NS-ARPA/1000

User/Programmer Reference Manual

Software Services and Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 91790-90020
E0495

Printed in U.S.A. April 1995
Seventh Edition

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard Company.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013.

Copyright © 1986-1993, 1995 by Hewlett-Packard Company

Printing History

The Printing History below identifies the edition of this manual and any updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this printing history page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the Manual Numbering File. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

First Edition .... Feb 1986 .... Rev. 2608
Update 1 .... Oct 1986 .... Rev. 4010
Second Edition .... Aug 1987 .... Rev. 5.0/5000
Update 1 .... Dec 1987 .... Rev. 5.0/5000
Update 2 .... Feb 1988 .... Rev. 5.05/5005
Update 3 .... Jan 1989 .... Rev. 5.1/5010
Third Edition .... Oct 1989 .... Rev. 5.16/5016
Update 1 .... May 1990 .... Rev. 5.2/5020
Fourth Edition .... Aug 1991 .... Rev. 5.24/5240
Fifth Edition .... Dec 1992 .... Rev. 6.0/6000
Sixth Edition .... Nov 1993 .... Rev. 6.1/6100
Seventh Edition .... Apr 1995 .... Rev. 6.2/6200

Preface

Hewlett-Packard Network Services for the HP 1000 (NS-ARPA/1000) provides the networking software that allows HP computer systems to communicate with each other.

Audience

The *NS-ARPA/1000 User/Programmer Reference Manual* is the primary reference source for users and programmers who will be writing or maintaining programs for NS-ARPA/1000 systems. The *NS-ARPA/1000 User/Programmer Reference Manual* should also be read by Network Managers before designing an NS network so that they will have a clear understanding of the full implications of various NS-ARPA/1000 functions and features.

Assumptions

Since the services described in this manual are both interactive and programmatic, this manual is intended for interactive users as well as programmers. As one of these interactive users or programmers, you should be familiar with the operating systems on the HP 1000, especially the RTE-A operating system. For those operations that deal with HP 3000 systems, a working knowledge of the Multiprogramming Executive (MPE) is also recommended. For those operations that deal with HP 9000 Series 800 systems, a working knowledge of the HP-UX operating system is also recommended. Network Managers, who have responsibility for generating and initializing nodes and configuring networks, should consult the *NS-ARPA/1000 Generation and Initialization Manual*, part number 91790-90030 and the *NS-ARPA/1000 Maintenance and Principles of Operation Manual*, part number 91790-90031.

Organization

Section 1	**Introduction**—presents an overview of NS-ARPA/1000, discussing the architecture of the network and introducing the User Services. This section also discusses the relation between NS-ARPA/1000 and its predecessor, DS/1000-IV. You are encouraged to read Section 1 before using the other sections for reference.
Section 2	**TELNET**—describes the commands, format, parameters and usage of the user interface program TELNET. TELNET provides a virtual terminal connection to any remote NS-ARPA/1000 node in your network.
Section 3	**FTP**—describes the commands, format, parameters, and usage of the File Transfer Protocol (FTP). FTP allows you to transfer files to and from remote nodes in your network. FTP also provides file management operations such as changing, listing, creating, and deleting remote directories.

Section 4	**Network File Transfer**—describes the commands, format, parameters and usage of the file copying program DSCOPY. DSCOPY allows you to copy files from one node to another in your network.
Section 5	**Network Interprocess Communication**—describes a set of programmatic calls that provide a data exchange interface between peer processes located at the same or different nodes in your network. Their format, parameters and usage are explained.
Section 6	**Remote Process Management**—describes a set of programmatic calls that provide remote scheduling, controlling, and terminating of programs located at the same or different HP 1000 nodes in your network. The format, parameters and usage are explained.
Appendix A	**FTP-NFT Comparison**—compares some of the features of FTP and NFT.
Appendix B	**Porting NetIPC Programs**—explains the differences and provides programming information to help you successfully port NetIPC programs between NS-ARPA/1000 and NS/9000 Series 800 systems.
Glossary	Defines NS-ARPA/1000 terms.
Bibliography	Other NS-ARPA/1000, NS/9000 Series 800, RTE-A, DS, DS/1000-IV, NS3000/V, NS3000/XL, X.25, and PC manuals are referred to by title within the text of this manual. To obtain the part numbers of these manuals, refer to this appendix.

Guide to NS-ARPA/1000 Manuals

The following are brief descriptions of the manuals included with the NS-ARPA/1000 product.

91790-90060 BSD IPC Reference Manual for NS-ARPA/1000 and ARPA/1000

Describes Berkeley Software Distribution Interprocess Communication (BSD IPC) on the HP 1000. BSD IPC on the HP 1000 offers a programmatic interface on the HP 1000 for multi-vendor connectivity to systems that offers BSD IPC 4.3.

91790-90020 NS-ARPA/1000 User/Programmer Reference Manual

Describes the user-level services provided by NS-ARPA/1000. The NS services are network file transfer (NFT), network interprocess communication (NetIPC), and remote program management (RPM). The ARPA services are TELNET and FTP. Because these are interactive and programmatic services, this manual is intended for interactive users as well as programmers. It should also be read by Network Managers before designing an NS-ARPA/1000 network so that they will have a clear understanding of the full implications of various NS-ARPA/1000 functions and features.

91790-90050 NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual

Describes the user-level services provided by the DS/1000-IV backward compatible services. These services are Remote File Access (RFA), DEXEC, REMAT, RMOTE, program-to-program communication (PTOP), utility subroutines, remote I/O mapping, remote system download to memory-based DS/1000-IV nodes only, and remote virtual control panel.

91790-90030 NS-ARPA/1000 Generation and Initialization Manual

Describes the tasks required to install, generate and initialize NS-ARPA/1000. This manual is intended for the Network Manager. Before reading this manual, the Network Manager should read the *NS-ARPA/1000 User/Programmer Reference Manual* to gain an understanding of the NS-ARPA/1000 user-level services. The Network Manager should also be familiar with the RTE-A operating system and system generation procedure.

91790-90031 NS-ARPA/1000 Maintenance and Principles of Operation Manual

Describes the NS-ARPA/1000 network maintenance utilities, troubleshooting techniques and the internal operation of NS-ARPA/1000. The Network Manager should use this manual in conjunction with the *NS-ARPA/1000 Generation and Initialization Manual*. This manual may also be used by advanced users to troubleshoot their applications.

91790-90040 NS-ARPA/1000 Quick Reference Guide

Lists and briefly describes the interactive and programmatic services described in the *NS-ARPA/1000 User/Programmer Reference Manual* and the *NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual*. The purpose of this guide is to provide a quick reference for users who are already familiar with the concepts and syntax presented in those two manuals.

The *NS-ARPA/1000 Quick Reference Guide* also contains abbreviated syntax for certain programs and utilities described in the *NS-ARPA/1000 Generation and Initialization Manual* and the *NS-ARPA/1000 Maintenance and Principles of Operation Manual*. For your convenience, the *NS-ARPA/1000 Quick Reference Guide* also contains a master index of NS-ARPA/1000 manuals. This is a combined index from the NS-ARPA/1000 manuals to help you find information that may be in more than one manual.

91790-90045 NS-ARPA/1000 Error Message and Recovery Manual

Lists and explains, in tabular form, all of the error codes and messages that can be generated by NS-ARPA/1000. This manual should be consulted by programmers and users who will be writing or maintaining programs for NS-ARPA/1000 systems. Because it contains error messages generated by the NS-ARPA/1000 initialization program NSINIT and other network management programs, it should be consulted by Network Managers.

91790-90054 File Server Reference Guide for NS-ARPA/1000 and ARPA/1000

Describes information on using and administering the HP 1000 file server, including runstring parameters, files needed for configuration, troubleshooting guidelines, and error messages.

5958-8523 NS Message Formats Reference Manual

Describes data communication messages and headers passed between computer systems communicating over Distributed System (DS) and Network Services (NS) links.

5958-8563 NS Cross-System NFT Reference Manual

Provides cross-system NFT information. It is a generic manual that is a secondary reference source for programmers and operators who will be using NFT on NS-ARPA/1000, NS3000/V,

NS3000/XL, NS/9000, NS for the DEC VAX* computer, and PC (PC NFT on HP OfficeShare Network). Information provided in this manual includes file name and login syntax at all of the systems on which NS NFT is implemented, a brief description of the file systems used by each of these computers, and end-to-end mapping information for each supported source/target configuration.

*DEC and VAX are U.S. registered trademarks of Digital Equipment Corporation.

Conventions Used in this Manual

NOTATION	DESCRIPTION
`nonitalics`	Words in syntax statements that are not in italics must be entered exactly as shown. Punctuation characters other than brackets, braces, and ellipses must also be entered exactly as shown. For example: `EXIT;`
italics	Words in syntax statements that are in italics denote a parameter that must be replaced by a user-supplied variable. For example: `CLOSE` *filename*
[ ]	An element inside brackets in a syntax statement is optional. Several elements stacked inside brackets means the user may select any one or none of these elements. For example: [A / B] User *may* select A or B or neither.
{ }	When several elements are stacked within braces in a syntax statement, the user must select one of those elements. For example: {A / B / C} User *must* select A or B or C.
...	A horizontal ellipsis in a syntax statement indicates that a previous element may be repeated. For example: [,*itemname*]...; In addition, vertical and horizontal ellipses may be used in examples to indicate that portions of the example have been omitted.
,	A shaded delimiter preceding a parameter in a syntax statement indicates that the delimiter *must* be supplied whenever (a) that parameter is included or (b) that parameter is omitted and any *other* parameter that follows is included. For example: *itema*[,*itemb*][,*itemc*] means that the following are allowed: *itema* / *itema*,*itemb* / *itema*,*itemb*,*itemc* / *itema*,,*itemc*

Δ	When necessary for clarity, the symbol Δ may be used in a syntax statement to indicate a required blank or an exact number of blanks. For example: `SET[(`*`modifier`*`)]Δ(`*`variable`*`);`
underlining	When necessary for clarity in an example, user input may be underlined. For example: `NEW NAME? ALPHA` (ALPHA underlined) Brackets, braces or ellipses appearing in syntax or format statements that must be entered as shown will be underlined. For example: `LET `*`var`*`[[`*`subscript`*`]] = `*`value`*` ` (inner brackets underlined) Output and input/output parameters are underlined. A notation in the description of each parameter distinguishes input/output from output parameters. For example: `CREATE (`*`parm1`*`,`*`parm2`*`,`*`flags`*`,`*`error`*`)` (parm2, flags, error underlined)
[ ]	The symbol [ ] may be used to indicate a key on the terminal's keyboard. For example, [RETURN] indicates the carriage return key.
[CONTROL] *char*	Control characters are indicated by [CONTROL] followed by the character. For example, [CONTROL]Y means the user presses the control key and the character Y simultaneously.

Table of Contents

Chapter 1
Introduction

Network Architecture 1-1
NS-ARPA/1000 User Services 1-3
    Where Described 1-3
Services and Link Availability 1-5
    The ARPA/Berkeley Services 1-5
    NS Common Services 1-6
    DS/1000-IV Compatible Services 1-7
Network Management Services and Features 1-7
NS-ARPA/1000 Programming Considerations 1-8
Node Names 1-8
IP Addresses 1-9
RTE-A Files and Directories 1-10
    CI File System 1-10
    FMGR Format 1-10

Chapter 2
TELNET

Overview 2-1
Application and Connectivity Considerations 2-2
    Connection Considerations 2-2
    Terminal Settings to DEC VAX Computers 2-3
    Chained TELNET Sessions 2-3
    Block Mode Considerations 2-5
        Troubleshooting Hints 2-5
Using TELNET 2-6
TELNET Operation 2-8
TELNET Commands 2-9
    ? 2-10
    CLOSE 2-11
    ESCAPE 2-12
    EXIT 2-14
    HELP 2-15
    INTERRUPT 2-16
    MODE 2-18
    OPEN 2-19
    QUIT 2-20
    RUN 2-21
    SEND 2-22
    STATUS 2-24

Chapter 3
FTP

Invoking FTP 3-2
FTP Operation 3-5
Terminating FTP 3-7
Temporarily Exiting FTP 3-7
Obtaining Help 3-8
RTE-A CI Files and Directories 3-8
    FMGR Cartridge Files 3-11
Transferring Files With FTP 3-11
    ASCII File Transfers 3-12
    Binary File Transfers 3-12
How FTP Treats Wild Card Characters 3-13
    $VISUAL Command Editing 3-14
FTP Commands 3-15
    ! 3-17
    ? 3-18
    3-19
    / 3-20
    APPEND 3-22
    ASCII 3-23
    BELL 3-24
    BINARY 3-25
    BYE 3-27
    CD 3-28
    CLOSE 3-29
    DEBUG 3-30
    DELETE 3-31
    DIR 3-32
    DL 3-34
    EXIT 3-36
    FORM 3-37
    GET 3-38
    GLOB 3-39
    HASH 3-41
    HELP 3-42
    LCD 3-43
    LL 3-44
    LS 3-45
    MDELETE 3-47
    MDIR 3-48
    MGET 3-49
    MKDIR 3-51
    MLS 3-52
    MODE 3-53
    MPUT 3-54
    NLIST 3-56
    OPEN 3-58
    PROMPT 3-59
    PUT 3-60
    PWD 3-61
    QUIT 3-62
    QUOTE 3-63
    RECV 3-64

REMOTEHELP 3-65
RENAME 3-66
RMDIR 3-67
RTEBIN 3-68
SEND 3-69
SITE 3-70
STATUS 3-71
STRUCT 3-72
SYSTEM 3-73
TR 3-74
TYPE 3-76
USER 3-77
VERBOSE 3-78

Chapter 4
Network File Transfer

Overview 4-1
Three-Node Model 4-2
File Copying Formats 4-3
Transparent Format 4-3
Interchange Format 4-3
Data Interpretation 4-4
Interactive Network File Transfer 4-5
Copy Descriptor 4-6
Using DSCOPY 4-10
HP 1000 File Names and Logons 4-11
HP 1000 File Masks 4-11
Interrupting the Copy Process 4-12
Examples 4-13
Transparent Format 4-13
Interchange Format 4-13
Optimizing Performance 4-13
NFT and DS/1000-IV Files 4-14
DSCOPY Commands 4-15
+CLEAR 4-16
+DEFAULT 4-17
+ECHO 4-19
+EX 4-20
+LL 4-21
+RU 4-22
+SHOW 4-23
+TRANSFER 4-24
+WD 4-25
? (HELP) 4-26
Programmatic Network File Transfer 4-27
DSCOPY 4-28
DSCOPYBUILD 4-29
Programmatic Examples 4-32

Chapter 5
Network Interprocess Communication

Overview 5-1
Sockets 5-2
Connections 5-2
  Naming, Socket Registry, and Path Reports 5-3
  Descriptors 5-3
  Establishing a Connection 5-4
    Creating a Call Socket 5-4
    Naming a Call Socket 5-5
    Looking Up a Call Socket Name 5-6
    Requesting a Connection 5-6
    Receiving a Connection Request 5-7
    Checking the Status of a Connection 5-8
    Summary of Calls Used in Connection Establishment 5-9
  Sending and Receiving Data Over a Connection 5-10
  Shutting Down a Connection 5-10
  Timing and Timeouts 5-11
  Additional NetIPC Calls 5-12
  Summary of NetIPC Calls 5-13
Synchronous and Asynchronous Socket Modes 5-14
  Read and Write Thresholds 5-14
Stream Mode 5-16
NetIPC Common Parameters 5-17
  Flags Parameter 5-17
    Pascal Programming Language 5-18
    FORTRAN 77 Programming Language 5-18
  Opt Parameter 5-19
  Data Parameter 5-21
    Type Coercion 5-23
  Result Parameter 5-23
  Socketname Parameter 5-23
  Nodename Parameter 5-24
Cross-System NetIPC 5-25
  Local NetIPC Calls 5-25
  Remote NetIPC Calls 5-27
  HP 1000 to HP 9000 NetIPC 5-28
  HP 1000 to HP 3000 NetIPC 5-29
  HP 1000 to PC NetIPC 5-31
Loading NetIPC Programs 5-32
Process Scheduling 5-32
  Remote HP 1000 NetIPC Process 5-32
  Remote HP 9000 NetIPC Process 5-33
  Remote HP 3000 NetIPC Process 5-33
  Remote PC NetIPC Process 5-33
NetIPC Syntax Conventions 5-34
  IPCCONNECT 5-35
    Cross-System Considerations 5-37
  IPCCONTROL 5-38
  IPCCREATE 5-40
    Cross-System Considerations 5-41
  IPCDEST 5-42
    Cross-System Considerations 5-44
  IPCGET 5-45

IPCGIVE 5-46
IPCLOOKUP 5-48
Race Conditions 5-49
IPCNAME 5-50
IPCNAMERASE 5-52
IPCRECV 5-53
Establishing a Connection 5-55
Receiving Data 5-55
Synchronous vs. Asynchronous I/O 5-56
Cross-System Considerations 5-59
IPCRECVCN 5-60
Synchronous vs. Asynchronous I/O 5-61
Cross-System Considerations 5-62
IPCSELECT 5-63
IPCSelect Call Bit Map Parameters 5-66
IPCSEND 5-67
Synchronous vs. Asynchronous I/O 5-68
Cross-System Considerations 5-68
IPCSHUTDOWN 5-70
Cross-System Considerations 5-71
Special NetIPC Calls 5-72
ADDOPT 5-73
ADROF 5-75
INITOPT 5-77
READOPT 5-79
Client-Server Program Examples 5-80
Server Program 5-80
Explanation of Server Using IPCSelect 5-81
Client Program 5-82
Cross-System NetIPC Program Examples 5-83
Pascal/1000 Client NetIPC Program 5-84
Pascal/1000 Server NetIPC Program 5-93
NetIPC Program Data Example 5-106
FORTRAN 77 Client NetIPC Program 5-107
FORTRAN 77 Server NetIPC Program 5-113
NS-ARPA/1000 NetIPC Program Examples 5-122
Pascal/1000 Example 1 5-122
Pascal/1000 Example 2 5-131
FORTRAN 77 Example 1 5-140
FORTRAN 77 Example 2 5-144

Chapter 6
Remote Process Management

Overview 6-1
Features of RPM 6-2
Summary of RPM Calls 6-3
RPM Programming Considerations 6-4
RPM Syntax Conventions 6-5
Flags Parameter 6-5
Opt Parameter 6-6
Result Parameter 6-6
Nodename Parameter 6-6
RPMCONTROL 6-7
RPMCREATE 6-10

Programs Scheduled by RPM Child Programs . . . 6-14
Terminating Dependent and Independent Child Programs . . . 6-15
Session-Sharing Among Child Programs . . . 6-16
RPMCREATE Options . . . 6-18
Adding Options Into the Opt Array . . . 6-19
RPMCreate Option 20000—Pass String . . . 6-21
RPMCreate Option 23000—Set Working Directory . . . 6-22
RPMCreate Option 23010—Restore Program . . . 6-23
RPMCreate Option 23020—Assign Partition . . . 6-24
RPMCreate Option 23030—Change Program Priority . . . 6-25
RPMCreate Option 23040—Modify Working Set Size . . . 6-26
RPMCreate Option 23050—Modify VMA Size . . . 6-27
RPMCreate Option 23060—Modify Code Partition Size . . . 6-28
RPMCreate Option 23070—Modify Data Partition Size . . . 6-29
RPMCreate Option 23080—Time Scheduling . . . 6-30
RPMCreate Option 23090—Program Scheduling (Immediate No Wait) . . . 6-32
RPMCreate Option 23100—Queue Program Scheduling . . . 6-35
RPMCreate Option 23110—Program Scheduling . . . 6-38
RPMGETSTRING . . . 6-39
RPMKILL . . . 6-41
RPM Program Examples . . . 6-42
Pascal/1000 RPM Parent Program . . . 6-42
Pascal/1000 RPM Child Program . . . 6-55
FORTRAN 77 RPM Parent Program . . . 6-57
FORTRAN 77 RPM Child Program . . . 6-63

Appendix A
NFT-FTP Comparison

System Type . . . A-1
Remote Logons . . . A-1
File Type . . . A-1
File Specification . . . A-2
File Size . . . A-2
Record Length . . . A-2

Appendix B
Porting NetIPC Programs

Overview . . . B-1
NS-ARPA/1000 and LAN/9000 . . . B-2
Path Report and Destination Descriptors . . . B-2
Socket Ownership . . . B-2
Socket Shut Down . . . B-2
Signals . . . B-3
TCP Checksum . . . B-3
Remote Process Scheduling . . . B-3
Remote NS-ARPA/1000 Process . . . B-4
Remote LAN/9000 Process . . . B-4
Case Sensitivity . . . B-4
NetIPC Calls . . . B-4
Unique NetIPC Calls . . . B-5
Common NetIPC Calls . . . B-5
Call Comparison . . . B-6

List of Illustrations

Figure 1-1 OSI Model 1-2
Figure 2-1 Logging On to a Remote Host With TELNET 2-1
Figure 2-2 Using TELNET to Reach a Distant Host 2-3
Figure 2-3 Using Different TELNET Escape Character 2-4
Figure 4-1 Three-Node Model 4-2
Figure 4-2 Interchange Format 4-3
Figure 4-3 NFT and DS/1000-IV 4-14
Figure 5-1 Telephone Analogy 5-2
Figure 5-2 IPCCreate (Processes A and B) 5-5
Figure 5-3 IPCName (Process B) 5-5
Figure 5-4 IPCLookUp (Process A) 5-6
Figure 5-5 IPCConnect (Process A) 5-7
Figure 5-6 IPCRecvCn (Process B) 5-8
Figure 5-7 IPCRecv (Process A) 5-9
Figure 5-8 Establishing a Connection with IPCLookUp 5-9
Figure 5-9 Establishing a Connection with IPCDest 5-10
Figure 5-10 Opt Parameter Structure 5-20
Figure 5-11 OPTARGUMENTS Structure 5-21
Figure 5-12 Vectored Data 5-22
Figure 5-13 Connection Established with IPCDest and IPCConnect 5-43
Figure 6-1 Parent-Child Relationship 6-1
Figure 6-2 Example of Child Programs Scheduling Another Program 6-15
Figure 6-3 Parent-Child Relationships When Session-Sharing 6-17

Tables

Table 1-1 Supported Connectivities for the ARPA/Berkeley Services 1-5
Table 1-2 Supported Connectivities for the NS Services 1-6
Table 1-3 Supported Connectivities for the DS/1000-IV Services 1-7
Table 2-1 TELNET Commands 2-9
Table 2-2 Illegal TELNET Escape Characters 2-12
Table 2-3 Illegal TELNET Interrupt Characters 2-16
Table 3-1 FTP Wild Card Characters 3-13
Table 3-2 FTP Commands 3-15
Table 3-3 FTP File Transfer Form, Mode, Structure, and Type 3-37
Table 3-4 FTP File Transfer Form, Mode, Structure, and Type 3-53
Table 3-5 FTP File Transfer Form, Mode, Structure, and Type 3-72
Table 3-6 FTP File Transfer Form, Mode, Structure, and Type 3-76
Table 4-1 RTE File Names and Logons 4-11
Table 5-1 Descriptor Summary 5-4
Table 5-2 NetIPC Calls 5-13
Table 5-3 Special NetIPC Calls 5-19
Table 5-4 NetIPC Calls Affecting The Local Process 5-26
Table 5-5 NetIPC Calls Affecting the Remote Process 5-27
Table 5-6 Cross-System NetIPC Calls (HP 1000—HP 9000) 5-28
Table 5-7 Cross-System NetIPC Calls (HP 1000—HP 3000) 5-29
Table 5-8 Cross-System NetIPC Calls (HP 1000—PC) 5-31
Table 5-9 Synchronous I/O Example 5-58
Table 5-10 Asynchronous I/O Example 5-58
Table 6-1 RPM Calls 6-3
Table 6-2 RPMCreate Options 6-13
Table 6-3 Where Sessions Are Created 6-17
Table B-1 Identical NetIPC Calls B-5
Table B-2 NS-ARPA/1000 and LAN/9000 Call Comparison B-6

1

Introduction

Hewlett-Packard Network Services for the HP 1000 (NS-ARPA/1000) is a data communications product that enables HP computer systems to exchange information and share resources in a *computer network*. A computer network is a collection of many types of equipment and software. The major components of a network are generally designated as *nodes* and *links*. A node is a computer system with its associated operating system and communication software. A node is connected to other nodes by communication links. Messages are sent to other computers over these communication links which may be physically hardwired or modem connections. The link includes the interface boards and cables.

The most significant feature of a network is *resource sharing*. Simply defined, resource sharing means that elements at each node are accessible from other nodes in the network. These elements may include disk files, printers, magnetic tapes, terminals, and other programs. One result of resource sharing is increased efficiency. For example, greater processing efficiency can be obtained when individual computers are dedicated to a specific type of processing. Any work of that specific type can be routed through the network for processing at the appropriate node.

Network Architecture

The architecture of NS-ARPA is based on the seven-layer *Open Systems Interconnection* (OSI) model developed by the International Standards Organization (ISO). Figure 1-1 shows the seven layers of the OSI model. This layered design offers a structured, modular approach to the different tasks that have to be performed in order to transmit and interpret data across a network. It is not necessary to know these architectural details in order to use the high-level services of NS-ARPA. However, some familiarity with the different tasks performed at the different levels may be helpful.

In the NS-ARPA/1000 network architecture, different transmission and communications tasks are assigned to logically distinct modules called *layers* or *levels*. The highest layer regulates user services while the lowest layer regulates the actual transmission of bits from one computer to another. At each layer one or more *protocols* are responsible for carrying out the appropriate tasks. A protocol is a set of rules governing a particular communication task. In a logical sense, the protocol entity at each level communicates with the corresponding protocol entity at the same level on another node. In reality, except for the physical transmission of data to another node, each protocol entity communicates with other protocols at the layer immediately above and below its own.

When a message is sent from one node to another in a network, it is first passed through the architectural levels at the source node. At one of the middle layers, the message is broken down into packets. At the lowest layer, the packets are actually sent across the physical communications link.

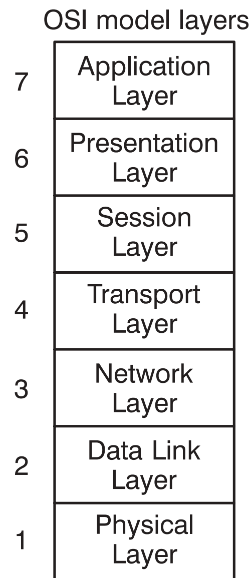


Figure 1-1. OSI Model

In NS-ARPA, the Application Layer, at the top of the hierarchy, corresponds to User Services such as file transfer, remote command execution and remote file access. The next two layers, Presentation and Session, define functions which contribute to these high-level services. There is no exact correspondence between NS-ARPA features and these layers, however.

The Transport Layer handles end-to-end communication between a source and a destination node, ensuring that a message from the source arrives at its destination in the proper form. The fragmentation of messages into packets may occur at this level. The Network Layer performs an addressing function, making sure that packets are acquired by the node to which they are addressed. The Data Link Layer governs the actual transmission of the packets over the communications link. (At this level the packets are technically known as frames.) The lowest layer, the Physical Layer, provides electrical and mechanical specifications for the transmission of bits across the link.

For more information on lower-level functions, refer to the *NS-ARPA/1000 Generation and Initialization Manual*, part number 91790-90030, and the *NS-ARPA/1000 Maintenance and Principles of Operation Manual*, part number 91790-90031.

NS-ARPA/1000 User Services

The user-level services provided by NS-ARPA, both interactive and programmatic, are known as User Services. The User Services available with the NS-ARPA/1000 product fit into one of the following categories:

- *ARPA/Berkeley Services*. These services are TELNET, File Transfer Protocol (FTP), and Berkeley Software Distribution Interprocess Communication (BSD IPC). The ARPA Services on the HP 1000 use standards defined by the Advanced Research Projects Agency (ARPA).

- *NS Common Services*. These services are Network File Transfer (NFT), Network Interprocess Communication (NetIPC), and Remote Process Management (RPM).

- *DS/1000-IV Compatible Services*. These services are also part of the DS/1000-IV product (the predecessor to NS-ARPA/1000). Almost all DS/1000-IV services are incorporated into NS-ARPA/1000. You can access the corresponding NS-ARPA/1000 services with the same commands and calls; the syntax remains the same. The DS/1000-IV Compatible Services provide:

  - *RTE-RTE services* that can be used for backward compatibility with DS/1000-IV nodes as well as for NS-ARPA/1000 to NS-ARPA/1000 communication.

  - *Transparent File Access (TRFAS), part of RTE-RTE services*. Also known as DS File Transparency, TRFAS allows you to access HP 1000 remote files using RTE file manipulation commands. However, because TRFAS is part of the RTE software, you should refer to your RTE documentation for more information on the syntax used to access remote files.

  - *RTE-MPE services* that can be used for backward compatibility with DS/3000 nodes as well as for NS-ARPA/1000 to NS3000/V communication.

Where Described

The NS-ARPA/1000 services (except TRFAS) are fully documented as follows:

NS-ARPA/1000 User/Programmer Reference Manual

- *Virtual Terminal (TELNET)*. Allows you to have a virtual terminal connection using the ARPA TELNET Protocol Standard, which is based on the Military Standard 1782 (MIL-STD-1782).

- *File Transfer Protocol (FTP)*. FTP is the file transfer program that uses the ARPA standard File Transfer Protocol (FTP). FTP also allows you to perform file management operations, such as changing, listing, creating, and deleting remote directories.

- *Network File Transfer (NFT)*. Allows you to copy files interactively or programmatically between NS-ARPA/1000 systems and other Hewlett-Packard computers in your network. This manual documents only NFT between NS-ARPA/1000 nodes. NFT between different types of systems is documented in the *NS Cross-System NFT Reference Manual*, part number 5958-8563. Refer to this manual for a list of supported computers.

- *Network Interprocess Communication (NetIPC)*. Allows autonomous processes running concurrently on different HP 1000 nodes to exchange information in a peer-to-peer manner. NetIPC can also be used between HP 1000 and other HP computer nodes.

- *Remote Process Management (RPM)*. Allows you to schedule, control, or terminate programs located at the same or different HP 1000 nodes in your network.

NS-ARPA/1000 BSD IPC Reference Manual

- *Berkeley Software Distribution Interprocess (BSD IPC)*. Allows client-server processes running on different nodes to exchange information in a peer-to-peer manner. Allows BSD IPC programs on the HP 1000 to communicate with programs on other systems that have BSD IPC 4.3.

NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual

- *Remote File Access (RFA)*. Enables you to perform I/O operations to files and peripherals located on HP 1000 and HP 3000 nodes.

- *Distributed Executive (DEXEC)*. Allows you to control I/O devices located at remote HP 1000 computers in your network. (DEXEC calls are the distributed equivalent to local RTE EXEC calls.)

- *REMAT*. Allows you to send RTE commands, or special REMAT commands, to any HP 1000 computer in your network.

- *RMOTE*. Creates an interactive session for you on a remote HP 3000 in your network, making your terminal appear to be directly connected to the other system.

- *Program-to-Program Communication (PTOP)*. Enables a “master” program on your local node to exchange information with and control the execution of a “slave” program on another HP 1000 or an HP 3000 node in your network.

- *Utility Subroutines*. Enable you to perform special tasks such as downloading absolute or memory-image program files to memory-based HP 1000 nodes and creating sessions at HP 3000 nodes.

Services and Link Availability

Whether or not a particular service is available is dependent on the type of link used to connect your local node to the system with which you want to communicate.

