter explains the process of remote booting and how to configure the computers involved in the process. The discussion also covers how to configure and use software for *diskless nodes*, which have no local mass storage devices. Computers that are booted remotely do not have to be diskless nodes, but they often are.

The bootstrap loading process (or booting) loads software into a computer's RAM using a program called the Bootstrap Loader. *Remote booting* loads a software image obtained across the network from a remote mass storage device, rather than one stored on a local disk. Remote booting is the standard way to boot diskless nodes, but you can boot any iRMX computer this way. This makes it possible to control common application software on a number of computers, by loading all the software from one *boot server*. The boot server is a computer that provides software to other nodes requesting a remote boot. A computer whose memory is loaded during remote booting is called the *boot client*. If the boot client is a diskless node, it also needs a *file server*. A file server is a computer that provides remote access to the files on its local hard disk to other nodes on the network.

Hardware and Software Requirements

Boot server This can be any computer running the iRMX OS, iNA 960 software, and the Boot Server software, which is supplied with the OS as a loadable job (*rbootsrv.job*). Note that this is the Remote Boot Server, not the MSA Boot Server (which is *bootserv.job*, used for booting within a Multibus II chassis).

See also: Creating Custom Server Applications, in this chapter

File server This can be any computer running the iRMX OS, iNA 960 software, and the iRMX-NET server (*rnetsrv.job*). The File Server software is one of the basic modules of iRMX-NET. The file server and boot server can be a single node that provides both file and boot services.

Boot client This is often a diskless node, but it does not have to be. Table 15-1 lists the CPU boards and NICs that can be used for a boot client.

Table 15-1. Boot Client Systems

System Bus*	CPU	NIC	Operating System**
PC	386 or higher	EtherExpress 16	RPC or RFW
MB2	SBC 486DX33 SBC 486DX66 SBC 486SX25	EWENET module	RPC
MB1	SBC 386/2X SBC 386/3X SBC 386/12 SBC 486/12	SBC 552A	III

* MB1 specifies Multibus I; MB2 specifies Multibus II; PC specifies PC Bus

** III specifies iRMX III OS; RPC specifies iRMX for PCs; RFW specifies DOSRMX.

For simplicity, this chapter often refers to the three remote boot client configurations as the Multibus I, Multibus II, and PC Bus systems. Remember, however, that remote booting also requires one of the supported CPU boards and NICs listed in Table 15-1 and that each hardware configuration runs only one or two versions of the iRMX OS. The following requirements also apply:

As a general rule, the boot client must contain, in PROM, an iRMX remote first stage bootstrap loader. However, this is not necessarily true for PCs.

PC Bus and Multibus II systems; there are two possible methods:

- The boot client NIC contains, in ROM, an iRMX remote first stage bootstrap loader. You create the ROM using software supplied with the iRMX OS and the instructions later in this chapter.
- To remotely boot DOSRMX on a PC or PC-compatible Multibus II board, you can boot DOS from a local disk or diskette. Then you can use the `loadrmx` command to remotely boot DOSRMX.

See also: **loadrmx**, *Command Reference*

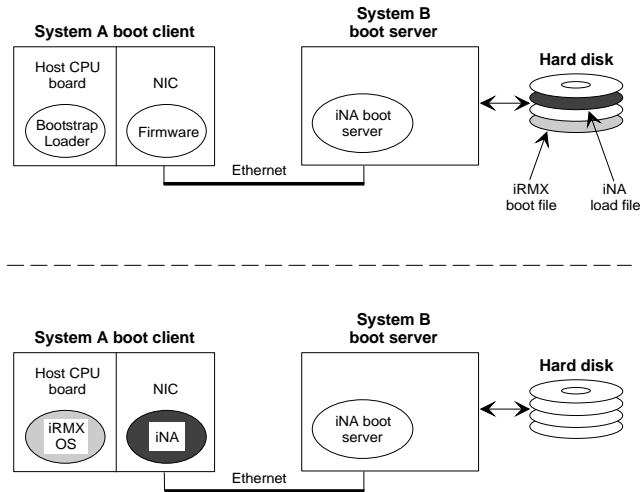
Multibus I systems

The CPU boards that support remote booting come with the iRMX remote first stage bootstrap loader in PROM. The first stage includes the remote device driver for the 552A board.

See also: *Bootstrap Loader Reference* for more about remote booting Multibus I systems

Overview of Remote Booting

Figure 15-1 shows a boot client and boot server before remote booting and again afterwards. The client is a Multibus I system, with the networking software running on a separate NIC.



iRMX is a registered trademark of Intel Corporation.

W-0918

Figure 15-1. Remote Booting the iRMX III OS, Start and Finish

The top diagram includes all the pieces required for remote booting the iRMX III OS:

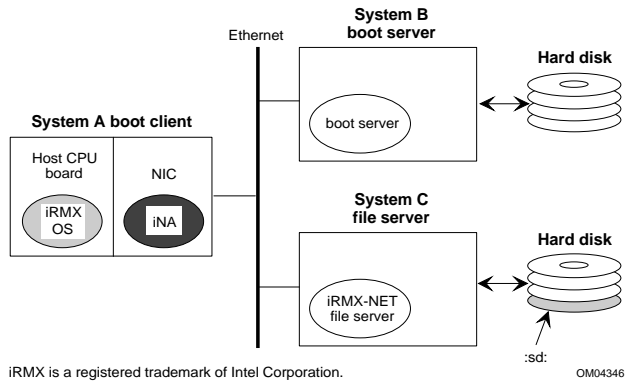
- A boot client (System A) with a first stage Bootstrap Loader and other remote boot firmware in PROM
- A boot server (System B) running the iNA Remote Boot Server software, which stores the files to be loaded onto the boot client: the iRMX III OS boot file, a remote third stage Bootstrap Loader, and an iNA 960 file

In the bottom diagram of Figure 15-1, booting is finished, and the iRMX OS and iNA 960 are running on the boot client.

If the boot client is a diskless node, during remote booting it attaches a file server and assigns it the logical name `:sd:`. Any references to `:sd:` on the diskless node use the remote disk provided by the file server.

Figure 15-2 shows the finish of remote booting when the boot client (System A) is a diskless node. During the process it has attached a file server (System C) across the

network. Here the file server (System C) and boot server (System B) are two separate nodes, but the same node could serve as both.



iRMX is a registered trademark of Intel Corporation.

Figure 15-2. Remote Booting a Diskless Node

In brief, the process of remote booting is:

1. You start by resetting the boot client.
2. The boot client sends a remote load request across the subnet. The request is a multicast message, so it does not go through routers into other subnets.
3. Each boot server that receives the message checks to see if it can service the request. If it can provide the service, it returns an accept message to the boot client; if it cannot, it does not respond.
4. Once it receives an accept message, the boot client notes the source address and sends a series of data request messages to that boot server.
5. The boot server responds with messages containing the requested data.
6. As it receives the data messages from the boot server, the boot client places the data into memory.
7. When the last of the data is loaded, the boot client software knows that loading is complete.

See also: [Creating Custom Server Applications](#), in this chapter, for more detail on the remote booting process

Configuring the Load Files

The software loaded onto the boot client varies with the version of the OS to be booted, as shown in Table 15-2. For booting, the files must be on the boot server. If you need new versions of the files, however, you can generate them on another host and copy them to the boot server. You might need to do this if the server does not have an ICU.

Table 15-2. Load Files for Remote Booting

File	iRMX III OS	iRMX for PCs	DOSRMX
iRMX OS boot file <i>*rsd.bck</i>	required	required	required
load-time configuration file <i>rmx.ini</i>	none - ICU configurable	optional	optional
remote 3rd stage bootstrap loader <i>*.rem32</i>	required	required	none - loaded from DOS
iNA 960 load file <i>ina*.32r</i>	optional	none - uses COMMputer	none - uses COMMputer

Operating System Boot File

The only file that is always required for remote booting is the OS boot file.

Generating an OS Boot File

To generate an OS boot file, configure a definition file using the ICU. Table 15-3 shows the remote boot definition files provided with the OS in the `/rmx386/icu` directory. The ICU restores the backup version of the file to a definition file with the extension `.def` instead of `.bck`. The `rsd` in the filenames means they are intended for use with a remote system device, which is the standard configuration for remote boot clients. The files contain all of the iRMX subsystems and the iRMX-NET File Consumer module, but not the iRMX-NET File Server software.

Table 15-3. ICU Definition Files for Remote Booting

File	CPU
38620rsd.bck	SBC 386/2X,3X
38612rsd.bck	SBC 386/12
48612rsd.bck	SBC 486/12
pccprsd.bck	PC with EtherExpress 16

⇒ **Note**

The `pccprsd.bck` file can run the DOSRMX OS (with DOS present) or iRMX for PCs (without DOS). You can also use this file on an SBC 486SX/DX board with an EWENET module. To do so, you need to make only one change.

Use the ICU to configure `pccprsd.bck`. Change the `iNA 960` file configured into the OS with the `OFN` parameter on the `ICMPJ` screen. For the EWENET module, specify either the `iewexpn` (NULL2) job or `iewexpe` (ES-IS) job, instead of the `iethxpn(e)` job used with the EtherExpress 16.

See also: *i*.job, System Configuration and Administration*

ICU Configuration

To create a diskless configuration of the iRMX III OS, generate the iRMX OS with the ICU in the usual way. You must change several ICU screens to replace the default file server name, FILESRV, with the name of the node you will use as the file server. Use its server name, which is registered with the Name Server.

See also: **listname** command, *Command Reference*

Invoke the ICU, using the definition file in Table 15-3 that matches the boot client's hardware, and make these changes:

1. For Multibus I boot clients only, on the Logical Names (LOGN) screen add this entry, where *server_name* is the name of the file server:

```
SD, server_name, REMOTE, 0H
```

⇒ Note

Only perform this step if ABR (automatic boot device recognition) is set to NO in the EIOS screen (this step applies only for remote booting a Multibus I client). If ABR=YES in the EIOS screen and you perform this step, the OS initialization will fail with EIOS initialization error 0005H.

2. On the User Definition File (UDF) screen, specify this parameter:

```
(MD) Master UDF Device    server_name
```

3. On the Client Definition File (CDF) screen, specify this parameter:

```
(CDD) CDF Device        server_name
```

You can also use the CDF screen to set a unique client name and password for the boot client board. Both parameters are case-sensitive. The default entries are *rmx* and *1234567*. You can use the defaults, but every diskless client using this configuration will have the same client name and password.

4. The definition file assumes that the file server is running the same version of the iRMX OS as the boot client. If it is a different version, specify the correct pathname for the configuration directory on the EIOS screen.

```
(CD) Configuration Directory path_name
```

The ICU generates an OS file in the local directory. The file has the same name as the definition file, but with a *.rem* extension.

5. Verify on the following screens that these parameters are set:

Screen	Parameter	
EIOS	ABR=Yes for a PC or Multibus II, ABR=No for Multibus I	
ABDR	DPN = <server name>	(only applies when ABR=Yes)
ABDR	DFD = Remote	(only applies when ABR=Yes)
MIP1	LD = No	(only applies when ABR=No)
GEN	RMB = Yes	
6. Invoke the generation submit file created by the ICU, **.csd*. This creates a boot file in the */rboot32* directory. The typical filename extensions are *.rem* or *.386*, as shown in Table 15-4.

The boot file is now ready for the default remote booting process. If you intend to boot it without an iNA 960 load file, instructions are provided later in this chapter.

See also: Creating the *ccinfo* File, in this chapter;
 User Definition File and Client Definition File, Chapter 2;
 ICU User's Guide and Quick Reference

Load-time Configuration File

For iRMX for PCs and DOSRMX only, use the load-time configuration file, *rmx.ini*, to override default parameters. You need a separate file if you are using different nodes for the boot server and the file server. By default, the OS looks for its system files on the boot server. You can also make any other optional changes in this file, like resetting the default client name and password.

To prepare a configuration file for remote booting:

1. Copy the default *rmx.ini* file to the */rboot32* directory. This file will be used by the boot client; be sure it is not a version that has been configured for the local node.

2. Edit the file server name:

```
DN = 'SD'           ; Device Name
```

where *SD* is the name of the file server, as registered with the Name Server.

3. You can also specify a unique client name and password for the boot client. Both parameters are case-sensitive:

```
CNN = 'client_name' ; Consumer Name
CNP = 'password'    ; Consumer Password
```

The default Consumer Name is 'rmx' and the default Consumer Password is '1234567'. You can use the defaults, but every diskless client using this configuration would have the same client name and password.

- In the EIOS block, add these parameters, where *server_name* is the name of the file server:

```
ADV = 'server_name'
AFD =05H
```

- In the HI block, add this parameter, where *init_file* is a file in the *:config:* directory to use as a replacement for *r?init:*

```
SCF = 'init_file'
```

- Translate the file into iNA load file format, using the **remini** command:

```
remini rmx.ini to rini_at.rem
```

This creates a configuration file for a PC Bus system. For Multibus II, use the filename *rini_mb2.rem*

See also: Load-time configuration, *System Configuration and Administration*; **remini** command, *Command Reference*

Remote Third Stage Bootstrap Loader

The *remote third stage* is provided with the iRMX Bootstrap Loader in the */bsl* directory. It is used for booting remote clients with iRMX for PCs or iRMX III OS. On DOSRMX systems, DOS loads the iRMX OS.

Third stage file names have a *.rem32* extension. Table 15-4 shows all the files and the OS boot files they are used with.

Table 15-4. Remote Third Stage Bootstrap Loader Files

Third Stage File	Default OS Boot File
exp.rem32	pccprsd.rem
38620.rem32	38620rsd.386
38612.rem32	38612rsd.386
48612.rem32	48612rsd.386

⇒ Note

When installed on a DOS file system, the third-stage filename extensions are truncated to *.rem* from *.rem32*.

On the iRMX III OS you can configure and generate a new third stage. Use the */bsl/br38.csd* submit file to create a Multibus I third stage, or */bsl/br3expgen.csd* for a Multibus II or PC Bus third stage.

To remote boot, the remote third stage that matches the OS boot file must be in the */rboot32* directory. Copy the appropriate file from the */bsl* directory. For example:

```
copy /bsl/exp.rem32 to /rboot32/exp.rem32
```

The remote third stage is different from other iRMX third stages in that it does not really load the OS into memory. The first stage and the firmware on the boot client, working with the iNA Remote Boot Server software, actually load both the OS and the remote third stage. The remote third stage receives control directly from the first stage and puts the processor into protected mode before starting execution of the OS. (Remote booting does not use a second stage, unlike booting from a local disk.)

See also: *Bootstrap Loader Reference* for more on the third stage for Multibus I

iNA 960 Load File

Preconfigured iNA 960 remote load files are provided with the OS for boot clients. Use the */net/ina*.32r* files, which are iNA 960 load file format. The *ina*.32l* files are for local use. You can boot the iRMX III OS without loading iNA 960, but this requires changes to the *ccinfo* and OS files.

Remotely booted iRMX for PCs and DOSRMX systems use a COMMMputer configuration. They do not have a separate NIC to be downloaded.

See also: *Creating the ccinfo File*, in this chapter

Generating a First Stage EPROM for the Boot Client

The boot client has no software to configure, but it must contain, in EPROM, an iRMX remote first stage bootstrap loader.

The hardware configuration required to remote boot the iRMX III OS includes Multibus I, the SBC 552A NIC, and CPUs that come with the first stage in ROM. The first stage includes the remote device driver R0 for the SBC 552A NIC. You can replace the first stage with one that you have configured.

See also: *Configuring the first stage, Bootstrap Loader Reference*, for Multibus I

Creating a First Stage for EtherExpress 16 or EWENET

For booting iRMX for PCs and DOSRMX, the first stage must be on the NIC. You create an EPROM for the EtherExpress 16 NIC (PC Bus) or EWENET module (Multibus II), using software supplied with the iRMX OS. This involves editing the configuration file, generating the first stage files, and burning a new EPROM.

On an iRMX III computer with an EPROM burner attached, complete these steps:

1. Move to the */bsl* directory to generate the EPROM.
2. Configure the *bexp.a86* file, if necessary. If you intend to boot iRMX for PCs on a Multibus II system and you want the bootstrap loader menu to be displayed, you can use the default configuration. If you need to edit the file, make a backup copy. Be sure you are not changing comment lines that begin with a semicolon (;) instead of the command lines.
 - To change the class code, replace the value 4001 in following line with one of the entries in the *:sd:net/ccinfo* file.

```
default_class_code DB '4001',00H
```

See also: Creating the ccinfo File, in this chapter

- To change the default menu display or check for a display adapter, replace the value OFFH in following line with the one appropriate to your system.

```
cmos_check_byte DB 0FFH ; Display Menu
```

Use 000H to suppress the menu, 01FH to indicate a Phoenix BIOS keyboard, or 014H for Standard BIOS equipment. If you use *cmos_check_byte* value 01FH or 014H to check for a display adapter, you also need to set the *cmos_check_mask* to specify whether the menu is displayed.

```
cmos_check_mask DB 001H ; Phoenix BIOS keyboard
```

A logical **and** is performed on the *cmos_check_byte* value 01FH or 014H and the *cmos_check_mask* value. If the result is not 0, the menu is displayed.