The following matrices show the services that are available to the types of systems that NS-ARPA/1000 can communicate with and the type of links that may connect them.

The first matrix shows the supported services and links for the ARPA/Berkeley Services: TELNET, FTP, and BSD IPC.

The ARPA/Berkeley Services

Table 1-1. Supported Connectivities for the ARPA/Berkeley Services

Link	Supported Connection from NS-ARPA/1000 to	Services
IEEE 802.3	NS-ARPA/1000	FTP, TELNET, BSD IPC
	ARPA/1000	FTP and TELNET
	ARPA Services/XL	FTP and TELNET
	ARPA/9000	FTP, TELNET, BSD IPC
	OfficeShare on the PC*	TELNET only
Ethernet	NS-ARPA/1000	FTP, TELNET, BSD IPC
	ARPA/1000	FTP and TELNET
	ARPA Services/XL	FTP and TELNET
	ARPA/9000	FTP, TELNET, BSD IPC
	ARPA Services/Vectra*	FTP, TELNET, BSD IPC
	ARPA on SUN workstations	FTP, TELNET, BSD IPC
X.25	NS-ARPA/1000	FTP, TELNET, BSD IPC
HDLC	NS-ARPA/1000	FTP, TELNET, BSD IPC

*When running ARPA between the HP 1000 and PC (Vectra), the PC must be the local host.

NS Common Services

The next matrix shows the supported services and links for the NS Common Services: Network File Transfer (NFT), Network Interprocess Communication (NetIPC), and Remote Process Management (RPM).

Table 1-2. Supported Connectivities for the NS Services

Link	Supported Connection from NS-ARPA/1000 to	Services
IEEE 802.3	NS-ARPA/1000	All NS Services
	NS3000/V	NFT and NetIPC
	NS3000/XL	NFT and NetIPC
	NS/9000	NFT
	LAN/9000	NetIPC
	OfficeShare on the PC*	NFT and NetIPC
Ethernet	NS-ARPA/1000	NFT and NetIPC
	NS/9000	NFT
	LAN/9000	NetIPC
X.25	NS-ARPA/1000	All NS Services
HDLC	NS-ARPA/1000	All NS Services

*To transfer files from a PC with NFT, you must initiate the transfer from the PC.

DS/1000-IV Compatible Services

The next matrix shows the supported services and links for the DS/1000-IV Compatible Services: Remote File Access (RFA), Distributed Executive (DEXEC), REMAT, REMOTE, and Program-to-Program Communication (PTOP).

The *DS Services* refer to the DS/1000-IV Compatible Services (RTE-RTE) if the link is between two HP 1000s, or the DS/1000-IV Compatible Services (RTE-MPE) if the link is between an HP 1000 and an HP 3000.

Table 1-3. Supported Connectivities for the DS/1000-IV Services

Link	Supported Connection from NS-ARPA/1000 to	Services
IEEE 802.3	NS-ARPA/1000	All DS Services, except Remote System Download and Remote VCP (DSVCP); RTE-A itself provides remote download and remote VCP over IEEE 802.3 LAN.
Ethernet	NS-ARPA/1000	All DS Services, except Remote System Download and Remote VCP
X.25	NS-ARPA/1000	All DS Services, except Remote System Download and Remote VCP
	DS/1000-IV	All DS Services, except Remote System Download and Remote VCP
	DS/3000	All DS Services
	NS3000/V	All DS Services
HDLC	NS-ARPA/1000	All DS Services
	DS/1000-IV	All DS Services
BISYNC	DS/3000	All DS Services
	NS3000/V	All DS Services

Network Management Services and Features

In addition to the User Services described above, NS-ARPA/1000 provides network management and link-level services. These services are documented in the *NS-ARPA/1000 Generation and Initialization Manual* and the *NS-ARPA/1000 Maintenance and Principles of Operation Manual*.

NS-ARPA/1000 Programming Considerations

Programs that use the NS-ARPA Common Services must be compiled in CDS.

Note DO NOT ALTER THE PRIORITIES OF ANY NS-ARPA/1000 SYSTEM PROGRAMS.

The priority of all the NS-ARPA/1000 programs must be higher than any user program which makes use of their capabilities. Be sure that your own programs do not have priorities higher than 30, that is, they are not in the range of 1 to 30. User programs with unnecessarily high priorities can delay necessary network processing and cause errors and/or poor performance.

For information about programming considerations for DS/1000-IV Compatible Services refer to the *NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual*, part number 91790-90050.

Node Names

Each computer system, or node, in an NS or an ARPA network has a *name*. (Node names are often referred to as *host names* in other ARPA systems.) NS-ARPA/1000 node names have the following syntax:

```
node[.domain[.organization]]
```

When all three parts of the node name are specified, it is called a *fully-qualified* node name. Each *node*, *domain*, and *organization* name is a maximum of 16 characters long. The maximum total length of a fully-qualified name is 50 characters. All alphanumeric characters are allowed, including the underscore (_) and dash (-) characters, but the first character of each field must be alphabetic. For example: `FOO.MKT.HP` would indicate node `FOO` in the `MKT` group (domain) of the `HP` Company (organization).

The *domain* and *organization* may be useful for grouping nodes and collections of nodes, but they currently have no special meaning regarding the structure of the network within the NS-ARPA/1000 product. In fact, most ARPA systems do not use the *domain* and *organization* fields in the node (host) names. These fields will simply be ignored if you supply them for your HP 1000 node name.

Currently, NS-ARPA/1000 does not support *ARPA domain names*.

When using the DS/1000-IV Compatible Services, you must use the remote node's Router/1000 node address assigned by the Network Manager.

IP Addresses

IP (Internet Protocol) addresses are used by nodes on the NS-ARPA network to uniquely identify each node. An IP address consists of two parts: a *network address*, which identifies the network; and a *node address*, which identifies a node within a network. A network address is concatenated with a node address to form the IP address, which then uniquely identifies a node within a network within a connected set of networks.

You may use the IP address, instead of the host node name, to identify a host when you invoke TELNET or FTP.

An IP address has the following format:

nnn.nnn.nnn.nnn

where *nnn* is a number from 000 to 255, inclusive. For example: 192.1.10.15. IP addresses are assigned to your node by your Network Manager during NS-ARPA/1000 initialization of your local host.

To find the node name and IP address of your host and other remote hosts network, use the A command in the NSINF information utility, such as in the example below.

Example

```
CI> NSINF

NSInf> A

                       LOCAL NAME AND ADDRESSES

 Local Name: BOBCAT.MKT.HP

IP address        LU  Status  Type   Station address    Multicast addresses

---------------  ---  ------  ----   -----------------  -------------------
192.006.001.002  134    UP     LAN   08-00-09-00-02-7C  09-00-09-00-00-01
                                                        09-00-09-00-00-02

                               GATEWAY TABLE

Destination net       Gateway       Down PID  Seg Size   Hops  Subnetwork Mask
---------------  -----------------  --------  --------   ----  ---------------
192.006.001.000     local net         LAN       1490        0  255.255.255.000
192.006.251.000  192.006.001.003    IEEE-802    1490      100  000.000.000.000

NSInf> E

CI>
```

RTE-A Files and Directories

RTE-A offers two types of file systems, CI hierarchical file system and FMGR cartridge file system. NS and ARPA Services are supported on both the CI and FMGR file systems.

For detailed information about RTE-A files and directories, refer to the *RTE-A User's Manual*, part number 92077-90002.

CI File System

A file name on the CI hierarchical file system can have up to 16 alphanumeric characters. In addition, a file can have a file extension, marked by a dot followed by an extension of up to four characters. The first character of the file name must be a letter. Examples of valid CI file names are `NOTES.DOC` and `TEST23`. Capitalization of file names is optional, because the HP 1000 always shifts the input to uppercase. On the HP 1000, the following three file names all refer to the same file: `TEST23`, `Test23`, `test23`.

Note HP 9000 hosts and other UNIX* hosts distinguish uppercase and lowercase in file names. Hence, `TEST23`, `Test23`, and `test23` refer to three different files on those systems.

For operations involving file names on the remote side, FTP on the HP 1000 transmits the file names as received by it. For operations involving file names on the HP 1000 side, FTP transmits the file names in lowercase to the remote side.

Discussion

FTP does not support sparse files (type 2 files with missing extents).

The maximum file path name, including the file name, is 63 characters. The file descriptor parameters—*`type`*, *`size`*, and *`recordlength`*—are optional. If specified, they must be given in the order shown above. To omit a parameter that is in front of another specified parameter, you must enter a colon as a place holder for the omitted parameter, such as: `FOOFILE:::4::200`. This file has type 4, and record length of 200 words; note that a colon serves as place holder for the omitted *`size`* parameter (between 4 and 200) in addition to the colon preceding the record length parameter.

FMGR Format

Some DS/1000-IV Compatible Services (RTE-RTE) and (RTE-MPE) cannot access non-FMGR files. Files used for RFA, REMAT, and all of the DS/1000-IV Compatible utility subroutines must be FMGR files. When one of these services requests a file name, it must be in FMGR format. RTE refers to this type of file descriptor as a *`namr`*.

---

*UNIX is a registered trademark of UNIX Systems Laboratories Inc. in the U.S.A. and other countries.

A *namr* consists of parameters that specify a file's name, security code (if one exists), the cartridge on which it resides, its type, size, and record length. Not all of the parameters are required. Unless otherwise noted by the particular service, each *namr* parameter (with the exception of *filename*) has a default value of zero.

The following is a description of the *namr* syntax:

filename[:*security code*][:*crn*][:*type*][:*size*][:*record length*]

Parameters

filename
: One to six character ASCII file name. Only printable characters can be used (! through ~). The colon (:) and comma (,) are not allowed. The first character of the file name must not be a blank or a number. Blanks may not be imbedded within the file name. Characters are not case-sensitive (all lowercase characters are upshifted). Each file name must be unique to the disk.

security code
: File security code. Can be a positive or negative integer or two ASCII characters represented as a positive integer. Range is from -32767 through 32767. The security code may be:

zero	File is unprotected. (This is the default value.)
positive integer	File is protected against alteration (write protection) or purging. May be read with any security code or none; may be purged only with correct or negative (two's complement) of correct code.
negative integer	File is fully protected (read and write protected). May only be referenced with correct negative code.

crn
: Cartridge identifier. May be a positive or negative integer or two ASCII characters represented as a positive integer. Range is from -32767 through 32767. It may be:

zero	On RTE-A systems, zero indicates that the file resides on a FMGR cartridge. (This is also true for RTE-6/VM with the hierarchical file system.) On other operating systems, zero indicates that the first available cartridge that satisfies the request will be used. (Zero is the default value on these systems.)
positive integer	Cartridge reference number by which the cartridge is identified.
negative integer	Logical unit number associated with the cartridge.

type

Each file descriptor has a file type parameter that indicates how the information in the file is organized. The file type is a number and is not to be confused with a a file type extension. There are standard RTE-A file types defined with the following characteristics:

-1 Symbolic link files. A symbolic link is a file that indirectly refers to another file. The symbolic link file itself contains a file descriptor that points to the new file.

0 An I/O device. Type 0 is used in accessing devices with file calls. There is no disk file or directory entry for type 0 files, and they do not have the other properties listed in this section.

1 Random access files. These do not have any structure information in them. These files contain fixed record lengths (128 words). They can be read and written very quickly.

2 Fixed-length record, random access files. The record length is defined when the files are created. They are usually user-created, large data files.

3 Type 3 and higher files are variable-length and higher record, sequential files suitable for use as text files. There is no difference in the handling of file types 3, 4, and 7. Type 3 is for general purpose files and can be used for text. This is the default file type when files are created with the CR command. Type 4 is recommended for text files. By convention, type 5 is used for Compiler or Assembler relocatable output files, type 6 is for program files that are memory-images of executable programs, and type 7 is for Compiler or Assembler absolute binary output files. Type 6 files are treated the same as type 1 files.

8 Type 8 and higher files are user-defined, with the following exceptions.

12 Byte stream files. These do not have any structure information in them. The directory entry for each type 12 file contains a pointer to the last byte in the file. The fields for number of records and record length are not defined for type 12 files.

6004 CALLS catalog files.

size

Decimal number of blocks in range of 1 through 32767. Indicates the space allocated to the file. Minimum number of blocks is 1.

record length

Decimal number of words in range 1 through 32767. Applies only to type 2 files. Type 1 files use 128-word records and other types use variable-length records.

2

TELNET

Overview

TELNET is used to communicate with another host using the TELNET protocol. The TELNET protocol is a standard ARPA service that provides a virtual terminal connection to a remote node on the network. TELNET enables you to logon to remote nodes on the network as if you were on a terminal directly attached to the remote system.

TELNET makes the fact that the session is remote almost entirely transparent. You enter commands and receive responses at your local terminal just as if your sessions were local. In reality, input and output to your local terminal pass through a “virtual” (as opposed to real, physical) terminal configured on the remote system. Your remote commands are transmitted over network connections, sent to the virtual terminal, and subsequently executed on the remote system.

TELNET is supported only on terminals directly connected or connected by modem to the MUX for A400 computers or to the HP 12040D MUX for other A-Series computers.

Figure 2-1 shows an illustration of TELNET virtual terminal service.

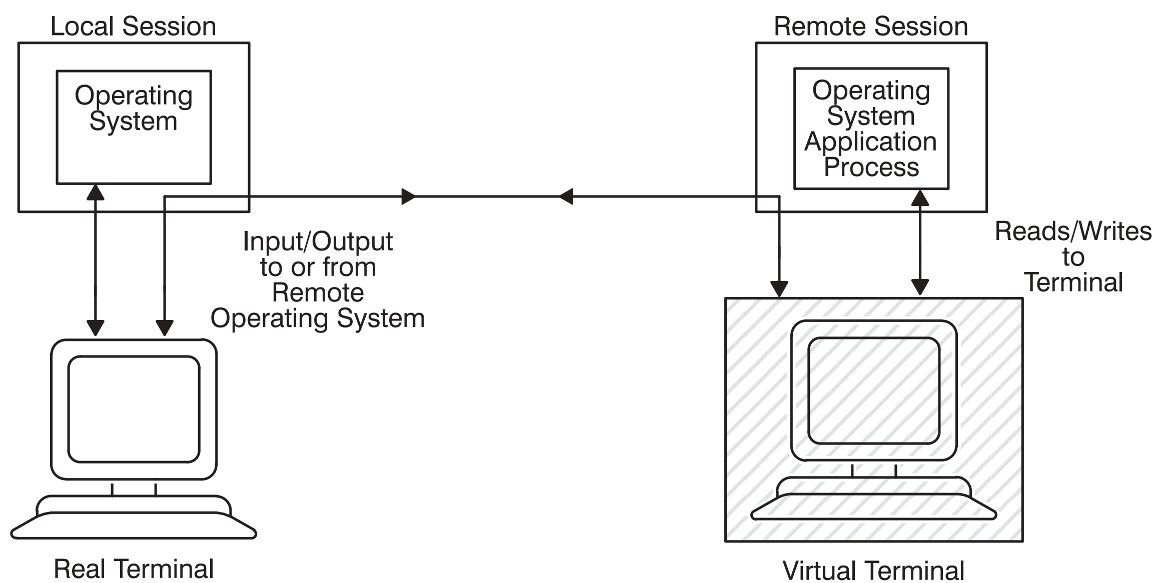


Figure 2-1. Logging On to a Remote Host With TELNET

Application and Connectivity Considerations

There are several considerations to keep in mind when using TELNET:

- In certain cases, it may take longer to send terminal data from the physical terminal over the network to the remote node than the time allowed by an application program. If the program fails to receive the needed data, it will result in error. User written applications that are expected to run over TELNET should be written with this in mind.
- TELNET does not support HP 12040D MUX firmware with revision earlier than 5.02.
- Make sure your application runs locally without errors before executing it over a TELNET connection.
- Different terminals and computers may have different configuration requirements. Refer to the following subsections for detailed information.
- Block mode applications have a limited number of supported configurations when using TELNET. Refer to the subsection "Block Mode Considerations" which follows shortly.

Connection Considerations

Table 1-1, "Supported Connectivities for the ARPA Services," in Section 1 of this manual lists all the supported cross-system TELNET connectivities between an HP 1000 and other systems.

There are several connectivity considerations:

- Only one connection for each TELNET user can be open at a time. *HP does not support multiple connections per each TELNET user*.
- A chained session is one where you have TELNET open to one computer and then you use TELNET from that computer to access another (a third) computer. Select a unique escape character for each host you wish to communicate with in a chained session. Refer to the subsection, "Chained TELNET Sessions" later in this section.
- For connections to any computer, always set the HP 1000 host terminal RECVPACE configuration (receive direction) to XON/XOFF.
- For block mode applications, terminals directly connected to an HP 1000 require XON/XOFF in both the transmit and receive directions. If the terminal is not set to XON/XOFF in both directions, a slow TELNET session may be overrun by the terminal and data will be lost or the application may hang.
- For block mode applications, terminals attached to the TS-8 with LSM 2.1 (or greater) software require XON/XOFF in only the *receive* direction. If XON/XOFF is set for the transmit direction, block mode applications may hang.
- You cannot initiate a remote session to a PC. Remote sessions between an HP 1000 and PC can only be initiated from the PC.
- When connecting to a remote DEC VAX computer, change your terminal configuration settings as described in the next subsection, "Terminal Settings to DEC VAX Computers."

Terminal Settings to DEC VAX Computers

If you are using TELNET on the HP 1000 to connect to a remote DEC VAX host, you should set the communication protocol of the HP 1000 host terminal to `XON/XOFF`. The steps are as follows:

1. On the HP 1000, enter `WH` to display information about your terminal. Locate your session number.
2. Execute this command to set your terminal to `XON/XOFF` protocol:
   `CI> cn,$session,34b,1b`
3. Use TELNET to log on to the remote DEC VAX host.
4. Once you are logged on to the DEC VAX host, execute this command:
   `$ set terminal/vt100`. You can put this command in your LOGIN.COM file for automatic execution whenever you log onto the DEC VAX system.
5. Set your terminal to ANSI term type. See your terminal documentation for instructions.
6. When you have completed your TELNET session on the DEC VAX host and returned to the local HP 1000 host, reset your terminal to HP term type. See your terminal documentation for instructions.
7. Restore the local host to `ENQ/ACK` protocol by executing: `CI> cn,$session,34b,2b`

Chained TELNET Sessions

Chaining makes it possible to hop across the network to different hosts. See Figure 2-2.

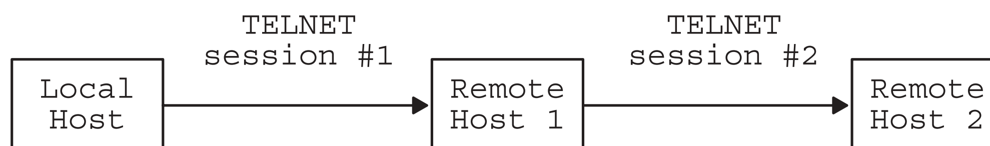


Figure 2-2. Using TELNET to Reach a Distant Host

If you chain several TELNET sessions, you may want to select a unique escape character for each host in the chain, using the `ESCAPE` command. Then you can escape to the node of your choice by issuing the appropriate escape character. (See Figure 2-3.)

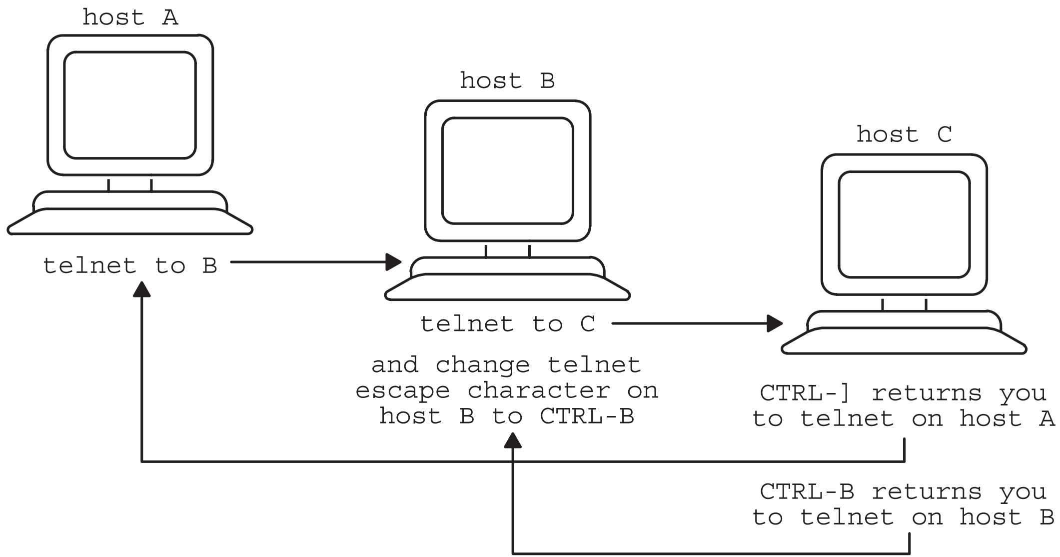


Figure 2-3. Using Different TELNET Escape Character

If all nodes use the same escape character, you can only escape to your *local* node; you cannot escape to an intermediate node. See the `ESCAPE` command description later in this section.

Note Do not define the escape character to be the same as the interrupt character (default interrupt is `CONTROL` `Y`). See the `INTERRUPT` command for details.

Also be aware that different systems have different illegal escape characters; be sure to check the appropriate system's TELNET documentation for its list of illegal characters.

If you chain TELNET sessions, the `QUIT` or `EXIT` command will terminate all sessions, close all connections, and return you to the local host. If, however, you log off the remote host, only the most recent TELNET session is closed. Any other chained sessions are still active.

If TELNET terminates abnormally or is aborted, any remote session chained from your session is automatically terminated. For example, if your session was the fourth out of five chained sessions, and it aborted, only the fourth and fifth sessions would abort.

Block mode applications over chained TELNET sessions are not supported.

Block Mode Considerations

The TELNET standard specifies a character mode protocol. In character mode, data is transmitted a character at a time as it is entered through the keyboard. Control codes, such as carriage return and linefeed, are also transmitted. Character mode is the normal operation of a terminal.

With block mode, data is not transmitted one character at a time. Instead, an entire block of data is typed in locally on the terminal. When the enter key is pressed, the data is transmitted from the terminal to the computer.

Block mode for the HP 1000 is technically defined in the *RTE-A Driver Reference Manual*, part number 92077-90011.

The following products support block mode applications to the HP 1000:

- ARPA/Vectra revision 2.0 (or later) with the Advlink B.02.00 Emulator. On ARPA/Vectra, the RS (record separator) is the default escape character for TELNET. The RS character is also a special character in block mode. The TELNET escape sequence on the PC must be changed to another character.
- ARPA/9000 revision 7.0 (or later) with direct connect terminals only. HPTERM is not supported.
- TS-8 with LSM 2.1 (or later) software.
- Data communications and Terminal Controller (DTC).

Block mode applications over TELNET are not supported on the PC Officeshare products.

Any TELNET user can communicate with a block mode application on the HP 1000 as long as the local terminal or terminal emulator can handle block mode I/O.

Block mode applications over chained TELNET sessions are not supported.

For block mode applications, terminals directly connected to an HP 1000 require XON/XOFF in both the transmit and receive directions. If the terminal is not set to XON/XOFF in both directions, a slow TELNET session may be overrun by the terminal and data will be lost or the application may hang.

For block mode applications, terminals attached to the TS-8 with LSM 2.1 (or greater) software require XON/XOFF in only the *receive* direction. If XON/XOFF is set for the transmit direction, block mode applications may hang.

Troubleshooting Hints

Here are some suggestions for isolating problems:

- Verify whether the terminal and/or terminal emulator supports block mode. Test your application on a directly-connected terminal on the local system. Then test again on the terminal emulator. Finally, test to a remote system over TELNET.

- Check that the block mode application can run between two HP 1000 systems before executing from a PC or HP 9000. You can also check your application using TELNET on the same HP 1000 system.
- Determine which call fails in the block mode application. Refer to the *RTE-A Driver Reference Manual*.

Using TELNET

TELNET is scheduled at the RTE Command Interpreter level (`CI>`). TELNET can be invoked with or without the *host* parameter.

Syntax

```
TELNET [,host]
```

Parameters

host — Specifies the remote node to which you want to log on. You may use the host's node name or IP address for the *host* parameter.

The syntax for the host's node name is shown here, and is further described under "Node Names" in Section 1 of this manual.

```
node[.domain[.organization]]
```

The syntax for the host's IP address is shown here, and is further described under "IP Addresses" in Section 1 of this manual.

```
nnn.nnn.nnn.nnn
```

If *host* is not specified, TELNET displays a prompt (`TELNET>`) and waits for you to enter a TELNET command. The commands are described under "TELNET Commands" later in this section.

Discussion

When TELNET is invoked with the *host* parameter, TELNET automatically performs an `OPEN` command to establish a connection with the specified remote host. If the remote connection is successful, the logon prompt for the remote system is displayed, allowing you to log on. The `OPEN` command is explained later in this section.

If TELNET is invoked without the `host` parameter, TELNET enters *command mode*, indicated by the TELNET prompt (`TELNET>`). In command mode, TELNET accepts and executes the TELNET commands listed in Table 2-1, "TELNET Commands." If you invoke TELNET without the `host` parameter, you must enter the TELNET `OPEN` command to establish a remote connection.

Only one connection for each user can be open at a time. *HP does not support more than one open connection per user*.

Once a connection has been opened, TELNET enters *input mode*. In this mode, text that is typed at your terminal is sent to the remote host. When you logout from the remote session, you will return to the local host.

To temporarily return to the local node and to issue TELNET commands, type the TELNET escape character. The default TELNET escape character is [CONTROL] `]`. You may change the default escape character with the `ESCAPE` command, which is explained later in this section. To return to the remote session before breaking the remote connection, type a carriage return at the TELNET prompt.

To terminate TELNET, enter the `EXIT` or `QUIT` command at the TELNET prompt. The `EXIT` and `QUIT` commands are explained later in this section.

TELNET Operation

The following example shows how to use TELNET.

Example	Explanation
CI>TELNET TELNET User Program Rev. 6100 ''Enter ? for help.	TELNET invoked at the local Command Interpreter prompt.
TELNET>	TELNET prompt is displayed.
TELNET>open Karl Opening to ... Karl *** Welcome to TELNET *** Please hit a <cr> to get the logon prompt.	OPEN establishes a connection to a remote system called Karl. Once the connection has been established, the remote node displays a logon prompt.
RETURN . RTE-A logon:	
RTE-A logon: amy	Log on to the remote system.
Password? ___	Password is not echoed to the terminal.
CI>RU,EDIT	Run EDIT at the remote node.
: :	
CI>	Finish editing.
CI>EX Finished Connection closed.	Terminate your remote RTE session with the RTE EX command. Note: this is *not* the TELNET EXIT command.
TELNET>EXIT	EXIT terminates TELNET and returns to the local operating system.

Note If TELNET is aborted or terminates abnormally at the local node after you log on to the remote node, you will be aborted from your remote session.

If an error occurs while running TELNET, an error message will be displayed on your terminal. Refer to "TELNET User Error Messages" in Section 4 of the *NS-ARPA/1000 Error Message and Recovery Manual* for a description of the TELNET error messages.

TELNET Commands

TELNET has 12 commands as listed in Table 2-1. These commands are explained in subsequent pages of this section.

Table 2-1. TELNET Commands

Command	Description
?	Displays TELNET commands and help information. Same as HELP.
CL[OSE]	Closes the remote connection and logs off the remote session.
ES[CAPE]	Defines the TELNET escape character.
EX[IT]	Closes the remote connection, logs off the remote session, and terminates TELNET. Same as QUIT.
HE[LP]	Displays TELNET commands and help information. Same as ?.
IN[TERRUPT]	Changes the TELNET remote interrupt character.
MO[DE]	Changes the data transmission to either line or character mode.
OP[EN]	Establishes a connection to a remote host.
QU[IT]	Closes the remote connection, logs off the remote session, and terminates TELNET. Same as EXIT.
RU[N]	Runs a local program.
SE[ND]	Sends special characters or commands to the remote node.
ST[ATUS]	Displays status of the TELNET remote connection.

?

Displays TELNET commands and help information. Same as the HELP command.

Syntax

```
? [command]
```

Parameters

command Any TELNET command listed in Table 2-1, "TELNET Commands."

If no command is specified, TELNET lists the TELNET commands, with a one-line description for each command. (See the example below.)

When a command is specified, TELNET displays a brief description of the command. (See the example for the HELP command later in this section.)

Example

The following example shows help information displayed by ?. The ^ character indicates the CONTROL key.

```
               ------------TELNET USER PROGRAM------------

TELNET>?

OP[EN] hostname                    - establish a connection to hostname
CL[OSE]                            - close connection to remote host
EX[IT]                             - terminate TELNET
QU[IT]                             - terminate TELNET
HE[LP]                             - print this help message
HE[LP] command                     - print description of command
?                                  - print this help message
? command                          - print description of command
??                                 - equivalent to HELP ?
ST[ATUS]                           - print state of TELNET connection
MO[DE] L[INE]                      - transmit data after line of input
MO[DE] C[HARACTER]                 - transmit data after each character
ES[CAPE] escape char               - define escape character
ES[CAPE] ^letter                   - define escape character
IN[TERRUPT] interrupt character    - define interrupt character
IN[TERRUPT] ^letter                - define interrupt character
SE[ND] IN[TERRUPT]                 - send interrupt character as data
SE[ND] E[SCAPE]                    - send escape character as data
SE[ND] B[REAK]                     - send TELNET "Break"
SE[ND] A[YT]                       - send TELNET "Are You There"
SE[ND] IP                          - send TELNET "Interrupt Process"
RU[N] program                      - run a program
```

CLOSE

Closes the remote connection and logs off the remote session.

Syntax

```
CL[OSE]
```

Discussion

The `CLOSE` command closes the connection to the remote host and terminates any remote session. You remain at the local node and in TELNET. The `CLOSE` command is similar to the TELNET `EXIT` and `QUIT` commands. However, unlike `EXIT` and `QUIT`, `CLOSE` does not return you to the local operating system unless you specified a *host* in the TELNET runstring.

The `CLOSE` command allows you to connect to other remote hosts during the same TELNET session.

Example

The following example shows the use of `CLOSE` to break a remote connection.

```
TELNET>CLOSE
Connection closed.
```

Breaks the remote connection.

```
TELNET>STATUS
No connection open.
Escape character = ^]
Interrupt character = ^Y
Transmission mode = Line
```

`STATUS` shows that the remote connection is closed (no connection open). `STATUS` is covered later in this section.

ESCAPE

Defines the TELNET escape character.

Syntax

```
ES[CAPE] escape_char
```

Parameters

escape_char Any seven-bit ASCII character except those listed below in Table 2-2. The default is CONTROL].

The escape character, when typed at the remote session, allows you to temporarily return to the local node. To go back to the remote node, enter a single carriage return at the TELNET prompt.

Table 2-2. Illegal TELNET Escape Characters

Decimal Value	ASCII Character	Terminal Keys
0	NUL (null)	CONTROL @
4	EOT (end of transmission)	CONTROL D
8	BS (backspace)	CONTROL H
10	LF (linefeed)	CONTROL J
13	CR (carriage return)	CONTROL M
17	DC1 (XON)	CONTROL Q
18	DC2	CONTROL R
19	DC3 (XOFF)	CONTROL S
24	CAN (cancel)	CONTROL X
25	EM (end of medium)	CONTROL Y
30	RS (record separator)	CONTROL ^
31	US (unit separator)	CONTROL _
127	DEL (rubout)	DEL

The characters in Table 2-2 cannot be defined as the remote escape character, because they have special terminal functions. Using one of these characters may cause communication problems with the remote node.