- If you change the *default_class_code*, *cmos_check_byte*, or *cmos_check_mask* parameters, also change the value in this line so that the file produces a checksum ending in 00 when you program the EPROM.

```
checksum_fix DB 0B8H
```

3. Invoke the submit file *bexp.csd*.

```
submit bexp
```

This step generates a located object file *bexprb.loc* for use with the iPPS PROM Programmer and a HEX file *bexprb.hex* for use with third party PROM programmers. This warning appears; this is normal.

```
WARNING 66: START ADDRESS NOT SPECIFIED IN OUTPUT MODULE
```

4. Program the EPROM.
 - If you are using a third party PROM programmer, move the *bexprb.hex* file to the proper location and follow the vendor's instructions to create the EPROM.
 - If you are using the iPPS PROM Programmer, follow the instructions in the next section.
5. If the process of programming the EPROM produces a checksum that does not end in 00, adjust the value of `checksum_fix` in the *bexp.a86* file and repeat the process.

Using the iPPS PROM Programmer

To place the first stage into an EPROM device, stay in the */bsl* directory and complete these steps:

1. Attach the physical device as *ipps*.

```
ad terminal_device as ipps p
```

Where *terminal_device* is the terminal device name for your system, for example, *t82530_0*.

2. Set the baud rate of the terminal to 2400 baud.

```
term :ipps: in=2400
```

3. Invoke the iPPS PROM Programmer:

```
ipps
```

4. At the iPPS prompt, enter:

```
i 86
t 2764
format bexprb.loc (92000)
3
1
1
0 to bexprb.rom
Y

copy bexprb.rom to b
copy b to bexprb.rom
Y
copy bexprb.rom to p
exit
```

The commands above are for programming a 2764 EPROM, using the *bexprb.loc* file prepared with the instructions in previous sections. This command (from the list above) copies the buffer back to a file.

```
copy b to bexprb.rom
```

The checksum reported in this step must end in 00, or the ROM BIOS will not execute the code. If necessary, adjust the *checksum_fix* variable in the *bexp.a86* configuration file and try again.

Installing the EPROM

To install the remote first stage in the boot client, complete these steps:

1. Plug the EPROM into an EtherExpress™ 16 or EWENET board.
2. Place the board in the boot client or, if it is diskless, in a computer with a disk drive.
3. Run the SOFTSET program from the DOS prompt to configure the board. Use the Manual Setup option to configure the board for the boot client. Set the boot ROM address parameter to any available address range.
4. Move the board to the boot client, if you configured it somewhere else.

See also: Manual configuration and installation in diskless workstations, *The Complete Guide to installing and configuring the Intel EtherExpress 16 and 16TP Network Adapters for ISA computers*

Configuring the Remote Boot Server

The boot server downloads OS and network communication software to remote nodes. It can handle simultaneous requests from multiple boot clients. The maximum number is an iNA 960 configuration option.

These are the basic steps for setting up a boot server:

1. Create the `:sd:net/ccinfo` file.
2. Sysload the boot server job `rbootsrv.job` using the proper command line parameters.

See also: `rbootsrv.job`, *System Configuration and Administration*

3. Place the load files in the directories specified in the `ccinfo` file.

Creating the `ccinfo` File

Every boot server must have a Class Code Information (`ccinfo`) file. This tells the server which files to send in response to a boot client's request, and in what order. The boot server reads the `ccinfo` file during initialization.

The `:sd:net/ccinfo` file is a binary file generated by the `bcl` utility from a template file, `:sd:net/ccinfo.bdf`. Each line of this file is a predefined entry for a different class code.

Class Codes

A *class code* is a 16-bit number that determines which files the boot server sends to a boot client, and in what order. Table 15-5 shows the default codes predefined in the *ccinfo.bdf* file. You can define your own class codes within the specified ranges.

Table 15-5. Class Code Ranges and Defaults

Host/Comm Images	Class Code	Default Value
Available for applications	0000H - 0FFFH	0000H
iNA 960	1000H - 1FFFH	1000H
DOS	2000H - 2FFFH	2000H
iRMX OS	3000H - 4FFFH	
III*	MB1**/386	3000H
III	MB1/486	4000H
RPC*	MB2**	4001H
RPC	PC**	4002H
RFW*	PC	4003H

* III specifies iRMX III OS; RPC is iRMX for PCs; RFW is DOSRMX

** MB1 specifies Multibus I; MB2 is Multibus II; PC is PC Bus

Every boot request includes a class code. When the boot client sends a boot request, the boot server checks whether it has a definition for the class code. If it does, the server sends all the files associated with that code to the boot client.

For example, the load files for booting a Multibus I boot client consist of a remote third stage, the OS, and usually an iNA 960 file. As you can see in the *ccinfo.bdf* file in Figure 15-3, by default all three files are matched to codes 3000H and 4000H. You could define a code 4004H to match only the first two files. A client broadcasting the code 4004H would receive only the third stage and OS files from the boot server.

```
4000 IS /rboot32/48612rsd.rem32, /rboot32/48612rsd.386,  
/net/ina552an.32r;  
  
4001 IS /rboot32/exp.rem32, /rboot32/rini_mb2.rem, /rboot32/pccprsd.rem;  
4002 IS /rboot32/exp.rem32, /rboot32/rini_at.rem, /rboot32/pccprsd.rem;  
4003 IS /rboot32/pcexprsd.rem;  
  
3000 IS /rboot32/38620rsd.rem32, /rboot32/38620rsd.386,  
/net/ina552an.32r;  
  
1032 IS /net/ina552an.32R;  
1033 IS /net/ina552ae.32R;
```

Figure 15-3. The :sd:net/ccinfo.bdf File

The files are sent in the order they are listed in the class code entry. This is important, because most of the files are translated with the N flag set on the **xl**ate command, which tells the boot client to look for the next file. When a file translated without the flag arrives, the boot client stops sending data messages to the server, and loading stops.

Both predefined and custom-generated files are translated to fit the order of the default *ccinfo.bdf* entries, as shown in Table 15-6.

Table 15-6. Remote Load File Translation

File	Xlate N Flag	Order of Loading
*.rem32 remote third stage	yes	first
rini_*.rem load-time configuration file	yes	before the OS
iRMX OS boot file		
*rsd.386 III**	yes	before iNA 960
pc*rsd.rem RPC** or RFW**	no	last
ina*.32r iNA 960 files	no	last

** III specifies iRMX III OS; RPC is iRMX for PCs; RFW is DOSRMX

However, two class codes in the *ccinfo.bdf* file, 1032H and 1033H, load iNA 960 separately. If you do this, you need to create a class code that loads the iRMX III OS without iNA 960. Add a line like this to the *ccinfo.bdf* file:

```
4004 IS /rboot32/myrsd.rem32, /rboot32/myrsd.386;
```

Then you must generate a new version of the iRMX III OS that is translated without the N flag. The ICU places the proper **xl**ate translation command line into the generation submit file it creates.

If you are not sure about the state of a particular file, the **unxl**ate command displays translation information about it.

See also: **xl**ate and **unxl**ate commands, *Command Reference*

Generating the *ccinfo* File

You can use the default entries in the *ccinfo.bdf* file or edit the file to change them.

1. Make a copy of the *:sd:net/ccinfo.bdf* file to use as a backup example.
2. Edit the file. Be sure not to change the order of the files in an entry.

If you are booting iRMX for PCs or DOSRMX without an *rini_at.rem* or *rini_mb2.rem* file, take the files out of the definitions for class codes 4001H and 4002H, or create a new code without it.

For example, you might have an DOSRMX system where you boot DOS locally but boot the iRMX OS remotely. In this case, you could include an *rmx.ini* file on the local DOS disk, and would not need a separate *rini_at.rem* file on the server.

3. Create the *ccinfo* file, using the **bcl** command:

```
attachfile :sd:net
bcl ccinfo.bdf ccinfo
```

See also: **bcl** and **inamon** commands, *Command Reference*

Make sure the *ccinfo* file is small enough to fit into the boot server's buffer in memory. The boot server is configured for a maximum file size of 1024 bytes.

See also: *rbootsrv.job*, *System Configuration and Administration*, to increase the default file size on the **sysload** command line

⇒ **Note**

If the *ccinfo* file is too large, no error message appears. Instead, only the bytes that fit into the buffer are loaded and any entries that do not fit are not supported by the boot server.

If the file is too large, use multiple boot servers, each supporting certain boot clients. Split the entries in the *ccinfo* file among the *ccinfo* files on different servers. Make sure they do not have different definitions of the same class codes. The boot client boots off the server that responds first, so the results would be unpredictable.

See also: iNA configuration values, Appendix A

Loading the Boot Server

After creating the *ccinfo* file, you must load the boot server. This reads the *ccinfo* file and tests the iNA Remote Boot Server software. If the boot server job is already running, first reboot the system. Then load *rbootsrv.job* with a **sysload** command.

See also: *rbootsrv.job*, *System Configuration and Administration*, for loading syntax and switches you can set on the command line

Installing the Load Files

Make sure the load files are on the boot server system, in the directories specified in the boot server's *ccinfo* file. Table 15-7 shows the defaults. The server uses the path names in the *ccinfo* file to find the load files.

Table 15-7. Default Directories for Load Files

File	Directory
iRMX OS boot file	:sd:rboot32
load-time configuration file	:sd:rboot32
remote third stage bootstrap loader	:sd:rboot32
iNA 960 file	:sd:net

At this point the boot server is ready to respond to requests from boot clients on the network.

Configuring the File Server

The file server is a node with a local hard disk that is used as a system device for a diskless node or a diskless host CPU. The server must provide remote access to all of the files that the iRMX OS and iRMX-NET assume are available during initialization. This means offering, as public directories, any root-level directories that are required. These public directories are listed in the file server configuration information later in this chapter.

Any iRMX OS generated using a default iRMX-NET configuration is ready to act as a file server. You do not need to make any special configuration changes, and the steps for setup are the same as for any other iRMX computer. If you are using a custom configuration of iRMX-NET, however, then these parameters must be set appropriately for the diskless boot client to initialize and function properly.

See also: Network jobs, *System Configuration and Administration*

This information outlines which iRMX-NET file server system parameters are important when the iRMX OS and iRMX-NET run in a diskless environment.

On the file server:

1. Make sure the server names of the file server and the boot client are in the Name Server object table.
2. Add the boot clients to the Client Definition File (CDF).
3. Enter the names of the boot clients and their terminal types into the *:config:terminals* file.

Loading Server Names into the Name Server Database

The server name of the file server must be available to the OS being booted, so the boot client can attach to the file server. This name should already be in the Name Server object table; use the **listname** command to check. If necessary, add this line to the `:sd:net/data` file:

```
fsname/nfs: TYPE=rmx: ADDRESS:;
```

where *f*sname is the server name of the file server.

See also: **listname** command, *Command Reference*

The boot client must also have a server name available, so the HI of the OS being booted can find its own name. It needs the name when it reads the `:config:terminals` file, which lists terminal types for diskless nodes by name.

To make this server a spokesman for the boot client, add a line like one of these to the `:sd:net/data` file:

```
mb1sys: TYPE=pt0005: ADDRESS=ssss#####00;  
mb2slot2: TYPE=pt0005: ADDRESS=ssss#####02;  
pcsys: TYPE=pt0005: ADDRESS=ssss#####00;
```

Where:

mb1sys The server names for boot clients, as defined in the ICU or the
mb2slot2 *rmx.ini* file during load file configuration.
pcsys

ssss The subnet ID configured into the iNA 960 job. The default number is 0001 (assuming the first subnet in multiple-subnet jobs). However, you can override the default by reconfiguring the job with the ICU or with an SNID parameter on the **sysload** command line when you load the job.

See also: *i*.job*, *System Configuration and Administration*

The Ethernet address of the boot client where the OS will run. PC Bus and Multibus II systems display the Ethernet address on the initial screen. On Multibus I systems, display the Ethernet address by attempting a remote boot.

The example lines above show a two-digit number following the Ethernet address. On a Multibus II system, specify the slot number of the boot client board. On a PC or Multibus I system, specify 00 after the Ethernet address.

Each boot client must have its own line in the file.

Reboot the file server or invoke the **loadname** command after you edit the `:sd:net/data` file.

See also: Adding a Server to the Name Server Object Table, Chapter 3;
Editing the `:sd:net/data.ex` File, Chapter 11, for more about the format of the entries

Adding Client Names to the CDF

Add the boot clients to the Client Definition File (CDF). The CDF on the file server must contain the client name and password for any boot clients. You defined the client name and password during configuration of the OS boot file on the boot server. The client name can be the same as the server name specified earlier, but it does not have to be. The client name and password are both case-sensitive.

See also: **modcdf** example, Chapter 5

Adding Server Names to the `:config:terminals` File

Enter the server name of the boot client and its terminal type into the `:config:terminals` file. The name must match the boot client's server name registered with the Name Server in the `:sd:net/data` file. For example:

```
//
1,<pcsys>
com1,,,any
//
1,<mb1sys>
t0,,,any
//
1,<mb2slot2>
t279_0,,,any
```

See also: Diskless workstations, *System Configuration and Administration*, for information about the format of the `:config:terminals` file

Remote Boot Start

When the software is in place on all of the nodes, and the file server and boot server are running, the remote booting can begin. Table 15-5 shows the predefined class codes you can use.

Booting Multibus I Systems

To boot the iRMX III OS on Multibus I clients, enter the SDM monitor by resetting the system. At the monitor prompt, specify a **boot** command line with a remote device name and a class code. For example:

```
b :r0: 4000H
```

Where:

:r0: The remote device name.

4000H The 16-bit class code. Make sure to add an H on the end for a hex value.

Leave a space between the remote device and the class code. If you omit the class code, the Bootstrap Loader uses the default value configured in the first stage.

See also: First stage configuration, *Bootstrap Loader Reference*

Booting Multibus II or PC Bus Systems

On PC Bus and Multibus II clients, a screen with a default class code appears when you reset the computer. Enter a different class code, if necessary, or allow the computer to reboot automatically using the default. A remote device name is not used.

System Initialization on a Diskless Node

During the initialization of the iRMX OS on the boot client, any file accesses go to the file server through iRMX-NET. The EIOS performs a logical **attachdevice** on the system device, but the first file access does not actually occur until the HI does an **attachfile** on that device. This happens when the HI creates the logical names specified in the ICU on the boot server when you configured the OS boot file. The HI also accesses the file server's *:config:terminals* file. If the user logs on as a dynamic user, the CLI accesses the file server to obtain the UDF file and any user files, such as *:prog:r?logon*.

During HI initialization, the system configuration file is invoked. On a diskfull node, the default system configuration file is `:config:r?init`, which submits the `:config:loadinfo` file. The `loadinfo` file contains **sysload** commands to load the system jobs.

Diskless nodes cannot use the `r?init` file or the `loadinfo` file; these typically apply to the server system. By default, the `*msd.bck` and `*rsd.bck` definition files (which apply to diskless nodes) are configured to use file `:config:initsrd` as the system configuration file, instead of `r?init`. If you want to use a different file than `:config:initsrd`, specify your file in the SCF parameter on the HI screen of the ICU or in the SCF parameter of the `rmx.ini` file (or your `rmx.ini` replacement for remote booting).

In the `:config:initsrd` file (or your alternate SCF file), submit a file like `:config:loadinfo` that contains **sysload** commands appropriate for the diskless node. For example, you might submit a file named `loadinfo.rsd`.

Regardless of what file is specified as the SCF, the HI initialization also submits a file of the same name with “2” appended, if such a file exists in the `:config:` directory. For example, the HI submits `:config:r?init2` on a diskfull node. With the default configuration of the `*msd.bck` and `*rsd.bck` definition files, the HI submits a file named `initsrd2` if it exists. If you change the system configuration file with the SCF parameter, the HI will submit your new SCF file as well as the same filename with “2” appended, if it exists.

⇒ **Note**

If you use the SCF mechanism on a DOS-controlled hard disk, specify an SCF filename with a maximum of 7 characters or with a maximum filename extension of 2 characters. This allows the HI to append a “2” and form a filename that fits the DOS 8.3 filename limits.