Do not define *escape_char* to be the same as the TELNET interrupt character. The TELNET interrupt character default is CONTROL Y and can be redefined by the INTERRUPT command.

Discussion

The TELNET ESCAPE command redefines the TELNET escape character. The new escape character remains in effect for the duration of the current connection or until another TELNET ESCAPE command is issued. If no *escape_char* is specified, the default (CONTROL]) will be used. Defining a different escape character is especially useful for chained sessions, as described in "Chained TELNET Sessions" ealier in this section.

When entered at the remote node, the TELNET escape character allows you to "escape" to your local node. The TELNET prompt is then displayed, and you can enter other TELNET commands. To return to the remote system, enter a single carriage return at the TELNET prompt.

Escape does *not* return you to the local operating system. To return to the local operating system, terminate TELNET with EXIT or QUIT or use the RUN program command to get CI. The escape character should only be used to temporarily return to the local node. To terminate a remote session and return to TELNET, use the appropriate logoff command for the remote system; e.g., EX command for RTE.

Example

In the following example, the ESCAPE command redefines the escape character as CONTROL A. When used at the remote node, CONTROL A returns the user to the local node.

TELNET>ESCAPE CONTROL A Escape character = ^A	ESCAPE defines CONTROL A as the escape character.
TELNET>open Karl	Connect to remote system called Karl.
: :	
CI>CONTROL A TELNET>	CONTROL A issued at the remote session returns user to the local node. TELNET prompt displayed at the local node.
TELNET>RETURN	Single carriage return at the TELNET prompt returns user to the remote session.
CI>	CI prompt at the remote session.

EXIT

Closes the remote connection, logs off the remote session, and terminates TELNET. Same as `QUIT`.

Syntax

```
EX[IT]
```

The TELNET `EXIT` command closes the connection to the remote host, logs off the remote session, terminates TELNET, and returns you to the operating system at the local node. The `EXIT` command is identical to the `QUIT` command.

The `CLOSE` command also closes the remote connection, but you remain in TELNET unless you specified *`host`* on the TELNET runstring, in which case you return to the local operating system.

HELP

Displays TELNET commands and help information. Same as `?`, described earlier.

Syntax

```
HE[LP] [command]
```

Parameters

command — Any TELNET command as listed in Table 2-1, "TELNET Commands."

When a command is specified, `HELP` displays a brief description of the command. (See example below.)

If no command is specified, `HELP` lists the TELNET commands, their syntax, and a one-line description for each command. (See the example for the `?` command, earlier in this section.)

Example

```
TELNET>HELP STATUS

ST[ATUS]                        - print state of TELNET connection

      This command is used to display the current state of the TELNET
      connection. Display items include:

      o   the current connection state (OPEN or CLOSED)
      o   the remote host name if a connection is open
      o   the currently defined TELNET escape character
      o   the currently defined TELNET interrupt character
      o   the current transmission mode (LINE or CHARACTER)
      o   a list of any TELNET network options which are currently
          enabled

TELNET>
```

INTERRUPT

Changes the TELNET remote interrupt character.

Syntax

```
IN[TERRUPT]  intr_char
```

Parameters

intr_char Any seven-bit ASCII character except those listed below in Table 2-3. The default is CONTROL Y.

The interrupt character is used to send a "BREAK" indication to the remote system without hitting the BREAK key on the terminal.

Table 2-3. Illegal TELNET Interrupt Characters

Decimal Value	ASCII Character	Terminal Keys
0	NUL (null)	CONTROL @
4	EOT (end of transmission)	CONTROL D
8	BS (backspace)	CONTROL H
10	LF (linefeed)	CONTROL J
13	CR (carriage return)	CONTROL M
17	DC1 (XON)	CONTROL Q
18	DC2	CONTROL R
19	DC3 (XOFF)	CONTROL S
24	CAN (cancel)	CONTROL X
27	ESC (escape)	CONTROL]
30	RS (record separator)	CONTROL ^
31	US (unit separator)	CONTROL _
127	DEL (rubout)	DEL

The characters in Table 2-3 cannot be defined as remote interrupt characters, because they have special terminal functions. Using one of these characters may cause communication problems with the remote node.

Do not define *intr_char* to be the same as the TELNET escape character. The TELNET escape character default is CONTROL] and can be redefined by the ESCAPE command.

Discussion

The TELNET `INTERRUPT` command redefines the TELNET interrupt character. The interrupt character is entered at the remote session to interrupt the current process on the remote node.

If no interrupt character is specified, the default (CONTROL Y) will be used.

Using the interrupt character is equivalent to hitting the BREAK key at the remote host.

To interrupt the remote session from the TELNET prompt, use `SEND IP`, which is explained later in this section.

Example

`CI>wh,al`	Run `wh,al` on a remote RTE node.
`:`	
`:`	
CONTROL Y	Hit the interrupt character (CONTROL Y) to interrupt.
`CM>`	The RTE `CM>` prompt is displayed.
`CM>`CONTROL	Hit carriage return to resume current process.

MODE

Changes the data transmission to either by line or by character mode.

Syntax

```
         { L[INE]       }
MO[DE]   { C[HARACTER]  }
```

Parameters

L[INE]	Sends data a line at a time from the local terminal to the remote node. In RTE-A, each line of data ends with a carriage return.
C[HARACTER]	Sends data a character at a time from the local terminal to the remote node. Each character is sent without waiting for an end-of-line character.

Discussion

As you enter characters at your local terminal, the characters are sent to the remote node. The characters are finally received by the application or Command Interpreter that you are running on that remote node. Characters can be transmitted either by line or by one character at a time. In line mode, you must enter a carriage return at the end of each line before the characters will be sent. In character mode, data is transmitted character by character. If no transmission mode is specified, the default line transmission mode will be negotiated.

Before a remote connection is established, the TELNET default data transmission mode is line mode. Once a remote connection is established, the data transmission mode automatically changes to character mode. When you change modes to line mode, the change takes a few seconds. Use the STATUS command to check when the connection has actually changed to line mode. The status will show “character mode,” until TELNET actually finishes negotiations and switches to “line mode”. RTE-A systems usually transmit data a line at a time.

CHARACTER data transmission may be used for applications that echo the characters themselves.

Note Character mode is very inefficient; performance is considerably better in line mode.

OPEN

Establishes a connection to a remote host.

Syntax

```
OP[EN] host
```

Parameters

host Specifies the remote node to which you want to log on. You may use the host's node name or IP address for the *host* parameter.

The syntax of the node name is shown here, and is further described under "Node Names" in Section 1 of this manual.

node[.*domain*[.*organization*]]

The syntax for the host's IP address is shown here, and is further described under "IP Addresses" in Section 1 of this manual.

nnn.nnn.nnn.nnn

If *host* is not specified, TELNET connects you to your local node.

Discussion

When TELNET is scheduled with the *host* parameter, TELNET automatically performs an OPEN command to establish a connection with the specified remote host. If the remote connection is successful, the logon prompt for the remote system is displayed, allowing you to log on. When you exit the remote session with the logout command for that system (in RTE, the EX command), TELNET automatically closes the remote connection, terminates TELNET, and returns to the local operating system.

If TELNET is *not* scheduled with the *host* parameter, TELNET enters command mode and displays the TELNET prompt (TELNET>). In command mode, TELNET accepts and executes the commands listed in Table 2-1, "TELNET Commands." In this case, you must enter the TELNET OPEN command to establish the remote connection. When you close the remote session, TELNET closes the remote connection, but you remain in TELNET, until a QUIT or EXIT command is issued. (See the example under "TELNET Operation," earlier in this section.)

Only one connection for each user can be open at a time. *HP does not support more than one open connection per user.*

Once a connection has been opened, TELNET enters input mode. In this mode, text that is typed at your terminal is sent to the remote host. To temporarily return to the local node and issue TELNET commands, type the TELNET escape character. The default TELNET escape character is CONTROL]. The ESCAPE command is explained earlier in this section.

If an error occurs while you are running TELNET, an error message will be displayed on your terminal. Refer to "TELNET User Error Messages" in Section 4 of the *NS-ARPA/1000 Error Message and Recovery Manual* for a description of the TELNET error messages.

QUIT

Closes the remote connection, logs off the remote session and terminates TELNET. Same as `EXIT`.

Syntax

```
QUIT
```

Discussion

The TELNET `QUIT` command closes the connection to the remote host, logs off the remote session, terminates TELNET, and returns you to the operating system at the local node. The `QUIT` command is identical to the `EXIT` command. The `CLOSE` command also closes the remote connection, but you remain in TELNET, unless *host* was specified on the TELNET runstring, in which case you return to the local operating system.

Note If you have chained TELNET sessions, the `QUIT` command terminates all sessions, closes all connections, and returns you to the host. If you log off the remote host, only the most recent TELNET session is closed. Any other chained sessions are still active.

RUN

Runs a program at the local node.

Syntax

```
RU[N] program
```

Parameters

program	The name of a program on the local system.

Discussion

The TELNET `RUN` command is identical to the RTE Command Interpreter `RU` command. If the specified program is scheduled successfully, TELNET will wait until it completes. Refer to the *RTE-A User's Manual* for more information on the `RU` command.

Note If you are using TELNET to communicate with systems other than the HP 1000, do not run programs that may alter the terminal configuration you established during your initial TELNET communication. Unpredictable results may occur.

SEND

Sends special characters or commands to the remote node.

Syntax

```
        ⎧ E[SCAPE]      ⎫
        ⎪ IN[TERRUPT]   ⎪
SE[ND] ⎨ A[YT]         ⎬
        ⎪ B[REAK]       ⎪
        ⎩ IP            ⎭
```

Parameters

E[SCAPE]

The SEND ESCAPE command sends the TELNET escape character as a data character to the remote node. Normally, the TELNET escape character is removed from any data sent to the remote node. The SEND ESCAPE command is needed and helpful when you *do* need to send the escape character as a data character.

If the application program running on the remote system requires you to input the current escape character, you can do one of two things:

- change the escape character for the duration of the program with the ESCAPE command
- press the escape character to return to the TELNET prompt (TELNET>), then use SEND ESCAPE to send the escape character as input to the remote application program. (See example below.)

IN[TERRUPT]

The SEND INTERRUPT command sends the interrupt character as a data character to the remote node. Normally, the interrupt character will suspend, interrupt, abort, or terminate the remote process. The SEND INTERRUPT command is needed and helpful when you *do* need to send the interrupt character as a data character.

If the application program running on the remote system requires you to input the current interrupt character, you can do one of two things:

- change the interrupt character for the duration of the program with the INTERRUPT command
- press the escape character to return to the TELNET prompt (TELNET>), then use SEND INTERRUPT to send the interrupt character as input to the remote application program.

A[YT]

The SEND AYT command asks the remote node to return evidence that the TELNET connection is still open. If the connection is still open, the remote node returns an affirmative response (e.g., terminal beep, yes, etc.). AYT stands for “Are you there?”

B[REAK]	The SEND BREAK command invokes a break at the remote node. This command is equivalent to pressing the BREAK key at the remote node.
IP	The SEND IP command sends an interrupt to the remote node. This command is equivalent to pressing the interrupt character (CONTROL Y) at the remote node.

Discussion

SEND performs two distinct functions:

- It allows you to send special character sequences as *data* to the remote node. See SEND ESCAPE and SEND INTERRUPT.
- It allows you to send special *commands* to the remote node. See SEND AYT, SEND BREAK, and SEND IP.

At the system console, pressing the BREAK key invokes VCP mode on HP 1000 nodes.

Example

The following example shows the steps to enter the escape character as input to an application program at the remote node.

Enter CNTRL-] to get last menu:	Remote application program requires the escape character (CONTROL]) as input.
CONTROL]	CONTROL] returns you to TELNET.
TELNET>	TELNET prompt displayed.
TELNET>SEND ESCAPE	SEND ESCAPE sends the escape character as input to the remote application program.
TELNET>RETURN	A single carriage return at the TELNET prompt returns user to the remote session.

STATUS

Displays the current state of the TELNET connection.

Syntax

```
ST[ATUS]
```

Discussion

The STATUS command displays information about the current TELNET connection. It shows the state of the connection (open or closed), the name of the remote host if connection is open, the currently defined escape character and interrupt character, and the data transmission mode (line or character).

To find more information about your local sessions, type the RTE-A command ru,wh,al at the TELNET prompt. For information about your remote sessions, enter the equivalent commands at the remote node.

Example

The following example shows the use of the STATUS command. The ^ represents the CONTROL key.

```
TELNET>STATUS
Connected to Karl.
Escape character = ^]
Interrupt character = ^Y
Transmission mode = Line
```

3

FTP

FTP is an ARPA Service that allows you to transfer files among HP 1000, HP 3000 XL, HP 9000, UNIX*, and non-UNIX network hosts that support ARPA Services. FTP is the file transfer program that uses the ARPA standard File Transfer Protocol (FTP).

FTP also allows you to perform file management operations, such as changing, listing, creating, and deleting remote directories.

This section provides a task-oriented introduction to FTP, followed by a command reference subsection that describes each FTP command in detail, organized in alphabetical order for easy reference.

The FTP commands currently supported on the HP 1000 are listed below:

File Operation Commands:	Directory Operation Commands:	Other Commands:
APPEND	CD	? or ??
ASCII	DIR	..
BELL	DL	/
BINARY	LCD	DEBUG
DELETE	LS	GLOB
FORM	MDIR	HELP
GET	MKDIR	LL
HASH	MLS	PROMPT
MDELETE	NLIST	REMOTEHELP
MGET	PWD	SITE
MODE	RENAME	STATUS
MPUT	RMDIR	SYSTEM
PUT		TR
RECV		USER
RENAME	**Invoke & Exit FTP Commands:**	VERBOSE
RTEBIN		
SEND	!	
STRUCT	BYE	
TYPE	CLOSE	
	EXIT	
	OPEN	
	QUIT	

*UNIX is a registered trademark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

Invoking FTP

FTP is scheduled at the RTE Command Interpreter level (`CI>`). FTP can be invoked with or without the following parameters.

Syntax

```
FTP [-i] [-l[filename]] [-n] [-tfilename] [-v] [-g] [-q]
    [-u[username:password]] [host]
```

Parameters

`-i` — Disables interactive prompting during multiple-file operations. Interactive prompting occurs during multiple file operations to let you selectively proceed with each file. You may use the FTP `PROMPT` command to toggle interactive prompting.

Default: Interactive prompting is enabled.

`-l[`*`filename`*`]` — Logs FTP output to the file specified in *`filename`* in addition to the user's terminal. If *`filename`* is omitted, then the file `FTP.LOG` is used. If *`filename`* is specified, there must be no space between `-l` and the file name. If the file specified by *`filename`* already exists, output is appended to the file.

You may also specify a log file with the FTP `LL` command. See the `LL` command for more details.

Default: Output is displayed on the user's terminal only.

`-n` — Disables auto-login. If auto-login is disabled, you must use the `USER` command to log in to a remote host. If auto-login is enabled, FTP prompts for a user name once a connection is established to a remote host. See the `USER` command for more details.

Default: Auto-login is enabled if FTP input is from the keyboard. Auto-login is disabled if FTP input is from a transfer file.

`-t`*`filename`* — Accepts input from the transfer file specified by *`filename`*. There must be no space between `-t` and the file name.

When you use a transfer file, FTP automatically tries to open the default log file, `FTP.LOG`, if a log file is not already open.

You may also use the FTP `TR` command to specify a transfer file. See the `TR` command for more details on FTP transfer files.

Default: FTP accepts command input from the user's terminal.

`-v` Enables verbose output. Verbose output displays all responses from any remote host to which you are connected. These responses indicate whether FTP commands completed successfully. Verbose output also displays file transfer statistics after the transfer completes.

You may toggle verbose output with the FTP `VERBOSE` command. See the `VERBOSE` command for details.

Default: Verbose output is enabled if FTP input is from the keyboard. Verbose output is disabled if FTP input is from a transfer file, unless you specify the `-v` option.

`-g` Disables file name globbing during multiple file operations. Globbing expands the wild card characters before proceeding with the multiple command. You may use the FTP `GLOB` command to toggle file name globbing.

Default: File name globbing is enabled.

`-q` Enables quiet mode for transfer files. The normal informative messages are not output to the terminal.

Default: Informative transfer file messages are output to the user's terminal.

`-u[username:password]` Specifies the user and password to use. FTP will use the username and password to automatically logon to the host system. If either one needs lower case characters, the string must be surrounded by back quotes (`‘`). An example is: `-u‘MyName:mypass‘`.

Interactive users will be automatically logged on with the first `OPEN` call. Subsequent calls to `OPEN` will prompt for the user and password. The default user will be the user specified on the command line.

A call to `OPEN` in a transfer file will always automatically log the user on using the username and the password specified in the command line. The `USER` command can be used to change the logon for a single `OPEN` call. Subsequent calls to `OPEN` will revert back to using the command line login.

host Specifies the host to which you want to log on. You may use the host's node name or IP address for the *host* parameter.

The syntax for the host's node name is shown here, and is further described under "Node Names" in Section 1 of this manual.

node[.*domain*[.*organization*]]

The syntax for the host's IP address is shown here, and is further described under "IP Addresses" in Section 1 of this manual.

nnn.nnn.nnn.nnn

If *host* is not specified, FTP displays the FTP prompt and waits for you to enter an FTP command. In this case, you must specify the OPEN command to open a connection to a host. The OPEN command is described later in the reference part of this section.

Discussion

FTP runstring parameters must be separated by one or more spaces or by a comma.

FTP runstring parameters may be specified in any order.

When FTP exits normally, the $RETURN parameters are set as follows:

$RETURN1 = 0 is successful. A non-zero value gives the total number of errors including FTP server errors.
$RETURN2 = last FMP error encountered if any.
$RETURN3 = last FTP error encountered if any.
$RETURN4 = last IPC error encountered if any.

FTP can be invoked from a user program by calling FMPRunProgram. The FTP runstring should include the -i and -t options. The transfer file should be an OPEN command (if the nodename is not in the runstring), a USER command, the file transfer commands, and a QUIT or BYE command.

FTP Operation

The following example shows a sample FTP session from an HP 1000 to a UNIX remote host.

```
CI> ftp -l
Logging to file ... FTP.LOG
FTP/1000 User Program Rev. 6100
Enter ? or ?? for help.
```

FTP invoked at the local Command Interpreter prompt. The `-l` option opens the default FTP log file, `FTP.LOG`. FTP messages are sent to this file in addition to the terminal.

```
ftp>
```

FTP prompt is displayed.

```
ftp> open sable
Connecting to ... sable
220 sable FTP server (Version $Revision: 16.2
    Mon Apr 29 20:45:42 GMT 1991) ready.
```

`OPEN` establishes a connection to a remote system called `sable`. Once the connection has been established, you are prompted for the user and password on the remote host.

```
(username: dwight) RETURN
331 Password required for dwight.
(password: dwight) ___
230 User dwight logged in.
Remote system type is UNIX Type: L8.
ftp>
```

Log on to the remote system. The password is not echoed on the terminal.

```
ftp> PWD
257 "/test/dwight" is the current directory.
```

`PWD` shows the name of the remote working directory.

```
ftp> CD EXAMPLES
250 CWD command successful.
```

`CD` connects to the remote `EXAMPLES` directory.

```
ftp> DIR
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls
    (192.6.70.19,33123) (0 bytes)
    total 30
-rw-rw-rw- 1 dwight  arpa1k   600 Feb  1 09:59 test1
-rw-rw-rw- 1 dwight  arpa1k   965 Feb  1 10:00 test2
-rw-rw-rw- 1 dwight  arpa1k  1691 Feb  1 10:00 test3
-rw-rw-rw- 1 dwight  arpa1k  5008 Feb  1 09:47 test4
-rw-rw-rw- 1 dwight  arpa1k  1350 Feb  1 10:04 test5
-rw-r--r-- 1 dwight  arpa1k  3655 Feb  1 09:46 track
226 Transfer complete.
383 bytes transferred in 1.43 seconds
    [ 0.26 kbytes/second ]
```

`DIR` lists the files in `EXAMPLES` directory.

```
ftp> GET TEST1 DATA1
200 PORT command successful.
150 Opening ASCII mode data connection for TEST1
    (192.6.70.19,33124) (600 bytes).
226 Transfer complete.
705 bytes transferred in 0.36 seconds
    [ 1.95 kbytes/second ]
ftp>
```

GET copies remote file TEST1 in the EXAMPLES directory to file DATA1 in the current working directory on the local host.

```
ftp> ?
```

? lists the currently supported FTP commands on the HP 1000.

```
FTP commands may be abbreviated. Commands are:

!        dl       mdir    quit       system
append   exit     mget    quote      tr
ascii    form     mkdir   recv       type
bell     get      mls     remotehelp user
binary   glob     mode    rename     verbose
bye      hash     mput    rmdir      ?
cd       help     nlist   rtebin     ??
close    lcd      open    send       /
debug    ll       prompt  site       ..
delete   ls       put     status
dir      mdelete  pwd     struct

ftp>
```

```
ftp> ? GET
```

? GET displays information about the GET command.

```
GET - transfers a remote file to a local file.
      Same as RECV.

ftp>
```

```
ftp> QUIT
221 Goodbye.
```

Quits FTP and returns to the CI prompt. The connection to the remote host is closed automatically.

```
CI>
```

Terminating FTP

You may terminate your FTP session with one of the following FTP commands:

`BYE`, `EXIT`, or `QUIT` to disconnect from the remote host and exit FTP.

`CLOSE` to disconnect from the remote host and remain in FTP. This allows you to connect to other remote hosts during the same FTP session.

`BYE`, `CLOSE`, `EXIT`, and `QUIT` are discussed in detail in the command reference part later in this section.

Temporarily Exiting FTP

You can temporarily exit FTP and return to the CI prompt on your HP 1000. This allows you to work on the local host and then return to FTP. You can either:

- execute a single command on your local host and automatically return to FTP by typing a single RTE program name after an exclamation point (`!`) at the FTP prompt:

```
ftp> ! programname
```

- work for an extended time on the local host. To do so, enter a single exclamation point (`!`) at the FTP prompt. This exits you to the CI prompt. To return to FTP, type the CI `EX` command at the CI prompt.

```
ftp> !

CI>

           :
           :

CI> EX

ftp>
```

Obtaining Help

You can obtain summary information about FTP commands with FTP's `HELP` commands. You can either list the FTP commands or get information about a specific FTP command.

To list the FTP commands available on the HP 1000, enter one of the following at the FTP prompt:

```
{ ?    }
{ ??   }
{ HELP }
```

To get information about a specific FTP command, type one of the following:

```
{ ?    }
{ ??   } ftp_command
{ HELP }
```

where *ftp_command* is an FTP command (or command abbreviation) listed in Table 3-2. You need to use a space or comma between the command keyword and the FTP command. For example:

`HELP GET` or `HELP,GET.`

RTE-A CI Files and Directories

A file name on the RTE-A CI system can have up to 16 alphanumeric characters. In addition, a file can have a file extension, marked by a dot followed by an extension of up to four characters. The first character of the file name must be a letter. Examples of valid RTE-A file names are NOTES.DOC and TEST23. Capitalization of file names is optional, because the HP 1000 always shifts the input to uppercase. On the HP 1000, the following three file names all refer to the same file: TEST23, Test23, test23.

Note HP 9000 hosts and other UNIX hosts distinguish uppercase and lowercase in file names. Hence, TEST23, Test23, and test23 refer to three different files on those systems.

HP 1000 FTP does *not* upshift the file names for remote hosts, such as UNIX, that distinguish uppercase and lowercase in file names.

The RTE-A CI file system has a hierarchical file structure. Files are catalogued in *directories*. Directories can also contain similar information about other directories, called *subdirectories*. Subdirectories have the same characteristics as directories; the term subdirectory means only that the directory is catalogued in the next higher level directory or subdirectory. Each account or logon on the HP 1000 has a default logon working directory. This directory is automatically made available to you when you log on.

If the HP 1000 file resides in the hierarchical file system, the file name syntax is as follows:

```
[/][directory/][subdir/] ... filename[:::type:size:recordlength]
                      - or -
[subdir/]filename::[directory][:type:size:recordlength]
```

Parameters

directory The directory containing the file. If the *directory* parameter is omitted, the default logon working directory is used.

In the first form of the syntax, if the initial slash (/) is omitted and the *directory* parameter is specified, the directory is assumed to be a subdirectory of the working directory.

In the second form of the syntax, if *directory* is omitted, you must have three colons (`:::`) between the file name and the file type, if a file type is specified, such as: `FOOFILE:::4`

subdir The directory contained under a directory. HP 1000 file system accepts many levels of directories.

filename Specifies the RTE-A file name, including an optional file extension.

type Specifies the file type. HP 1000 files can have the following file types:

-1 Symbolic link files. A symbolic link is a file that indirectly refers to another file. The symbolic link file itself contains a file descriptor that points to the new file.

0 An I/O device. Type 0 is used in accessing devices with file calls. There is no disk file or directory entry for type 0 files, and they do not have the other properties listed in this section.

1 Random access files. These do not have any structure information in them. These files contain fixed record lengths (128 words). They can be read and written very quickly.

2 Fixed-length record, random access files. The record length is defined when the files are created. They are usually user-created, large data files.

3 Type 3 and higher files are variable-length and higher record, sequential files suitable for use as text files. There is no difference in the handling of file types 3, 4, and 7. Type 3 is for general purpose files and can be used for text. This is the default file type when files are created with the CR command. Type 4 is recommended for text files. By convention, type 5 is used for Compiler or Assembler relocatable output files, type 6 is for program files that are memory-images of executable programs, and type 7 is for Compiler or Assembler absolute binary output files. Type 6 files are treated the same as type 1 files.

8 Type 8 and higher files are user-defined, with the following exceptions.

	12	Byte stream files. These do not have any structure information in them. The directory entry for each type 12 file contains a pointer to the last byte in the file. The fields for number of records and record length are not defined for type 12 files.
	6004	CALLS catalog files.

size Specifies how many blocks of disk space the file needs. One block is 128 words (256 bytes or characters).

recordlength Specifies the file's record length, in words, for fixed record files, especially type 2 files.

Discussion

The maximum file path name, including the file name, is 63 characters.

FTP does not support sparse files (type 2 files with missing extents).

The file descriptor parameters—*type*, *size*, and *recordlength*—are optional. If specified, they must be given in the order shown above. To omit a parameter that is in front of another specified parameter, you must enter a colon as a placeholder for the omitted parameter, such as: `FOOFILE:::4::300`. This file has type 4, and record length of 300 words; note that a colon serves as placeholder for the omitted *size* parameter (between 4 and 300) in addition to the colon preceding the record length parameter.

In a file transfer, if file descriptor parameters are specified for the source file, they are used for the destination file, unless the destination file specifies different file descriptor parameters. Note that it is recommended that the source and destination file use the same file descriptor parameters.

When a Revision 6.0 or later FTP client discovers that an FTP server is another Revision 6.0 or later HP 1000, the client transfers files to and from the server such that the original file attributes—file type, file size, and record size—are retained. The user does not have to specify anything except the file name. FTP automatically sets the transfer type to `BINARY`.

For HP 1000 server systems that are pre-6.0, or for other server systems such as HP-UX, an additional command, `RTEBIN`, sets the transfer type to `BINARY`; for subsequent `PUT` or `MPUT` commands from the Revision 6.0 FTP client, the file type, size, and record length are added to the destination file descriptor.

For example, a `PUT` of file `FOO` results in a destination file name of:

    FOO:::type:size:recordlength

This enables such files to be transferred to an HP-UX system and transferred back without loss of their attributes.

Example

Note that the first example below is a shorthand for the second example. They both transfer a local type 6 file to the remote node. The source and destination files have the same file type and file name.

```
ftp> PUT TEST:::6

ftp> PUT TEST:::6 TEST:::6
```

FMGR Cartridge Files

If the HP 1000 file resides on a FMGR cartridge, the file name syntax is as follows:

`filename[:security code][:crn][:type][:size][:record length]`

where `sc` is the security code and `crn` is the cartridge reference number. The `type`, `size`, and `recordlength` parameters are the same as described for a hierarchical file. (See the RTE-A CI Files and Directories, Parameters subsection earlier in this section.)

For more information about RTE-A files and directories, refer to the *RTE-A User's Manual*, part number 92077-90002.

Transferring Files With FTP

FTP offers the following file transfer operations:

- `GET` or `RECV`—transfers a file from the remote to the local host.
- `MGET`—transfers multiple files from the remote to the local host.
- `PUT` or `SEND`—transfers a file from the local to the remote host.
- `MPUT`—transfers multiple files from the local to the remote host.

When you transfer files with FTP, you are *copying* these files from one place to another. Transfers do *not* move or delete the original files.

If no part of the target file path is specified, the source path is used for the target. In this case, any directory path you specify as part of the source file must also exist on the target host. Otherwise, FTP will not transfer the file.

The example below transfers a remote file `user/examples/test` to file `user/examples/test` on the local host. Note that in order for this `GET` command to work, you must have a directory path of `/user/examples` on your local (target) host.

```
ftp> GET /user/examples/test
```

The example below transfers a remote file `user/examples/test` to file `foo` on the current working directory of the local host.

```
ftp> GET /user/examples/test foo
```

If no directories are specified, FTP transfers the files between the default working directory of the remote and local hosts.

```
ftp> PUT prog1 prog2
```

ASCII File Transfers

ASCII file transfer should be used for transferring files containing ASCII data (that is, file types 3 and 4). File types 1, 2, 5, and 6 usually contain binary data and will cause unpredictable results if you use ASCII file transfer.

The default file attributes for an ASCII file are:

File Type = 4
Record Length = 256 words

FTP ASCII file transfers use the above default file attributes, unless you change them in the file description. The maximum record length allowed for ASCII records is 2048 words (4096 bytes). Records larger than 2048 words are truncated during file transfer.

You may use the `ASCII` command to set file transfer to ASCII type. Refer to the `ASCII` command, explained later in this section, for more details about transferring ASCII files.

Binary File Transfers

You can specify binary file transfers with the `BINARY` or `RTEBIN` commands. Files are transferred in binary mode automatically when both systems are Revision 6.0 or later HP 1000s.

When using binary file transfers, if both systems are *not* Revision 6.0 or later HP 1000s, it is strongly recommended that you specify the destination file type, size, and record length (where record length applies to type 2 files only). If you do not specify these destination file attributes and you are transferring to an HP 1000 system, then the destination default file type is 1, record length 128 words.

For binary file transfers, if the destination file size specified is greater than the original (source) file size, FTP fills the remaining portion to the end of the file with null characters.

Refer to the `BINARY` and `RTEBIN` commands for more information about transferring binary files.

Example

The following shows file transfers in the case where one of the systems involved in the transfer is *not* a Revision 6.0 (or later) HP 1000 system. The examples show the `GET` command used to transfer files of type 1 through type 6, where file `FOO1` is of type 1, `FOO2` is of type 2, and so on. In these examples, the target file will have the same file type and file name as the source file.

```
ftp> BINARY

ftp> GET FOO1

ftp> GET FOO2:::2:100:56

ftp> ASCII

ftp> GET FOO3:::3

ftp> GET FOO4

ftp> BINARY
```

```
ftp> GET FOO5:::5

ftp> GET FOO6:::6
```

Note that for file transfers other than file types 1 and 4, the file type specification is required to ensure that the target file has the same file type as the source file. File type 1 and type 4 are the default file types for binary and ASCII transfers, respectively.