The `r?init2` file on a diskfull node typically submits these TCP/IP-related files with **esubmit** commands:

```
/etc/tcpstart.csd
/etc/nfsstart.csd
/etc/tcpd.csd
```

To run TCP/IP on diskless nodes, use your SCF file with “2” appended (for example, `initsrd2`) to submit similar files that properly set up TCP/IP for those nodes.

See also: *TCP/IP and NFS for the iRMX Operating System*

If Remote Booting Fails

If it gets no response to its boot request message, the boot client waits one second and retransmits the message. After transmitting three times, the boot client gives up and returns an error.

Many other problems on the boot client, boot server, file server, or network can prevent remote booting. If it fails, correct the problem and try again.

Troubleshooting

These messages could appear on the boot client.

REMOTE BOOTING NOT SUPPORTED ON NIC

The NIC is incorrect; replace it with a supported one.

:R0: DEVICE DOES NOT EXIST

Incorrect version of the First Stage Bootstrap Loader is in PROM, or the remote device, :R0:, was not configured into the Bootstrap Loader. Replace PROMS on the CPU board with the Bootstrap Loader from the iRMX III OS. (Multibus I only)

See also: Configuring the first stage, *Bootstrap Loader Reference*, for details on burning new PROMs that include the remote device, :R0:.

Error 02H

No boot server responded to the remote boot requests of the boot client. Several problems can cause this.

The boot server could not find the `:sd:net/ccinfo` file during initialization. Check these items on the boot server:

1. The `rbootsrv.job` is loaded.
2. The `ccinfo` file exists.
3. The `:sd:net/ccinfo.bdf` file was translated into the `ccinfo` file, using **bcl**.
4. The `:sd:net` directory is a public directory.

See also: Setting Up Public Directories, Chapter 4

The boot server could not find the files listed in the `ccinfo` file for the specified class code. Make sure the first directory in the path for each file is a public directory. The default is `/rboot32`.

The boot server did not honor the boot request for a particular class code. Check these items on the boot server:

1. The *ccinfo.bdf* file contains an entry for that class code. The class code specified in the *ccinfo.bdf* file is assumed to be a hexadecimal value.
2. All of the *ccinfo* file fits into the boot server's buffer in memory.
3. Entries in the *ccinfo* file for other class codes do not specify the boot client's Ethernet address.

See also: **bcl** command, *Command Reference*

The physical connection between the boot client and the boot server has been broken. Reconnect the systems to the network.

06 Undefined Operation

The wrong third stage Bootstrap Loader was used. Use the remote third stage shipped with the iRMX OS. If the boot client is a Multibus I host, make sure that the remote third stage contains a module located at 1060H. Use *unxlate* to check.

See also: **unxlate** command, *Command Reference*

If the boot client is a diskless node, these messages could also appear.

HI INITIALIZATION ERROR: 0021H

Several problems can cause this.

The boot client did not attach to the file server. Make sure the file server name has been entered into a Name Server database.

See also: Adding a Server to the Name Server Object Table, Chapter 3

The boot client could not find the file to be assigned a logical name. Make sure that the file server has offered as public those directories needed for the logical names. The logical names are defined in the *:config:loadinfo* file or the Public Directory screen of the ICU. The default directories needed are listed in this chapter.

See also: Making Local Files Accessible to Other Nodes, Chapter 4

The boot client is executing a different OS than the file server and the path name of the configuration directory on the ICU's EIOS screen was not changed to reflect the different OS. Reconfigure the OS boot file to use *:sd:rmx386/config* for the correct file server OS.

A Multibus I boot client is trying to use *:R0:* as the system device. Turn off automatic device recognition in the EIOS screen of the ICU, and add an entry for a remote SD in the logical names screen of the ICU. Regenerate the OS boot file.

HI INITIALIZATION ERROR: 004BH

The file server cannot find a client name and password for the boot client in the Client Definition File (CDF). Use **modcdf** on the file server to add the information.

See also: Adding a Client to the CDF, Chapter 3

The system boots, but the recovery user comes up, not the login prompt. Check for these two problems:

The name of the boot client is not in the file server's *:config:terminals* file.

See also: *:config:terminals* file, *System Configuration and Administration*

The name of the boot client is not in the Name Server object table on the file server.

Enter a type 0005H object in the */net/data* file and invoke the **loadname** command on the file server to load the name.

See also: Adding an Object to the Name Server Object Table, Chapter 11

Creating Custom Server Applications

This section provides additional information about the Boot Service provided by the iNA NMF, for those who are implementing custom servers. This includes the boot service message fields recognized by the NMF and the iNA boot file format. A more detailed discussion of the NMF Boot Service architecture and protocol is beyond the scope of this manual.

Boot Request and Response

The boot request and boot response messages have this format.

Boot Request Message Fields

```
char          destination_address[6]    = "01AA00FFFFFF"
char          source_address[6]
unsigned short LLC_PDU_length           = 7h
char          destination_LSAP_selector = 8h
char          source_LSAP_selector     = 8h
char          control                   = 3h
char          reserved                  = 0h
char          command                   = 4h
unsigned short class_code
```

The fields `destination_address` and `LLC_PDU_length` through `command` must have the values indicated following the equal signs.

Boot Response Message Fields

```
char          destination_address[6]
char          source_address[6]
unsigned short LLC_PDU_length           = 5H
char          destination_LSAP_selector = 8H
char          source_LSAP_selector     = 8H
char          control                   = 3H
char          reserved                  = 0H
char          command                   = 5H
```

The fields `LLC_PDU_length` through `command` must have the values indicated following the equal signs.

Loading Operation

Once the boot request and response is completed, the actual loading operation begins.

From the class code specified in the boot request, the boot server knows exactly which modules the boot client needs. The boot server breaks the modules into an arbitrary number of blocks of unspecified size. The size of each block must be small enough to fit into a boot response packet, which can contain up to 1496 bytes of data. The protocol enables any number of modules to be loaded.

The boot client begins transmitting requests for blocks of data. Block numbers start at 0 and increase by 1 for each successive request. The algorithm for transmitting these requests is the same as described earlier for the initial boot request. Upon receiving a response from the boot server, the boot client processes the response and transmits a request for the next block of data.

When the boot client has received all blocks of data, it simply stops transmitting requests for more. The boot server then times out waiting for the next request from the client and releases resources that were reserved for the timed-out client.

When the iRMX first stage Bootstrap Loader receives the indication that the remote load is done, it jumps to location 1060H in memory.

The data request and data response messages have this format.

Data Request Message Fields

char	destination_address[6]	
char	source_address[6]	
unsigned short	LLC_PDU_length	= 7h
char	destination_LSAP_selector	= 8H
char	source_LSAP_selector	= 8H
char	control	= 3H
char	reserved	= 0H
char	command	= 6H
unsigned short	block	

The fields `LLC_PDU_length` through `command` must have the values indicated following the equal signs. The `block` field is the data block number. Block numbers start at 0 and increase by 1 for each successive request.

Data Response Message Fields

char	destination_address[6]	
char	source_address[6]	
unsigned short	LLC_PDU_length	= 7h + n
char	destination_LSAP_selector	= 8H
char	source_LSAP_selector	= 8H
char	control	= 3H
char	reserved	= 0H
char	command	= 7H
unsigned short	block	
char	data[n]	

The fields `LLC_PDU_length` through `command` must have the values indicated following the equal signs. The `LLC_PDU_length` value must be 7H plus the length of the data field, including padding. The `block` field is the number of the data block in the data field.

On Multibus I systems a portion of the remote third stage has been loaded into memory, starting at 1060H. The code at this location jumps to the configurable start of the rest of the remote third stage. The default location of this remote third stage is 6000H. The remote third stage then sets up the GDT, IDT, and TSS. The information for the GDT, IDT, TSS and start of the OS is located in 14 bytes starting at 1050H. The 14 bytes are added by the **xlate** utility and are always part of the iRMX OS. The third stage places the processor into protected mode and starts the execution of the OS. On PC Bus and Multibus II systems the process is similar.

See also: **xlate** and **unxlate** commands, *Command Reference*, for information about how the firmware determines where to place the data; GDT slots, *System Configuration and Administration*; IDT and task states, *System Concepts*

Boot Module Format

This load file (or boot module) format is the one recognized by the boot client firmware on Intel network communication boards. This is the format of the iNA load file.

```
typedef struct boot_module {
    unsigned char    command;
    unsigned long   load_addr;
    unsigned short  length;
    unsigned long   execution_addr;
    unsigned char   memory_image[1];
} BOOT_MODULE;
```

command Indicates whether to execute the module and whether to load more modules. Only the four low-order bits are meaningful, as shown below. Bits 4 through 7 must be 0.

Bit	Value	Meaning
0	0	This is the last load module.
	1	Other modules follow this one.
1	0	Save the execution address to jump to when a GO command is received.
	1	Jump to the execution address immediately after loading the module.
2	0	The load and execution addresses are in pointer format, typically used for a module to load onto a communication controller.
	1	The load and execution addresses are absolute addresses, typically used for a module to load onto the boot client host.
3	0	Load this module onto a communication controller.
	1	Load this module into host memory.

Bits 0 and 1 act together and have this meaning:

	Bit 0	Bit 1
	0	1
0	Load complete Wait for GO	Load complete Execute now
1	More to load Save execute address	More to load But jump to execute address now

<code>d_addr</code>	The address to begin writing the memory image. This must be a 32-bit value that is meaningful to the firmware on the target being loaded.
<code>length</code>	The number of bytes in <code>memory_image</code> , from 0 to 64K.
<code>execution_addr</code>	The execution address of the loaded memory image. This must be a 32-bit value that is meaningful to the firmware on the target being loaded.
<code>memory_image</code>	The data to load.

Using **SUPPLY_BUFFER** and **TAKEBACK_BUFFER**

Two NMF commands support remote load applications. The **SUPPLY_BUFFER** and **TAKEBACK_BUFFER** commands provide a mechanism for the NMF to pass remote load requests and responses that it does not support between applications (for example, between custom boot clients and boot servers).

The application is free to define a protocol for remote load request and response messages. There is, however, a constraint on the `reserved` and `command` fields of the messages; they must be set to 0.

For a custom remote load application, the NMF boot server is not required in the iNA NMF configuration used by the application. However, the `NMF_SUBNET_FUNCTIONS` macro call must be included in the iNA NMF configuration used by the application. All preconfigured iNA modules supplied with iRMX-NET include this macro except for those specifically for the SBX 552 board.

For the NMF to pass messages between the applications, buffers are needed to store the incoming and outgoing message packets. If packets are received by the NMF that contain commands NMF doesn't recognize and no buffers have been supplied to handle them, the message packets are dropped; no acknowledgment is sent to the source.

When the NMF receives a Data Link packet with the NMF LSAP, the command field is checked to determine if the NMF recognizes the command. If it does, the NMF executes the command.

If the NMF does not recognize the command and a buffer has been supplied, the packet is placed in the buffer and a request block pointing to it is forwarded to the application.

The application responds to the command by writing the response data to another buffer and then responding to the source of the Data Link packet in an agreed-upon manner (e.g., using EDL or datagram services). The response to the command contains a pointer to the buffer containing the response data.

This capability makes it possible for the application to implement custom NMF-level commands that the application executes, rather than the iNA NMF.

The `SUPPLY_BUFFER` command supplies buffers for the purpose described above and the `TAKEBACK_BUFFER` command releases them.

See also: `SUPPLY_BUFFER` and `TAKEBACK_BUFFER` commands,
Chapter 14

□ □ □

Internetwork Routing 16

Network addressing and internetwork routing concepts are introduced in Chapters 8 and 9 of this manual; read those chapters before this one.

Internetwork Routing Protocols

The iNA 960 ES-IS software supports two internetwork routing methods:

- Static, which uses the MAP 2.1 routing scheme for mapping NSAP addresses to subnet addresses.
- End system to intermediate system (ES-IS), which implements the protocol described in IS 9542.

The main difference between static and ES-IS routing is the way routing information is initially defined and subsequently maintained in end systems and intermediate systems. In both static and ES-IS routing, routing information is defined by tables located in end systems and intermediate systems.

Static Routing

In the Static routing scheme, you build routing tables on each end system and intermediate system in the network when you initially configure them. If the network configuration and/or membership changes, you must update the routing tables explicitly, using some mechanism such as the iNA 960 Network Management Facility (NMF).

ES-IS Routing

In the ES-IS routing protocol, iNA 960 builds routing tables on each end system and intermediate system dynamically as it starts up and comes onto the network. Part of the startup process of an end system is to notify the intermediate systems in the subnet of its existence. Similarly, intermediate systems notify end systems of their existence when they start up. If the subnet configuration and/or membership changes, (whether temporary or permanent), the protocol provides a mechanism for automatic periodic updates to the routing tables. This ensures that routing information for inactive systems is removed where necessary throughout the subnet.

If an end system needs to send a message to some destination, it sends the message to an intermediate system which forwards the message. If the destination is directly reachable from the end system that sent the message, the intermediate system sends the end system the information necessary to send future messages directly to that destination.

An intermediate system does not have to be present in a subnet. In the absence of an intermediate system, end systems can determine each other's existence and correctly exchange messages.

Using Static and ES-IS Routing Together

ES-IS routing does not dynamically define or update the information necessary to deliver messages between intermediate systems in different subnets. You must set up the table containing this information when you configure the intermediate systems. Thus, when you configure systems to perform internetwork routing, you must use a combination of Static and ES-IS routing. On the ISs (intermediate systems, or routers), you set up static routing tables to point to each IS attached to the same subnet, and also to indicate which IS to use to get to any subnets that could possibly be accessed (including subnets anywhere that you might want to send or receive messages from).

Once you have set up the static routing on all ISs, iNA 960 automatically performs ES-IS routing to locate ESs (end systems). You do not have to set up static routing tables on ESs, nor do you have to indicate ESs in the static routing tables you set up on ISs.

See also: Chapter 9 for examples of setting up static routing with the Multibus II subnet

Routing Tables

iNA 960 makes internetwork routing decisions based on information defined in a set of tables maintained at all end systems and intermediate systems. The tables are objects that are part of the internal database maintained by the iNA 960 software.

See also: Static and ES-IS router objects, Appendix C

The routing tables at end systems define mappings of NSAP address to subnet address for the local subnet. The routing tables at intermediate systems define the same information as end systems, plus information on remote subnets and other intermediate systems. The Network Layer uses the routing tables to determine how to deliver a message.

Application Access to Routing Tables

You can access the routing tables as iNA 960 routing objects. To perform network routing, the application uses the NMF commands `READ_OBJECT` and `SET_OBJECT`.

See also: `READ_OBJECT`, Chapter 14

The remainder of this chapter describes the command buffer and response buffer structures that are unique to the Static and ES-IS routing objects. The Static routing objects are different than the ES-IS routing objects and they have different command and response buffer structures. The application references these buffers in the `READ_OBJECT` and `SET_OBJECT` request blocks.

Reading and Setting Static Routing Objects

The Static routing objects are four routing tables:

- The Local Subnet Table, specifying all of the subnets to which the system is physically connected.
- The Specific Router Table, specifying all of the intermediate systems that can be reached directly from the system.
- The Destination Subnet Table, specifying all of the subnets that can be reached through the intermediate systems listed in the Specific Router Table. This table associates a remote subnet with the specific intermediate system through which it can be reached.
- The Default Router Table, specifying a single intermediate system to use if none of the systems in the Specific Router Table works. Only end systems use this table.

All of the parameter values associated with the Static routing objects can be read and most of them can be set. The objects that cannot be set are all in the Local Subnet Table: the subnet name, the subnet address, and the transmit packet size. These are set when the system containing the table is configured.

Command and Response Buffers for Static Routing

Use these structures for command and response buffers when reading and setting Static routing objects. Most of the fields in the two structures are the same; the field descriptions follow the structure definitions. The structures are provided as typedefs in the include files for routing structures.

See also: Include Files, Chapter 10;
 Programming with Structures, Chapter 10

⇒ **Note**

The application must supply a filled-in command buffer for both the SET_OBJECT and READ_OBJECT commands.

Command Buffer

The command buffer pointer in the READ_OBJECT and SET_OBJECT commands references an array of one or more objects to read or set. Set the command buffer pointer to reference the `stat_routing_cmd` structure below.