How FTP Treats Wild Card Characters

You can use wild card characters in the file path for the FTP listing commands (DIR, DL, LS, NLIST, MDIR, and MLS) and for multiple file operation commands (MPUT, MGET, and MDELETE). These wild card characters represent a set of characters or character strings and are a “shorthand” way of specifying a set of directory or file names. Different file systems have their own set of wild card characters, so the wild card characters that are valid depend on the file system that processes the FTP command.

The following table is a quick reference to the meaning of the wild card characters supported on the HP 1000 and UNIX file systems.

Table 3-1. FTP Wild Card Characters

Character	Matches
@	Any string, including a null string. The file system processing the FTP command must be an HP 1000 CI file system.
-	Any single character. The file system processing the FTP command must be an HP 1000 CI file system.
*	Any string, including a null string. The file system processing the FTP command must be a UNIX file system.

For more information about wild card characters, refer to the documentation on the specific file systems that you will be using.

The dash (-) and at-sign (@) wild card characters can be used only in the file name. They have no special meaning in the directory and subdirectory names.

Note Wild card characters are always expanded (that is, always take effect) for listing commands: DIR, DL, LS, NLIST, MDIR, and MLS. The following example lists all files in the /TEST directory with the extension FTN:

```
DIR /TEST/@.FTN
```

If file name globbing is enabled, wild card characters are expanded for the multiple file operations commands: MDELETE, MGET, and MPUT. By default, file name globbing is enabled. Globbing may be toggled on and off by the GLOB command or disabled by the -g option when invoking FTP.

Wild card characters are *not* expanded for single file transfer commands: GET, PUT, SEND, and RECV.

Refer to the GLOB command for more information about file name globbing.

Examples

The following MPUT command is used to transfer all files on the current working directory of the local HP 1000 host to the working directory of the remote host. The MGET command is used to transfer all files on the current working directory of a remote UNIX host to the local host.

```
ftp> MPUT @                    (The local file system is an HP 1000)
ftp> MGET *                    (The remote file system is a UNIX system)
```

Note If you use a wild card character that is not recognized by the file system (such as the asterisk character on an HP 1000), the file system treats the character "as is." For example, FTP will try to list a file called * with the following command to a remote HP 1000 host: DIR *.

$VISUAL Command Editing

For VC+ systems, FTP supports the $VISUAL command editing modes (for example, EMACS or VI) through use of the CMNDO monitor. To use this feature, make sure your home directory is set, using the PATH program to set UDSP #0, and create file FTP.STK in that directory as a type 3 or 4 file.

To use CMNDO, set the $CMNDO_FTP or $CMNDO environment variable by entering one of the following commands from CI (and restart FTP):

```
CI> set -x CMNDO_FTP = T          Use CMNDO from FTP, but not other utilities.
```

or

```
CI> set -x CMNDO = T              Use CMNDO from all utilities that support it.
```

If you have $CMNDO set to TRUE but do not wish to use CMNDO from FTP enter:

```
CI> set -x CMNDO_FTP = F
```

Refer to the "Command Editing" chapter in the *RTE-A User's Manual*, part number 92077-90002, for more information on the $VISUAL command editing modes and the CMNDO monitor.

FTP Commands

ARPA/1000 supports the FTP commands listed in Table 3-2. These commands are documented in detail on the following pages of this section.

FTP accepts the unique abbreviation for the FTP command keyword, as shown in Table 3-2.

FTP command parameters must be separated by one or more spaces or by a comma.

If you specify more than the number of parameters allowed for a command, FTP displays the proper syntax for the command on your screen, followed by the FTP prompt. You may then reenter the command.

If you omit a required parameter, FTP prompts you for the parameter.

Table 3-2. FTP Commands

Command	Description
`!`	Invokes CI on the local host.
`?[?]`	Displays FTP commands and help information. Same as `HELP`.
`..`	Sets the working directory on the remote host to the parent directory.
`/`	Displays the FTP command stack.
`AP[PEND]`	Transfers *local_file* to the end of *remote_file*.
`AS[CII]`	Sets the FTP file transfer type to `ASCII`. This is the default type.
`BE[LL]`	Sounds a bell after each file transfer completes.
`BI[NARY]`	Sets the FTP file transfer type to `BINARY`.
`BY[E]`	Closes the remote connection and exits from FTP. Same as `EXIT` and `QUIT`.
`CD`	Sets the working directory on the remote host to the specified *remote_directory*.
`CL[OSE]`	Closes the remote connection and remains in FTP.
`DEB[UG]`	Prints the commands that are sent to the remote host.
`DEL[ETE]`	Deletes the specified *remote_file* or empty *remote_directory*.
`DI[R]`	Writes an extended directory listing of a remote directory or file to the terminal or to a *local_file*.
`DL`	Writes an extended directory listing in RTE-A DL format to the terminal or to a *local_file*.
`E[XIT]`	Closes the remote connection and exits from FTP. Same as `BYE` and `QUIT`.
`F[ORM]`	Sets the FTP file transfer form to the specified format. The only supported format is `non-print`.
`G[ET]`	Transfers *remote_file* to *local_file*. Same as `RECV`.
`GL[OB]`	Toggles file name globbing.
`HA[SH]`	Toggles hash-sign (#) printing for each data block transferred. The size of a data block is 1024 bytes.
`HE[LP]`	Displays FTP commands and help information. Same as `?` and `??`.
`LC[D]`	Sets or displays the local working directory.

Command	Description
LL	Specifies a log file to which FTP sends the commands and miscellaneous messages ordinarily displayed to the user's terminal.
LS	Writes an extended directory listing of a remote directory or file to the terminal or to a *local_file*.
MDE[LETE]	Deletes multiple *remote_files*.
MDI[R]	Writes an extended directory listing of remote directories or files to a *local_file*.
MG[ET]	Transfers multiple *remote_files* to the local system, using the same file names.
MK[DIR]	Creates a *remote_directory*.
ML[S]	Writes an abbreviated directory listing of remote directories or files to a *local_file*.
MO[DE]	Sets the FTP file transfer mode to the specified mode. The only supported mode is stream.
MP[UT]	Transfers multiple *local_files* to the remote system, using the same file names.
N[LIST]	Writes an abbreviated directory listing of a remote directory or file to the terminal or to a *local_file*.
O[PEN]	Establishes a connection to the remote host.
PR[OMPT]	Toggles interactive prompting.
PU[T]	Transfers *local_file* to *remote_file*. Same as SEND.
PW[D]	Writes the name of the remote working directory to the terminal.
QUI[T]	Closes the remote connections and exits FTP. Same as BYE and EXIT.
QUO[TE]	Sends arbitrary FTP server commands to the remote host.
REC[V]	Transfers *remote_file* to *local_file*. Same as GET.
REM[OTEHELP]	Requests help information from the remote host.
REN[AME]	Renames a *remote_file* or *remote_directory*.
RM[DIR]	Deletes an empty *remote_directory*.
RT[EBIN]	Sets the FTP file transfer type to BINARY. PUT will create destination file names with the full RTE file descriptor.
SE[ND]	Transfers *local_file* to *remote_file*. Same as PUT.
SI[TE]	Performs server-specific services.
STA[TUS]	Writes the current status of FTP to the terminal.
STR[UCT]	Sets the FTP file transfer structure to the specified structure. The only supported structure is file.
SY[STEM]	Shows the remote system type.
TR	Specifies an input file from which to get FTP commands.
TY[PE]	Sets the FTP file transfer type to the specified type. ASCII and BINARY are the types currently supported.
U[SER]	Logs into the remote host on the current connection, which must already be open.
V[ERBOSE]	Toggles verbose output. When verbose output is enabled, FTP displays responses from the remote host.

Invokes CI or runs the specified program on the local HP 1000 host.

Syntax

```
! [prog_name]
```

Parameters

prog_name	Any program that you can execute singly. Once the command is executed, you automatically return to FTP. A space or comma must separate the exclamation mark and the program name. If no command is specified after the exclamation mark, you will remain in CI until you execute the CI `EX` command. This allows you to run multiple CI commands before returning to FTP.

Example

The exclamation point after the FTP prompt exits you to the CI prompt. To return to FTP, use the CI `EX` command.

```
ftp>!

CI> wh,al

         :
         :

CI> EX

ftp>
```

?

Displays FTP commands and help information. You may use a single question mark (?) or double question marks (??). Same as HELP command.

Syntax

```
?[?] [command]
```

Parameters

command	Any FTP command listed in Table 3-2, "FTP Commands." If no command is specified, FTP lists the currently supported FTP commands. When a command is specified, FTP displays a brief description of the command. A space or comma must separate the question mark and the *command* parameter.

Example

The following example shows help information displayed by ?, without and with a specified command.

```
ftp>?

FTP commands may be abbreviated. Commands are:

!       dl      mdir    quit       system
append  exit    mget    quote      tr
ascii   form    mkdir   recv       type
bell    get     mls     remotehelp user
binary  glob    mode    rename     verbose
bye     hash    mput    rmdir      ?
cd      help    nlist   rtebin     ??
close   lcd     open    send       /
debug   ll      prompt  site       ..
delete  ls      put     status
dir     mdelete pwd     struct

ftp> ? GET

GET - transfers a remote file to a local file. Same as RECV.

ftp>
```

Sets the working directory on the remote host to the parent directory.

Syntax

```
..
```

Discussion

The double period (..) command is a shorthand for the `CD ..` command. See the `CD` command for details.

Example

If you are currently in `/USER/EXAMPLES` directory, a double period (..) command places you in the `/USER` directory.

/

Displays the FTP command stack. Used for time saving purpose of not having to retype an FTP command. Similar to the command stack display function (/) in RTE-A.

Syntax

```
  ┌           ┐
  │ linecount │
/ │ /...      │
  │ .text     │
  └           ┘
```

Parameters

linecount	Optional command line count integer, from 1 to 12, that specifies the number of command lines from the last command entered to be displayed.
/...	Optional extra slashes, up to 12 slashes, that you may specify. If two extra slashes are specified, the last two commands executed are displayed. If three extra slashes are specified, the last three commands are displayed, and so on.
.text	String of text that FTP uses to search the command stack. The text must be preceded by a period (.). FTP displays commands in the stack that contain the specified string of text.

Discussion

You may execute a command in the command stack by first displaying it with the / command, then move the cursor to the desired command, edit it if necessary, and press carriage return.

A single slash (/), without any parameters, displays the last twelve command lines. If there are less than 12 command lines in the stack, then only the existing command lines in the stack are displayed.

The extra slashes and the *linecount* parameter are mutually exclusive.

If you have set environment variables to enable $VISUAL command editing, then the full features of the command stack editor apply. See the preceding section on $VISUAL command editing.

The following examples show what is displayed by different / commands:

Command Syntax	Number of Lines Displayed
/	Last 12 command lines displayed.
/8	Last 8 command lines displayed.
///	Last 2 command lines displayed.

Example

The first / command displays the last 3 commands of the command stack. The second / command displays commands that contain the string “ll”.

```
ftp> /3
pwd
cd /DWIGHT/EXAMPLES
dir

ftp> /.ll
ll 1
ll ftp.log

ftp>
```

APPEND

Transfers a local file to the end of a remote file.

Syntax

```
AP[PEND] local_file [remote_file]
```

Parameters

local_file	Specifies a valid file on the local host to be appended to the remote file.
remote_file	Specifies a valid file path on the remote host to append the local file. If the *remote_file* does not exist, FTP creates it before appending the local file. If the *remote_file* parameter is omitted, FTP uses the *local_file* name as the *remote_file* name.

Discussion

For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section.

APPEND is not supported for binary file transfers. APPEND is also not supported for ASCII file transfers involving file types 1, 2, and 6.

Example

The following example appends file TEST1, from the local working directory, to the file FULLTEST in directory /USER/TEST on the remote host.

```
ftp> APPEND TEST1 /USER/TEST/FULLTEST
ftp>
```

ASCII

Sets the file transfer type to `ASCII`.

Syntax

```
AS[CII]
```

Discussion

The file transfer types supported for FTP are: ASCII and binary.

ASCII type should be used for transferring ASCII files. ASCII type should also be used for transferring files between unlike host systems (for example, between an HP 1000 and a UNIX host).

Binary transfer should be used for transferring files between similar operating systems and transferring files for archiving. See the `BINARY` command for more details.

When ASCII file transfer type is specified, the following occurs:

- *Transferring to an HP 1000*:

  The <CR> and <LF> characters at the end of each record are removed as the record is written to disk.

- *Transferring from an HP 1000*:

  The <CR> and <LF> characters are appended to each record before it is sent over the data connection.

When you use ASCII file transfer, make sure your file does not contain either the <LF> character or the <CR> character as part of the data; otherwise, the file may be corrupted after the transfer.

The default attributes used in ASCII type are:
File Type = 4
Record Length = 256 words

FTP ASCII file transfer uses the above default file attributes, unless you change them in the file description. The maximum record length allowed for ASCII transfer is 2048 words (4096 bytes).

The default attributes may be changed by specifying file descriptor parameters as part of the file name, such as: `/EXAMPLE/TESTFILE:::3:150`

The syntax for HP 1000 file names and parameters is shown earlier in this section, under "RTE-A Files and Directories."

BELL

Specifies that a bell sound is generated after each file transfer completes. This command toggles.

Syntax

```
BE[LL]
```

Discussion

By default, the BELL sound is disabled.

BINARY

Sets the FTP file transfer type to `BINARY`.

Syntax

```
BI[NARY]
```

Discussion

The supported FTP file transfer types are: ASCII and binary. See the `ASCII` command for details.

Once the file transfer type is specified as binary, it remains in effect until you close the remote connection or until you specify ASCII by using the `ASCII` command or `TYPE ASCII` command.

ASCII transfer type should be used for transferring ASCII files and for transferring files between unlike host systems (for example, between an HP 1000 and a UNIX host).

Binary transfer type should be used in two kinds of file transfers:

- Transferring files between similar operating systems.
- Transferring files for archiving (for storage and not access). The files would not be in a meaningful format if accessed on the destination host after the transfer. You would need to transfer the files back to the same type of operating system from which they originated to access them in a meaningful form.

For binary file transfers, if the destination file size specified is greater than the original (source) file size, FTP fills the remaining portion to the end of file with null characters.

The default attributes used in binary transfer type to/from a pre-6.0 system are:

File Type = 1
Record Length = 128 words

The default attributes may be changed by specifying file descriptor parameters as part of the file name, such as:

```
/EXAMPLE/TESTFILE:::3:150
```

The syntax for HP 1000 file name and parameters is shown earlier in this section, under “RTE-A Files and Directories.” For more information on transferring binary files with FTP, refer to “Binary File Transfers” earlier in this section.

Note When using binary file transfer, you must specify the target file type to be the same as the source file type. This can be accomplished by specifying the file type in the source or target file name (see the second and third examples below). If you do not specify the target file type to be the same as the source file, the target file defaults to type 1, and the result is unpredictable if source and target file types do not match.

Transfers to/from Revision 6.0 or later systems do not need the file types specified. FTP automatically transfers the source file type.

Examples

The following command transfers a type 1 file. The target file will have the same file name as the source file in this example.

```
ftp> PUT LOCALFILE
```

The following command transfers a type 2 file, `SOURCE`, to a type 2 file on the remote host, `TARGET`. Note the type 2 specification for the target file.

```
ftp> PUT SOURCE TARGET:::2::200
```

The following command transfers a file, `TEST`, to the remote host with the same file name and file type. FTP uses the file specification of the source file for the destination file.

```
ftp> PUT TEST:::6
```

Closes the remote connection and exits from FTP. Same as EXIT and QUIT.

Syntax

```
BY[E]
```

Discussion

The FTP BYE command closes the connection to the remote host, logs off the remote session, terminates FTP, and returns you to the operating system of the local node. The BYE command is identical to the EXIT and QUIT commands.

If you want to close the remote connection but remain in FTP, use the CLOSE command. See the CLOSE command for more information.

CD

Sets the working directory on the remote host to the specified directory.

Syntax

```
CD remote_directory
```

Parameters

remote_directory Specifies a valid directory on the remote host to be the working directory.

By default, the working directory is the default login directory. To change to another directory on the remote host, you should use the `CD` command.

Discussion

To change to the parent directory, you may use double periods (`..`) or the `CD ..` command.

To determine the current working directory on the remote host, use the `PWD` command, described later in this section.

The `LCD` command is the equivalent command for the local host. `LCD` sets a specified directory as the working directory on the local host.

For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section.

CLOSE

Closes the remote connection and remains in FTP.

Syntax

```
CL[OSE]
```

Discussion

The CLOSE command closes the remote connection and logs off the remote host, but does not exit from FTP. This allows you to log on to another remote host and transfer files all within the same FTP session.

CLOSE resets the file transfer type to the default mode, ASCII.

To close the remote connection and exit from FTP, use the BYE, EXIT, or QUIT command.

DEBUG

Prints the commands that are sent to the remote host. Used for debugging the current FTP session. This command toggles debug mode.

Syntax

```
DEB[UG]
```

Discussion

When debug mode is enabled, FTP displays the FTP server commands that are sent to the remote host along with any parameters that are needed to execute the FTP commands. This enables the user to verify which command is being sent to the server.

By default, the debug mode is disabled.

Example

The following session shows an LS command executed with debug mode on and then off.

```
ftp> DEBUG
Debugging on.

ftp> LS
---> PORT 192,6,70,19,128,57
200 PORT command successful.
---> NLST
150 Opening data connection for /bin/ls (192.6.70.19,32825) (0 bytes).
test1
test2
test3
test4
test5
tracker_form
226 Transfer complete.
49 bytes transferred in 0.55 seconds [ 0.8 kbytes/second ]

ftp> DEBUG
Debugging off.

ftp> LS
200 PORT command successful.
150 Opening data connection for /bin/ls (192.6.70.19,32825) (0 bytes)
test1
test2
test3
test4
test5
tracker_form
226 Transfer complete.
49 bytes transferred in 0.22 seconds [ 0.22 kbytes/second ]

ftp>
```

DELETE

Deletes the specified remote file or remote directory.

Syntax

```
DEL[ETE]  remote_file
```

Parameters

remote_file	Specifies a valid file path on the remote host to be deleted. This can be a file or an empty directory. For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section. If the specified file does not exist or is not an empty directory, FTP displays a warning and ignores the command.

Discussion

Refer to MDELETE, later in this section, for deleting multiple remote files.

DIR

Writes an extended directory listing of a remote directory or file to the terminal or to an output file.

Syntax

```
DI[R] [remote_listing] [local_file]
```

Parameters

remote_listing Specifies the remote directory or file mask from which a directory listing is to be generated. If this parameter is not specified, a directory listing of the remote working directory is generated.

local_file Specifies the output file on the local host to store the directory listing. If this parameter is not specified, the directory listing is written to your terminal.

Discussion

The DIR command requests an extended directory listing from the remote server. DIR, without any parameters, lists the current remote working directory to your terminal.

To omit the *remote_listing* parameter but specify the *local_file* parameter, use a comma as a placeholder for the first parameter, such as

```
DIR,,local_file
```

This command lists the current remote working directory to the specified local file.

The following table shows the FTP commands available for listing remote directories and files. See the individual commands for listing a single directory or file (DIR, DL, LS, NLIST) for samples of listing formats.

FTP Remote Listing Commands

Listing a Single Directory or File	Listing Multiple Directories or Files
DIR—extended listing	MDIR—extended listing
DL—extended RTE listing	
LS—extended listing	MLS—abbreviated listing
NLIST—abbreviated listing	

For more information about the HP 1000 file name syntax, refer to “RTE-A Files and Directories” earlier in this section.

Example

The following is an example of output returned from an FTP 1000 server:

```
ftp> DIR

200 PORT command successful.
150 Opening data connection for /bin/ls -l (192.6.70.19,33123) (0 bytes).
total 30
-rw-rw-rw-   1 dwight   ns1000       600 Feb  1 09:59 test1
-rw-rw-rw-   1 dwight   ns1000       965 Feb  1 10:00 test2
-rw-rw-rw-   1 dwight   ns1000      1691 Feb  1 10:00 test3
-rw-rw-rw-   1 dwight   ns1000      5008 Feb  1 09:47 test4
-rw-rw-rw-   1 dwight   ns1000      1350 Feb  1 10:04 test5
-rw-r--r--   1 dwight   ns1000      3655 Feb  1 09:46 tracker_form
226 Transfer complete.
383 bytes transferred in 1.43 seconds   [ 0.26 kbytes/second ]

ftp>
```

DL

Writes an extended RTE-A directory listing of a remote directory or file to the terminal or to an output file.

Syntax

```
DL [remote_listing] [local_file]
```

Parameters

remote_listing Specifies the remote directory or file mask from which a directory listing is to be generated. If this parameter is not specified, a directory listing of the remote working directory is generated.

local_file Specifies the output file on the local host to store the directory listing. If this parameter is not specified, the directory listing is written to your terminal.

Discussion

DL, without any parameters, lists the current remote working directory to your terminal in RTE-A DL format. The DL command requests an RTE-A format directory listing from a Revision 6.0 (or later) FTP 1000 server. The command only works when a Revision 6.0 (or later) FTP client communicates with a Revision 6.0 (or later) FTP server.

To omit the *remote_listing* parameter but specify the *local_file* parameter, use a comma as a placeholder for the first parameter, such as:

```
DL,,local_file
```

This command lists the current remote working directory to the specified local file.

The following table shows the FTP commands available for listing remote directories and files. See the individual commands for listing a single directory or file (DIR, DL, LS, NLIST) for samples of listing formats.

FTP Remote Listing Commands

Listing a Single Directory or File	Listing Multiple Directories or Files
DIR—extended listing	MDIR—extended listing
DL—extended RTE listing	
LS—extended listing	MLS—abbreviated listing
NLIST—abbreviated listing	

For more information about the HP 1000 file name syntax, refer to “RTE-A Files and Directories” earlier in this section.

Example

```
ftp> DL
200 Type set to A.
200 PORT command successful.
150 Opening data connection for file list.
directory /WARRENO/TRACES

    name      ex   prot   type  blks   words  recs    addr/lu

FAIL1.FMT          rw/r      3  1589  203322  4841  993264/16
FAIL1.TRC          rw/rw     2   256   32768    32   18560/16
FAIL2.TRC          rw/rw     2  2048  262144   256  856400/16
FAIL3K.TRC         rw/r      2  1024  131072   128  344832/16
FMTRC.RUN          rw/r      6   728   93184   728  387056/16
HEX1.FMT           rw/r      3   635   81186  1933 1002432/16
HEXFORMAT.FMT      rw/r      3   567   72450  1725  994864/16
HEXTR.CMD          rw/r      4     2     240    42    2816/16
HEXTR1.FMT         rw/r      3   567   72450  1725  985904/16
IPFRAG.TRC         rw/r      2  1024  131072   128  858448/16
LAVERNE.FMT      * rw/r      3  2304  215502  5131    8384/16
LAVERNE.TRC        rw/r      2   256   32768    32   18816/16
NICE10.FMT         rw/r      3   571   73038  1739 1053952/16
NICE11.FMT         rw/r      3   568   72702  1731  977184/16
NSTRACE.FMT        rw/r      3   318   40698   969  358816/16
NSTRGOLD.TRC       rw/r      2   256   32768    32    3072/16
NS TRACE.TRC       rw/rw     2  1024  131072   128  852320/16
OCTALTR1.FMT       rw/r      3   567   72450  1725  986480/16
OCTFORMAT.FMT      rw/r      3   567   72450  1725  995440/16
TEST.FMT         * rw/r      4   768   90132  2146   57152/16
XHEX.FMT           rw/r      3   567   72450  1725  996016/16

226 Closing data connection.
1312 bytes transferred in 0.44 seconds         [ 2.98 kbytes/second ]
200 Type set to I.
ftp>  221 Service closing control connection.
ftp>
```

EXIT

Closes the remote connection and exits from FTP. Same as BYE and QUIT.

Syntax

```
E[XIT]
```

Discussion

The FTP EXIT command closes the connection to the remote host, logs off the remote session, terminates FTP, and returns you to the operating system of the local node. The EXIT command is identical to the BYE and QUIT commands.

If you want to close the remote connection but remain in FTP, use the CLOSE command. See the CLOSE command for more information.

Sets the FTP file transfer form to the specified format. The only supported format is `non-print.`

Syntax

```
F[ORM] format
```

Parameters

format Specifies the file transfer format. Currently the only supported format is `non-print.`

Non-print format specifies that no vertical format information is contained. Normally, this format is useful for files destined for processing or just storage. If the file is passed to a printer process, the process may assume standard values for spacing and margins.

Table 3-3 lists the FTP commands used to define file transfer form, mode, structure, and type.

Table 3-3. FTP File Transfer Form, Mode, Structure, and Type

Command	Supported Parameters	Meaning
FORM	non-print	non-print specifies that no vertical format information is contained in the file.
MODE	stream	Data is transmitted as a stream of bytes.
STRUCT	file	File is considered to be a continuous sequence of data bytes.
TYPE*	A[SCII]	Data is converted to the standard 8-bit ASCII representation for transfer. Also see the ASCII command.
	B[INARY]	Data is sent as it appears on disk. Also see the BINARY command.

* ASCII file transfer type is the default when both systems are not Revision 6.0 (or later) HP 1000s. To specify ASCII or binary, you may also use the ASCII or BINARY command, respectively.

GET

Transfers a remote file to a local file. Same as RECV.

Syntax

```
GE[T] remote_file [local_file]
```

Parameters

remote_file Specifies a valid file path on the remote host to be copied to the local host.

local_file Specifies the file on the local host to copy into. If this parameter is not specified, FTP uses the remote file path as the local file path.

If a local file name is specified without a directory, the current working directory on the local host is used.

Caution If a local file with the same file name already exists before the file transfer, it is overwritten without warning.

Discussion

Refer to MGET, later in this section, for transferring multiple remote files.

For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section.

Example

The first GET command transfers /user/example/file1 to /user/example/file1 on the local host. If a directory path /user/example/ does not exist on the local host, FTP will generate an error and will not execute the GET command.

The second GET command transfers file foofile in the remote working directory to foofile in the local working directory.

```
ftp> get /user/example/file1
ftp> get foofile
```

GLOB

Toggles file name globbing (expansion) for multiple file operations.

Syntax

```
GL[OB]
```

Discussion

When file name globbing is enabled, FTP expands wild card characters in multiple file and directory operations. In other words, FTP uses the wild card characters as wild cards and not as the characters they normally represent. The wild card characters used depend on the type of the remote host. See the subsection "How FTP Treats Wild Card Characters" earlier in this section for more information about the use of wild card characters.

If globbing is enabled, wild card characters are expanded for the multiple file operations, such as `MDELETE`, `MGET`, and `MPUT`.

File name globbing may be disabled in one of two ways:

- With the `GLOB` command described here.
- With the `-g` option when invoking FTP. Refer to "Invoking FTP" earlier in this section for more information about this option.

When file name globbing is disabled, the wild card characters are treated "as is" and not as wild cards. So, without globbing, the following command transfers just one file called `@.TXT`:

```
ftp> MPUT @.TXT
```

Note The wild card characters are always expanded for the listing commands: `DIR`, `DL`, `LS`, `NLIST`, `MDIR`, and `MLS`.

Example

The following example shows transferring multiple files with globbing on and then off. (In both cases shown here, interactive prompting is enabled.)

```
ftp> mput @.pas @.ftn
mput ftp.pas (Yes, No, Break, Stop Asking)[Yes]? Y
mput ftpsv.pas (Yes, No, Break, Stop Asking)[Yes]? Y
mput test.pas (Yes, No, Break, Stop Asking)[Yes]? N
mput test.ftn (Yes, No, Break, Stop Asking)[Yes]? N
```

```
mput test2.ftn (Yes, No, Break, Stop Asking)[Yes]? Y

ftp> glob
Globbing off.

ftp> mput @.pas @.ftn

mput @.pas (Yes, No, Break, Stop Asking)[Yes]? N

mput @.ftn (Yes, No, Break, Stop Asking)[Yes]? N

ftp>
```

HASH

Specifies the printing of a hash sign (#) for each data block transferred. The size of the data block is 1024 bytes. This command toggles the printing of hash signs.

Syntax

```
HA[SH]
```

Discussion

By default, hash sign printing is disabled.

HELP

Displays FTP commands and help information. Same as `?` or `??`.

Syntax

```
HE[LP] [command]
```

Parameters

command — Any FTP command listed in Table 3-2, “FTP Commands.”

When a command is specified, FTP displays a brief description of the command.

If no command is specified, FTP lists the currently supported FTP commands.

Example

The following example shows help information displayed by `HELP`, without and with a specified command.

```
ftp>HELP

FTP commands may be abbreviated. Commands are:

!          dl         mdir      quit          system
append     exit       mget      quote         tr
ascii      form       mkdir     recv          type
bell       get        mls       remotehelp    user
binary     glob       mode      rename        verbose
bye        hash       mput      rmdir         ?
cd         help       nlist     rtebin        ??
close      lcd        open      send          /
debug      ll         prompt    site          ..
delete     ls         put       status
dir        mdelete    pwd       struct

ftp> HELP GET

GET - transfers a remote file to a local file. Same as RECV.

ftp>
```

LCD

Sets the local working directory to the specified directory.

Syntax

```
LC[D] [local_directory]
```

Parameters

local_directory Specifies a valid directory on the HP 1000 host to be the local working directory.

If *local_directory* is not specified, FTP returns the user to their home directory.

Refer to “RTE-A Files and Directories” earlier in this section for more information about directories on the HP 1000.

Discussion

If *local_directory* is not specified, FTP returns the user to their home directory, which is defined by setting UDSP#0. If this is not set, LCD prints out the current local working directory.

The CD command is the equivalent command for the remote host. CD sets a specified directory as the working directory on the remote host.

LL

Specifies a local log file to which FTP sends commands and messages in addition to displaying them on the user's terminal.

Syntax

```
LL [local_file]
```

Parameters

local_file Specifies a valid file name on the HP 1000 host as a log file.

All terminal output generated by FTP is logged into this file in addition to your terminal. This allows you to check the log file later for errors and information regarding the transfer.

If *local_file* is not specified, FTP prompts you for the log file name.

If the file already exists, output is *appended* to it.

To close the log file, use `LL,1`.

Discussion

You may specify an FTP log file in one of two ways:

- With the `LL` command as described here.
- With the `-l` option when you invoke FTP. See "Invoking FTP" earlier in this section.

If an FTP log file is already open when you attempt to open another log file, you will get an error on the second log file, and FTP will continue sending log messages to the first log file.

If an error occurs while FTP tries to open a log file (for example, the file is already open), FTP automatically tries to open up the default log file, `FTP.LOG`, so that there is no information lost.

Note FTP treats as a comment line any line that has an asterisk (`*`) as the first character. You may want to use this feature to send comments to a log file. See the example below.

Example

```
ftp> LL TEST.LOG
ftp> * This line is treated and sent to the log file as a comment.
```

Writes an extended directory listing of a remote directory or file to your terminal or to a local file on the HP 1000.

Syntax

```
LS [remote_listing] [local_file]
```

Parameters

remote_listing Specifies the remote directory or file mask from which a directory listing is to be generated. If this parameter is omitted, `LS` lists the remote working directory.

local_file Specifies a valid file path on the local HP 1000 host to store the directory listing. If this parameter is omitted, the directory listing is displayed on your terminal.

Discussion

`LS`, without any parameters, lists the current remote working directory to your terminal.