In the SET_OBJECT command, each object in the array specifies this `stat_rout_info` structure of data. In READ_OBJECT command buffers, the `stat_rout_info` structure need not be present; fill in the `obj_cmd_struct` structure and set `cmd_len` to 0.

```
typedef struct stat_rout_info {
    unsigned char    router_name_len;
    unsigned char    router_name[12];
    unsigned char    subnet_name_len;
    unsigned char    subnet_name[12];
    unsigned char    lifetime;
    unsigned char    afi;
    unsigned char    subnet_no_len;
    unsigned char    subnet_no[6];
    unsigned char    addr_len;
    unsigned char    addr[12];
    unsigned short   tx_pkt_size;
} STAT_ROUT_INFO;

typedef struct obj_cmd_struct {
    unsigned short   object;
    unsigned short   modifier;
    unsigned short   cmd_len;
    STAT_ROUT_INFO   cmd_info;
} OBJ_CMD_STRUC;

typedef struct stat_routing_cmd {
    unsigned char    num_obj;
    OBJ_CMD_STRUC    obj_info[1]; /* set to num_obj */
} STAT_ROUTING_CMD;
```

Response Buffer

The response buffer pointer in the READ_OBJECT and SET_OBJECT commands references an array of one or more objects for which information is returned. In the SET_OBJECT command, the application need not read the data returned in the response buffer, but the buffer must be large enough to hold the entire structure. Set the response buffer pointer to reference the stat_routing_resp structure below.

```
typedef struct stat_rout_info {
    unsigned char    router_name_len;
    unsigned char    router_name[12];
    unsigned char    subnet_name_len;
    unsigned char    subnet_name[12];
    unsigned char    lifetime;
    unsigned char    afi;
    unsigned char    subnet_no_len;
    unsigned char    subnet_no[6];
    unsigned char    addr_len;
    unsigned char    addr[12];
    unsigned short   tx_pkt_size;
} STAT_ROUT_INFO;

typedef struct obj_resp_info {
    unsigned short   object;
    unsigned short   modifier;
    unsigned char    status;
    unsigned short   resp_len;
    STAT_ROUT_INFO   resp_info;
} OBJ_RESP_INFO;

typedef struct stat_routing_resp {
    unsigned char    num_obj;
    OBJ_RESP_INFO    obj_info[1]; /* set to num_obj */
} STAT_ROUTING_RESP;
```

Field Descriptions for Command and Response Buffers

`num_obj`

The number of objects being read or set by the command.

`obj_info`

An array of structures where each structure contains the information pertaining to the object being read or set.

`object`

The object ID of the routing table to act upon.

See also: Appendix C for the Static routing table ID numbers

`modifier`

For the `READ_OBJECT` command buffer, this is an index number specifying the table entry to read. The first entry in a table has index number 1. This field is not used in a `READ_OBJECT` response buffer or in a `SET_OBJECT` command.

`status`

Present only in the response buffer; it contains a status code indicating the result of the requested NMF operation.

See also: `READ_OBJECT` command, Chapter 14, for codes

`cmd_len`

In the command buffer this field must be set to 50 bytes.

`cmd_info`

In the command buffer of a `SET_OBJECT` command this is a structure specifying values to set. The structure is unused in `READ_OBJECT` command buffers.

`resp_len`

In the response buffer this value is set to 50 bytes.

`resp_info`

In the response buffer this is a structure containing returned parameter values for the table entry.

`router_name_len`

The actual length in bytes of the router name specified in the `router_name` field (not the length of the array containing the name). This field is not use when reading or setting Local Subnet Table objects.

`router_name`

An application-specified alias. Router names are ASCII character strings up to 12 characters long, stored as an array of bytes. A `SET_OBJECT` command buffer for the Specific Router or Default Router Tables that specifies a router name already in the table causes the object values for that router name to be overwritten with the values specified in the remaining fields of the structure. This field is not used when reading or setting entries in the Local Subnet Table.

`subnet_name_len`

The actual length in bytes of the subnet name specified in the `subnet_name` field (not the length of the array containing the name). For the Local Subnet Table, the value of this parameter can be changed only by reconfiguring the system, not by using the `SET_OBJECT` command.

In `SET_OBJECT` commands for the other tables, specifying a subnet name length of 0 causes the existing table entry for the specified router name, or subnet number in the case of the Destination Subnet Table, to be deleted from the table.

`subnet_name`

The name of the local subnet that is chosen when the system is configured. This field is not used when setting Destination Subnet Table entries; the association of a subnet with a router in this table uses the router name and the subnet number.

`lifetime`

The maximum time a message can remain in the network if it cannot be delivered to its destination end system. This value is specified in units of 0.5 seconds up to a maximum of 255 units (127.5 seconds). If the message lifetime expires before delivery of the message to the destination end system, the message is discarded. Notification may be sent to the message originator, but message reception is not guaranteed. This field is not used when reading or setting entries in the Specific Router table.

`afi`

The authority and format identifier (AFI) portion of the Initial Domain Part (IDP) of the NSAP address for the subnet. By definition, the AFI for Static routing NSAP addresses is fixed at 49H. This field is not used when setting entries in the Specific Router or Default Router tables.

`subnet_no_len`

The actual length in bytes of the subnet ID specified in the `subnet_no` field (not the length of the array containing the number). This field is not used when setting entries in the Specific Router or Default Router tables.

subnet_no

The 2-byte subnet ID. This field is not used when setting entries in the Specific Router or Default Router tables. For the Destination Subnet Table, this is the ID number of a subnet that is reachable through the router specified in the `router_name` field.

addr_len

The actual length in bytes of the subnet address specified in the `addr` field (not the length of the array containing the address). This field is not used when setting entries in the Local Subnet or Destination Subnet tables, and not used when reading entries in the Destination Subnet Table.

addr

The subnet address. This field is not used when setting entries in the Local Subnet or Destination Subnet tables and when reading entries in the Destination Subnet table. When reading or setting entries in the Specific Router and Default Router tables, this field specifies the subnet address of the router. When reading entries in the Local Subnet Table, this field contains the local subnet address.

See also: Subnet Address, Chapter 8

tx_pkt_size

The maximum size of a data link transmit packet. This field is only used in the response buffer for a `READ_OBJECT` command on the Local Subnet table.

Reading and Setting ES-IS Routing Objects

Part of the ES-IS routing objects are these six routing tables. In addition to these six routing tables, there are several objects relating to routing operations and the size of the routing tables.

See also: ES-IS routing objects, Appendix C

The ES-IS tables are:

- Local End System Table
- Intermediate System Hello Table
- Static Intermediate System Table
- Reachable NSAP Address Table
- Subnet Table
- Local NSAP Address Table

Local End System Table Contains information about the end systems physically attached to the same subnet as the system containing the table (that is, all of the end systems reachable in one hop). This table is typically present in all end systems and intermediate systems. This table is the first one searched to map an NSAP address to a subnet address.

Intermediate System Hello Table Contains routing information derived from received intermediate system Hello PDUs. When an end system searches the Local End System Table for an NSAP address and does not find it there and there is no Static Intermediate System Table present on the system, it chooses a router from this table and maps the NSAP address to the subnet address of that router. This table is present only in end systems.

Static Intermediate System Table Is primarily an intermediate system table; however, it may also be present in end systems. This table is not updated dynamically. Instead, it is built when the system is configured and can later be modified using iNA 960 NMF commands. The information in this table identifies the intermediate systems that are available on local subnets. This table and the Reachable NSAP Address Table are used by an intermediate system when it determines that the destination end system is not available on the local subnet. In this case, the NPDU is forwarded to an intermediate system chosen from this table.

Reachable NSAP Address Table	Is always present in intermediate systems. Like the Static Intermediate System Table, it may also be present in end systems and is also built when the system is configured and can be modified with NMF commands. Each entry in this table corresponds to an entry in the Static Intermediate System Table. Entries in this table contain an NSAP address prefix. All NSAP addresses beginning with that prefix are considered to be reachable through the intermediate system identified by the Static Intermediate System Table entry corresponding to the entry in this table.
Subnet Table	<p>Contains information about each of the local subnets to which the system is physically connected. This table is present in both end systems and intermediate systems. For each subnet, the table identifies:</p> <ul style="list-style-type: none"> • The name of the subnet. • The lifetime value to use when sending a PDU on the subnet. • The local SNPA-ID of the subnet. • A flag indicating whether or not to use the ES-IS routing protocol over the subnet. This flag is useful when the subnet described by the table entry is of a type that does not support multicast addresses (e.g., X.25).
Local NSAP Address Table	Specifies the NSAP addresses available on the local system. In the case of intermediate systems, the first entry in this table is used as the Network Entity Title (NET). The NSAP addresses in this table do not specify the NSAP selector portion of the NSAP address; this is specified at initialization time by the network service user. This table is present in both end systems and intermediate systems.

Command and Response Buffers for ES-IS Routing

Use these structures for command and response buffers when reading and setting ES-IS routing objects. Most of the fields in the two structures are the same; the field descriptions follow the structure definitions. The structures are provided as typedefs in the include files for routing structures for your use.

See also: Include Files, Chapter 10;
 Programming with Structures, Chapter 10

In READ_OBJECT request blocks, both command and response buffers are used. The command buffer specifies the objects to read; the response buffer is filled with the values read.

In SET_OBJECT request blocks, the command buffer specifies the objects to set and the data to write to the objects. The application need not read the data returned in the response buffer, but the buffer must be large enough to hold the entire structure.

Command Buffer

The command buffer pointer in the READ_OBJECT and SET_OBJECT commands references an array of one or more objects to read or set. Set the command buffer pointer to reference the `es_is_nmf_cmd_buf` structure below. In the SET_OBJECT command, each object in the command buffer array specifies a structure of data in the `cmd_info` field. In READ_OBJECT command buffers, the `cmd_info` field need not be present; fill in the `es_is_cmd` structure and set `cmd_len` to 0.

The data in the `cmd_info` field varies depending on which routing table is being accessed. The structures for the various tables are described after the response buffer section.

```
typedef struct es_is_cmd {
    unsigned short    object;
    unsigned short    modifier;
    unsigned short    cmd_len;
    OBJ_CMD_STRUC     cmd_info;    /* object-specific */
} ES_IS_CMD;

typedef struct es_is_nmf_cmd_buf {
    unsigned char     num_obj
    ES_IS_CMD         obj_info[1]; /* set to num_obj */
} ES_IS_NMF_CMD_BUF;
```

Response Buffer

The response buffer pointer in the READ_OBJECT and SET_OBJECT commands references an array of one or more objects for which information is returned. In the SET_OBJECT command, the application need not read the data returned in the response buffer, but the buffer must be large enough to hold the entire structure. Set the response buffer pointer to reference the `es_is_nmf_resp_buf` structure below.

The data returned in the `resp_info` field varies depending on which routing table is being accessed. The structures for the various tables are described in following sections.

```
typedef struct es_is_resp {
    unsigned short    object;
    unsigned short    modifier;
    unsigned char     status;
    unsigned short    resp_len;
    OBJ_RESP_INFO     resp_info;    /* object-specific */
} ES_IS_RESP;

typedef struct es_is_nmf_resp_buf {
    unsigned char     num_obj
    ES_IS_RESP        obj_info[1]; /* set to num_obj */
} ES_IS_NMF_RESP_BUF;
```

Field Descriptions for Command and Response Buffers

`num_obj`

The number of objects being read or set by the command.

`obj_info`

An array of structures where each structure contains the information pertaining to the object being read or set.

`object`

The object ID of the routing table to act upon.

See also: ES-IS routing objects, Appendix C, for ID numbers

`modifier`

For the `SET_OBJECT` command, this field is not used for most objects. The objects for which the modifier has meaning are the Local NSAP Address and the Reachable NSAP Address tables.

For the `READ_OBJECT` command buffer, this is an index number specifying the table entry to read. The first entry in a table has index number 1. An index beyond the end of the table results in an end-of-table status (82H) in the response buffer. Table entries are not maintained in any particular order; a table must be read sequentially to find the desired value.

`status`

Present only in the response buffer; it contains a status code indicating the result of the requested NMF operation.

See also: `READ_OBJECT` command, Chapter 14, for codes

`cmd_len`

The length in bytes of data in the `cmd_info` field.

`cmd_info`

In the command buffer of a `SET_OBJECT` command this is an object-specific structure containing values to set. The structure is unused in `READ_OBJECT` command buffers.

`resp_len`

The length in bytes of data in the `resp_info` field.

`resp_info`

In the response buffer of a `READ_OBJECT` command this is an object-specific structure containing the values read from the object.

Data in the `cmd_info` and `resp_info` fields may be any of these structures described in these sections:

- `local_es_table_cmd_struct`
- `ish_table_cmd_struct`
- `static_is_table_cmd_struct`
- `nsap_addr_reachable_cmd_buf`
- `subnet_table_cmd_struct`
- `local_nsap_table_cmd_struct`

The Local End System Table Structure

```
typedef struct local_es_table_cmd_struct {
    unsigned char    nsap_addr_len;
    unsigned char    nsap_addr[20];
    unsigned char    subnet_addr_len;
    unsigned char    subnet_addr[12];
    unsigned char    subnet_name_len;
    unsigned char    subnet_name[12];
    unsigned short   holding_time;
} LOCAL_ES_TABLE_CMD_STRUCT;
```

`nsap_addr_len`

The actual length of the NSAP address being read or entered into the table (not the length of the 20-byte array that contains the address). The NSAP address may be fewer than twenty bytes long, but a full 20-byte array must be used to hold the address.

`nsap_addr`

The NSAP address being read or entered, including an NSAP selector (encoded as an array of bytes with the selector as the last byte). The address can be at most twenty bytes long. This NSAP address is entered into the table and mapped to the subnet address specified in this field. If this NSAP address is already present in the routing table, the parameter values for it are replaced by the values specified in this command.

`subnet_addr_len`

The actual length of the subnet address (not the length of the array).

`subnet_addr`

The subnet address to which the preceding NSAP address maps.

`subnet_name_len`

The actual length of the subnet name (not the length of the array).

`subnet_name`

An ASCII string, encoded as an array of bytes, identifying the subnet where the subnet address is located. The subnet names are defined by the system builder when the system is configured. A null subnet name (a length of 0) indicates that the preceding NSAP address and its associated subnet address are to be deleted from the table.

`holding_time`

The number of seconds this information is valid. The value 0xFFFFH indicates infinite time. After the specified time expires, if the information specified in this command has not been somehow refreshed, it is removed from the table.

The Intermediate System Table Structure

```
typedef struct ish_table_cmd_struct {
    unsigned char    net_entity_title_len;
    unsigned char    net_entity_title[20];
    unsigned char    subnet_address_len;
    unsigned char    subnet_address[12];
    unsigned char    subnet_name_len;
    unsigned char    subnet_name[12];
    unsigned short   holding_time;
} ISH_TABLE_CMD_STRUCT;
```

`net_entity_title_len`

The actual length in bytes of the Network Entity Title (not the length of the array that contains the NET).

`net_entity_title`

The Network Entity Title (NET) of the intermediate system (IS) that is reached by the subnet address. A NET has the same syntax as an NSAP address, but is distinct from any NSAP address. NETs should have an NSAP selector (the last byte) of 0.

`subnet_address_len`

The actual length in bytes of the subnet address (not the length of the array).

`subnet_address`

The subnet address where the intermediate system identified by the preceding NET can be reached.

`subnet_name_len`

The actual length of the subnet name (not the length of the array).

`subnet_name`

An ASCII string, encoded as an array of bytes, identifying the subnet where the subnet address is located. The subnet names are defined by the system builder when the system is configured. A null subnet name (a length of 0) indicates that the preceding NET and its associated subnet address are to be deleted from the table.

`holding_time`

The number of seconds this information is valid. The value 0xFFFFH indicates infinite time. After the specified time expires, if the information specified in this command has not been refreshed by receipt of an Intermediate System Hello, the information is removed from the table.

The Static Intermediate System Table Structure

```
typedef struct static_is_table_cmd_struct {
    unsigned char    net_entity_title_len;
    unsigned char    net_entity_title[20];
    unsigned char    subnet_address_len;
    unsigned char    subnet_address[12];
    unsigned char    subnet_name_len;
    unsigned char    subnet_name[12];
    unsigned char    router_name_len;
    unsigned char    router_name[12];
    unsigned short   num_prefixes;
} STATIC_IS_TABLE_CMD_STRUCT;
```

`net_entity_title_len`

The actual length in bytes of the Network Entity Title (NET). This is not the length of the array that contains the NET.

`net_entity_title`

The Network Entity Title (NET) of the intermediate system (IS) that is reached by the subnet address. A NET has the same syntax as an NSAP address, but is distinct from any NSAP address. NETs should have an NSAP selector (the last byte) of 0.

`subnet_address_len`

The actual length in bytes of the subnet address (not the length of the array).

`subnet_address`

The subnet address where the intermediate system identified by the preceding NET can be reached.

`subnet_name_len`

The actual length of the subnet name (not the length of the array).

`subnet_name`

An ASCII string, encoded as an array of bytes, identifying the subnet where the subnet address is located. The subnet names are defined by the system builder when the system is configured. A null subnet name (a length of 0) indicates that the preceding NET and its associated subnet address are to be deleted from the table.

`router_name_len`

The actual length in bytes of the router name (not the length of the array).

`router_name`

An application-defined name for the router; any string of up to 12 characters. A SET_OBJECT command specifying a router name for which an entry already exists in the table causes the parameter values for that router to be overwritten with the new values specified in the command. Specifying a router name that does not exist in the table creates a new entry in the table.

num_prefixes

A read-only parameter (it cannot be set) whose value is the number of NSAP prefixes registered in the Reachable NSAP Address Table that corresponds to this Static Intermediate System Table entry.

The Reachable NSAP Address Table Structure

A SET_OBJECT command on the Reachable NSAP Address table uses the modifier field of the command buffer. If the modifier field is set to 0, a table entry for the specified router is created. If the modifier field is set to a not 0 value, the table entry for the specified router is deleted.

```
typedef struct nsap_addr_reachable_cmd_buf {
    unsigned char    router_name_len;
    unsigned char    router_name[12];
    unsigned char    nsap_prefix_len;
    unsigned char    nsap_prefix[20];
} NSAP_ADDR_REACHABLE_CMD_BUF;
```

router_name_len

The actual length in bytes of the router name (not the length of the array).

router_name

An application-defined name for the router; any string of up to twelve characters. All NSAP addresses starting with the prefix specified in the fields are considered reachable through the router specified by this name. The router name must be specified in the Static Intermediate System Table before this command can be used.

nsap_prefix_len

The actual length in bytes of the NSAP prefix (not the length of the array).

nsap_prefix

Specifies which NSAP addresses may be reachable through the router specified in the preceding Router Name field. All NSAP addresses beginning with the prefix are reachable through the specified router. For example, suppose all of the NSAP addresses reachable by a router named HF1 began with 49 0001. The prefix 490001H would be registered in this table with the router name HF1. If all the NSAP addresses reachable by a router named HF2 began with 49 0002, the prefix 490002H would be registered in this table with the router name HF2. The length of the NSAP prefix array is twenty bytes so that full NSAP addresses can be mapped.

The Subnet Table Structure

```
typedef struct subnet_table_cmd_struct {
    unsigned char    subnet_name_len;
    unsigned char    subnet_name[12];
    unsigned char    lifetime;
    unsigned char    use_protocol;
    unsigned char    nsap_prefix_len;
    unsigned char    nsap_prefix[10];
    unsigned long    snpa_id;
} SUBNET_TABLE_CMD_STRUCT;
```

`subnet_name_len`

The actual length of the subnet name (not the length of the array).

`subnet_name`

An ASCII string identifying the subnet to which the rest of the information in the structure pertains. The subnet names are defined when the system is configured. Subnet names cannot be deleted from the table; only the values for the parameters can be changed.

`lifetime`

The maximum time a message originating on this subnet can remain in the network if it cannot be delivered to its destination end system. This value is specified in units of 0.5 seconds up to a maximum of 255 units (127.5 seconds). If the message lifetime expires before delivery of the message to the destination end system, the message is discarded. Notification may be sent to the message originator, but reception is not guaranteed.

`use_protocol`

A flag indicating whether to send and receive ES-IS Hello and Redirect function PDUs over the subnet defined by this structure. This flag is false (Hello and Redirect PDUs will not be sent or received) when its value is 0 and true (Hello and Redirect PDUs will be sent and received) when its value is not 0.

This flag serves as a mask for the send and receive hello flags. If the send hello flag is true and this flag is false, End System Hellos will not be sent. If the send hello flag is false and this flag is true, End System Hellos will still not be sent.

This flag is useful to intermediate systems that are attached to multiple subnets, that use the ES-IS routing protocol, and where one of the attached subnets does not support multicast addressing. For example, if one of the attached subnets is an X.25 subnet and the others are all 802.3 subnets, the flag can be set false for the X.25 subnet, and true for all of the others.

`nsap_prefix_len`

The actual length in bytes of the NSAP prefix (not the length of the array).

`nsap_prefix`

Used for compatibility with the Null2 (inactive subset) and Static ES-IS Network Layer configurations. The prefix is added to the source subnet address of a received Null2 network PDU (NPDU) in order to form the source NSAP address. This is necessary because the Null2 subnet address does not contain the source NSAP address.

This field also indicates which NSAP addresses exist on the local subnet. If the local addressing flag is true, all destination NSAP addresses are matched against the NSAP prefixes in the Local Subnet Table entries. If a match is found, the destination is directly reachable on the local subnet and the portion of the NSAP address immediately following the matching prefix is used as the subnet address of the destination.

This parameter is known as the AFI and subnet number combination in Null2 addressing. For example, a value for this parameter for use with Null2 addressing is 490003H.

`snpa_id`

A read-only parameter whose value is the source LSAP selector for outgoing subnet PDUs (SNPDUs). The value of this parameter is defined when the system is configured.

The Local NSAP Address Table Structure

A `SET_OBJECT` command on the Local NSAP Address table uses the `modifier` field of the command buffer. If the `modifier` field is set to 0, a table entry for the specified NSAP address is created. If the `modifier` field is set to a not 0 value, the table entry for the specified NSAP address is deleted.

```
typedef struct local_nsap_table_cmd_struct {
    unsigned char      nsap_address[1];
} LOCAL_NSAP_TABLE_CMD_STRUCT;
```

`nsap_address`

The NSAP address to enter, delete, or read from the table. This NSAP address is a string of bytes; the length of which is specified by the `cmd_len` field in the `SET_OBJECT` or `READ_OBJECT` command buffer structure. The NSAP address must not include the NSAP selector; the selector is dynamically determined by the Network Layer at run time.



iRMX-NET and iNA 960 Transport Configuration Values

A

This appendix describes the values preconfigured into these network files:

iNA 960 download files	These are files loaded onto the NIC by MIP jobs. The MIP jobs are <i>ipcl2.job</i> for a PC, <i>i552a.job</i> in a Multibus I system, or <i>icemb2.job</i> in a Multibus II system. Table A-1 lists the download files.
MIP jobs	Jobs that interface to iNA 960 download files or to iNA 960 COMMputer jobs running on another board
iRMX-NET jobs	The file server <i>rnetserv.job</i> and client, <i>remotefd.job</i>
See also:	Network Software Implementation, Chapter 7; <i>i*.job, System Configuration and Administration</i>

Files Containing iNA 960 Transport Software

The iNA 960 transport software exists in one of two places. It is either an iNA 960 file downloaded to a NIC, or an iNA 960 COMMputer job running on the same board as the OS. This section describes the files and their preconfigured values.

iNA 960 Download Files

MIP jobs in a COMMengine environment load iNA transport software from disk to the NIC. The iNA 960 download filenames have the form:

```
INA<board><a>.32<l>
```

Where:

<board>	An encoded board name.
<a>	An N or E specifying the type of transport address: N stands for Null2 and E stands for ES-IS format. If your application programs the network addresses or uses a network with multiple subnets, you could use an ES-IS network, which is routable.
<l>	An L specifying a local load file or an R specifying remote load.

To find the iNA release version to which the file applies, use the **version** command.

Table A-1 lists the iNA 960 files available for downloading, depending on your system type and NIC. As shown in the table, most files are specific to certain versions of the OS.

Table A-1. iNA 960 Download Files

Filename	Board	Configuration	System*	OS**
inapl2n.32l	PCL2	Null2 local load	PC	RPC, RFW
inapl2an.32l	PCL2A	Null2 local load	PC	RPC, RFW
ina552an.32l	SBC 552A	Null2 local load	MB1	III
ina552an.32r	SBC 552A	Null2 remote load	MB1	III
ina552ae.32l	SBC 552A	ES-IS local load	MB1	III
ina552ae.32r	SBC 552A	ES-IS remote load	MB1	III
ina530n.32l	SBC 186/530	Null2 local load	MB2	all
ina530e.32l	SBC 186/530	ES-IS local load	MB2	III
ina560n.32l	MIX 386/560	Null2 local load	MB2	all
ina560e.32l	MIX 386/560	ES-IS local load	MB2	III

* MB1 indicates Multibus I, MB2 indicates Multibus II

** III indicates the iRMX III OS, RPC indicates iRMX for PCs, RFW indicates DOSRMX

For ICU-configurable systems, you can specify the download file in the appropriate MIP job screen: MIP1 for Multibus I, CEBI for Multibus II, or MIPAT for PCs. On a Multibus II system you can override the configuration in the `rq_mip_xx` parameter of the BPS file. In the DOSRMX or iRMX for PCs OS, you can specify the download file in the `rmx.ini` file.

See also: `rq_mip_xx`, *MSA for the iRMX Operating System*; `rmx.ini` file, *System Configuration and Administration*

Table A-2 lists values configured into the iNA 960 download files, according to the NIC. For values configured into iNA 960 COMMputer jobs, see the default values on screens in the ICU. ICU.

Table A-2. iNA 960 Download File Configuration

Parameter	PCL2	PCL2A	552A	186/530	386/560	
Name Server						
Maximum number of objects	Null2	25	80	40	50	80
	ES-IS	20	25	25	80	80
Maximum length of value (Null2/ES-IS)		90	90/32	90	40/90	90
TSAP ID - initiator		4200	4200	4200	4200	4200
TSAP ID - responder		4300	4300	4300	4300	4300
Retry timeout (msec.)		614	614	614	820	820
TSAP ID - file server		1000	1000	1000	1000	1000
TSAP ID - file consumer		1100	1100	1100	1100	1100
Transport Layer						
Maximum TSAP length (bytes)		32	32	32	32	32
Maximum network address length (bytes)		20	20	20	20	20
Number of virtual circuits	Null2	101	160	92	201	201
	ES-IS	25	101	87	201	201
Number of datagram TSAPs	Null2	30	30	30	30	30
	ES-IS	19	30	30	30	30
Retransmission timeout - dynamic (sec.)	Null2	0.1 - 1.0	0.1 - 1.0	0.1 - 1.0	0.1 - 1.0	0.1 - 1.0
	ES-IS	0.2 - 1.0	0.2 - 1.0	0.1 - 1.0	0.2 - 1.0	0.2 - 1.0
Inactivity timeout (sec.)		30	30	30	30	30
Closing abort timeout (sec.)		6	6	6	6	6
Open window timeout (sec.)		1	1	1	1	1
Maximum window size		15	15	15	15	15
Data Link Layer						
Number of Rx buffers (*bytes)	Null2	17*1510	40*1510	230*256	40*1600	20*1510
	ES-IS	14*1510	32*1510	200*256	40*1600	20*1510
Number of Tx buffers (*bytes)	Null2	5*1500	20*1500	4*1500	20*1500	4*1500
	ES-IS	2*1500	5*1500	4*1500	4*1500	4*1500
Number of EDL LSAPs		16	16	16	16	16

iNA 960 COMMputer Jobs

The iNA 960 COMMputer jobs are available either as first-level jobs that you include with the ICU or as loadable jobs that you install with the **sysload** command. For values configured into iNA 960 COMMputer jobs, see the default values on screens in the ICU.

See also: *i*.job, System Configuration and Administration*

Configuration of iNA 960 MIP Jobs

Table A-3 lists values configured into the MIP jobs.

Table A-3. MIP Job Configuration

Parameter	ipcl2.job	i552a.job	icemb2.job
NIC (see Note 1)	not applicable	SBC 552A	186/530
Multibus II port ID	not applicable	not applicable	505H
Number of external mailboxes	10	10	10
Number of internal ports	10	10	10
Default wakeup port (see Note 2)	360H	8B4H	not applicable
Default interrupt level (see Note 2)	21H	48H	not applicable
Default download file (see Note 1)	/net/inapcl2n.32l	/net/ina552an.36l	/net/ina530n.32l
Boot address	0CC000H	1040H	not applicable
Off-board layers (on the NIC)	Name Server External Data Link Transport Virtual Circuit Transport Datagram Network Management Facility		

Note 1: Change these values in the *rmx.ini* file or the ICU, or with a BPS parameter for the iRMX III OS.

Note 2: This is specified in the **sysload** invocation for DOSRMX.

See also: MIP configuration values on the IPMPJ, MIP1, MIP2, and MIPAT screens of the ICU

Configuration of iRMX-NET Jobs

Table A-4 lists values configured into the iRMX-NET jobs. You can change some of these values in the *rmx.ini* file or in the ICU.

Table A-4. iRMX-NET Configuration

Parameter	Value
User Administration	
User Definition File	:sd:rmx386/config/udf
Client Definition File	:sd:rmx386/config/cdf
Loadname File	/net/data
Default client node name	rmx
Default client password	1234567
File Server	
Max virtual circuits (VCs, 1 per client)	20
Max users per client VC	5
Max client jobs per VC	30
Max file attachments per VC	100
Max open files per VC	40
Max open files per client job	30
Max outstanding client requests	35
Number of small read/write buffers (* size in bytes)	37*1488
Number of large read/write buffers (* size in bytes)	3*10240
File Consumer	
Max remote file driver requests	21
Max server connections	20
Data Link packet size (bytes, + 12 for header)	1488
Wait for server connection (sec.)	45
Wait for server response to request (sec.)	60
Apex File Access (AFA)	
Max public devices	5
Max public directories	30
Max concurrent I/O requests (AFA tasks)	15
Support DOS and UNIX file attributes	yes
Support DOS wildcard file delete	yes
Max concurrent file searches (wildcard)	32
Hold file connection after wildcard search (sec.)	3
Remote boot server	yes
Public devices	sd: and :bb:
Public directories (display with the publicdir command)	



Data Flow in MIP and COMMputer Jobs

B

Your application requests iNA 960 network services through special data structures called request blocks. As far as the application is concerned, the interchange of request blocks is the same whether you use a COMMputer job or a MIP job. This appendix describes how the interchange actually occurs for the different kinds of job.

Data Interchange with the MIP

A MIP job provides a way to exchange request blocks between the application, which runs with the OS, and iNA 960 software that runs on a separate NIC. What we call a MIP job runs on the board with the OS. However, the MIP actually includes not only the MIP job, but also a MIP interface to iNA 960 on the NIC itself. As shown in the diagram below, the MIP interface on the NIC is called an iNA 960 environment interface.

Figure B-1. MIP Protocol Model

The particular system environment determines how the MIP actually exchanges the iNA 960 request blocks. The exchange may occur over a system bus, or a host board

and NIC may share memory. This is why each MIP job and iNA 960 download file are configured for a specific system environment.

To the application job, the MIP is completely transparent because a single high-level call, **cq_comm_rb**, exchanges request blocks. The MIP handles the lower-level primitive functions that perform the exchange for a specific physical environment.

To the iNA 960 transport service, the MIP is completely transparent, because access to it and other environmental resources occur through an environment interface. The iNA 960 transport service requests logical functions, which the environment interface converts to physical resource functions.

The iNA 960 software includes three MIP jobs for specific bus types:

- *i552a.job* for Multibus I
- *ipcl2.job* for PCs
- *icemb2.job* for the Multibus II MIP, also called the LAN Controller Interface (LCI)

The following sections describe these MIPs in more detail.

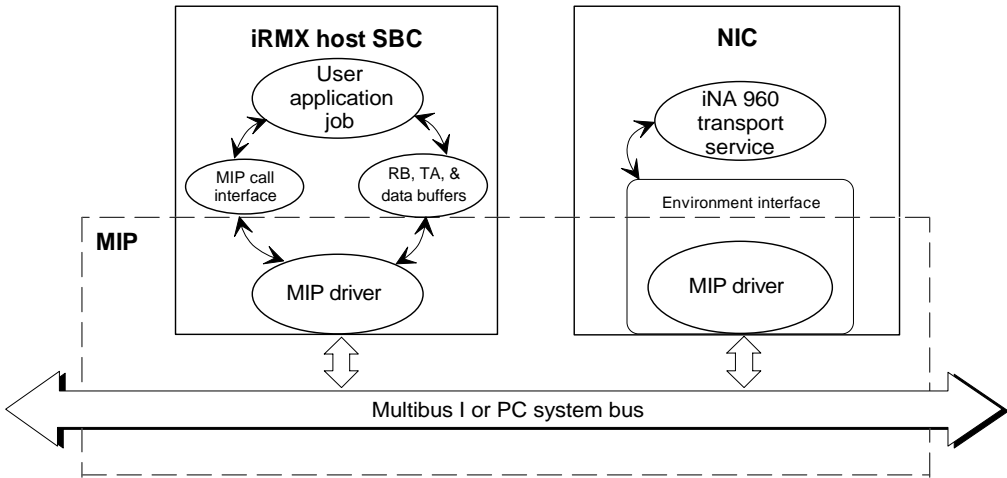
See also: Network Software Implementation, Chapter 7, for COMMengine and COMMputer environments

Multibus I and PC Bus MIP

In the Multibus I and PC Bus system architectures, the application runs on an iRMX host single-board computer and the iNA 960 software runs on a separate NIC. The iNA 960 request blocks, transport address buffers, and user data are not physically exchanged over the bus. Because host memory is dual ported, only the buffer addresses in host memory are exchanged over the bus.