To omit the *remote_listing* parameter but specify the *local_file* parameter, use a comma as a placeholder for the first parameter, such as:

```
LS,,local_file
```

This command lists the current remote working directory to the specified local file.

The following table shows the FTP commands available for listing remote directories and files. See the individual commands for listing a single directory or file (`DIR`, `DL`, `LS`, `NLIST`) for samples of listing formats.

FTP Remote Listing Commands

Listing a Single Directory or File	Listing Multiple Directories or Files
`DIR`—extended listing	`MDIR`—extended listing
`DL`—extended RTE listing	
`LS`—extended listing	`MLS`—abbreviated listing
`NLIST`—abbreviated listing	

Example

```
ftp> LS

200 PORT command successful.
150 Opening data connection for /bin/ls -l (192.6.70.19,33123) (0 bytes).
total 30
-rw-rw-rw-   1 dwight   ns1000       600 Feb  1 09:59 test1
-rw-rw-rw-   1 dwight   ns1000       965 Feb  1 10:00 test2
-rw-rw-rw-   1 dwight   ns1000      1691 Feb  1 10:00 test3
-rw-rw-rw-   1 dwight   ns1000      5008 Feb  1 09:47 test4
-rw-rw-rw-   1 dwight   ns1000      1350 Feb  1 10:04 test5
-rw-r--r--   1 dwight   ns1000      3655 Feb  1 09:46 tracker_form
226 Transfer complete.
383 bytes transferred in 1.43 seconds   [ 0.26 kbytes/second ]

ftp>
```

Deletes multiple remote files.

Syntax

```
MDE[LETE] remote_file [remote_file ...]
```

Parameters

remote_file Specifies a valid file path on the remote host to be deleted. This can be a file or an empty directory.

The ellipsis (. . .) means that you may specify multiple remote files or empty directories, delimited by a comma or by one or more blank spaces.

You may use wild card characters in the remote file names.

Discussion

If interactive mode is on (the default), FTP prompts you with a confirmation for each specified file before deleting. The confirmation prompt looks like this:

```
mdelete filename (Yes, No, Break, Stop Asking)[Yes]?
```

where `Yes` (or carriage return) will proceed with the file deletion; `No` will abort this file's deletion and proceed to the next file; `Break` will abort `MDELETE` and return to the FTP prompt; and `Stop Asking` will delete the rest of the files specified by `MDELETE` without further confirmation. (See the first example below.)

If interactive mode is disabled, FTP simply deletes the specified remote files without any confirmation prompt. (See the second example below.)

If globbing is enabled (the default), FTP expands the wild card characters before the deletion is executed. However, if globbing is disabled, FTP tries to delete the specified file or directory "as is." Refer to the `GLOB` command for more information.

If a file specified by `MDELETE` does not exist, FTP ignores this file and does not issue an error.

Example

```
ftp> mdelete prog1 prog2 prog3              (interactive mode on)

mdelete prog1 (Yes, No, Break, Stop Asking)[Yes]? Y

mdelete prog2 (Yes, No, Break, Stop Asking)[Yes]? N

mdelete prog3 (Yes, No, Break, Stop Asking)[Yes]? Y

ftp> PROMPT                                 (interactive mode off)
Interactive prompting disabled.

ftp> mdelete file1,file2,file3

ftp>
```

MDIR

Writes an extended directory listing of multiple remote directories or files to a local file.

Syntax

```
MD[IR] remote_listing [remote_listing ...] local_file
```

Parameters

remote_listing	Specifies the remote directories or file masks from which a directory listing is to be generated. The ellipsis (...) means that you can specify multiple remote directories or files, delimited by a comma or by one or more blank spaces.
local_file	A valid file path on the local HP 1000 host to store the remote listing. This parameter is required, because MDIR does not output the remote listing to the terminal. FTP always uses the last parameter in the MDIR command string as the *local_file*.

Discussion

FTP prompts you for a confirmation of the *local_file* to be used. Note that if *local_file* already exists, it is overwritten.

If a directory or file specified by *remote_listing* does not exist, FTP ignores the MDIR command and does not create the output directory listing.

The following table shows the FTP commands available for listing remote directories and files. See the individual commands for listing a single directory or file (DIR, DL, LS, NLIST) for samples of extended and abbreviated listing formats.

FTP Remote Listing Commands

Listing a Single Directory or File	Listing Multiple Directories or Files
DIR—extended listing	MDIR—extended listing
DL—extended RTE listing	
LS—extended listing	MLS—abbreviated listing
NLIST—abbreviated listing	

MGET

Transfers multiple remote files to the local host.

Syntax

```
MG[ET] remote_file [remote_file ...]
```

Parameters

remote_file Specifies a valid file path on the remote host for the file to be transferred.

The ellipsis (. . .) means that you may specify multiple remote files, delimited by a comma or by one or more blank spaces.

You may use wild card characters in the remote file names.

The files are transferred to local files with the same directory paths and names as the source files.

Caution If a local file with the same file name already exists before the file transfer, it is overwritten without warning.

Discussion

If interactive mode is on (the default), FTP prompts you with a confirmation for each specified file before transferring. The confirmation prompt looks like this:

```
mget filename (Yes, No, Break, Stop Asking)[Yes]?
```

where `Yes` (or carriage return) will proceed with the file transfer; `No` will abort this file transfer and proceed to the next file; `Break` will abort `MGET` and return to the FTP prompt; and `Stop Asking` will transfer the rest of the files specified by `MGET` without further confirmation. (See the example below.)

If interactive mode is disabled, FTP transfers the specified remote files without any confirmation.

If globbing is enabled (the default), FTP expands the wild card characters before the file transfer is executed. However, if globbing is disabled, FTP tries to transfer the files “as is.” Refer to the `GLOB` command for more information.

If a file specified with `MGET` does not already exist, FTP ignores this file and does not issue an error.

Example

```
ftp> mget test1 test2 test3
mget test1 (Yes, No, Break, Stop Asking)[Yes]? Y
mget test2 (Yes, No, Break, Stop Asking)[Yes]? Y
mget test3 (Yes, No, Break, Stop Asking)[Yes]? Y
ftp>
```

MKDIR

Creates a directory on the remote host.

Syntax

```
MK[DIR] remote_directory [lu]
```

Parameters

remote_directory Specifies a valid directory path on the remote host.

If a directory with the specified name already exists, FTP issues a warning and ignores the command.

For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section.

lu Specifies the LU on which the remote directory will reside.

Discussion

If you specify an LU for a remote host that is not an HP 1000, unpredictable results may occur.

The LU may be included in the directory name.

MLS

Writes an abbreviated directory listing of multiple remote directories or files to a local file.

Syntax

```
ML[S] remote_listing [remote_listing ...] local_file
```

Parameters

remote_listing — Specifies the remote directories or file masks from which a directory listing is to be generated.

The ellipsis (. . .) means that you may specify multiple remote directories or files, delimited by a comma or by one or more blank spaces.

local_file — A valid file path on the local host to store the remote listing. This parameter is required, because MLS always outputs the remote listing to a local file and not to a terminal. FTP always uses the last parameter in the MLS runstring as the *local_file*.

Discussion

FTP prompts you with a confirmation of the *local_file* to be used. Note that if *local_file* already exists, it is overwritten.

If a directory or file specified by *remote_listing* does not exist, FTP ignores the MLS command and does not create the output directory listing.

The following table shows the FTP commands available for listing remote directories and files. See the individual commands for listing a single directory or file (DIR, DL, LS, NLIST) for samples of extended and abbreviated listing formats.

FTP Remote Listing Commands

Listing a Single Directory or File	Listing Multiple Directories or Files
DIR—extended listing	MDIR—extended listing
DL—extended RTE listing	
LS—extended listing	MLS—abbreviated listing
NLIST—abbreviated listing	

MODE

Specifies the file transfer mode.

Syntax

```
MO[DE] mode_name
```

Parameters

mode_name — A valid FTP file transfer mode. The only currently supported mode is `stream`.

Discussion

Stream mode specifies that the data is transmitted as a stream of bytes. There is no restriction on the representation type used. If the structure is a file structure (which is the default), the end-of-file is indicated by the sending host closing the data connection and all bytes are data bytes.

Table 3-4 lists the FTP commands used to define file transfer form, mode, structure, and type.

Table 3-4. FTP File Transfer Form, Mode, Structure, and Type

Command	Supported Parameters	Meaning
FORM	non-print	non-print specifies that no vertical format information is contained in the file.
MODE	stream	Data is transmitted as a stream of bytes.
STRUCT	file	File is considered to be a continuous sequence of data bytes.
TYPE*	A[SCII]	Data is converted to the standard 8-bit ASCII representation for transfer. Also see the ASCII command.
	B[INARY]	Data is sent as it appears on disk. Also see the BINARY command.

* ASCII file transfer type is the default when both systems are not Revision 6.0 (or later) HP 1000s. To specify ASCII or binary, you may also use the ASCII or BINARY command, respectively.

MPUT

Transfers multiple local files to the remote host.

Syntax

```
MP[UT] local_file [local_file ...]
```

Parameters

local_file	Specifies a valid file path on the local host to be transferred to the remote host. The ellipsis (. . .) means that you can specify multiple local files, delimited by a comma or by one or more blank spaces. You may use wild card characters in the file names. The wild card characters are expanded if globbing is enabled (the default). The files are transferred to the remote host, under the same directory and file names as the source files.

Caution If a remote file with the same file name already exists before the file transfer, it is overwritten without warning.

Discussion

If interactive mode is on (the default), FTP prompts you with a confirmation for each specified file before transferring. (See the example below.) The confirmation prompt looks like this:

```
mput filename (Yes, No, Break, Stop Asking)[Yes]?
```

where `Yes` (or carriage return) will proceed with the file transfer; `No` will abort this file transfer and proceed to the next file; `Break` will abort `MPUT` and return to the FTP prompt; and `Stop Asking` will transfer the rest of the files specified by `MPUT` without further confirmation. (See the example below.)

If interactive mode is disabled, FTP simply transfers the specified remote files without a confirmation prompt.

If globbing is enabled (the default), FTP expands the wild card characters before the file transfer is executed. However, if globbing is disabled, FTP tries to transfer the files “as is.” Refer to the `GLOB` command for more information.

If a file specified by `MPUT` does not exist, FTP ignores this file and does not issue an error.

Note MPUT forces all destination file names to be lowercase. This is done to follow the UNIX industry norm of using lowercase for filenames.

Example

```
ftp> mput foo1 FOO2 FOO3
mput foo1 (Yes, No, Break, Stop Asking)[Yes]? Y
mput foo2 (Yes, No, Break, Stop Asking)[Yes]? Y
mput foo3 (Yes, No, Break, Stop Asking)[Yes]? Y
ftp>
```

NLIST

Writes an abbreviated directory listing of a remote directory or file to your terminal or to a local file on the HP 1000.

Syntax

```
N[LIST] [remote_listing] [local_file]
```

Parameters

remote_listing Specifies the remote directory or file mask from which a directory listing is to be generated. If this parameter is omitted, NLIST lists the remote working directory.

local_file Specifies a valid file path on the local HP 1000 host to store the directory listing. If this parameter is omitted, the directory listing is displayed on your terminal.

Discussion

NLIST, without any parameters, lists the current remote working directory to your terminal.

To omit the *remote_listing* parameter but specify the *local_file* parameter, use a comma as a placeholder for the first parameter, such as:

```
NLIST,,local_file
```

This command lists the current remote working directory to the specified local file.

The following table shows the FTP commands available for listing remote directories and files. See the individual commands for listing a single directory or file (DIR, DL, LS, NLIST) for samples of listing formats.

FTP Remote Listing Commands

Listing a Single Directory or File	Listing Multiple Directories or Files
DIR—extended listing	MDIR—extended listing
DL—extended RTE listing	
LS—extended listing	MLS—abbreviated listing
NLIST—abbreviated listing	

Example

```
ftp> NLIST
200 PORT command successful.
150 Opening data connection for /bin/ls (192.6.70.19,32825) (0 bytes)
test1
test2
test3
test4
tracker_form
226 Transfer complete.
49 bytes transferred in 0.22 seconds [ 0.22 kbytes/second ]
```

OPEN

Establishes a connection with a specified remote host.

Syntax

```
O[PEN] host
```

Parameters

host — Specifies the host to which you want to log on. You may use the host's node name or IP address.

The syntax for the host node name is shown here, and is further described under "Node Names" in Section 1 of this manual.

node[.*domain*[.*organization*]]

The syntax for the host IP address is shown here, and is further described under "IP Addresses" in Section 1 of this manual.

nnn.nnn.nnn.nnn

Discussion

The OPEN command connects you to the remote host and then prompts you for a remote login and password if auto-login is enabled (the default). If auto-login is disabled (when input is from a transfer file), you must use the USER command to log in to a remote host. See the USER command for more details.

There are two ways to establish connection to a remote host. You may

- Use the OPEN command at the FTP prompt, as described here.
- Specify the remote host when you invoke FTP. See "Invoking FTP" earlier in this section.

FTP can only have one connection open at a time. If you issue an OPEN command when a connection to a remote host already exists, FTP issues a warning and ignores the command.

Note *Logging in to an HP 9000*: For security reasons, you can only log in to accounts that have passwords associated with them. You must be able to supply FTP with a valid login name and password on the remote host.

PROMPT

Toggles interactive prompting for multiple file operations.

Syntax

```
PR[OMPT]
```

Discussion

Interactive prompting occurs during multiple file operations to allow you to selectively proceed with each file. By default, interactive mode is enabled.

Interactive prompting is also used by the `MDIR` and `MLS` command to confirm the use of the local file as an output file.

Interactive mode may be disabled in one of two ways:

- With the `PROMPT` command described here.
- With the `-i` option when you invoke FTP. See "Invoking FTP" earlier in this section.

If interactive mode is disabled, FTP simply performs multiple file operations without any confirmation. If interactive mode is disabled, you must use the `PROMPT` command to enable it.

Example

The following example shows deleting multiple files with interactive mode on and then off.

```
ftp> mdelete prog1 prog2 prog3                (interactive mode on)

mdelete prog1 (Yes, No, Break, Stop Asking)[Yes]? Y

mdelete prog2 (Yes, No, Break, Stop Asking)[Yes]? N

mdelete prog3 (Yes, No, Break, Stop Asking)[Yes]? Y

ftp> PROMPT                                   (interactive mode off)
Interactive prompting disabled.

ftp> mdelete test1 test2 test3

ftp>
```

PUT

Transfers a local file to the remote host. Same as `SEND`.

Syntax

```
PU[T] local_file [remote_file]
```

Parameters

local_file Specifies a valid file path on the local host to be transferred.

remote_file Specifies a valid file path on the remote host to be transferred into. If this parameter is omitted, FTP uses the local file path as the file name on the remote host.

If a remote file name is specified without a directory, the current working directory on the remote host is used.

Caution If a remote file with the same file name already exists, it is overwritten without warning.

Discussion

Refer to `MPUT`, earlier in this section, for transferring multiple files.

For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section.

Example

The first `PUT` command transfers file `/user/example/test` to file `foo` on the working directory of the remote host.

The second example transfers file `/user/example/test` to `/user/example/test` on the remote host. Note that the directory path `/user/example/` must exist on the remote host, or FTP will generate an error and not execute the `PUT` command. This is because no target file path was specified, so FTP uses the source file path as the target file path.

```
ftp> PUT /user/example/test foo
ftp> PUT /user/example/test
```

Writes the name of the remote working directory to the terminal.

Syntax

```
PW[D]
```

Discussion

To change the remote working directory, use the CD command.

Example

```
ftp> PWD
257 "/DWIGHT/EXAMPLES" is the current working directory.

ftp>
```

QUIT

Closes the remote connection and exits from FTP. Same as BYE and EXIT.

Syntax

```
QUI[T]
```

Discussion

The FTP QUIT command closes the connection to the remote host, logs off the remote session, terminates FTP, and returns you to the operating system of the local node. The QUIT command is identical to the BYE and EXIT commands.

If you want to close the remote connection but remain in FTP, use the CLOSE command. See the CLOSE command for more information.

QUOTE

Sends arbitrary server commands to the remote host.

Syntax

```
QUO[TE] arguments
```

Parameters

arguments — Specifies a valid FTP server command to be sent to the remote host. The *arguments* are sent "as is," including commas if included.

Discussion

The QUOTE command is used to send an FTP server command to the remote host. This is sometimes useful for debugging when you need to activate an FTP server command out of the usual command sequence. To generate a list of server commands, use the REMOTEHELP command.

Note that when using the QUOTE command, the responses received from the server command might put FTP out of sequence.

Example

FTP's REMOTEHELP command is used to list the FTP server commands, then the QUOTE command is used to send the server commands, XPWD and CDUP.

```
ftp> remotehelp
214- The following commands are recognized (* =>'s unimplemented).
  USER    PORT    STOR    MSAM*   RNTO    NLST    MKD    CDUP
  PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD   XCUP
  ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD    STOU
  SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD   SIZE
  REIN*   MODE    MSND*   REST    XCWD    HELP    PWD    MDTM
  QUIT    RETR    MSOM*   RNFR    LIST    NOOP    XPWD

ftp> quote xpwd
257 "/DWIGHT/EXAMPLES" is the current working directory.

ftp> quote cdup
200 CWD command successful.

ftp> quote xpwd
257 "/DWIGHT" is the current working directory
```

RECV

Transfers a remote file to the local host. Same as `GET`.

Syntax

```
REC[V] remote_file [local_file]
```

Parameters

remote_file Specifies a valid file path on the remote host to be transferred to the local host.

local_file Specifies a file on the local host to be copied into.

If this parameter is omitted, the local file will have the same directory path and file name as the remote file.

If a local file name is specified without a directory, the current working directory of the local host is used.

Caution If a local file with the same file name already exists, it is overwritten without warning.

Discussion

For more information about the HP 1000 file name syntax, refer to "RTE-A Files and Directories" earlier in this section.

Example

The first `RECV` command transfers file, `/USER/EXAMPLE/FOO`, on the remote host to file, `TEST1`, on the working directory of the local host.

The second `RECV` command transfers file, `/USER/EXAMPLE/FOO`, on the remote host to file, `/USER/EXAMPLE/FOO`, on the local host. If the directory path `/USER/EXAMPLE` does not exist on the local host, FTP generates an error and does not execute the `RECV` command. This is because the *local_file* parameter was omitted.

```
ftp> RECV /USER/EXAMPLE/FOO  TEST1
ftp> RECV /USER/EXAMPLE/FOO
```

REMOTEHELP

Displays the currently supported FTP server commands on the remote host.

Syntax

```
REM[OTEHELP]  [command]
```

Parameters

command Any FTP server command on the remote host. FTP displays help information on the specified server command from the remote host.

If *command* is omitted, FTP displays a list of currently supported FTP server commands on the remote host.

Example

The following example shows FTP server commands and help information on a command displayed by REMOTEHELP.

```
ftp> REMOTE
214- The following commands are recognized (* =>'s unimplemented).
  USER    PORT    STOR    MSAM*   RNTO    NLST    MKD     CDUP
  PASS    PASV    APPE    MRSQ*   ABOR    SITE    XMKD    XCUP
  ACCT*   TYPE    MLFL*   MRCP*   DELE    SYST    RMD     STOU
  SMNT*   STRU    MAIL*   ALLO    CWD     STAT    XRMD    SIZE
  REIN*   MODE    MSND*   REST    XCWD    HELP    PWD     MDTM
  QUIT    RETR    MSOM*   RNFR    LIST    NOOP    XPWD

ftp> REMOTE APPE
214 SYNTAX: APPE file-name

ftp>
```

RENAME

Renames a remote file or remote directory.

Syntax

```
REN[AME] remote_old remote_new
```

Parameters

remote_old Specifies the original name of a remote file or remote directory to be renamed.

remote_new Specifies the new name for the remote file or remote directory.

If *remote_new* already exists, FTP issues a warning and ignores the command.

RMDIR

Removes an empty directory from the remote host.

Syntax

```
RM[DIR] remote_directory
```

Parameters

remote_directory Specifies a valid directory path on the remote host to be removed. The directory must be empty or FTP issues a warning and ignores the command.

RTEBIN

Sets the transfer type to BINARY such that for subsequent PUT or MPUT commands if only the file name is specified, then the file type, size, and record length are included in the destination file descriptor.

Syntax

```
RT[EBIN]
```

Discussion

The RTEBIN command has two functions:

1. It sets the transfer type to BINARY.
2. It causes FTP to add the file type, size, and record length to the destination file descriptor(s) when the user does a PUT or MPUT. The files thus retain this information in their file names on non-RTE-A systems. RTEBIN is recommended when using PUT or MPUT to a pre-Revision 6.0 RTE 1000 system; this will preserve the file attributes and improve performance.

To disable the RTEBIN command feature, reset the transfer type to ASCII or binary with the ASCII, BINARY, or TYPE commands.

Limitations:

1. The RTEBIN command should not be used if the remote file system does not support the colon (:) character as a legal character in a file name.
2. The RTEBIN command affects only the PUT and MPUT commands by adding the file type, size, and record length to the destination file descriptor, if they are not specified. That is, it will PUT (or MPUT) destination files (with their full file descriptors) to the remote server system but it will *not* PUT (MPUT) files with their full file descriptors if you are performing a file transfer from a non-6.0 system, unless you specifically designate the file descriptors.

Example

```
ftp> RTEBIN
200 Type set to I.

ftp> PUT CI.RUN
```

This will create a file named CI.RUN:::6:528:128 on the remote system.

SEND

Transfers a local file to the remote host. Same as PUT.

Syntax

```
SE[ND] local_file [remote_file]
```

Parameters

local_file	Specifies a valid file path on the local host to be transferred.
remote_file	Specifies a valid file path on the remote host to be transferred into. If this parameter is omitted, FTP uses the local file path as the remote file name. If a remote file name is specified without a directory, the current working directory of the remote host is used.

Caution If a remote file with the same file name already exists, it is overwritten without warning.

Discussion

For more information about the HP 1000 file name syntax, refer to “RTE-A Files and Directories” earlier in this section.

Example

The first SEND command transfers a local file on the working directory, LFILE, to the working directory on the remote host.

The second SEND command transfers file /user/example/test to file foo on the working directory of the remote host.

```
ftp> SEND LFILE
ftp> SEND /user/example/test foo
```

SITE

Sends arguments, verbatim, to the server host as a `SITE` command.

Syntax

```
SI[TE] arguments
```

Parameters

arguments — Specifies a valid FTP server `SITE` command to be sent to the remote host. The *arguments* are sent "as is," including commas if included.

Discussion

The `SITE` command is used to pass commands that request server-specific functions. The user must use a `REMOTEHELP SITE` command to list the functions that the server supports. No user-callable `SITE` functions are currently implemented in the HP 1000 FTP server.

Example

```
ftp>

ftp> open sable
Connecting to ... sable
220 sable FTP server (Version $Revision: 16.2
    Mon Apr 29 20:45:42 GMT 1991) ready.

(username: dwight) [RETURN]
331 Password required for dwight.
(password: dwight) ___
230 User dwight logged in.
Remote system type is UNIX Type: L8.

ftp>

ftp> site umask

200 Current UMASK is 027
```

STATUS

Writes the current status of FTP to the terminal.

Syntax

```
STA[TUS]
```

Example

The following example shows a display after executing the STATUS command.

```
ftp> STATUS
Connected to: sable

Auto Login : ON    Form : NON-PRINT   Mode      : STREAM   Type    : ASCII
Bell       : OFF   Glob : ON          Prompt    : ON       Verbose : ON
Debug      : OFF   Hash : OFF         Structure : FILE

Log file: FTP.LOG
```

STRUCT

Sets the FTP file transfer structure to the specified structure.

Syntax

```
STR[UCT] struct_name
```

Parameters

struct_name Specifies the FTP file transfer structure. Currently, the only supported file transfer structure is `file`.

Discussion

File structure means that there is no internal structure and the file is considered to be a continuous sequence of data bytes.

Table 3-5 lists the FTP commands used to define file transfer form, mode, structure, and type.

Table 3-5. FTP File Transfer Form, Mode, Structure, and Type

Command	Supported Parameters	Meaning
FORM	non-print	non-print specifies that no vertical format information is contained in the file.
MODE	stream	Data is transmitted as a stream of bytes.
STRUCT	file	File is considered to be a continuous sequence of data bytes.
TYPE*	A[SCII]	Data is converted to the standard 8-bit ASCII representation for transfer. Also see the ASCII command.
	B[INARY]	Data is sent as it appears on disk. Also see the BINARY command.

* ASCII file transfer type is the default when both systems are not Revision 6.0 (or later) HP 1000s. To specify ASCII or binary, you may also use the ASCII or BINARY command, respectively.

SYSTEM

Returns the type of operating system running on the server.

Syntax

```
SY[STEM]
```

Example

The following example shows a `SYSTEM` command sent to an RTE-A system:

```
ftp> SYSTEM
215 RTE-A
ftp>
```

The following example shows a `SYSTEM` command sent to an HP-UX system:

```
ftp> SYSTEM
215 UNIX Type: L8
ftp>
```

If the server does not support the `SYSTEM` command, the following message is displayed:

```
502 Command not implemented.
```

TR

Specifies a local command input file (also called a transfer file) containing FTP commands.

Syntax

```
TR local_file
```

Parameters

local_file — Specifies a local transfer file containing FTP commands. FTP executes the commands in this file. The `TR` command lets you execute FTP from a command file rather than entering each FTP command via your terminal keyboard.

You may include any valid FTP commands in the transfer file.

FTP terminates when the `EXIT`, `BYE`, or `QUIT` command is executed. If the end-of-file is found before any of these commands, control is passed back to FTP.

Discussion

When you run FTP using a transfer file, FTP attempts to open the `FTP.LOG` log file if a log file is not already open. FTP log files are opened with the `LL` command or the `-l` option in the FTP runstring. FTP logs messages and other information from this FTP session into the log file. See the `LL` command for more information on FTP log files.

You may specify an FTP transfer file in two ways:

- With the `TR` command as described here.
- With the `-t` option when you invoke FTP. See “Invoking FTP” earlier in this section.

You may include a `TR` command within a transfer file; that is, you may have chained `TR` transfer files. However, FTP does not provide a way to return to the original transfer file at the point of entry; that is, FTP always reads a transfer file sequentially from the first line of the file. Care should be taken so that you do not end up in an endless loop when using multiple transfer files. For example, if two transfer files (tr1 and tr2) are used, commands in tr1 are executed first and then control is passed from tr1 to tr2 with the exection of the command “tr tr2” within the file tr1. If control is passed back to tr1 from within tr2, the first line of tr1 will be the next one executed. This will result in an endless loop and any commands that come after the “tr tr2” command in tr1 will not be executed.

If you are using a transfer file to log on to a remote host, you must explicitly use the `USER` command to log on to the remote host. This is necessary because auto-login is disabled when input is from a transfer file.

In a transfer file, FTP treats any text on a line following an asterisk (`*`) as a comment.

Example

The following is a simple transfer file to log in to a remote host, transfer two files between the remote and local hosts, then log out.

```
******************************************************
*             ***** SAMPLE FTP TRANSFER FILE *****
******************************************************
*
* Enable Verbose Mode
*
VERBOSE
*
******************************************************
*
* Send output to NORA.LOG
*
LL NORA.LOG
*
******************************************************
* Log into remote host SABLE, account DWIGHT, password MAC
*
OPEN SABLE
USER DWIGHT MAC
*
******************************************************
* Connect to directory /EXAMPLES/SOURCES
*
CD /EXAMPLES/SOURCES
*
******************************************************
* Get a file from remote host to local host
*
GET TIMER.PAS /TEST/NEW_TIMER.PAS
*
******************************************************
* Put a file from local host to remote host
*
PUT PERSONAL/STATUS_REPORT /DWIGHT/THIS_MONTHS_STATUS
*
******************************************************
* Generate a directory listing and display FTP status
*
DIR
STATUS
*
******************************************************
* USE EXIT, BYE, OR QUIT TO RETURN TO CI.
* USE CLOSE OR NOTHING TO RETURN TO FTP
*
EXIT
*******************************************************
```

TYPE

Sets the FTP file transfer type to the specified type.

Syntax

```
TY[PE] [type_name]
```

Parameters

type_name — Specifies the FTP file transfer type. Currently, the only supported FTP file transfer types are ASCII or A for ASCII transfer type and BINARY, B, or I (for Image) for binary transfer type. If this parameter is omitted, TYPE displays the current FTP file transfer type on the terminal.

The default file transfer type is ASCII when both systems are not Revision 6.0 (or later) HP 1000s.

Discussion

You may also use the ASCII or BINARY command to specify ASCII or binary file transfer type, respectively. Refer to these commands for more details.

Table 3-6 lists the FTP commands used to define file transfer form, mode, structure, and type.

Table 3-6. FTP File Transfer Form, Mode, Structure, and Type

Command	Supported Parameters	Meaning
FORM	non-print	non-print specifies that no vertical format information is contained in the file.
MODE	stream	Data is transmitted as a stream of bytes.
STRUCT	file	File is considered to be a continuous sequence of data bytes.
TYPE*	A[SCII]	Data is converted to the standard 8-bit ASCII representation for transfer. Also see the ASCII command.
	B[INARY]	Data is sent as it appears on disk. Also see the BINARY command.

* ASCII file transfer type is the default when both systems are not Revision 6.0 (or later) HP 1000s. To specify ASCII or binary, you may also use the ASCII or BINARY command, respectively.

USER

Logs on as a different user on the currently connected remote host.

Syntax

```
U[SER]  [user_name]  [password]
```

Parameters

user_name	Specifies the account on the remote host to log on. If the *user_name* is not specified, FTP prompts you for it.
password	Specifies the password for the account, if required. If a password is required and not specified, FTP prompts you for it.

Discussion

If you are using a transfer file to log on to a remote host, you must use the USER command. See the TR command for details on FTP transfer files.

To log on to a remote host with the USER command, you must already have an open connection to the remote host. There are two ways to establish a connection to the remote host:

- With the OPEN command. See OPEN for details.
- With the *host* parameter when you invoke FTP. See "Invoking FTP" earlier in this section.

You do not need to use the USER command if auto-login is enabled and input is interactive (from the keyboard). With auto-login, FTP automatically prompts you for the user name and password after you open a connection to the remote host. By default, auto-login is enabled. (See "Invoking FTP" for details on auto-login.)

FTP accepts only one login per connection. If you are currently logged into an account and you issue a USER command to connect to another HP 1000 host, FTP logs out of the previous account before attempting to log in to the new account. If you want to connect to another host that is not an HP 1000, you must close the connection before issuing the USER command; otherwise you will receive an error with the USER command.

Note *Logging in to an HP 9000 remote host*: For security reasons, you can only log in to accounts that have passwords associated with them. You must be able to supply FTP with a valid login and password on the remote host.

VERBOSE

Specifies verbose output. This command toggles verbose output.

Syntax

```
V[ERBOSE]
```

Discussion

Verbose output displays all responses from any remote host to which you are connected. These responses tell you whether or not FTP commands completed successfully.

By default, verbose output is enabled if FTP input comes from your keyboard. Verbose output is disabled if FTP input comes from an FTP transfer file. See the `TR` command for details on FTP transfer files.

When verbose output is disabled, FTP performs the command issued and simply redisplays the FTP prompt.

Verbose output may be enabled in one of two ways:

- With the `VERBOSE` command described here.
- With the `-v` option when you invoke FTP. See "Invoking FTP" earlier in this section.