Access to the shared host memory (using the system bus) by the iNA 960 transport service occurs through a logical window. This logical window is managed by the iNA 960 environment interface on the NIC and maps a portion of the NIC's local address space to the part of the host memory address space that contains the buffers pertaining to the user's iNA 960 service requests. To the iNA 960 transport service, the shared memory is part of its local address space.

The MIP model for the Multibus I and PC Bus system architecture is shown in Figure B-2. The MIP consists basically of two drivers, one for the iRMX host and one for the NIC.



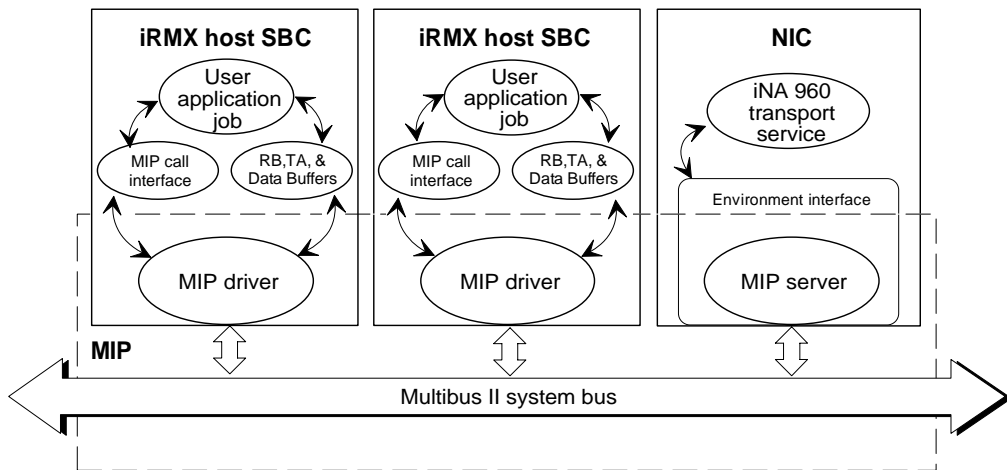
W2049

Figure B-2. Multibus I and PC Bus MIP Model

Multibus II MIP

The MIP for the Multibus II system architecture is also known as the LAN Controller Interface (LCI). In the Multibus II system architecture, there may be multiple user applications running on multiple iRMX host single-board computers and a single instance of the iNA 960 software running on a single NIC. The iNA 960 request blocks, transport address buffers, and user data are physically exchanged over the bus using the Multibus II message passing service. The MIP puts all three data structures into a message, transmits the message over the bus, and on the other end parses the message into the three data structures. The request block goes on to its destination, and the transport address and user data buffers are copied to local memory.

In addition to the Multibus II transport service, the MIP for this environment provides server functions for the iNA 960 environment interface. The Multibus II MIP shown in Figure B-3 consists of MIP jobs for each iRMX host and a MIP server for the NIC, which is part of the iNA 960 download file.



W2052

Figure B-3. Multibus II MIP Model

Data Interchange in a COMMputer Job

In the COMMputer architecture shown in Figure B-4, the application and the iNA 960 software are jobs running under the iRMX OS. In this environment, iNA 960 request blocks and referenced user data are stored in local memory and there is no MIP.

Figure B-4. COMMputer MIP Model



iNA 960 Network Objects

This appendix lists and describes the iNA 960 network objects and events that can be accessed through the Network Management Facility. The list includes the object ID number, object type, access permissions, the size of the object, and a brief description. The objects are listed according to which iNA subsystem they belong to.

The object types are:

Counter	A 16- or 32-bit counter that records the number of times a particular action occurs. It is an unsigned integer and may be either of two types: Wrap-around, which resets to 0 on overflow. This type is indicated by WCounter in the table. Sticky, which sticks at its highest value on overflow. This type is indicated by SCounter in the table.
Threshold	A 16-bit threshold value for the number of events that may occur before the net agent notifies the net manager. Threshold objects are used in event notification.
Timerval	A 32-bit timer value specified in milliseconds.
Time	An ASCII string which shows the time in a year, month, day, hour, minutes, and seconds format.
Parameter	Adjusts the actual operation of a layer (Intel private objects only).
Value	Anything that is not one of the types listed above.

Each object is assigned access permissions that are enforced by the NMF functions. The access permissions are:

- R READ_OBJECT is allowed
- S SET_OBJECT is allowed
- C READ_AND_CLEAR_OBJECT is allowed

In Table C-1, the first two characters of the object ID represent the Data Link subsystem. Substitute the characters 2x in the ID column with:

Value	Subsystem
20H	Data Link for boards with the 82586 component, including the first MIX560 board in the system
21H	Data Link for SBX 586 board
22H	Data Link for the second MIX560 board in the system
23H	Data Link for the third MIX560 board in the system
24H	Data Link for boards with the 825595TX component, including the EtherExpress PRO/10 and SBC P5090 (See also Table C-2)
25H	DEC 21143 component, SBC P5200 PC-compatible boards, all versions

Table C-1. 802.3 Data Link Objects

ID	Type	Access	Size	Description
2x00H	Value	R	BYTE	Data Link type: returned value is 2xH
2x01H	Value	R	WORD_32	Line speed: physical transmission rate in bits/second
2x02H	Value	R	6 BYTES	Host ID: hardware MAC (Ethernet) address
2x03H	WCounter	RC	WORD_32	Total number of packets sent
2x04H	SCounter	RC	WORD_16	Primary collisions: number of packets transmitted that had at least 1 collision
2x05H	SCounter	RC	WORD_16	Secondary collisions: total of collisions encountered after a primary collision
2x06H	SCounter	RC	WORD_16	Number of packets discarded because the maximum number of collisions was exceeded
2x07H	WCounter	RC	WORD_32	Total of packets received from the network
2x08H	WCounter	RC	WORD_16	Total of packets dropped due to CRC errors
2x09H	WCounter	RC	WORD_16	Packets dropped due to alignment errors
2x0AH	WCounter	RC	WORD_16	Resource errors: number of times the Data Link service ran out of resources
2x0BH	WCounter	RC	WORD_16	DMA overruns: number of times a received packet was dropped because of an 82586 DMA overrun error (the 82586 does not provide statistics on transmit DMA underruns)
2x0CH	WCounter	RC	WORD_16	Number of restarts: total of times software has reset the 82586 due to a lockup failure

continued

Table C-1. 802.3 Data Link Objects (continued)

2x0DH	Value	R	Variable	Multicast address list: List of 48-bit multicast addresses for which the subnet is listening
2x0EH	WCounter	RC	WORD_32	RawEDL: number of frames lost due to lack of posted EDL buffers
ID	Type	Access	Size	Description
2x0FH	WCounter	RC	WORD_32	RawEDL: total frames delivered to EDL user
2x10H	WCounter	RC	WORD_32	Bytes received by Data Link for RawEDL

Table C-2 lists objects that are specific to the Data Link subsystem for iNA 960 software on boards that use the 825595TX component, including the EtherExpress PRO/10 and SBC P5090.

Table C-2. 802.3 Data Link Objects With the 825595TX Component

ID	Type	Access	Size	Description
2411H	Value	RC	WORD_32	Total transmission errors
2412H	Value	RC	WORD_32	Number of late collision errors
2413H	Value	RC	WORD_32	Number of lost carrier errors
2414H	Value	RC	WORD_32	Number of underrun errors
2415H	Value	RC	WORD_16	Number of short frames received
2416H	Value	RC	WORD_16	Number of received packets with collisions
2417H	Value	R	WORD_16	Number of packets to be transmitted onchip
2418H	Value	R	BYTE	Flag indicates if EarlyTx is on
2419H	Value	R	BYTE	Flag indicates if EarlyRx is on
241AH	Value	R	BYTE	I/O speed

Table C-3 lists objects that are specific to the Data Link subsystem for iNA 960 software on boards that use the DEC21143 component, including various PCI-based NICs and SBC P5200.

Table C-3. 802.3 Data Link Objects With the DEC21143 Component

ID	Type	Access	Size	Description
2511H	Value	RC	WORD_32	Total transmission errors
2512H	Value	RC	WORD_32	Number of late collision errors
2513H	Value	RC	WORD_32	Number of lost carrier errors
2514H	Value	RC	WORD_32	Number of underrun errors
2515H	Value	RC	WORD_16	Number of short frames received
2516H	Value	RC	WORD_16	Number of received packets with collisions
2517H	Value	R	WORD_16	Number of packets to be transmitted onchip
2518H	Value	R	BYTE	Flag indicates if EarlyTx is on
2519H	Value	R	BYTE	Flag indicates if EarlyRx is on
251AH	Value	R	BYTE	I/O speed

Table C-4 lists objects that are specific to the Data Link subsystem for iNA 960 software that uses the Multibus II subnet.

Table C-4. 802.3 Data Link Objects for the Multibus II Subnet

ID	Type	Access	Size	Description
2F00H	Value	R	BYTE	Data Link type: returned value is 2FH
2F01H	Value	R	WORD_32	Line speed: physical transmission rate in bits/second
2F02H	Value	R	6 BYTES	Host ID: the MAC (Ethernet) address
2F03H	WCounter	RC	WORD_32	Total number of packets sent
2F04H	WCounter	RC	WORD_32	Total number of packets received
2F05H	WCounter	RC	WORD_16	Resource errors: number of times the Data Link service ran out of resources
2F06H	Value	R	Variable	Multicast address list: List of 48-bit multicast addresses for which the subnet is listening
2F07H	WCounter	RC	WORD_32	RawEDL: number of frames lost due to lack of posted EDL buffers
2F08H	WCounter	RC	WORD_32	RawEDL: total frames delivered to EDL user
2F09H	WCounter	RC	WORD_32	Bytes received by Data Link for RawEDL
2F0AH	Value	RC	WORD_32	Number of rq_send errors
2F0BH	Value	R	Variable	<p>Indicates which slots are using the Multibus II subnet. If you access this value programmatically, it is an array of 20 7-byte fields. The first 7-byte field applies to slot 0, on up to slot 19. The first 6 bytes of each field contain the Ethernet address for that slot. The last byte is either 0 (does not use Multibus II subnet) or 0FFH (uses Multibus II subnet).</p> <p>If you access this value with inamon, the slots that use the Multibus II subnet are indicated with an asterisk; Ethernet addresses are not displayed.</p>

Table C-5. IP Network Layer Objects

ID	Type	Access	Size	Description
3140H	WCounter	RC	WORD_32	Number of null header packets sent
3141H	WCounter	RC	WORD_32	Number of null header packets received
3142H	WCounter	RC	WORD_32	Number of non-null header packets sent
3143H	WCounter	RC	WORD_32	Number of non-null header packets received
3144H	WCounter	RC	WORD_32	Number of packets discarded due to lack of Network Layer resources
3145H	WCounter	RC	WORD_32	Number of error PDUs received
3147H	WCounter	RC	WORD_16	Number of packets discarded due to checksum failure
3148H	WCounter	RC	WORD_16	Number of packets discarded due to lifetime exceeded in transit
3149H	WCounter	RC	WORD_16	Number of packets discarded due to ISO protocol or procedure violations
314AH	WCounter	RC	WORD_16	Number of packets with unrecognized destination NSAP selector
314BH	Threshold	RSC	WORD_16	Threshold for number of packets with lifetime exceeded: an event is generated if this threshold is exceeded

Table C-6. Router Objects - Static

ID	Type	Access	Size	Description
3801H	Table	RS	struct*	Local Subnet Table
3802H	Table	RS	struct*	Specific Router Table
3803H	Table	RS	struct*	Default Router Table
3804H	Table	RS	struct*	Destination Subnet Table

* See Chapter 16 for structure definitions

Table C-7. Router Objects - ES-IS

ID	Type	Access	Size	Description
3900H	Value	R	WORD_16	Maximum number of entries allowed in the Local End System Table
3901H	WCounter	R	WORD_16	Number of Local End System Table entries currently used
3902H	Value	R	WORD_16	Maximum number of entries allowed in the Intermediate System Hello Table
3903H	WCounter	R	WORD_16	Number of Intermediate System Table entries currently used
3904H	Value	R	WORD_16	Maximum number of entries allowed in the Static Intermediate System Table
3905H	WCounter	R	WORD_16	Number of Static Intermediate System Table entries currently used
3906H	Value	R	WORD_16	Maximum number of entries allowed in the Reachable NSAP Address Table
3907H	WCounter	R	WORD_16	Number of Reachable NSAP Address Table entries currently used
3908H	Value	R	WORD_16	Maximum number of entries allowed in the Local NSAP Address Table
3909H	WCounter	R	WORD_16	Number of Local NSAP Address Table entries currently used
390AH	Value	R	WORD_16	Maximum number of entries allowed in the Subnet Table
390BH	WCounter	R	WORD_16	Number of Subnet Table entries currently used
390CH	Flag	RS	BYTE	Indicates whether to use Refresh Redirect function; true if not 0
390DH	Flag	RS	BYTE	Indicates whether to use Configuration Notification function; true if not 0
390EH	Value	RS	WORD_16	Specifies how frequently to send End System Hellos, in 500 ms. units
390FH	Value	RS	WORD_16	Specifies how frequently to send Intermediate System Hellos, in 500 ms. units
3910H	Value	RS	WORD_16	Specifies a holding time to send with End System Hellos, in 500 ms. units

continued

Table C-7. Router Objects - ES-IS (continued)

ID	Type	Access	Size	Description
3911H	Value	RS	WORD_16	Specifies a holding time to send with Intermediate System Hellos, in 500 ms. units
3912H	Flag	RS	BYTE	Indicates whether to send End System Hellos; true if not 0
3913H	Flag	RS	BYTE	Indicates whether to send Intermediate System Hellos; true if not 0
3914H	Flag	RS	BYTE	Indicates if Redirect PDUs should be transmitted when necessary; true if not 0
3915H	Flag	RS	BYTE	Indicates if End System Hellos should be received and processed; true if not 0
3916H	Flag	RS	BYTE	Indicates if Intermediate System Hellos should be received and processed; true if not 0
3917H	Flag	RS	BYTE	Indicates if Redirect PDUs should be received and processed; true if not 0
3918H	Flag	RS	BYTE	Indicates if checksums should be sent with ES-IS PDUs; true if not 0
3919H	Table	RS	struct *	Local NSAP Address Table
391AH	Table	RS	struct *	Local End System Table
391BH	Table	RS	struct *	Intermediate System Hello Table
391CH	Table	RS	struct *	Static Intermediate System Table
391DH	Table	RS	struct *	Reachable NSAP Address Table
391EH	Table	RS	struct *	Subnet Table
391FH	Value	RS	BYTE	Array containing the subnet address for multicast messages to all End Systems; address length is the first byte of the array.
3920H	Value	RS	BYTE	Array containing the subnet address for multicast messages to all Intermediate Systems; address length is first byte of array.

continued

* See Chapter 16 for structure definitions

Table C-7. Router Objects - ES-IS (continued)

ID	Type	Access	Size	Description
3921H	Flag	RS	BYTE	MAP 2.1 compatibility; true if not 0. This enables sending null header PDUs if there is an entry for the destination NSAP address in the Local End System Table or if the address is recognized through local addressing (local addressing flag is true and the NSAP address matches a NSAP prefix in the Subnet Table).
3922H	Flag	RS	BYTE	Local addressing; true if not 0. This enables decomposition of the destination NSAP address to attempt to match it to an NSAP prefix in the Subnet Table.
3923H	Flag	RS	BYTE	iNA 960 Release 1 addressing; true if not 0. This enables recognition of iNA 960 R1 format network addresses: R1 addresses always contain the destination subnet address.
3924H	Flag	R	BYTE	Automatically configure local NSAP address; true if not 0. This causes iNA 960 to determine its own NSAP address when it initializes; this flag can only be set in the iNA 960 configuration file, not with the NMF SET_OBJECT command.
3937H	Value	R	WORD_16	Maximum number of entries allowed in the Multicast NSAP Address Table.
3938H	WCounter	R	WORD_16	Number of Multicast NSAP Address Table entries currently used.
3939H	Value	R	WORD_16	The unique multicast address used by the Name Server.