Example

The following example shows a file transfer with verbose output, then another file transfer without verbose output. Note the numbers `200`, `150`, and `226` are FTP command reply codes.

```
ftp> GET REMOTEDATA                                     (verbose output on)

200 PORT command successful.

150 Opening data connection for REMOTEDATA ...

226 Transfer complete.

5123 bytes received in 5 seconds ...

ftp> VERBOSE                                            (verbose output off)

ftp> GET REMOTEDATA2

ftp>
```

4

Network File Transfer

Overview

Network File Transfer (NFT) is an NS Common Service that enables you to copy files between NS systems in your network. Files can be copied interactively using the program DSCOPY, or from within a program by invoking the `Dscopy` call.

NFT includes features that allow you to:

- *Copy files between other non-HP 1000 systems*. NS is implemented on the HP 1000, HP 3000, HP 9000, and PC computers. NFT enables you to copy files between NS systems in your network. Refer to the *NS Cross-System NFT Reference Manual*, part number 5958-8563 for information on using NFT among these systems.
- *Copy remote files*. Using NFT at your local system, you can copy files from your system to a remote node, from a remote node to your system, and between remote systems. No user intervention at the remote system or systems is required.
- *Translate file attributes*. Translation of file attributes is performed transparently when files are copied between different types of systems. You can also explicitly convert file attributes.
- *Copy groups of files*. Directories and groups of files can be copied between NS-ARPA/1000 systems with a single command.
- *Access remote accounts*. Files under any account can be accessed if you provide the correct logon and password.
- *Copy all types of HP 1000 files*. You can copy FMGR files as well as files in the hierarchical file system. Sparse files (files that are missing intermediate records) can also be copied.

Note NFT cannot be used on HP 1000 computers to copy files to or from non-disk devices.

Three-Node Model

NFT utilizes a *three-node model* to copy files between systems. Under the three-node model there are three logical participants in the file copy process:

- The *Initiator*. Located on the system where the copy request originates, the Initiator receives the user request and *initiates* the copy process.
- The *Producer*. Located on the same node as the source file, the Producer accesses that file and *produces* the data that is to be copied.
- The *Consumer*. Located on the same node as the target file, the Consumer *consumes* the data and writes it into the target file.

All three participants are logically distinct. They may be three separate processes on three separate nodes, or any two, or all three, may reside on the same node. This is because the copy request does not have to originate from either the source or the target node.

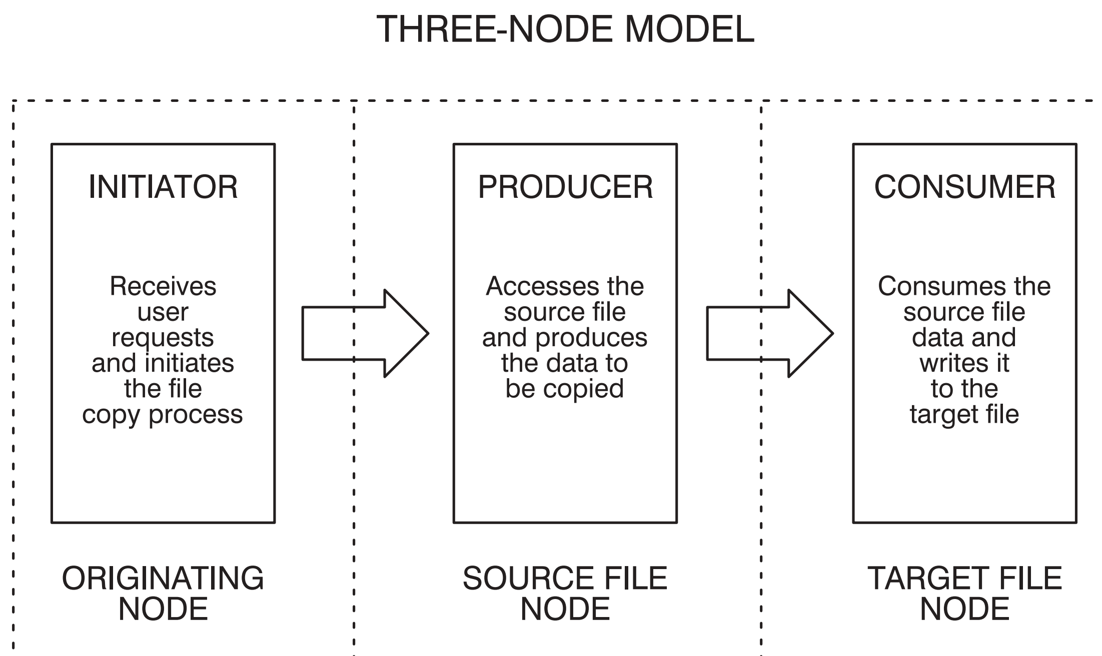


Figure 4-1. Three-Node Model

File Copying Formats

NFT uses two file copying formats: *Transparent Format* and *Interchange Format*.

Transparent Format

Transparent Format is invoked by default when files are copied between NS-ARPA/1000 systems. Transparent Format does not alter a file's attributes, but simply copies the file. It should be used when you want a low-overhead, maximum-speed file copy process between systems.

Interchange Format

When the source and the target system are different types of computers (for example, one is an NS/9000 node and one is an NS-ARPA/1000 node), files copied from one to the other must be converted to *Interchange Format*. Interchange Format consists of a set of attributes that describe a file in a standard way so that it can be understood by any NS system.

Interchange Format is invoked by default whenever you use NFT to copy a file residing on one type of system to another. You can also invoke Interchange Format explicitly by specifying one or more Interchange Format options when copying a file. These options specify how the source file will be read and how it will be stored in the target file.

When a file is copied using Interchange Format, it is translated into Interchange Format at the source system before being copied to the target system. At the target system, the file is mapped from Interchange Format into the target system's file format. Interchange Format's standard file attributes enable the target computer to map the source file into a target file that has attributes that match the source file's as closely as possible.

Figure 4-2 is a conceptual view of Interchange Format.

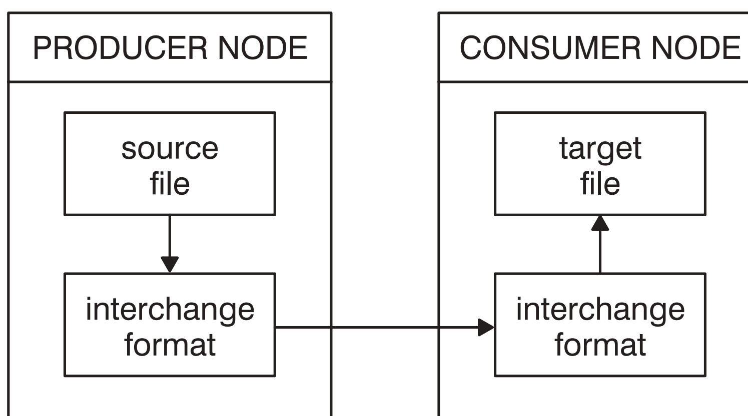


Figure 4-2. Interchange Format

RTE-A Type 6 files cannot be moved in Interchange Format.

Refer to the *NS Cross-System NFT Reference Manual* for information on copying files from one type of computer to another.

Data Interpretation

Although the purpose of Interchange Format is to create an accessible target file on a system of a different type, it does not ensure that the target file will be usable. This is because Interchange Format changes a file's attributes only; it does *not* perform data interpretation. Interchange Format can create an unusable target file if the target system has a different representation for the data present in the source file.

For example, if a file that contains floating point numbers is copied to a different type of computer, there is no guarantee that the target system will be able to read the data as floating point. Consequently, the usability of your target files must be determined by the applications that use them.

Note Files copied to or from an HP 1000 in Interchange Format cannot have records longer than 4400 bytes. Records can be truncated using the `RSIZE` option. (The `RSIZE` option is described in the "Copy Descriptor" and "DSCOPYBUILD" subsections later in this section.)

Interactive Network File Transfer

You can use NFT interactively by running the program DSCOPY.

Syntax

```
DSCOPY [ ,copydescriptor  ]
       [ ,dscopycommand   ]
```

Parameters

copydescriptor A copy descriptor. May be a maximum of 256 characters long. The syntax of *copydescriptor* is provided on the following page.

dscopycommand A DSCOPY command. The DSCOPY commands are described later in this section.

Discussion

DSCOPY can be scheduled with a DSCOPY command, a copy descriptor, or without parameters. (The DSCOPY commands and copy descriptor are described in the following pages.) DSCOPY enters interactive mode, or terminates, depending on how it is scheduled.

When scheduled with a DSCOPY command (other than an +EX to exit), DSCOPY executes the command and then enters interactive mode and displays the DSCOP> prompt. For example

```
ru,dscopy,+echo
DSCOP>
```

When scheduled with a copy descriptor, DSCOPY executes the request and then terminates. For example

```
ru,dscopy,memo.txt  to  memo.txt>cricket.ind.hp
CI>
```

When scheduled without parameters, DSCOPY enters interactive mode and prompts for commands until it is terminated. Termination will occur if the the +EX command is entered to exit. For example

```
ru,dscopy
DSCOP>
```

If DSCOPY discovers an error while executing a command or copy descriptor it will print an error message to the list file or device. Error codes, and the total number of errors that occurred, can also be retrieved after DSCOPY terminates by examining the contents of the P-globals (FMGR), or the return variables (CI). If FMGR is the command interpreter, the first P-global, 1P, will contain the total number of errors that occurred, if any, and 2P will contain the most recent error code. If CI is the command interpreter, $RETURN1 will contain the total number of errors that occurred, if any, and $RETURN2 will contain the most recent error code.

For a complete listing of the DSCOPY error messages, refer to “DSCOPY Error Messages” in the *NS-ARPA/1000 Error Message and Recovery Manual*, part number 91790-90045.

Copy Descriptor

The copy descriptor allows you to specify the files or directories you wish to copy.

Syntax

sfile[[*slogon*]][>*snode*] {ΔTOΔ | ,} *tfile*[[*tlogon*]][>*tnode*][,*option*][,*option*]...]

Parameters

sfile — The source file; the name of the file to be copied. Refer to "File Masks" later in this section for an explanation of how HP 1000 file masks may be used in this parameter. (The *NS Cross-System NFT Reference Manual* explains file name syntax at other NS systems.)

[*slogon*] — The logon and password, if any, at the node where the source file resides. Must be enclosed in brackets ([ ]). This parameter is *required* if the source node is a remote multiuser HP 1000. NS-ARPA/1000 is not supported on a single user HP 1000 computer. (The *NS Cross-System NFT Reference Manual* explains logon and password syntax at other NS systems.)

Default: If `slogon` is omitted and the source node is the local node, the account under which DSCOPY was scheduled is used.

>*snode* — The name of the source node. Must be preceded by ">." The syntax of NS node names is provided in Section 1, "Introduction," of this manual.

Default: You may omit the organization, organization and domain, or all parts of the node name. If the organization, or organization and domain, are omitted, the local organization and/or domain will be used. If the entire node name is omitted, it will default to the local node.

tfile — The target file; the name the source file will acquire at the target node. Refer to "File Masks" later in this section for an explanation of how HP1000 file masks may be used in this parameter. (The *NS Cross-System NFT Reference Manual* explains file name syntax at other NS systems.)

[*tlogon*] — The logon and password, if any, at the target node. Must be enclosed in brackets ([ ]). This parameter is *required* if the target node is a remote multiuser HP 1000. (The *NS Cross-System NFT Reference Manual* explains logon and password syntax at other NS systems.)

Default: If `tlogon` is omitted and the target node is the local node, the account under which DSCOPY was scheduled is used.

>*tnode* — The name of the target node. Must be preceded by ">." The syntax of NS node names is provided in Section 1, "Introduction," of this manual.

Default: You may omit the organization, organization and domain, or all parts of the node name. If the organization, or organization and domain,

are omitted, the local organization and/or domain will be used. If the entire node name is omitted, it will default to the local node.

option — May be one or more of the options described below; there is no limit to the number of options you can specify. Each option must be separated by a comma, semicolon or space, but different delimiters cannot be used in the same copy descriptor. If conflicting options are given (for example, `ASCII` and `BINARY`), DSCOPY will issue a warning and the *last option given will take precedence*.

Default: DSCOPY will use Transparent Format. Interchange Format is used if the `ASCII`, `BINARY`, `FIXED`, `FSIZE`, `RSIZE`, `STRIP`, or `VARIABLE` options are specified.

The first eight options described below cause Interchange Format to be used. RTE-A Type 6 files cannot be moved in Interchange Format.

The following explanations describe how the Interchange Format options operate when the source and target nodes are both NS-ARPA/1000 systems. Refer to the *NS Cross-System NFT Reference Manual* for information on the operation of these options in regard to other NS systems.

`AS[CII]` — Specifies that records contain printable ASCII characters and that spaces should be used as padding when creating fixed length records. This option may be used in conjunction with the `STRIP` option to indicate that spaces should be stripped from the ends of records.

Default: If the source file is ASCII, the target file will be ASCII.

`BI[NARY]` — Specifies that records contain binary information and that null characters (numeric zeros) should be used as padding when creating fixed length records. This option may be used in conjunction with the `STRIP` option to indicate that nulls should be stripped from the ends of records.

Default: If the source file is binary, the target file will be binary.

`FI[XED]` — Specifies that source file records should be formed into fixed length records. (Record size can be specified using the `RSIZE` option and the type of padding used can be specified using the `ASCII` or `BINARY` options.)

Default: For HP 1000 type 1 and 2 source files, the target file will have fixed length records. For other types of HP 1000 files, the target file will have variable length records.

FS[IZE]= *filesize*	Specifies how much space (*filesize*) to allocate for the target file. If the target file has fixed length records, *filesize* is in records. If the target file has variable length records, *filesize* is the number of maximum size records. This option can be used instead of the HP 1000 file descriptor size parameter to specify the size of an HP 1000 target file. *Default*: The target file will be the same size as the source file.
IN[TERCHANGE]	Overrides the default copy format and causes the file or files to be copied using Interchange Format. RTE-A Type 6 files cannot be moved in Interchange Format. *Default*: DSCOPY will use Transparent Format. Interchange Format is also used if the ASCII, BINARY, FIXED, FSIZE, RSIZE, STRIP, or VARIABLE options are specified.
RS[IZE]= *recordsize*	Specifies the record size (*recordsize*) in bytes. If fixed length records are being produced, *recordsize* is the size of each record. If variable length records are being produced, *recordsize* limits the size of the largest record and records may be padded or truncated. You cannot copy files with records longer than 4400 bytes to or from an HP 1000. *Default*: The target file will have the same record size as the source file.
ST[RIP]	Strips any record padding from the ends of records. You can use this option to create variable length records from fixed length records. (Also see the VARIABLE option.) The type of padding to strip is based on the type of the source file. For HP 1000 type 4 files, spaces are stripped. In other HP 1000 file types, null characters are stripped. You can use this option in conjunction with RSIZE to truncate records. Records will be truncated before padding is stripped. *Default*: Padding is not stripped.
VA[RIABLE]	Specifies that source file records should be formed into variable length records. The maximum size of a variable length record may be given using the RSIZE option. *Default*: For HP 1000 type 1 or 2 files, the target file will have fixed length records. For all other HP 1000 file types, the target file will have variable length records.

The next four options do not invoke either Interchange Format or Transparent Format. They can be used when a file is copied in either format and do not affect the attributes of the target file. They can also be used in conjunction with Interchange Format options.

`MO[VE]`
: Purges the source file after it has been successfully copied to the target system. DSCOPY will issue a warning if the file cannot be purged. You must have proper access rights, including any security code, to purge the file. If a directory is copied, the files within the directory and any subdirectories will be purged, but the directory and subdirectories will not be purged.

  Default: The source file is not purged.

`OV[ER]`
: Causes a copy of the source file to overwrite an existing target file, beginning with the first record. If the source file is larger than the existing target file, NFT will copy as much of the source file as will fit in the existing file space and will then return an error. If the source file is smaller than the target file, then the contents of the existing file that extend beyond the end of the copied source file will remain in the target file. If the target file does not exist, a new file will be created. The attributes of the source and target files must match, or NFT will return an error message.

  If you do not specify this option, existing target files will not be overwritten.

`QU[IET]`
: Suppresses the printing of warnings and file names to the list file or device. Error messages cannot be suppressed.

  Default: Warnings, file names, and error messages are printed to the list file.

`RE[PLACE]`
: If the target file exists, this option causes it to be purged and a new file created by the same name. The original file is purged only after the new file is copied successfully to the target system in a scratch file.

  Default: The target file is not replaced and an error message is returned if it already exists.

`SI[LENT]`
: Suppresses the printing of warnings, file names, and error messages to the list file or device. Same as the `QUIET` option, except that error messages are also suppressed. Recommended for programmatic use.

  Default: Warnings, file names, and error messages are printed to the list file.

Using DSCOPY

The following rules should be observed when you use DSCOPY. Some have been discussed earlier but are repeated here so that they may be easily referenced.

- *Copy Descriptor Syntax*. A space must precede and follow the "`TO`" if it is used to separate source and target specifications in a copy descriptor. If a comma is used, no spaces are necessary.

- *Copy Descriptor Length*. A copy descriptor may be a maximum of 256 characters long. If your copy descriptor exceeds this character limit, you may default portions of it using the `+DEFAULT` command. (The `+DEFAULT` command is described later in this section.)

- *Case Sensitivity*. Copy descriptor file names and logon strings are upshifted by CI and FMGR if they are typed as part of the DSCOPY runstring; they are *not* upshifted when typed in response to a DSCOPY prompt or when read from a command file via the DSCOPY command `+TR`. The grave accent (`) may be used to prevent CI from casefolding.

- *Logons and Multiuser*. The copy descriptor parameters, *`slogon`* and *`tlogon`* parameters are required with HP 1000 computers. NS-ARPA/1000 also requires that the HP 1000 computer must be a multiuser HP 1000 computer. "Multiuser" is part of the 92078A Virtual Code Package known as VC+. For information about logons and passwords at other NS systems, refer to the *NS Cross-System NFT Reference Manual*.

- *Line Continuation*. Only one copy descriptor may be issued per line. If a single copy descriptor exceeds one line, you can append a continuation character ("`&`") to the descriptor. If spaces are placed between the copy descriptor and the continuation character they are considered part of the copy descriptor. When DSCOPY encounters the continuation character it will prompt you for the remainder of the copy descriptor. If you wish to modify a copy descriptor that ended with a continuation character you can type `CONTROL` `Y` and a carriage return. DSCOPY will then flush the copy descriptor. If you end a copy descriptor with a continuation character and then discover the copy descriptor is complete, a carriage return entered at the continuation prompt will cause the copy descriptor to be executed.

- *RTE File Names and Logons*. Because RTE file names can consist of the control characters used in the copy descriptor ("`[`", "`+`" and "`>`"), some file names may be misinterpreted by DSCOPY. To ensure that DSCOPY correctly interprets file names that contain these characters, you can use single quotations marks as shown in Table 4-1.

- *Protection Mode and Update Time*. Files copied from one HP 1000 to another using Transparent Format retain their protection mode and update time at the target system. (Protection modes and update time are explained in the *RTE-A User's Manual*.)

- *DS/1000-IV Files*. Files can be copied from a DS/1000-IV node to an NS-ARPA/1000 node and then onto another NS-ARPA/1000 node. The DS/1000-IV file descriptor must be enclosed in single quotes. Also, the DS/1000-IV node *must* be connected to an NS-ARPA/1000 node on a LAN network.

Table 4-1. RTE File Names and Logons

File Name/Logon Typed	File Name/Logon as Interpreted by DSCOPY
`'['Filename[JON/99]`	File: `[Filename` Logon: `JON/99`
`'+Filename>'[JON.GROUP]`	File: `+Filename>` Logon: `JON.GROUP`
`''Filename`	`'Filename`

Note Unlike CI, DSCOPY uses single quotation marks to quote control characters. CI uses grave accents.

HP 1000 File Names and Logons

The logon and file name specifications used at NS-ARPA/1000 nodes are identical to those you usually use at your HP 1000. NFT uses the following defaults in the source and target file specifications for HP 1000 nodes:

- If a subdirectory is included in the file specification and the directory is omitted, the default logon working directory for the logon specified is searched for the subdirectory.
- If only a file name is specified (i.e., no directory or subdirectory is specified), the default logon working directory for the logon specified is used.

HP 1000 File Masks

By using file masks in the source file name and target file name parameters of your copy descriptors, you can:

- *Copy groups of files*. A single copy descriptor can copy multiple files if a file mask is used for the source file name.
- *Create target file names*. You can create target file names from source file names by substituting a file mask for part, or all, of the target file name. A file mask can be used for both the file name and the file's type extension in the target file name. If an entire directory of files is copied, a target file mask can be used to cause the directory, subdirectories and files to retain their original names at the target node. In the following example, the runstring creates a target file named `wilma.pas` at the HP 1000 target node:

```
ru,dscopy,fred.pas,wilma.@
```

All of the file mask features defined by the RTE-A file system can be used as source file masks in the *sfile* parameter. DSCOPY appends the "d" qualifier to the source file name whenever a file is copied using a wildcard mask in the source file parameter. (If any directory matches the mask, the "d" qualifier causes all files in that directory to be copied. This can be overridden with the "n" mask qualifier. The "d" and "n" qualifiers are explained in detail in the *RTE-A User's Manual*.) If a source file is masked to copy a directory, the files within the directory will retain their hierarchical structure at the target node.

Only the wildcard character "at" (@) can be used in the *tfile* parameter to form target file names; you cannot use any of the other RTE-A file mask features. DSCOPY uses this wildcard character as a target file mask in the same way as the Command Interpreter unless the source file is sparse. If the source file is sparse, DSCOPY will ignore the file type, size and record length parts of the file mask. This is done to preserve the integrity of the file at the target node.

For an extensive discussion of HP 1000 source and target file masks, refer to the *RTE-A User's Manual*.

Interrupting the Copy Process

You can interrupt the copy process by entering breakmode and setting DSCOPY's break bit while DSCOPY is executing. When DSCOPY senses that its break bit has been set, it will prompt you to choose a function to be performed. If you do not want to execute one of the functions offered, you can exit breakmode by typing a carriage return and DSCOPY will resume execution.

In the following example, a key is struck, breakmode is entered and DSCOPY prompts for a function. You may type A to Abort, C to Cancel, S for Status information, or H for Help.

```
CM> br,dscopy

Dscopy: Abort, Cancel, Status, Help (CR to continue)__
```

The following is an explanation of each DSCOPY breakmode command:

- *Abort*. Terminates DSCOPY and saves the portion of the target file that has been created thus far. You can also use the A command to exit an active transfer file and return control to the scheduling terminal. Although you can abort a file copy at any time, the target file may be in an inconsistent state if aborted prematurely.

- *Cancel*. Terminates DSCOPY and purges the target file. You can also use the C command to exit an active transfer file and return control to the scheduling terminal. You can cancel a file copy at any time.

- *Status*. Indicates the percentage of the file that has been transmitted to the target node; not all of this data may actually have been *received* at the target. This number is not exact and should be considered an estimate.

- *Help*. Provides an explanation of the A, C and S commands.

Examples

The following examples show files being copied with DSCOPY in both Transparent and Interchange Formats.

Transparent Format

In this example, a global directory called games and all subdirectories and files are copied to a subdirectory on a remote NS-ARPA/1000 node. The directory games will first be created within joesfiles if it does not already exist. The target file mask used after joesfiles will cause games, and all the subdirectories and files in games, to retain their original names. The MOVE option is specified so that the source files are purged after they are successfully copied to the remote node. The QUIET option is specified to suppress all output, except error messages, to the list file or device. (Note the space before and after the TO, the lack of a space before the continuation character, and the "Continue: " prompt.)

```
ru,dscopy,/games.dir to /joesfiles/@[joe]&
Continue: >cricket.ind.hp,move,quiet
```

Interchange Format

In this example, a file called ipc1.lst is copied to a remote NS-ARPA/1000 node called mantis.ind.hp. The RSIZE option is used to create 10 byte records in the target file. If the source file contains records that are longer than 10 bytes, they will be truncated and DSCOPY will issue a warning message.

```
ru,dscopy,ipc1.lst to /listfiles/@[liz]>mantis.ind.hp,rsize=10
```

Optimizing Performance

DSCOPY copies files across data communications connections that it establishes after receiving the copy descriptor. After a file is copied, DSCOPY maintains these connections so that the next copy descriptor issued can make use of them. If the next copy descriptor cannot utilize the same connections, the connections are dismantled.

Because setting up connections can be time consuming, you should group copy descriptors that can utilize the same connections together whenever possible. For example, if the target logon or target node name changes from one copy descriptor to the next, the connection between the source and target computers is dismantled but the connection between DSCOPY and the computer where the source file resides is maintained. If the source logon or source node name changes, however, all connections are dismantled.

In the following examples, files are copied from an HP 1000 called mantis.ind.hp to an HP 1000 called cricket.ind.hp. The user initiates NFT from another node, an HP 1000 called butterfly.ind.hp. (Thus, the *initiator* is located at butterfly.ind.hp, the *producer* is located at mantis.ind.hp, and the *consumer* is located at cricket.ind.hp.) Although a different logon is used in the second copy descriptor, the connection between

DSCOPY at the user's node and `mantis.ind.hp`, the computer where the source files exist, is maintained.

```
ru,dscopy
dscopy> /meetings/minutes.txt[liz]>mantis.ind.hp to memos/minutes.txt&
Continue: [liz]>cricket.ind.hp
DSCOP> /programs/proga.ftn[liz] to progs/proga.ftn&
Continue: [jacquie]>cricket.ind.hp
```

NFT and DS/1000-IV Files

NFT is also supported between a DS/1000-IV node and NS 3000/V and NS/9000 Computers, provided that the DS/1000-IV node is connected to an NS-ARPA/1000 node on the LAN network (see Figure 4-3 below). In this case, the DS/1000-IV node cannot be the initiator.

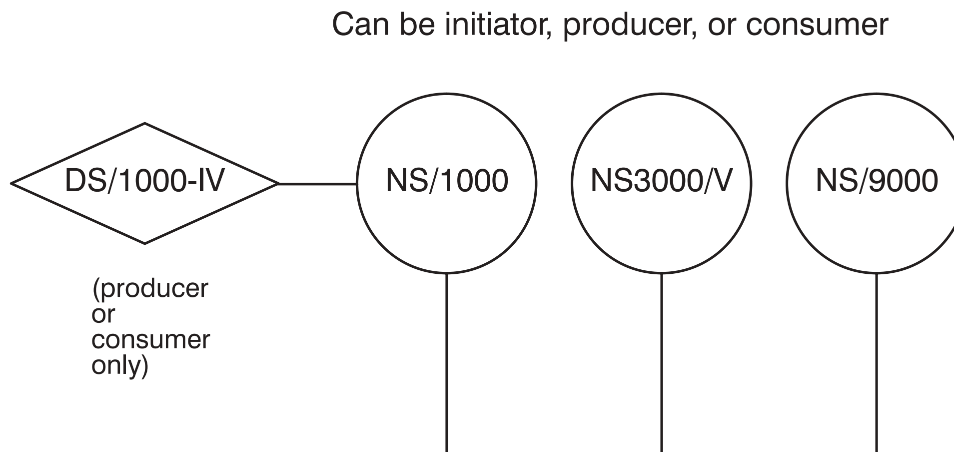


Figure 4-3. NFT and DS/1000-IV

DSCOPY Commands

In addition to the copy descriptor, ten commands can be used with the DSCOPY program:

- *+CLEAR*. Clears all the copy descriptor defaults previously set with the `+DEFAULT` command.
- *+DEFAULT*. Sets defaults for selected portions of subsequently issued copy descriptors.
- *+ECHO*. Causes commands to be echoed, or not echoed, to the list file or device.
- *+EX*. Exits DSCOPY.
- *+LL*. Changes the list file or device.
- *+RU*. Runs a program from within DSCOPY.
- *+SHOW*. Shows all currently active copy descriptor defaults set with the `+DEFAULT` command.
- *+TRANSFER*. Transfers control to a command file or device.
- *+WD*. Displays or changes the current working directory.
- *?*. Requests help information for any DSCOPY command or copy descriptor option. Can also be used to provide a general help summary.

Each command, with the exception of `?`, must begin with a plus ("+") so that DSCOPY can distinguish it from a copy descriptor. Only one command can be issued per line.

+CLEAR

Clears all currently active copy descriptor defaults that have been set with the +DEFAULT command.

Syntax

```
+CL[EAR]
```

Discussion

Refer to the +DEFAULT command for more information.

Sets defaults for portions of subsequently issued copy descriptors.

Syntax

```
+DE[FAULT],copydescriptor
```

Parameters

copydescriptor	A copy descriptor. Refer to the copy descriptor description for the syntax of *copydescriptor*.

Discussion

You can use the +DEFAULT command to set your own defaults for any copy descriptor parameter with the exception of the source and target file names. DSCOPY will use these defaults when a copy descriptor is issued that omits parameters for which defaults were set. If you do not use +DEFAULT, DSCOPY's own defaults are used when parameters are omitted. (Refer to the copy descriptor syntax in this section for an explanation of DSCOPY's defaults.) By defaulting portions of a copy descriptor, you can specify a copy descriptor in excess of the 256 character limit.

If a portion of a subsequently issued copy descriptor conflicts with a default set with +DEFAULT, the copy descriptor settings will take precedence and DSCOPY will issue a warning. Issuing a copy descriptor that conflicts with a default set with the DEFAULT command does *not* change the default setting.

After using +DEFAULT to set a remote logon for the *slogon* or *tlogon* parameters, you may want to use the local logon temporarily. To do this, type a null logon string ("[ ]") in your copy descriptor. This will cause the default logon (the account under which DSCOPY was scheduled) to take precedence over the default set with the +DEFAULT command. DSCOPY will issue a warning.

If the *snode* or *tnode* parameters have been defaulted and you want to use the local node name temporarily, this can be done by specifying the local node name, or a null node name (> only), in the source node and target node parameters of your copy descriptor. DSCOPY will issue a warning.

Examples

The following command sets the defaults for the source logon (donald), the source node (cricket.ind.hp), the target node (mantis.ind.hp), and two options (ASCII and QUIET):

```
DSCOP> +default,[donald]>cricket.ind.hp,>mantis.ind.hp,ascii,quiet
```

You can selectively clear and reset the defaults specified in the previous example by reissuing the +DEFAULT command with new defaults. In the following example, both the source logon and source node defaults (set to [donald] and >cricket.ind.hp in the preceding example) are

cleared by specifying null strings. The ASCII option is changed to BINARY. Because they are not reset, the target node and QUIET defaults are unchanged.

```
DSCOP> +DEFAULT,[]>,,BINARY
```

The +DEFAULT command can save typing if multiple files must be copied between the same two nodes, but the files cannot be selected by using a source file mask. In the following example, three files are copied from a remote HP 1000 node named node1.lab.hp to another remote HP 1000 node named node2.mktg.ind. Defaults are established with the +DEFAULT command for the source and target logons and node names. The "at" (@) wildcard character is used in the *tfile* parameter so that the file names will be retained at the target node.

```
ru,dscopy,+default,[jack]>node1.lab.hp  to  [jill]>node2.mktg.hp

DSCOP> memo.txt to @
DSCOP> helpfile to @
DSCOP> lastfile to @
```

+ECHO

Causes commands to be echoed, or not echoed, to the list file or device.

Syntax

```
+EC[HO] [,ON ]
        [,OFF]
```

Parameters

ON	Causes commands to be echoed to the list file or device. This is the default if +ECHO is issued without a parameter.
OFF	Turns echo off. (DSCOPY does not echo commands to the list file or device by default.)