Table C-8. Transport Layer Objects - Virtual Circuit Connection Independent

ID	Type	Access	Size	Description
4000H	Value	R	BYTE	Virtual circuit type: returned value is 0.
4001H	Value	R		Connection ID vector: WORD_16 array where each not 0 element is an allocated connection ID. Size of this object is twice as many bytes as the maximum number of connections.
4002H	Value	R	BYTE	ISO transport number: version number of the ISO VC subsystem.
4003H	Value	R	WORD_16	Max VCs: maximum number of connections supported by the VC subsystem.
4004H	Value	R	WORD_16	Same as 4003H.
4005H	Value	R	WORD_16	Same as 4003H.
4006H	Value	R	WORD_16	Active CDB's: number of connection databases currently open but not necessarily established.
4007H	Value	R	WORD_16	CDB size: size in bytes of a connection database.
4008H	Parameter	RS	WORD_16	Default persistence count: number of times the local transport entity attempts to establish a connection when the remote transport entity explicitly rejects the connection attempt. The default count is assigned to new connections that request it.
4009H	Parameter	RS	WORD_16	Default abort timeout: amount of time (in units of 51 ms.) an unacknowledged segment is transmitted before automatically aborting the connection. The default timeout is assigned to new connections that request it. Value OFFFFH indicates that an automatic abort is never to occur (this does not apply to sending a disconnect request without a response).

continued

Table C-8. Transport Layer Objects - VC Connection Independent (continued)

ID	Type	Access	Size	Description
400AH	Parameter	RS	WORD_32	Default retransmit timeout: initial amount of time (in 100 microsecond units) the Transport Layer waits before retransmitting an unacknowledged TPDU. This value is used on all new connections. The retransmit timeout may be subsequently altered by a dynamic algorithm.
400BH	Parameter	RS	WORD_32	Minimum retransmit timeout: minimum time (in 100 microsecond units) the Transport Layer will ever wait before transmitting an unacknowledged TPDU. The initial value is configurable.
400CH	Parameter	RS	WORD_16	Closing abort timeout: amount of time (in 51 ms. units) for which the Transport Layer will attempt to send a connection close request without receiving a response before aborting the connection.
400DH	Parameter	RS	WORD_32	Flow control window timeout. Once a connection is established, the local transport entity sends flow control window acknowledgement packets (AK TPDUs) to the remote entity at regular intervals, to signal to the remote entity that it is still functioning when there is no other activity on the connection. These packets also inform the remote transport of the most current local flow control window status. This object specifies the time interval (in 100 microsecond units) between these packets.
400EH	Parameter	RS	WORD_16	Inactivity maximum count: number of times local transport transmits an unacknowledged flow control window acknowledgement packet (AK TPDU) before aborting the connection.
400FH	SCounter	RC	WORD_16	Total duplicate TPDUs rejected: total number (over all connections) of received TPDUs rejected due to duplicate sequence numbers.

continued

Table C-8. Transport Layer Objects - VC Connection Independent (continued)

ID	Type	Access	Size	Description
4010H	SCounter	RC	WORD_16	Total checksum errors: The total number (over all connections) of received TPDU's that were rejected because of checksum errors.
4011H	SCounter	RC	WORD_16	Total retransmission: The total number of times (over all connections) that acknowledgeable TPDU's were retransmitted.
4012H	SCounter	RC	WORD_16	Total resource errors: The total number (over all connections) of data TPDU's discarded because receive buffers were not available.
4013H	Value	R	BYTE	Maximum NSAP address length: The maximum length of a remote NSAP address.
4014H	Value	R	BYTE	Maximum TSAP selector length: The maximum length of local or remote TSAP selectors.
4015H	Value	R	WORD_16	Local NSAP selector: The NSAP selector bound to the local Network Layer.
4018H	Parameter	RS	WORD_16	Default connection negotiation options.
4019H	Parameter	RS	BYTE	Maximum TPDU Size: The value (specified as a power of 2) used for maximum TPDU size in the negotiation phase of connection establishment by the local transport entity.
401AH	Parameter	RS	BYTE	An additional option field (encoded as in ISO 8073) assumed to be requested by a remote entity when no such option parameter is in the received TPDU.
401BH	Parameter	RS	BYTE	The maximum TPDU size (specified as a power of 2) assumed, when no size is specified by a remote entity in the received TPDU.
401CH	Parameter	RS	WORD_16	Maximum normal window size: largest receive buffer credit that can be reported on a connection by the local transport entity to a remote transport entity on a connection using normal sequence number format.

continued

Table C-8. Transport Layer Objects - VC Connection Independent (continued)

ID	Type	Access	Size	Description
401DH	Parameter	RS	WORD_16	Maximum extended window size: largest receive buffer credit that can be reported on a connection by the local transport entity to a remote transport entity on a connection using extended sequence number format.
401EH	Parameter	RS	WORD_16	Minimum credit: smallest receive buffer credit that can be reported on a connection by the local transport entity to a remote transport entity.
401FH	Parameter	RS	WORD_32	Open window timeout: interval (in 100 microsecond units) between successive acknowledgements (AK TPDUs) that announce the opening of a previously closed credit window to avoid flow control deadlock.
4020H	Parameter	RS	WORD_16	Maximum open window count: maximum number of open window AK TPDUs transmitted before the sender assumes that the remote transport entity has received the open window credit information. When this count is reached, the local transport entity stops transmitting open window AK TPDUs.

Table C-9. Map 2.1 Transport Objects

ID	Type	Access	Size	Description
4040H	WCounter	RS	WORD_32	Total number of bytes of application data sent over Transport Layer VCs (does not include datagrams).
4041H	WCounter	RS	WORD_32	Total number of bytes of application data received over Transport Layer VCs (does not include datagrams).
4042H	WCounter	RS	WORD_32	Total bytes of expedited data sent.
4043H	WCounter	RS	WORD_32	Total bytes of expedited data received.
4044H	WCounter	RS	WORD_32	Total number of TPDUs successfully transmitted.
4045H	WCounter	RS	WORD_32	Total number of TPDUs retransmitted (overlaps object 4011H).
4046H	WCounter	RS	WORD_32	Total number of TPDUs received.

continued

Table C-9. Map 2.1 Transport Objects (continued)

ID	Type	Access	Size	Description
4047H	WCounter	RS	WORD_32	Total number of data TPDU retransmitted.
4048H	WCounter	RS	WORD_16	Total number of retransmitted AK TPDU.
404AH	WCounter	RS	WORD_16	Total number of application disconnect requests.
404BH	WCounter	R	WORD_16	Number of open connections (same as object 4006H, Active CDBs).
404CH	WCounter	RS	WORD_16	Total number of Type 1 connection refusals: connection exceeds node connection limit.
404DH	WCounter	RS	WORD_16	Total number of Type 2 connection refusals: all others.
404EH	WCounter	RS	WORD_16	Total of successful inbound connections.
404FH	WCounter	RS	WORD_16	Total of successful outbound connections.
4050H	WCounter	RS	WORD_16	Total of unsuccessful inbound connections.
4051H	WCounter	RS	WORD_16	Total of unsuccessful outbound connections.
4052H	WCounter	RS	WORD_16	Total number of timed out connections.
4053H	WCounter	RS	WORD_16	Total of connect request retransmissions.
4054H	Timerval	R	WORD_32	Maximum local acknowledge time, between receipt of TPDU and transmission of acknowledgement. This object is set when the system is configured.
4055H	Timerval	R	WORD_32	Maximum local retransmission time; set when the system is configured.
4056H	Integer	R	WORD_16	Maximum number of retransmissions allowed; set when the system is configured.
4057H	Timerval	R	WORD_32	Default connection inactivity time: how long a connection can be inactive before a disconnect request is sent. This is set when the system is configured.
4058H	Integer	R	WORD_16	Maximum TPDU size in bytes; set when the system is configured.
4059H	WCounter	RS	WORD_16	Total number of protocol errors.
405AH	WCounter	RS	WORD_16	Total number of invalid received TPDU.

Table C-10. Map 2.1 Transport Objects - Virtual Circuit Connection Dependent

ID	Type	Access	Size	Description
4081H	Value	R	Variable	Local TSAP selector for the connection (specified in the modifier field of the NMF READ_OBJECT request block). The first byte of the value is the length of the selector.
4082H	Value	R	Variable	Remote NSAP address: of the entity at the remote end of the connection. If the application performs a partially specified or unspecified passive open, this object will be 0 until the connection is established. The first byte of the value is the length of the address.
4083H	Value	R	Variable	Remote TSAP selector for the specified connection. The first byte of the value is the length of the selector.
4084H	Value	R	BYTE	Connection State: For descriptions, see the state parameter under connection dependent status in the Transport STATUS command.
4085H	Value	R	WORD_16	Remote connection ID: the reference of the specified connection, set after the connection is established.
4086H	Parameter	RS	WORD_16	Persistence Count: number of times a connection request is retransmitted when the remote entity explicitly refuses it.
4087H	Parameter	RS	WORD_16	Abort Timeout for this connection: amount of time (in units of 51 ms.) an unacknowledged segment is transmitted before automatically aborting the connection.
4088H	Parameter	RS	WORD_32	Retransmit Timeout for this connection: amount of time (in 100 microsecond units) the Transport Layer waits before retransmitting an unacknowledged TPDU. This is determined by a dynamic algorithm.
4089H	Value	R	WORD_32	Next Transmit Sequence Number: to be used with the next TPDU transmitted (not always the highest number).
408AH	SCounter	RC	WORD_16	Duplicate TPDU's Rejected: total of duplicate received TPDU's discarded by the transport entity for this connection.

continued

**Table C-10. Map 2.1 Transport Objects - Virtual Circuit Connection Dependent
(continued)**

ID	Type	Access	Size	Description
408BH	SCounter	RC	WORD_16	Retransmitted TPDU: total number of times an unacknowledged TPDU has been retransmitted for this connection.
408CH	SCounter	RC	WORD_16	Resource Errors: total of times that TPDU received on this connection were rejected because receive buffers were not available.
408DH	Value	R	WORD_16	Client Options: specified by the client in the connection request.
408EH	Value	R	BYTE	Class Options: the ISO class of services and sequence number format negotiated on this connection are: 40H - Class 4 and normal (7-bit) format 42H - Class 4 and extended (31-bit) format
408FH	Value	R	BYTE	Additional Options: negotiated on the connection, where only bits 0 and 1 are meaningful. The values are: 0 - no expedited service and checksum 1 - expedited service and checksum 2 - no expedited service and no checksum 3 - expedited service and no checksum
4090H	Value	R	BYTE	Maximum TPDU Size (as a power of 2), negotiated for this connection.
4091H	Value	R	WORD_16	Maximum TPDU Data Length: maximum length in bytes of data that can be sent in one TPDU. This is the smaller of the Maximum TPDU Size or the configured maximum NSDU size, minus the header length.
4092H	Value	R	WORD_16	Inactivity Count: number of times an inactivity AK has been sent without response from the remote entity.
4093H	Value	R	Variable	Local NSAP selector for this connection; the first byte is the length of the selector.

Table C-11. Map 2.1 Transport Objects - Transport Datagram

ID	Type	Access	Size	Description
4100H	Value	R	BYTE	Datagram Type: returned value is 1.
4101H	Value	R	BYTE	Datagram receive queue size: maximum number of TSAP selectors for which the client can post buffers.
4103H	SCounter	RC	WORD_16	Total number of datagrams transmitted.
4104H	SCounter	RC	WORD_16	Total number of datagrams received.
4105H	SCounter	RC	WORD_16	Total datagram resource errors: number of datagrams discarded due to lack of buffers.
4106H	SCounter	RC	WORD_16	Total datagram checksum errors: number of datagrams discarded due to checksum errors.
4107H	SCounter	RC	WORD_16	Total datagram address errors: number of datagrams discarded due to illegal address fields in the header.

Table C-12. NMF Objects

ID	Type	Access	Size	Description
8049H	Time	RS	17 BYTES	System time.
804AH	WCounter	R	WORD_16	Time reset counter: This is incremented every time the system time is set.
804BH	Value	R	2 BYTES	iNA 960 version number.

Table C-13. Network Layer Events

ID	Name	Description
3100H	PDU Lifetime Threshold Exceeded	The number of packets discarded because they were unclaimed longer than the PDU lifetime threshold value.

Table C-14. Transport Layer Events

ID	Name	Description
4000H	Abnormal Transport Layer Connection Abort	The provider of an established transport connection has terminated the connection.
4001H	Transport Layer Bad Destination Address	The destination TSAP address in a connection request does not exist; there is no entity waiting at the destination TSAP selector. This applies to both incoming and outgoing PDUs.
4002H	Transport Layer Protocol Violation	Some violation of Transport Layer protocol has occurred



Related Documentation

The manuals listed here provide additional network information. Many are available from your sales representative:

- *iRMX Virtual Terminal Software User's Guide*
- *OpenNET PCL2 for DOS Installation Guide*
- *PCL2 LAN Controller User's Guide*
- *SV-OpenNET User's Manual*
- *SV-OpenNET Installation and Administration Manual*
- *SBC/SXM 552A IEEE, 802.3 Communications Controller User's Guide*
- *32-Bit Local Area Network (LAN) Component User's Manual*



/net/data file, 18, 135
:config/
 terminals file, 352
:sd:net/data file, 38, 39, 42, 51, 58, 135
:sd:net/data.ex file, 18, 135
? (question mark) character, 52

A

abort timeout, 406, 407, 411
abort timeout value, 228
ACCEPT_CONNECT_REQUEST, Transport
 command, 190
active open, 186
ADD_NAME, Name Server command, 150
ADD_SEARCH_DOMAIN, Name Server
 command, 153
address format, 181
address match tests, 202
addressing authority, 72
addressing network buffers, 178
Administrative Unit, see AU
AFI, 72, 74
Apex File Access (AFA) module, 65
ATTACH_AGENT, NMF command, 306
attachdevice command, 27, 31, 46, 54
AU (administrative Unit), iRMX-NET
 configuration example, 37
AU (Administrative Unit), iRMX-NET, 11
 master node, 12, 20
 Master UDF, 12, 20
 security, 12
AWAIT_CLOSE, Transport command, 193
AWAIT_CONNECT_REQUEST_CLIENT,
 Transport command, 196
AWAIT_CONNECT_REQUEST_TRAN,
 Transport command, 196

AWAIT_EVENT, NMF command, 309

B

bcl command, 346
bexp.a86 file, 343
bexp.csd file, 344
BIOS, 65
boot client, 294, 304, 333, 335
 adding name to CDF, 352
 hardware and OS requirements, 333
boot file
 configuration of, 339
 format of, 361
 generating, 338
boot requests, 358
boot response, 358
boot server, 294, 304, 333, 335
 configuring, 346
 load files for, 350
booting
 diskless nodes, 333
 remote, 333
bootstrap loader, 333
br38.csd file, 341
br3expgen.csd file, 341
broadcast subnet, 71
buffers
 addressing network, 178
 availability test, 202
 contiguous, 184
 maximum transport protocol length, 231
 network data, 108
 noncontiguous, 184

C

carrier errors, 399, 400
case sensitivity, 49, 55, 56

- ccinfo file, 343
 - creating, 346
 - generating, 348
- ccinfo.bdf file, 347
- CDB (connection database), 186
 - maximum number of, 210
- CDF (Client Definition File), 352
- CDF (Client Definition File), 11, 12, 13, 22
- CHANGE_VALUE, Name Server command, 155
- checksum errors, 408
- chgid utility, UNIX, 255
- circuit, virtual, 2
- class codes, 347, 348
- client, 2
 - iRMX-NET, 3
 - iRMX-NET, name, 11, 22
 - validation, 13
 - verified, 12, 22
 - verifying, 11
- Client Definition File, see CDF
- client-based protection, iRMX-NET, 12, 13
- CLOSE, Transport command, 206
- collision errors, 399, 400
- collisions, 398, 399, 400
- COMMengine, 60, 144
- COMMputer, 40, 60, 144
 - data flow in, 395
- Configuration Notification function, 403
- CONFIGURE, Data Link command, 264
- configuring
 - AU, 37
- CONNECT, Data Link command, 266
- connection database, 186
- connectionless transport, 178
- connections
 - establishing network, 184, 186
 - reference ID, 210
 - terminating, 184, 188
- contiguous buffers, 184
- copy command, 28, 29
- cq*.ext files, 109
- cq*.h files, 109
- cq_comm_multi_status call, 114
- cq_comm_ptr_to_dword call, 111, 116
- cq_comm_rb call, 111, 117
- cq_comm_status call, 121

- cq_create_comm_user call, 111, 123
- cq_create_multi_comm_user call, 124
- cq_delete_comm_user call, 111, 126
- cqtransp.h file, 179
- cqtransp.lit file, 179
- CSMA/CD, 256

D

- data
 - buffers, 108
 - structures, 110
 - transferring, 186
- Data
 - Link layer, 255
 - Link objects, 398
 - for 82595TX, 399
 - for DEC21143, 400
 - for Multibus II subnet, 401
- data file (iRMX-NET), 18
- Data Link
 - type, 401
- data.ex file, 18, 135
- datagram objects, 413
- datagram service, 178, 189, 235, 255
- DEC21143 component
 - Data Link objects for, 400
- dedicated server, 2
- Default Router Table, 368, 402
- default TSAP selector, 180
- delete request block data structure, 127
- DELETE_NAME, Name Server command, 157
- DELETE_PROPERTY, Name Server command, 159
- DELETE_SEARCH_DOMAIN, Name Server command, 161
- deletename command, 141
- Destination Subnet Table, 368, 402
- DETACH_AGENT, NMF command, 312
- detachdevice command, 28, 46, 54
- dir command, 28
- directories
 - /bsl, 343
 - /net, 18
 - /rmx386/demo/network, 147
 - /rmx386/jobs, 27

- disconnect
 - message, 195
 - request, 208
- DISCONNECT, Data Link command, 269
- diskless nodes, booting, 333
- DLSAP (destination LSAP), 258
- domain, 72
- DOS, 4
 - client, 51
 - client, 49, 52
 - filenames, 51
 - interoperability, 47
 - pathname, 51
 - server, 49, 51
 - system, 49, 50
 - user, 49
 - wildcard characters, 52
- DOSRMX
 - networking, 17
- DUMP, NMF command, 313
- dynamic name resolution, 8
- dynamic routing, 74, 75

E

- EarlyRx, 399, 400
- EarlyTx, 399, 400
- ECHO, NMF command, 316
- EDL (External Data Link), 255, 256
- end of message, 231
- end system, 69
- End System Hellos, 403, 404
- enetinfo, UNIX command, 140
- EOM, 231
- EOT (end of transmission), 231
- EPROM
 - programming first stage in, 343
- error messages
 - remote boot, 355
- ES-IS
 - address, 75
 - objects, 403
 - protocol, 75
 - routing, 74, 365
- EtherExpress, 257, 334, 338
 - programming EPROM for, 343

- Ethernet address, 73, 139, 140, 142, 144, 398, 401
 - changing, 272
 - in NSAP address, 74, 75
 - of spokesman, 168
- events
 - Network Layer, 413
 - Transport Layer, 414
- EWENET module, 257, 334, 338
 - programming EPROM for, 343
- examples
 - applications, 147
 - applications, 178
 - AU configuration, 37
 - copying the CDF, 30
 - copying the UDF, 29
 - iRMX-NET setup, 37
 - modcdf command, 44
- exp.rem32 file, 341
- expedited data, 187, 222, 238
 - buffer, 236
- External Data Link Interface, 256

F

- FCTSAP, 143
- File Consumer, 66
- file server, 333
 - configuring, 350
 - specifying name of, 351
- File Server, 66
- files
 - access rights in UNIX, 57
 - access rights in UNIX, 57
 - access rights to, 32
 - accessing remote, 27
 - granting remote access to, 31
 - iNA 960 download files, 385
 - transparent access to, 2
 - with ? in name, 52
- findname command, 141
- flow control, 177, 238
- flow control window timeout, 407
- FLUSH, Data Link command, 271
- FSTSAP, 143

G

- general topology subnet, 71
- GET_NAME, Name Server command, 163
- GET_SEARCH_DOMAIN, Name Server command, 166
- GET_SPOKESMAN, Name Server command, 168
- GET_VALUE, Name Server command, 170
- getaddr command, 141
- getname command, 141
- group ID, UNIX, 11

H

- hardware requirements
 - boot client, 333
- header files, 109
- header packets, 402
- home directory, UNIX, 11
- host memory, 303

I

- IA_SETUP, Data Link command, 272
- ICU (Interactive Configuration Utility)
 - iRMX-NET configuration, 11, 15, 22
- IEEE 802 LAN, 257
- iNA 960, 59, 60
 - configuration values, 387
 - download files, 385
 - files, 385
 - Network Management Facility (NMF), 144
 - Null2, 140
 - subnet, 11
 - subnet number, 144
 - Transport Address, 144
 - Transport Software, 11, 60
 - Transport Software, 144
- ina*.32l files, 342
- ina*.32r files, 342
- ina530n.32l file, 349
- inactivity count, 412
- inamon command, 23
- INANLNUM, 143
- INARDY object, 112

- INARELNUM, 143
- INASUBNETxx, 144
- include files, 109
- Intermediate System Hello Table, 403, 404
- Intermediate System Hellos, 403, 404
- intermediate systems, 69
 - hello table, 374
- International Standards Organization, see ISO
- internetwork routing, 69, 73
 - ES-IS, 365
 - protocol, 74
 - static, 365
- interoperability
 - iRMX-NET, 4
- IP
 - addressing, 74
 - routing objects, 402
- iRMX, 51
 - and UNIX compatibility, 56
 - client, 56
 - files, 51
 - interoperability, 47
 - logon name, 11
 - root, 51
 - static user, 21
 - Super user, 20
 - symbols, 56
 - UDF (User Definition File), 11, 12, 13, 14, 20, 22
 - user directories, 20
 - World user, 21
- iRMX III, 40
- iRMX-NET
 - AU (Administrative Unit), 11, 12, 20, 22
 - CDF (Client Definition File), 11, 12, 13, 22
 - client, 3, 7
 - client name, 11, 22
 - client-based protection, 12, 13
 - configuration values, 388
 - default parameters, 38, 42
 - features, 3
 - ICU configuration, 11, 15, 22
 - interoperability, 4
 - loadable jobs, 388
 - load-time configuration, 11, 15, 22
 - Master UDF, 56

- Name Server, 8, 9, 18, 51
- network access, 15
- nodes, 58
- security, 12
- server, 3
- server name, 8, 18, 19
- setup example, 37
- software, 7
- software, 47
- spokesman node, 9
- ISO (International Standards Organization), 59
 - OSI Model, 59
 - protocols, 61
 - reason codes, 185
 - services, 61

J

- jobs
 - iRMX-NET, 388
 - MIP, 388
 - remotefd.job, 7

L

- LAN (Local Area Network), 1, 2
 - Controller Interface (LCI), see LCI
- lanstatus command, 23
- LCI (LAN Controller Interface), 394
- line speed, 401
- line terminators, 56
- LIST_TABLE, Name Server command, 173
- listname command, 132, 141, 351
- literal files, 109
- LLC (logical link control), 61, 71, 256
- load file
 - format of, 361
- load files, 350
- loadinfo file, 15
- loadname command, 18
- loadname command, 27, 50, 132, 134, 135, 136, 138
- loadrmx command, 334
- load-time configuration
 - iRMX-NET, 11, 15, 22
- local
 - end system table, 374

- node, 2
- object table, 141
- Local
 - NSAP Address Table, 375
 - Subnet Table, 368
- Local End System Table, 403, 404
- Local NSAP Address Table, 403, 404
- Local Subnet Table, 402
- Logical Link Control, see LLC
- logicalnames command, 28
- login shell, UNIX, 11
- logon name, iRMX, 11
- lookup_object call, 112
- LP486, 257
- LSAP (link service access point), 73, 74
 - identifiers, 258

M

- MAC (media access control), 61, 71, 73, 74, 256
- mailbox, 111
- MAP 2.1, 405
- MAP2.1 objects, 409
- MC_ADD, Data Link command, 274
- MC_REMOVE, Data Link command, 276
- Media Access Control, see MAC
- MIP, 60, 117, 119, 120, 128, 393
 - configuration values, 388
 - configuring, 67
 - errors, 119
 - jobs, 388
 - LAN Controller Interface (LCI), 392, 394
 - Multibus I, 393
 - Multibus II, 392, 394
 - PC Bus, 393
- MIX 386/560, 257
- MIX 560, 257
- modcdf command, 11, 22, 43, 44, 53, 56
- modself, UNIX command, 53
- Multibus I, 40, 60, 139, 143, 144
- Multibus II, 40, 41, 60, 139, 143, 144
- Multibus II subnet, 401
 - Data Link objects for, 401
- multicast address, 274, 399, 401
 - Name Server, 405
- multicast messages, 404

Multicast NSAP Address Table, 405
MYHOSTIDxx, 144
MYNAMExx, 145

N

name resolution, dynamic, 8
Name Server, 8, 18, 131
 example, 10
 multicast address, 405
 operation, 9
 using pointers, 108, 147
Name Server object table, 9, 18, 50, 58, 131,
 139, 144, 174
 adding objects, 134
 deleting objects, 141
 entries at initialization, 142
 listing local information, 141
negotiation option tests, 202
net agent, 295
net manager, 295
net start, MS-Net command, 50
net use, MS-Net command, 50, 53
netadm, UNIX command, 56, 58
network
 access, iRMX-NET, 15
 address, 18
 address, 8
 management, 64
 object, 9
 peer-to-peer, 2
 security, iRMX-NET, 12
 topology, 69
 user definition, 14
 user definition, 11
 user validation, 13, 14
Network
 Management Facility, 293
 Service Access Point, 72
Network File Access (NFA) protocols, 47
network files
 preconfigured values, 385
Network Layer, 63
 addressing, 71, 74
 configuration, 76
 events, 413
network objects, 397

networking
 DOSRMX, 17
 iRMX features, 1
NIC (Network Interface Card), 257
NMF, 293
 and remote booting, 362
 boot requests, 358
 boot response, 358
 objects, 413
nmfcfg.a86 file, 349
node, 2
 client, 2
 diskless, 333
 server, 2
noncontiguous buffers, 184
ns_get_host_id procedure, 144
NSAP (Network Service Access Point), 71,
 72, 74
 address, 175, 179
 selector, 72
NSAP address, 411
NSAP address length, 408
NSAP selector, 408
NSCOMMENGINE, 144
NSDONE, 145
NSDU (network service data unit), 177, 231
null TSAP selector, 180
Null2 addressing, 74

O

object table, iRMX-NET Name Server, 9, 18,
 131, 139, 144
 adding objects, 134
 deleting objects, 141
 entries at initialization, 142
objects
 Data Link, 398, 399, 400, 401
 ES-IS, 403
 iNA 960, 293, 397
 IP, 402
 MAP2.1, 409
 Name Server, 131
 network, 9, 131
 NMF, 413
 static routing, 402
 Transport Layer, 406

- offer command, 33
- OPEN, Transport command, 209
- OpenNET, 3, 4, 47, 49, 53, 132
 - Local Area Network, 1
 - networking, 1
 - user definition, 11
- options
 - iNA 960, 412
- OSI Reference Model, 59

P

- packets
 - discarded, 402, 413
 - received, 401
 - sent, 401
- packets received, 398
- padding
 - Data Link packet, 291
 - structures, 110
- partially specified TSAP address, 183
- passive open, 186
- password command, 11, 20, 43, 49, 55
- PC Bus, 53, 60, 139, 140, 143
- pccprsd.bck file, 338
- PCL2
 - PCL2 R3.0, 51
 - PCL2(A), 257
- peer-to-peer network, 2
- permit command, 34, 57
- persistence count, 228, 406, 411
- physical addresses, 108
- Physical Layer, 255
- pointers
 - Name Server, 108, 147
 - translating, 108
- point-to-point subnet, 70
- POST_RPD, Data Link command, 278
- promiscuous mode, 274
- publicdir command, 33

Q

- question mark (?) character, 52

R

- RAW_POST_RECEIVE, Data Link command, 282
- RAW_TRANSMIT, Data Link command, 286
- RawEDL, 399, 401
- rb_common data structure, 118
- rbootsrv.job, 333, 349
- Reachable NSAP Address Table, 375, 403, 404
- READ_AND_CLEAR_OBJECT, NMF command, 321
- READ_CLOCK, Data Link command, 288
- READ_MEMORY, NMF command, 319
- READ_OBJECT, NMF command, 321
- reason codes, 185
- receive buffers, 216
- RECEIVE_ANY, iNA command, 211
- RECEIVE_DATA, Transport command, 214
- RECEIVE_DATAGRAM, Transport command, 217
- RECEIVE_EXPEDITED_DATA, Transport command, 220
- Redirect PDUs, 404
- Refresh Redirect function, 403
- remini command, 341
- remote
 - booting, 333
 - node, 2
- remote access
 - across AUs, 14, 21
 - iRMX-NET, 15
 - within an AU, 13
- remote boot, 353
 - error messages, 355
 - failures, 355
 - troubleshooting, 355
- remote booting, 337
- Remote File Driver (RFD), 7
- remote third stage, 341
- remove command, 34
- rename command, 57
- request block, 107, 117
 - data structure, 118
 - MIP response codes, 119
- resource management, 107
- restarts, 398

retransmit timeout, 407, 411
RFD (Remote File Driver), 65
rmx.ini file, 15, 22, 340
RNETSRV, 145
routing
 network packets, 69
 tables, 75, 366
rq_create_mailbox call, 111
rq_lookup_object call, 112
rq_receive_message call, 111

S

SBC 186/530, 257
SBC 386/12(S), 334
SBC 386/2X, 334
SBC 386/3X, 334
SBC 486/12(S), 334
SBC 486/133SE, 257
SBC 486/166SE, 257
SBC 486DX33, 334
SBC 486DX66, 334
SBC 486SX25, 334
SBC 552A, 257, 334
SBC P5090, 257
SBX 586 Multimodule, 257
SDM
 booting from, 353
SEND_CONNECT_REQUEST, Transport
 command, 223
SEND_DATA, Transport command, 229
SEND_DATAGRAM, Transport command,
 233
SEND_EOM_DATA, Transport command,
 229
SEND_EXPEDITED_DATA, Transport
 command, 236
server, 2
 dedicated, 2
 iRMX-NET, 3
 names, iRMX-NET, 8, 18, 19
server_name object, 145
server-based security, 14
service information, inside back cover
SET_MEMORY, NMF command, 319
SET_OBJECT, NMF command, 321

setname command, 19, 27, 43, 50, 132, 134,
 144
SLSAP (source LSAP), 258
SNPA (subnet point of attachment), 73
Specific Router Table, 368, 402
specified TSAP address, 183
spokesman node, iRMX-NET, 9, 132
Static Intermediate System Table, 374, 403,
 404
static routing, 74, 365
 address, 75
 objects, 368, 402
static user, 21
STATUS, Transport command, 239
structures
 padding, 110
 using, 110
Subnet Table, 375, 403, 404
subnetwork, 11, 54, 55, 69, 70
 address, 73, 314
 iNA 960, 11
Super user, iRMX, 20
SUPPLY_BUFFER, NMF command, 329,
 362
supported architectures, 60
SV-OpenNET, 55, 58, 132
 SV4-OpenNET, 132
 SV4-OpenNET R2.0, 53
 SV-OpenNET R3.2.3, 53

T

TAKEBACK_BUFFER, NMF command,
 332, 362
TCP/IP, 1
third stage bootstrap loader, 341
TLCOMMENGINE, 144
TPDU (transport protocol data unit), 232
TPDU Size, 408, 412
transferring data, 186
translating pointers, 108
transmission errors, 399, 400
TRANSMIT, Data Link command, 289
transparent file access, 2
transport address, 18, 181
Transport Layer, 63, 175
 events, 414

- objects, 406
- transport protocol, 177
- TSAP (Transport Service Access Point), 71
 - address, 175
 - address buffer, 179
 - address format, 181
 - selector, 175, 179
- TSAP address, 414
- TSAP selector, 411
- TSAP selector length, 408
- TSDU (transport service data unit), 229
- typedef, 110

U

- UDF (User Definition File), 11, 12, 13, 14, 20, 22
- underrun errors, 399, 400
- UNIX, 4, 139
 - /net/data file, 58
 - and iRMX compatibility, 56
 - file access, 54
 - files, remote, 56
 - group ID, 11
 - home directory, 11
 - interoperability, 47
 - login shell, 11
 - pathnames, 56
 - server, 140
 - server, 53, 132
 - subnetwork, 11
 - symbols, 56
 - system, 53, 55, 58
 - tools, 56
- unspecified TSAP address, 183
- user
 - defining, OpenNET, 11
 - definition, 11, 14
 - groups, UNIX, 55, 57
 - static, 21
 - Super, 20
 - validation, 13, 14
 - verified, 11, 12, 22
 - World, 21
- User Administration module, 66
- User Definition File, see UDF

V

- validating
 - client, 13
 - user, 13, 14
- VC, see virtual circuit
- verified client, 11, 12, 22
- verified user, 11, 12, 22
- virtual circuit, 406
- virtual circuit (VC), 2, 177
 - maximum number of, 210
- virtual root directory, 51

W

- WITHDRAW_DATAGRAM_RECEIVE_BUFFER, Transport command, 249
- WITHDRAW_EXPEDITED_BUFFER, Transport command, 251
- WITHDRAW_RECEIVE_BUFFER, Transport command, 253
- work directory, 51
- World user, 21, 53, 57

X

- xlate command, 348