Discussion

If echo is ON, your commands are echoed to the list file or device. If echo is OFF, commands are not echoed. You may want to turn echo "ON" when you are transferring control to a command file because it allows you to see which command is being executed.

+EX

Exits DSCOPY.

Syntax

```
+EX
```

Discussion

Use the +EX command to terminate DSCOPY.

Changes the list file or device.

Syntax

```
+LL,lfiledev
```

Parameters

lfiledev — The name of a list file or the LU of a device.

Discussion

The +LL command changes the list file or device to that specified in the *lfiledev* parameter. The default list device is the LU number of the scheduling terminal. Warnings and file names are written to the list file or device unless suppressed by the QUIET option in the copy descriptor. Errors are always printed to the list file or device.

+RU

Runs a program from within DSCOPY.

Syntax

```
+RU,progname
```

Parameters

progname The name of the program to be run.

Discussion

If the program is scheduled successfully, DSCOPY will wait until it completes. The DSCOPY +RU command is identical to the Command Interpreter RU command. Refer to the *RTE-A User's Manual* for more information.

+SHOW

Shows all currently active copy descriptor defaults set with the +DEFAULT command.

Syntax

```
+SH[OW]
```

Discussion

You can use this command to confirm that the proper defaults have been set with the +DEFAULT command.

+TRANSFER

Transfers control to a command file or device.

Syntax

```
+TR[ANSFER],cmdfiledev
```

Parameters

cmdfiledev	The name of the command file or the LU of the device that will have control.

Discussion

The +TRANSFER command allows you to transfer control to a command file or device. The commands and copy descriptors that are entered in the command file or at the device should be identical to those you would enter interactively in response to the DSCOP> prompt. When the last command is executed, control is returned to DSCOPY.

Comments can be included in a command file by beginning a line with an asterisk (*). Any command that begins with an asterisk is ignored by DSCOPY.

DSCOPY commands can be nested but they cannot be stacked. You can transfer control to a command file from within another command file, but control will never return to the initial command file.

Examples

The following command transfers control to a command file called commands.cmd.

```
ru,dscopy,+transfer,commands.cmd
```

The contents of file commands.cmd are as follows:

```
memo.txt  to  memo.txt[joe]>cricket.ind.hp
+wd
```

When the commands in commands.cmd are executed, file memo.txt will be copied to a remote node called cricket.ind.hp and the working directory will be displayed. After +WD command is executed, DSCOPY will enter interactive mode.

+WD

Displays or changes the current working directory.

Syntax

```
+WD[,directoryname]
```

Parameters

directoryname The name of the new working directory. May be a subdirectory.

Discussion

If used with a parameter, this command changes the current working directory to the directory specified in *directoryname*. The +WD command will display the current working directory if used without a parameter. This command is identical to the Command Interpreter WD command. Refer to the *RTE-A User's Manual* for more information.

? (HELP)

Requests help information for any command or copy descriptor option.

Syntax

```
?[,commandoption]
```

Parameters

commandoption Any DSCOPY command or copy descriptor option.

Discussion

The ? command will provide information on any command or copy descriptor option. If no parameter is given, ? will provide a general help summary including DSCOPY command and copy descriptor options.

Programmatic Network File Transfer

Two calls are provided to copy files programmatically: `DscopyBuild` and `Dscopy`. The `DscopyBuild` call creates a copy descriptor that is used by the `Dscopy` call to copy the file or files specified.

Two programmatic examples, one in Pascal/1000 (Version 2) and one in Fortran 77, are provided at the end of this section.

DSCOPY

Copies a file or files.

Syntax

```
DSCOPY(builtdescriptor,result)
```

Parameters

builtdescriptor	*Character array (FORTRAN); String (PASCAL)*. A buffer of variable length that contains a copy descriptor or a DSCOPY command. The *builtdescriptor* parameter may be created programmatically by calling DscopyBuild. (DscopyBuild is described later in this section.)
result	*Array of 16-bit integers*. A five-word array returned by Dscopy. The first word contains the number of errors that occurred while the file, or files, were being copied. The second word returns the error code, if any; zero is returned if the file or files are copied successfully. (If multiple files are copied, the error code is the result of the last attempted file copy.) The last three words of this parameter are reserved for future use. The DSCOPY error codes are described in the *NS/1000 Error Message and Recovery Manual*.

Discussion

If a copy descriptor is specified in the *builtdescriptor* parameter, DSCOPY will execute the request and then terminate. Control will then be returned to the calling program.

If a DSCOPY command is specified in the *builtdescriptor* parameter, DSCOPY will execute the command and then enter interactive mode. However, if the command is +EX (to exit), DSCOPY will terminate and control will be returned to the calling program.

If your program is written in Pascal/1000, Version 2, you must set the FIXED_STRING option before declaring Dscopy. In addition, the routines SetStrLen and StrMax must be used to initialize the *builtdescriptor* string. FIXED_STRING, SetStrLen and StrMax are described in the *Pascal/1000 Reference Manual*.

If your program is written in Pascal/1000, Version 1, you must use the routine StrDsc to convert the *builtdescriptor* string to a format that can be processed by both the calling program and DSCOPY. This routine is described in the *RTE-A Programmer's Reference Manual*.

Builds a copy descriptor to be used in the `Dscopy` call.

Syntax

```
DSCOPYBUILD(builtdescriptor,sfile,slogon,snode,tfile,tlogon,
            tnode,options,rsize,fsize)
```

Parameters

builtdescriptor — *Character array (FORTRAN); String (PASCAL)*. The returned copy descriptor to be used in the `Dscopy` call. Will be blank-padded if less than the length declared.

sfile — *Character array (FORTRAN); String (PASCAL)*. The source file; the name of the file to be copied. Refer to "File Masks" earlier in this section for an explanation of how file masks may be used in this parameter. (The *NS Cross-System NFT Reference Manual* explains file name syntax at other NS systems.)

slogan — *Character array (FORTRAN); String (PASCAL)*. The logon and password, if any, at the node where the source file resides. Do *not* enclose in brackets. This parameter is *required* if the source node is a remote multiuser HP 1000. NS-ARPA/1000 is not supported on a single user HP 1000 computer. (The *NS Cross-System NFT Reference Manual* explains logon and password syntax at other NS systems.)

Default: If this parameter is a string of blanks and the source node is the local node, the account under which the program is running is used.

snode — *Character array (FORTRAN); String (PASCAL)*. The name of the source node. The syntax of NS node names is described in Section 1, "Introduction," of this manual.

Default: If this parameter is a string of blanks, *snode* will default to the local node. You may omit the organization and domain, or substitute a string of blanks for this parameter. If the organization, or organization and domain, are omitted, the local organization and/or domain will be used. If a string of blanks is used, the node name will default to the local node.

tfile — *Character array (FORTRAN); String (PASCAL)*. The target file; the name the source file will acquire at the target node. Refer to "File Masks" earlier in this section for an explanation of how HP 1000 file masks may be used in this parameter. (The *NS Cross-System NFT Reference Manual* explains target file syntax at other NS systems.)

tlogon — `Character array (FORTRAN); String (PASCAL)`. The logon and password, if any, at the target node. Do *not* enclose in brackets. This parameter is *required* if the source node is a remote multiuser HP 1000. NS-ARPA/1000 is not supported on a single user HP 1000 computer. (The *NS Cross-System NFT Reference Manual* explains logon and password syntax at other NS systems.)

Default: If this parameter is a string of blanks and the target node is your local node, the account under which the program is running is used.

tnode — `Character array (FORTRAN); String (PASCAL)`. The name of the target node. The syntax of NS node names are described in Section 1, "Introduction," of this manual.

Default: If this parameter is a string of blanks, *snode* will default to the local node. You may omit the organization and domain, or substitute a string of blanks for this parameter. If the organization, or organization and domain, are omitted, the local organization and/or domain will be used. If a string of blanks is used, the node name will default to the local node.

options — `32-bit integer`. A two-word (32-bit) parameter which identifies specific options. An option is included if its corresponding bit is set. If no bits are set, no options are specified. The options and their corresponding bits are listed below (zero represents the least significant bit). These options are equivalent to those that can be used with DSCOPY interactively.

For an explanation of the meaning of the following options, refer to the "Copy Descriptor" discussion in this section.

0	Reserved for future use.
1	ASCII
2	BINARY
3	Reserved for future use.
4	FIXED
5	INTERCHANGE
6	MOVE
7	OVERWRITE
8	QUIET
9	REPLACE
10	STRIP
11	VARIABLE
12	SILENT
13 through 31	Reserved for future use.

rsize	*32-bit integer*. Appends the RSIZE option to the *builtdescriptor*. The value in *rsize* is in bytes. If fixed length records are being produced, *rsize* is the size of each record. If variable length records are being produced, *rsize* limits the size of the largest record and records may be padded or truncated. If *rsize* is zero, the RSIZE option is not appended to the *builtdescriptor* and the target file will have the same record size as the source file. You cannot copy files with records longer than 4400 bytes to or from an HP 1000.
fsize	*32-bit integer*. Appends the FSIZE option to the *builtdescriptor*. The value in *fsize* specifies how much space to allocate for the target file. If the target file has fixed length records, *fsize* is in records. If the target file has variable length records, *fsize* is the number of maximum size records. You can use this option instead of the HP 1000 file descriptor size parameter to specify the size of an HP 1000 target file. If *fsize* is zero, the FSIZE option is not appended to the *builtdescriptor* and the target file will be the same size as the source file.

Discussion

If your program is written in Pascal/1000, Version 2, you must set the FIXED_STRING option before declaring DscopyBuild. In addition, the Pascal routines SetStrLen and StrMax must be used to initialize the *builtdescriptor* string prior to calling DscopyBuild. FIXED_STRING, SetStrLen and StrMax are described in the *Pascal/1000 Reference Manual*.

If your program is written in Pascal/1000, Version 1, you must use the routine StrDsc to convert the *builtdescriptor* string to a format that can be processed by both the calling program and DSCOPY. This routine is described in the *RTE-A Programmer's Reference Manual*.

Programmatic Examples

Below are two example programs, one in Pascal and one in FORTRAN.

```
$PASCAL '91790-16239 REV.5240 <860303.1238>'
$CDS$
$ CODE_CONSTANTS OFF $
$ DEBUG $

{}
{       NAME: COPY
{     SOURCE: 91790-18239
{      RELOC: 91790-16239
{       PGMR: VH
{}
{  MODIFICATION HISTORY
{
{  DATE     PGMR  DESCRIPTION
{
{  053091   VH    Modified to get nodename and user/passwd.
{}
PROGRAM COPY (input,output);

CONST

    FIXED_OPT     = 16;
    QUIET_OPT     = 256;
    REPLACE_OPT   = 512;

TYPE
    CommandType   = String [150];
    Integer16     = -32768..32767;
    FileNameType  = String [64];
    LogonType     = String [30];
    NodeNameType  = String [20];
    FiveWordsType = ARRAY [1..5] OF Integer16;

VAR
    command       : CommandType;
    options       : Integer;
    result        : FiveWordsType;
    source_name   : FileNameType;
    target_name   : FileNameType;
    nodename      : NodeNameType;
    login         : LogonType;

$FIXED_STRING ON$

PROCEDURE Dscopy
    (VAR command  : String;
     VAR result   : FiveWordsType);EXTERNAL;
```

```
PROCEDURE DscopyBuild
   (VAR command      : String;
        source_file  : FileNameType;
        source_logon : LogonType;
        source_node  : NodeNameType;
        target_file  : FileNameType;
        target_logon : LogonType;
        target_node  : NodeNameType;
        options      : Integer;
        rsize        : Integer;
        fsize        : Integer);EXTERNAL;

BEGIN {main program}

{}
{ get nodename
{}
prompt ('Enter node name: ');
readln (nodename);

{}
{ get login/passwd.
{}
prompt ('Enter login/passwd: ');
readln (login);

{read the source and target file names}

prompt ('Enter source name: ');
readln (source_name);

prompt ('Enter target name: ');
readln (target_name);

{set bit for each desired option in the options bit array}

options := FIXED_OPT + QUIET_OPT + REPLACE_OPT;

{initialize the command string}

SetStrLen (command, StrMax (command));

{default the source logon, source node, and set RSIZE = 80 bytes}

DscopyBuild (command, source_name, ' ', ' ', target_name,
             login, nodename, options, 80, 0);

Dscopy (command, result);

{print the result}

writeln ('Total errors: ', result [1]);
writeln ('Error code: ', result [2]);

END. {main program}
```

```
FTN77,L
$CDS ON
      PROGRAM COPY(4,99),91790-16240 REV.5240 <860303.1238>
C
C       NAME: COPY
C     SOURCE: 91790-18240
C      RELOC: 91790-16340
C       PGMR: VH
C
C MODIFICATION HISTORY
C --------------------
C DATE   PGMR   DESCRIPTION
C 053091 VH     modified to take nodename and login/passwd.
C
      CHARACTER nodename*50
      CHARACTER login*32
      CHARACTER command*150, source_name*64, target_name*64
      INTEGER*4 options, fsize, rsize
      INTEGER result(2)

      PARAMETER (FIXED_OPT=16, QUIET_OPT=256, REPLACE_OPT=512)

      write (1,'("Enter node name: _")')
      read  (1,'(A50)') nodename

      write (1,'("Enter login/passwd: _")')
      read  (1,'(A32)') login

      WRITE (1,'("Enter source name: _")')
      READ  (1,'(A64)') source_name

      WRITE (1,'("Enter target name: _")')
      READ  (1,'(A64)') target_name

      options = FIXED_OPT + QUIET_OPT + REPLACE_OPT
      rsize = 80
      fsize = 0

      CALL DscopyBuild (command, source_name, ' ', ' ', target_name,
     +                       login, nodename, options, rsize, fsize)

      CALL Dscopy (command, result)

      WRITE (1,'("Total errors: _",I4)') result(1)
      WRITE (1,'("Error code: _",I4)') result(2)

      STOP

      END
```

5

Network Interprocess Communication

Overview

Network Interprocess Communication (NetIPC) is an NS Common Service that enables processes on the same or different NS-ARPA/1000 nodes to communicate using a series of programmatic calls.

The form of interprocess communication offered by NetIPC is more flexible than that provided by PTOP. PTOP is described in the *DS/1000-IV Compatible Services Reference Manual*. Because the relationship between NetIPC processes is peer-to-peer rather than master-to-slave, NetIPC processes are more independent than PTOP processes where the "master" process is in control of communication. NetIPC and PTOP are compared in the *DS/1000-IV Compatible Services Reference Manual*.

Processes that use NetIPC calls gain access to the communication services provided by the network protocols utilized by NS-ARPA/1000. NetIPC does not encompass a protocol of its own, but acts as a generic interface to the protocols underlying all of the NS-ARPA/1000 application services.

Network interprocess communication between an HP 1000 and other types of HP computers is also available with the NetIPC service. A NetIPC program on an HP 1000 is able to communicate with a peer NetIPC program on an HP 9000 computer, HP 3000 computer, or PC. This functionality between two processes on two different computer systems systems is called *cross-system NetIPC*. Information about cross-system NetIPC is explained in the subsection, "Cross-System NetIPC," later in this section.

The "Porting NetIPC Programs" appendix of this manual describes programming considerations when porting HP 1000 NetIPC programs to run under HP 9000 and vice versa.

The remainder of this NetIPC section is arranged as follows:

- Provides conceptual information about network interprocess communication—sockets and connections.
- Explains some of the common parameters used in the NetIPC calls.
- Summarizes cross-system NetIPC considerations between HP 1000 and other types of HP computers.
- Mentions ways to schedule a remote process.
- Explains the NS-ARPA/1000 NetIPC calls.

Sockets

NetIPC processes communicate with each other by means of s*ockets*. Processes make use of sockets via the NetIPC calls to establish connections and exchange data. The Transport Layer's Transmission Control Protocol (TCP) regulates the transmission of data to and from sockets. Although data must pass through the control of lower-level protocols and, if necessary, through intervening nodes, these details are transparent to NetIPC processes when they send and receive data. A brief description of the NS-ARPA/1000 network architecture is provided in the Introduction to this manual. For more detailed information, refer to the *NS-ARPA/1000 Generation and Initialization Manual*.

Connections

Before a connection can be established between two NetIPC processes, each process must create a *call socket*. A call socket is roughly analogous to a telephone handset with multiple buttons or extensions. Call sockets are used to create and connect *virtual circuit* (*VC*) *sockets*. When two VC sockets are connected, they become the endpoints of a connection called a *virtual circuit*, or a *virtual circuit connection*.

While a call socket is analogous to a telephone with multiple extensions, a VC socket is analogous to one of the extensions on that telephone. Figure 5-1 is an illustration of this telephone analogy.

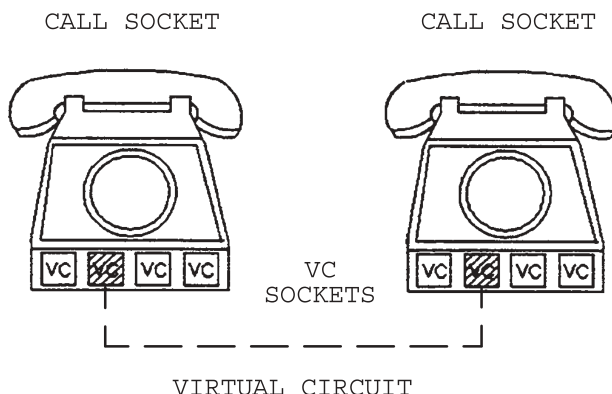


Figure 5-1. Telephone Analogy

Virtual circuits are the basis for interprocess communication. Once a virtual circuit is established, the two processes that created it may use it to exchange data. *Only VC sockets can be used to pass data between processes; data cannot be passed through call sockets*. A virtual circuit has two major properties:

- It is a dedicated link, accessible only to the two processes that established the connection.
- It provides reliable service, guaranteeing that data will not be corrupted, lost, duplicated or received out of order.

Naming, Socket Registry, and Path Reports

When a NetIPC process initiates a connection to a peer process, it must reference a call socket that was created by that peer process. To gain access to another process's call socket, a NetIPC process must reference the socket's *name*.

NetIPC processes may assign ASCII-coded names to their call sockets. Each NS-ARPA/1000 node has a *socket registry* that contains a listing of all the named call sockets that reside at that node. Pursuing the telephone analogy begun earlier, the socket registry could be compared to a telephone directory: a call socket name is inserted in the local socket registry in much the same way as a person's name is placed in a local telephone directory.

NetIPC processes reference call sockets created by other processes by passing a socket name and the corresponding node name to the socket registry software. The socket registry determines which socket is associated with the name and formats the address information pertaining to that socket into a *path report* which it returns to the inquiring process. When a path report is returned to a process, it tells the process how it can send messages to the associated socket.

Using the socket registry to gain access to another process's call socket is similar to using directory assistance to find a person's telephone number because a path report, like a telephone number, is an address that can be used to direct a call to a particular destination.

Descriptors

NetIPC processes reference call sockets, VC sockets and path reports with *descriptors*. Descriptors are returned to processes when certain NetIPC calls are invoked. An explanation of these descriptors and the NetIPC call, or calls, that are used to obtain them follows.

- *Call Socket Descriptor*. A call socket descriptor refers to a call socket. A process obtains a call socket descriptor by invoking `IPCCreate` (to create a call socket) or `IPCGet` (to get a call socket descriptor given away by another process). When a call socket descriptor is obtained with either one of these calls, the call socket it refers to is said to be *owned* by the calling process.

- *Path Report Descriptor*. A path report descriptor refers to a path report. The path report contains addressing information that is used by the calling process to direct requests to a certain call socket at a certain node. A process obtains a path report descriptor by invoking either `IPCLookUp` (to look up the name of a call socket in a specific socket registry), `IPCGet` (to obtain a path report descriptor given away by another process), or `IPCDest` (to create a path report descriptor).

- *VC Socket Descriptor*. A VC socket descriptor refers to a VC socket. A VC socket is the endpoint of a virtual circuit connection between two processes. A VC socket descriptor is returned by `IPCRecvCn` and `IPCConnect` after an initial dialogue takes place over a connection formed by call sockets. A process can also obtain a VC socket descriptor given away by another process by invoking `IPCGet`.

Table 5-1. Descriptor Summary

Descriptor Type	Parameter Name	Description	Returned as Output From
call socket descriptor	*calldesc*	Refers to a call socket. A call socket is used to build a VC socket.	IPCCREATE IPCGET
path report descriptor	*pathdesc*	Refers to a path report. A path report contains addressing information that is used to direct requests to a certain call socket at a certain node.	IPCLOOKUP IPCGET IPCDEST
VC socket descriptor	*vcdesc*	Refers to a VC socket. A VC socket is the endpoint of a virtual circuit connection between two processes.	IPCCONNECT IPCRECVCN IPCGET

Establishing a Connection

The steps needed to establish a virtual circuit connection are described in the following examples. Although only two processes are shown, this is not meant to imply that communication cannot exist between more than two processes. Either or both of the processes shown can establish virtual circuit connections with other processes. Secondary or auxiliary connections can also be set up between the same two processes.

The following paragraphs are a call-by-call explanation of how a virtual circuit connection is built. The telephone analogy that was used to explain call sockets, VC sockets, and virtual circuits is continued as each call is compared to a certain aspect of the telephone system.

Usually the two processes are executing at the same time or one process starts executing first and then schedules the other process. NetIPC itself does not provide a call to schedule a peer process. However, there are other services that allow you to schedule a process at an NS-ARPA/1000 node. For more information refer to "Process Scheduling" later in this section.

Creating a Call Socket

Interprocess communication is initiated when Process A and Process B each create a call socket by invoking the NetIPC call `IPCCreate`. This is illustrated in Figure 5-2 below. As explained previously, a call socket is roughly analogous to a telephone with multiple extensions (see Figure 5-1). `IPCCreate` returns a *call socket descriptor* in its `calldesc` parameter that refers to the call socket, or "telephone," that was created. This call socket descriptor is used in subsequent NetIPC calls.

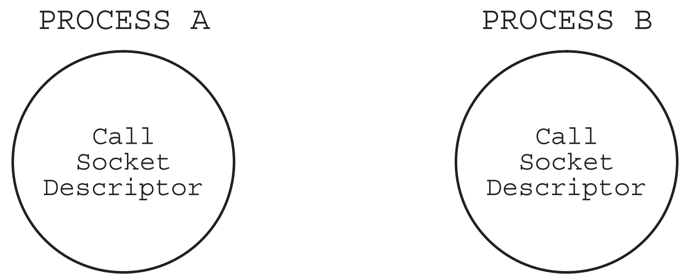


Figure 5-2. IPCCreate (Processes A and B)

Naming a Call Socket

Process B names its call socket by calling `IPCName`. This is illustrated in Figure 5-3 below. The name assigned by Process B is placed in the socket registry at the node on which Process B is running. The name Process B assigns to its call socket must also be known to Process A because Process A must reference it later in its `IPCLookUp` call. The name Process B assigns to its call socket must also be unique to its node. Although call sockets do not have to be named, a process cannot gain access to another process's call socket if the call socket is not named. The socket must be named and be in the socket registry at Process B's node when Process A calls `IPCLookUp`.

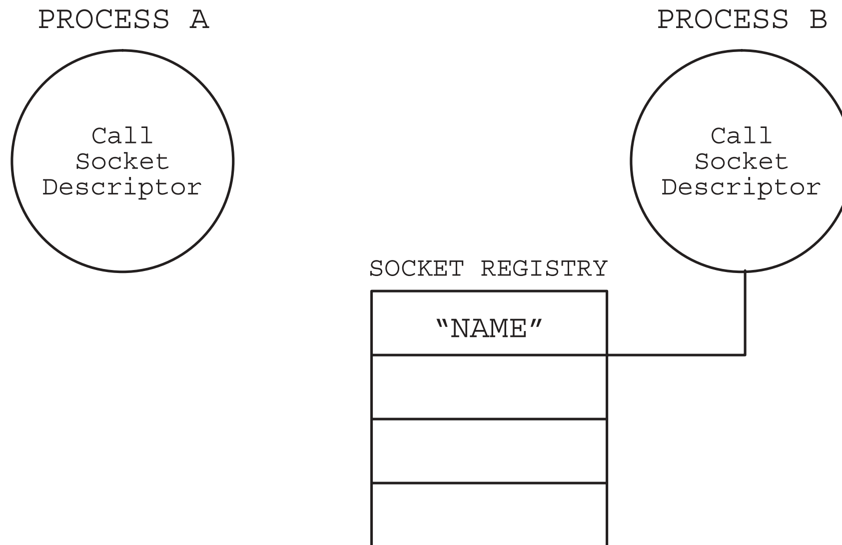


Figure 5-3. IPCName (Process B)

An alternative to using `IPCName` and `IPCLookUp` to name a socket and then obtain its destination descriptor is available through the use of the `IPCDest` call. `IPCDest` enables you to identify the remote socket by its TCP port address. Refer to the description of `IPCDest` later in this section for more information.

Looking Up a Call Socket Name

Process A must know the name assigned to Process B's call socket. It calls `IPCLookUp` to "look up" the name of the call socket in the socket registry at the node where Process B resides. `IPCLookUp` returns a *path report descriptor* in its *`pathdesc`* parameter. The path report described indicates the location of Process B's call socket. This is illustrated in Figure 5-4 below.

Compared to the telephone system, `IPCLookUp` is similar to directory assistance: Process A calls the "operator" (`IPCLookUp`), and gives him/her a "city" (*`location`* parameter) and a "name" (*`socketname`* parameter). Using the "city," the operator looks for the name in the proper "telephone directory" (socket registry). Once the name is found, the operator returns a "telephone number" (*`pathdesc`* parameter) to the caller.

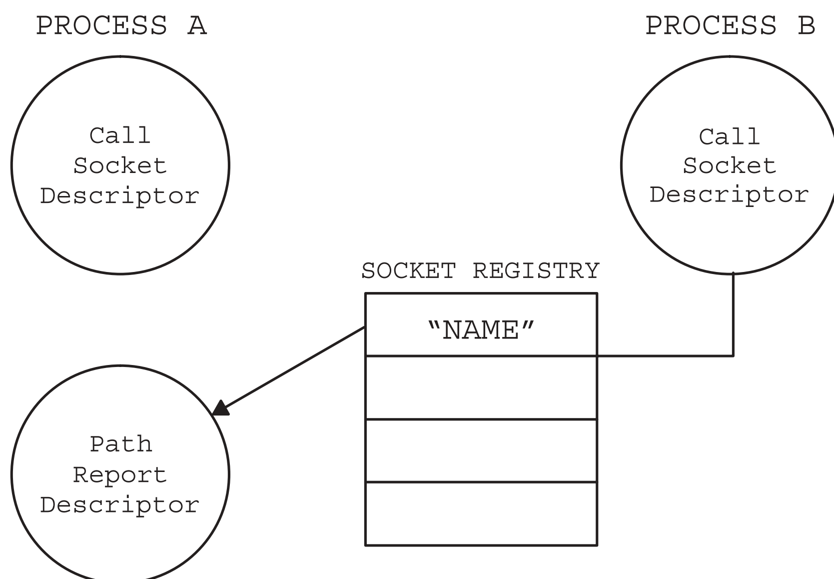


Figure 5-4. IPCLookUp (Process A)

You can also use `IPCDest` to obtain a path report descriptor for a call socket with a *particular protocol address*. A call socket is created by using the `IPCCreate` call with the PROTOCOL ADDRESS option.

Requesting a Connection

Process A specifies the path report descriptor returned by `IPCLookUp` and the call socket descriptor returned by `IPCCreate` in its `IPCConnect` call. With these two parameters, `IPCConnect` requests a virtual circuit connection between Process A and Process B. `IPCConnect` returns a *VC socket descriptor* in its *`vcdesc`* parameter that refers to the VC socket endpoint of the connection at Process A. This is illustrated in Figure 5-5 below.

`IPCConnect` is a non-blocking call; it does not suspend the execution of the calling process. Because of this, `IPCConnect` could be compared to dialing a phone, but not waiting for an answer. (The differences between blocking and non-blocking calls are explained in detail in "Asynchronous and Synchronous Socket Modes" later in this section.)

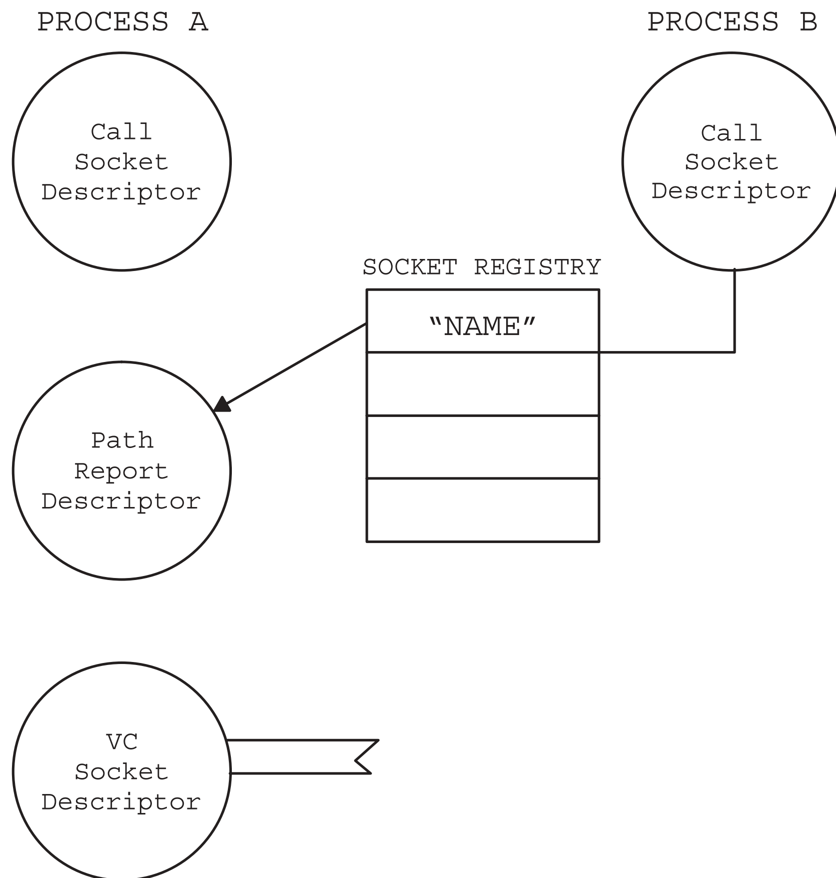


Figure 5-5. IPCConnect (Process A)

Receiving a Connection Request

Using the call socket descriptor returned by its `IPCCreate` call, Process B calls `IPCRecvCn` to receive any connection requests. In this example, Process B will receive a connection request from Process A. (Process A "dialed its telephone" to call Process B when it called `IPCConnect`.) `IPCRecvCn` returns a *VC socket descriptor* in its *`vcdesc`* parameter. This VC socket is the endpoint of the virtual circuit at Process B. This is illustrated in Figure 5-6. The connection will *not* be fully established until Process A calls `IPCRecv`. Compared to the telephone system, `IPCRecvCn` is similar to "hearing the telephone ring and answering it."

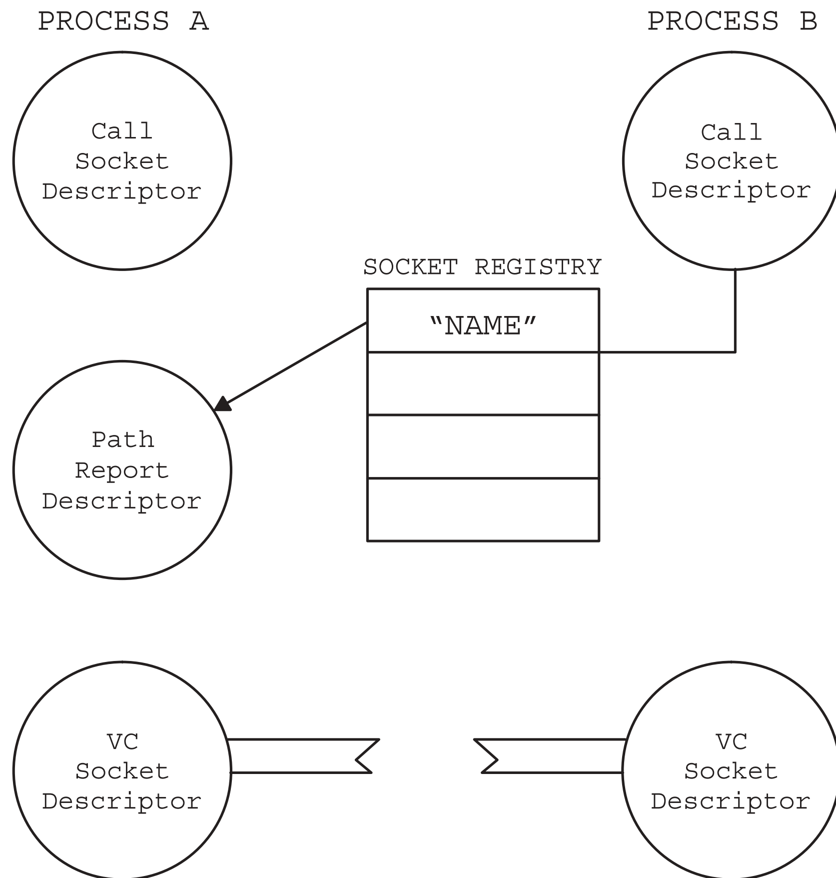


Figure 5-6. IPCRecvCn (Process B)

Checking the Status of a Connection

Process A calls `IPCRecv` using the VC socket descriptor returned by its `IPCConnect` call. `IPCRecv` returns the status of the connection (successful/unsuccessful) initiated by `IPCConnect`. If the status is successful, the connection has been established and Process A and Process B can “converse” over the new virtual circuit. This is illustrated in Figure 5-7.

Compared to the telephone system, `IPCRecv` is similar to “listening to hear if the phone was answered.” (Since `IPCConnect` was compared to “dialing a phone, but not waiting for an answer,” calling `IPCRecv` could be described as completing the connection request initiated by `IPCConnect`.)

`IPCRecv` can also be used to receive data. This function is discussed in the `IPCRecv` call discussion later in this section.

PROCESS A

PROCESS B

Call Socket Descriptor

Call Socket Descriptor

SOCKET REGISTRY

"NAME"

Path Report Descriptor

VC Socket Descriptor

VIRTUAL CIRCUIT CONNECTION

VC Socket Descriptor

Figure 5-7. IPCRecv (Process A)

Summary of Calls Used in Connection Establishment

Figure 5-8 illustrates the sequence of NetIPC calls that is used to establish a virtual circuit connection. This figure summarizes the information presented in Figure 5-2 through Figure 5-7.

PROCESS A	PROCESS B
IPCCreate ()	IPCCreate()
	IPCName()
IPCLookUp ()	
IPCConnect ()	
	IPCRecvCn()
IPCRecv()	

PROCESS A:

1. Create call socket
2. "Look up" name
3. Request connection
4. Check status of connection

PROCESS B:

1. Create call socket
2. Name call socket
3. Receive connection request

Figure 5-8. Establishing a Connection with IPCLookUp

Figure 5-9 summarizes a different way to establish a virtual circuit connection using IPCDest.

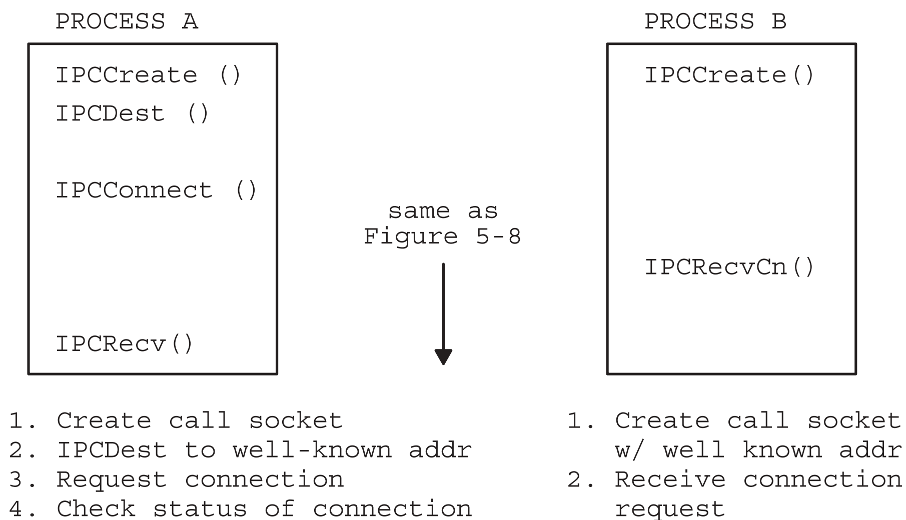


Figure 5-9. Establishing a Connection with IPCDest

Steps 3 and 4 are the same for Process A in both figures. IPCLookUp and IPCName specify a node name whereas IPCDest specifies a well-known address (integer). Note that the advantage of using IPCLookUp is that names might be easier to remember and use. With IPCDest, the address must be unique and other processes must cooperate and not use that same address.

Sending and Receiving Data Over a Connection

Once a virtual circuit connection is established, both processes can send and receive data using the NetIPC calls IPCSend and IPCRecv. IPCSend is used to send data on an established connection. Invoking IPCSend is analogous to "speaking" over a telephone connection. IPCRecv is used to receive data on an established connection. The use of IPCRecv is similar to "listening" at your telephone handset. (Note that IPCRecv has a dual function: to establish a virtual circuit connection as well as to receive data on a previously established connection.)

Shutting Down a Connection

The NetIPC call IPCShutDown releases a descriptor and any resources associated with it. IPCShutDown can be called to release a call socket descriptor, a path report descriptor or a VC socket descriptor. How IPCShutDown functions depends on which type of descriptor is referenced. Refer to the discussion of IPCShutDown later in this section for more information on releasing call socket descriptors and path report descriptors.

Before terminating, a process should close its virtual circuit connections by calling IPCShutDown to release its VC socket descriptors. Because IPCShutDown takes effect very quickly, any data that is in transit on the connection, including any data that has already been queued on the destination VC socket, may be destroyed before its intended recipient is able to receive it. As a result, the processes that share a connection must cooperate to ensure that no data is lost. In order to release a connection without losing data, two processes can take the following steps:

- Process A sends a “last message” to Process B via an `IPCSend` call. This message contains data that will be recognized by Process B as a termination request. Process A then calls `IPCRecv` to wait for Process B’s “last message.”
- Process B calls `IPCRecv` to receive Process A’s “last message” and then sends its own “last message” to Process A via `IPCSend`. Process B’s message contains data that will be recognized by Process A as a confirmation of its termination request. Process B then calls `IPCRecv` to wait for an error indicating that Process A has closed the connection.
- Process A receives Process B’s “last message” via a call to `IPCRecv` and calls `IPCShutDown` to release its VC socket descriptor and close the connection.
- Process B’s `IPCRecv` call receives a “remote aborted the connection” error (error code 64). It then calls `IPCShutDown` to release its own VC socket descriptor.

Timing and Timeouts

When setting up and using a virtual circuit connection, timing is critical at several points:

- When a process calls `IPCLookUp` to “look up” the name of a call socket in the socket registry of a remote node.
- When a process calls `IPCRecvCn` to receive a connection request from another process.
- When a process first calls an `IPCRecv` after an `IPCConnect` (to check the status of a connection).
- When a process sends and receives data with `IPCSend` and `IPCRecv`.

When a process attempts to look up a socket name in the appropriate socket registry, the name must be there or a “name not found” error (error code 37) will be returned to the calling process. When two processes are running concurrently, it may be difficult to ensure that a socket name is placed in the socket registry prior to being “looked up” by another process. This problem is referred to as a *race condition* because the two processes are “racing” to see which one will access the socket registry first. Several ways to avoid this race situation are outlined in the discussion of `IPCLookUp` later in this section.

If the NetIPC calls `IPCRecvCn`, `IPCSend` and `IPCRecv` are used synchronously, it may be necessary to alter the synchronous timeout value by calling `IPCControl.` (The default synchronous timeout is 60 seconds.) The synchronous timeout determines:

- How long `IPCRecvCn` will suspend the calling program while waiting for a connection request.
- How long `IPCSend` will suspend the calling program if it cannot immediately obtain the buffer space needed to accommodate its data.
- How long `IPCRecv` will suspend the calling program if its request for data cannot be satisfied or if the referenced connection cannot be established.

For more information on synchronous I/O, refer to “Synchronous and Asynchronous Socket Modes” later in this section.

Additional NetIPC Calls

Once a virtual circuit is established between processes, descriptors can be given away, names can be erased, and other functions can be performed. The following NetIPC calls are provided in addition to those described in the previous paragraphs to enable you to perform these functions. A brief introduction to each call and its use follows. (A complete description of these and all of the NetIPC calls is provided in the following pages.)

- *`IPCControl`*. Performs special operations on sockets such as enabling synchronous or asynchronous mode, changing synchronous timeout values, and setting read and write threshold values.

- *`IPCDest`*. Returns a path report descriptor that the calling process can use to establish a connection to another process. Using this call is an alternative to naming the call socket with `IPCName` and acquiring a path report descriptor with `IPCLookUp`.

- *`IPCGet`*. The companion call to `IPCGive`. Receives a descriptor given away by a process that has called `IPCGive`. This call is similar to `IPCLookUp` because it enables your process to acquire a descriptor that can be used in subsequent NetIPC calls.

- *`IPCGive`*. The companion call to `IPCGet`. Releases ownership of a descriptor to NetIPC so that it can be acquired by another process via a call to `IPCGet`.

- *`IPCNamErase`*. Does the reverse of `IPCName`: it removes a name associated with a socket or path report descriptor from the socket registry. Only the owner of a descriptor can remove its name.

- *`IPCSelect`*. Allows a process to detect and/or wait for the occurrence of any of several events across multiple sockets. `IPCSelect` can report: (1) whether the socket has any data queued to it; (2) whether the socket can accommodate any new data that might be sent out through it; and (3) whether the socket has some exceptional condition associated with it. Compared to the telephone system, `IPCSelect` allows you to perform complex "switchboard" operations.

Summary of NetIPC Calls

The following table summarizes the NetIPC calls described in this section.

Table 5-2. NetIPC Calls

Call	Description
IPCCONNECT	Requests a virtual circuit to another program and returns a VC socket descriptor which identifies a VC socket endpoint at the calling program.
IPCCONTROL	Performs special operations on sockets such as enabling synchronous and asynchronous modes, changing the synchronous timeouts, and setting read and write thresholds.
IPCCREATE	Creates a call socket for the calling program.
IPCDEST	Returns a path report descriptor that the calling process can use to establish a connection to another process.
IPCGET	Receives a descriptor given away by another program.
IPCGIVE	Releases ownership of a descriptor to NetIPC so that the descriptor can be acquired by another program via a call to IPCGet.
IPCLOOKUP	Searches the socket registry for a socket name and returns a path report descriptor that indicates how to get to the destination call socket.
IPCNAME	Associates a name with a call socket descriptor or path report descriptor and places it in the local node's socket registry.
IPCNAMERASE	Removes a name associated with a call socket descriptor or path report descriptor from the socket registry.
IPCRECV	Establishes a virtual circuit, or receives data on a previously established connection.
IPCRECVCN	Receives a connection request from another program and returns a VC socket descriptor that describes a VC socket endpoint at the calling program.
IPCSELECT	Enables a program to detect and/or wait for the occurrence of any of several events across multiple call or VC sockets.
IPCSEND	Sends data to another program on a virtual circuit.
IPCSHUTDOWN	Releases a descriptor and any resources associated with it.

Synchronous and Asynchronous Socket Modes

When a send operation is performed on a socket, data is moved out of a process into an outbound transmission buffer. Similarly, when a receive operation is performed on a socket, data is moved from an inbound transmission buffer into a process. Sometimes a send or receive request cannot be immediately satisfied. In the case of `IPCSend`, an empty transmission buffer may not be available; an `IPCRecv` request may not be satisfiable because data-filled transmission buffers are not queued on the referenced socket. When either of these situations occur, NetIPC must decide whether to fail the request or suspend the process until the request can be satisfied. This decision is based upon whether the socket being manipulated is in *synchronous* or *asynchronous* mode.

Sockets are automatically placed in synchronous mode when they are created. When a socket is in synchronous mode, send and receive requests that reference it cause the calling process to be suspended if the requests cannot be immediately satisfied. A process that has been suspended will remain suspended until the request is satisfied, a synchronous timeout occurs, or an error is detected. Each synchronous socket has a timer associated with it that can be modified with an `IPCControl` call. This timer determines how long a NetIPC call will block the socket while waiting for its request to be satisfied. A NetIPC call will not be able to block forever unless the synchronous timeout value is set to zero with a call to `IPCControl.`

Three NetIPC calls, `IPCSend`, `IPCRecv` and `IPCRecvCn`, support asynchronous as well as synchronous I/O. In addition, `IPCConnect` is by definition an asynchronous call. (The remaining NetIPC calls support only synchronous I/O.) Sockets can be placed in asynchronous mode by calling `IPCControl` and specifying request code 1 in the *`request`*s parameter. Send and receive requests directed against a socket in this mode do not cause the calling process to be suspended if the requests cannot be immediately satisfied. Instead, a “would block” error (error code 56) is returned and the process is free to perform other tasks before retrying the request.

Read and Write Thresholds

For efficiency, a process using asynchronous sockets must be able to determine whether a VC socket can satisfy an `IPCSend` or `IPCRecv` call *before* the request is issued. The `IPCSelect` call addresses this problem by providing socket status information. Included in this information is whether or not:

- A VC socket is *readable* (it can satisfy an `IPCRecv` call).
- A VC socket is *writeable* (it can satisfy an `IPCSend` call).

`IPCRecv` determines whether or not a VC socket is readable by examining the socket’s *read threshold*. A VC socket is considered readable if it can immediately satisfy an `IPCRecv` request for a number of bytes *equal to or greater than* its read threshold. The read threshold is used by `IPCSelect` to check if there are *at least* that many bytes queued on the socket ready for reading.

Similarly, `IPCSend` determines whether or not a VC socket is writeable by examining the socket’s *write threshold*. A VC socket is considered writeable if it can immediately satisfy an `IPCSend` request for a number of bytes *equal to or greater than* its write threshold. The write threshold is used by `IPCSelect` to check if there are *at least* that many bytes in the system ready to be used as a buffer space for writing to a particular socket.

`IPCSelect` will not return accurate status information unless a socket's read and write thresholds are set to the correct number of bytes. The thresholds default is one byte each. (A VC socket's read and write thresholds can be set by calling `IPCControl`. Refer to the discussion of this call for more information.) The number of bytes that can be sent or received on a socket should determine the correct read and write threshold settings. As a general rule, *set a socket's read threshold to the same number of bytes as the length of the data you expect to receive on that socket*. Similarly, *set a socket's write threshold to the same number of bytes you expect to send on that socket*. Consider the following example: Process B will always issue `IPCSend` calls with 64 bytes of data on VC socket X. Therefore, socket X's write threshold should also be 64 bytes. Similarly, if Process B expects to issue 64-byte `IPCRecv` requests on socket X, socket X's read threshold should be set to 64 bytes as well.

If you expect to receive variable length data on a particular VC socket, the socket's read threshold should be set to the length of the *shortest* amount of data expected. If you expect to send variable length data on a particular VC socket, the socket's write threshold should be set to the length of the *longest* amount of data you expect to send.

Note The read and write thresholds are used exclusively by the `IPCSelect` call. They have no effect on other NetIPC calls.

For more information on using sockets in asynchronous mode, refer to the discussions of `IPCSelect`, `IPCControl`, `IPCSend` and `IPCRecv`.

Stream Mode

All data transfers between NetIPC processes are in *stream mode*. Stream mode adheres to the Transport Layer's Transmission Control Protocol (TCP). In stream mode, data is transmitted in a stream of bytes; there are no end-of-message or end-of-data markers. This means that the data received by an individual `IPCRecv` call may not be equivalent to data sent by an individual `IPCSend` call. In fact, the data received may contain part of the data or multiple sets of data sent by multiple `IPCSend` calls. Although no attempt is made to preserve boundaries between data sent at different times, the data received will always be in the correct order (in the order that the data was sent).

You may specify the maximum number of bytes that you are willing to receive through a parameter of the `IPCRecv` call. When the call completes, this parameter will contain the number of bytes *actually* received. The amount of data received will never be more than the amount that was requested, but it may be less. Whether or not an `IPCRecv` call will receive less data than it requested is determined by the DATA_WAIT bit of the *`flags`* parameter. If the DATA_WAIT bit is set, `IPCRecv` will never receive less than the requested amount. If the DATA_WAIT bit is *not* set, `IPCRecv` may receive less data than was requested.

If an `IPCRecv` call requests more data than is queued on a VC socket, one of the following situations will result:

- If the VC socket is in synchronous mode, the calling process will suspend until enough data is queued to satisfy the `IPCRecv` request. If enough data does not arrive within the synchronous timeout period to satisfy the request, a "timeout" error (error code 59) will be returned.
- If the VC socket is in asynchronous mode, a "would block" error (error code 56) will be returned.

For more information on synchronous and asynchronous I/O, refer to the previous discussion titled "Synchronous and Asynchronous Socket Modes" and to the discussions of `IPCSend` and `IPCRecv` later in this section.

NetIPC Common Parameters

The *flags*, *opt*, *data*, *result*, *socketname*, and *nodename* parameters are common to many NetIPC calls. The *flags*, *opt*, and *result* parameters are also common to the Remote Process Management (RPM) calls. RPM calls are explained in the "Remote Process Management" section later in this manual. These calls can be used to schedule remote NetIPC programs.

The *opt* parameter provides functionality for NetIPC and RPM calls; *opt* usually has *data* associated with it. The *flags* parameter enables or disables certain functions for NetIPC calls. The *result* parameter returns error codes for NetIPC calls. The *socketname* and *nodename* parameters identify sockets and nodes, respectively. The following descriptions should help you better understand the meaning, use, and structure of each of these parameters.

Flags Parameter

The *flags* parameter is a bit map of 32 *special request bits*. By setting bits in the *flags* parameter, you can invoke various services in IPCConnect, IPCRecv, IPCRecvCn, and IPCSend calls.

The NetIPC calls IPCControl, IPCCreate, IPCDest, IPCGet, IPCGive, IPCLookUp, and IPCShutDown also include a *flags* parameter, but in these calls this parameter is reserved for future use. However, the *flags* parameter must be initialized to zero before it is used in these calls. The parameter must also be cleared *after* it is used in these calls. This precaution should be taken because NetIPC calls that do not use the *flags* parameter on input may return non-zero values in *flags* on output.

NetIPC and RPM calls assume that the bits in the *flags* parameter are numbered from left to right with the most significant bit being bit *one*, and the least significant bit being bit 32.

```
MSB
1 2 3 4 5 6 ...             32
```

Pascal, NetIPC, and RPM

```
MSB
31 30 29 28 ...              0
```

FORTRAN

The IPCSelect bit map parameters are also numbered from left to right as in the NetIPC *flags* parameter. Refer to "IPCSelect Call Bit Map Parameters" later in this section.

The following paragraphs explain how the *flags* parameter is declared and manipulated in Pascal and FORTRAN.

Pascal Programming Language

In Pascal/1000, the *flags* parameter is represented as follows:

```
    TYPE

         flags_type  = packed array [1..32] of boolean;

    VAR

         flags : flags_type;
```

`flags [1]` refers to the high order bit in the boolean array; `flags [32]` refers to the low order bit. To set a bit in the array, assign the value `TRUE` to the desired bit. For example, `flags[22]:=TRUE` would set bit 22 of the *flags* array. A clear bit would be assigned the value `FALSE`. If you do not want to set any of the bits in the *flags* array, but you want to be certain that all of the bits are clear, you may make *flags* type `INTEGER` and assign it the value zero.

FORTRAN 77 Programming Language

In FORTRAN 77, the *flags* parameter must be declared as a 32-bit integer (`INTEGER*4`). The simplest way to set a bit in this parameter is to use the FORTRAN 77 library function `ibset(`*a*`,`*b*`)`. The *flags* parameter is passed in the first argument (*a*) and the bit position you want to set is passed in the second argument (*b*). Multiple bits can be set by repeating the `ibset` function.

In the following FORTRAN 77 example, flags bit 22 is set in the *flags* parameter:

```
    INTEGER*4 flags
    INTEGER*4 ibset

C   The flags value is subtracted from 32 so that the proper
C   bit is set. This maps ibset’s bit numbering convention into
C   NetIPC’s.

C   Set Bit 10 in FORTRAN, which is Bit 22 in NetIPC
    flags = ibset(flags,(32–22))
```

```
MSB
1 2 3 4 5 6 ...     22...                32   NetIPC flags

MSB
31 30 29 28 ...      10 9 8 7 6 5 4 3 2 1 0   FORTRAN
```

Opt Parameter

The *opt* parameter allows you to request optional services when invoking certain NetIPC and RPM calls. It enables calls that include the *opt* parameter to accept an arbitrary number of arguments that are either protocol or operating system specific. To help you distinguish between a *flag* parameter and an *opt* parameter, remember that the *opt* parameter is an array and usually has data associated with it.

You can invoke services from the *opt* parameter in the NetIPC calls, `IPCConnect`, `IPCCreate`, `IPCRecv`, `IPCRecvCn`, and `IPCSend`.

The NetIPC calls, `IPCDest` and `IPCShutDown`, also include an *opt* parameter, but in these calls this parameter is reserved for future use. However, the *opt* parameter must be initialized to zero before it is used in these calls.

Because the *opt* parameter is an array with a complex structure, NetIPC provides a special set of calls that allow your processes to view the parameter as a packed array of bytes (Pascal) or an array of words (FORTRAN). Table 5-3 summarizes these calls. A complete description of each call is provided in "Special NetIPC Calls" at the end of this section.

Table 5-3. Special NetIPC Calls

Call	Description
`ADDOPT`	Adds an argument and its associated data to an *opt* parameter.
`ADROF`	Obtains the byte address of any byte within a data object.
`INITOPT`	Initializes an *opt* parameter so that arguments can be added.
`READOPT`	Obtains the option code and argument data associated with an *opt* parameter argument.

Before you can invoke a NetIPC or RPM call that includes an *opt* parameter, you must prepare the parameter by using the following *opt* parameter calls:

- First, `InitOpt` must be called to initialize the *opt* parameter. This call allows you to specify how many arguments will be placed in this parameter.
- Next, `AddOpt` must be called to add an argument and its associated data to the *opt* parameter. An `AddOpt` call can add only one argument at a time, so you must call it multiple times if you want to add multiple arguments to the *opt* parameter.

The two other *opt* parameter calls are `AdrOf` and `ReadOpt`. The `ReadOpt` call allows you to obtain option code and argument data associated with a certain *opt* parameter. The `AdrOf` call enables you to obtain a byte address that can be placed in the byte address field of a data vector.

The following diagrams are provided to illustrate the general form of the *opt* parameter after it has been initialized with the special NetIPC call `InitOpt`. In Figure 5-10, the following portions of the *opt* parameter are:

- `OPTLENGTH` represents the combined length of the `OPTARGUMENTS` and `DATA` portions of the *opt* parameter:

```
OPTLENGTH = 8 * OPTNUMARGUMENTS + DATA
```

OPTLENGTH takes up two bytes in the *opt* parameter.

- OPTNUMARGUMENTS represents the number of arguments or entries placed in the parameter. OPTNUMARGUMENTS takes up two bytes.
- OPTARGUMENTS is an area containing the arguments themselves; each argument is 8 bytes in length. Figure 5-11 illustrates its structure.
- DATA is where the data associated with the arguments is stored. The length of DATA is variable.

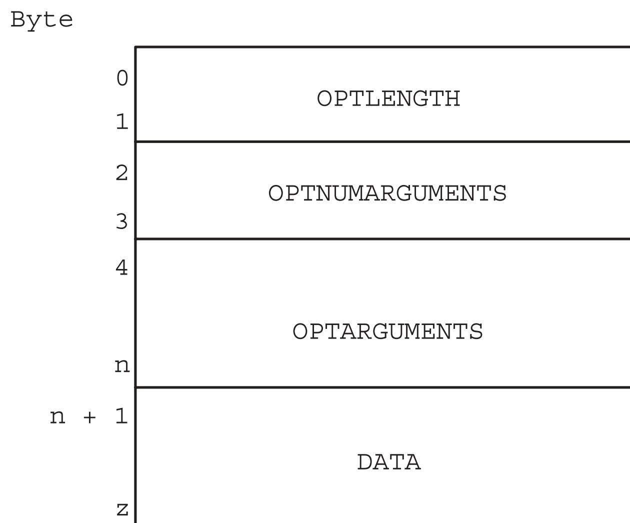


Figure 5-10. Opt Parameter Structure

The length (in bytes) of the *opt* parameter, including any data associated with it, can be determined with the following formula. This formula can also be used to determine the *opt* parameter length *before* coding your application.

```
total_length_of_opt := 4 + 8 * OPTNUMARGUMENTS + DATA;
```

OPTNUMARGUMENTS is the number of arguments that will be placed in the parameter and DATA is the length of the data associated with all of the arguments. The value of total_length_of_opt is the minimum size needed for the *opt* parameter. For most NetIPC programs, an average *opt* parameter is 60 to 100 bytes.

Figure 5-11 illustrates the structure of an *opt* parameter argument, OPTARGUMENTS:

- OPTIONCODE is the option code associated with the argument being added.
- OFFSET is the number of bytes offset into the *opt* record where the data associated with an argument is located.
- DATALENGTH is the length of the data associated with the argument.

This information is added to the *opt* parameter with the special NetIPC call AddOpt. (An example of adding an argument to the *opt* parameter is provided in the discussion of AddOpt later in this section.)

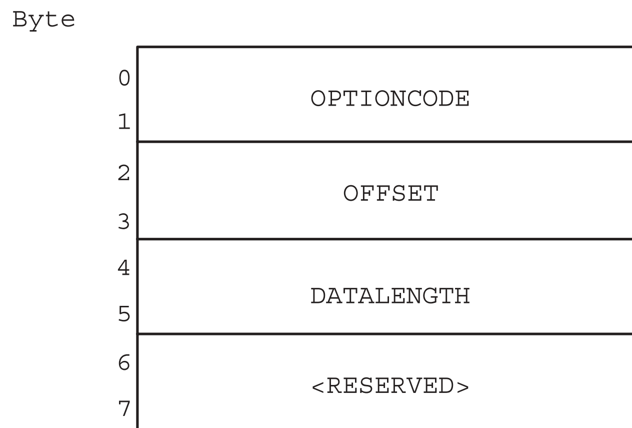


Figure 5-11. OPTARGUMENTS Structure

Data Parameter

The data parameters present in `IPCControl`, `IPCSend` and `IPCRecv` may reference data vectors or data buffers.

Unlike a data buffer, which is a structure containing actual data, a data vector is a structure that can *describe* several *data objects*. The description of each object consists of a byte address and a length. The byte address describes where the object is located and the length indicates how much data the object contains. Any kind of data object (arrays, portions of arrays, records, simple variables, etc.) can be described by a data vector.

When a data vector is used to identify data to be sent, it describes where the data is located. This is referred to as a *gathered write*. When a data vector is used to identify data to be received, it describes where the data is to be placed. This is referred to as a *scattered read*.

Using data vectors may be more efficient than using data buffers in certain circumstances. For example, a process that sends data from several different buffers must call `IPCSend` several times, or copy the data into a packing buffer prior to sending it, if its *`data`* parameter is a data buffer. However, if its *`data`* parameter is a data vector, the process may describe all of the buffers in the *`data`* parameter and transfer it using one `IPCSend` call.

Figure 5-12 is an example of a data vector and the data objects that it represents. The data vector describes the characters “HERE IS THE DATA.”

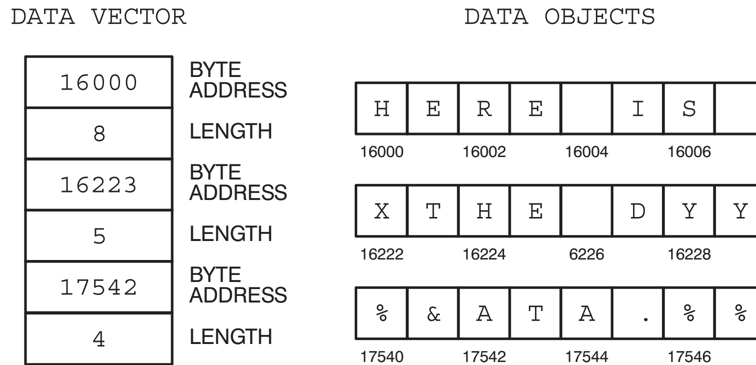


Figure 5-12. Vectored Data

Note Because neither Pascal/1000 or FORTRAN permit manipulation of byte addresses, a special routine, `AdrOf`, is provided to allow you to construct data vectors. `AdrOf` is described in "Special NetIPC Calls" at the end of this section.

Because NetIPC uses 16-bit addressing, NETIPC (`IPCSend`, `IPCRecv`, `IPCControl`) cannot access data with 32-bit addressing. Therefore, data in EMA (Extended Memory Area) cannot be accessed directly.

To access data in EMA, you should copy data from EMA to your local area, then access it with the NetIPC call.

When a data parameter refers to a data vector, the length of the data parameter (usually called *`dlen`*) refers to the *length of the structure containing the vector*. For example, if an `IPCSend` call were to reference the data vector in Figure 5-12 above, its *`dlen`* parameter would be 12 bytes. Each byte address and length totals 4 bytes; hence, each pointer (byte address) to a data object is 2 bytes long. There are three sets of byte addresses and lengths. Therefore, 4 * 3 = 12. Each length in a data vector must be greater than or equal to zero.

Type Coercion

A single data parameter can be used to represent either vectored or unvectored data in a Pascal/1000 program if type coercion is performed. This is useful when both vectored and unvectored data will be referenced. The following is an example of type coercion with a data parameter named `data_buffer`.

```
type vectored_array = array [0..10] of data_area
data_area           = record
                           location  : int16;
                           length    : int16;
                      end;

type byte_array      = packed array [0..64] of byte;
data_buffer          = record
                           case boolean of
                           true   : (data    : byte_array);
                           false  : (vect    : vectored_array);
                      end;
```

In the above example, the `vectored_array` type would be referenced by specifying `data_buffer.vect`. The `byte_array` type (for unvectored data) would be referenced by specifying `data_buffer.data`.

Note Since the data location descriptors contain machine-specific information, code using the vectored option may not be portable to other machines.

Result Parameter

Every NetIPC call has a *`result`* parameter. If an error occurs when a program uses a NetIPC call, an error code is returned to this parameter. The *NS-ARPA/1000 Error Message and Recovery Manual* lists and explains the NetIPC error codes.

Socketname Parameter

The NetIPC calls `IPCName`, `IPCNamErase`, `IPCLookUp`, `IPCGive`, `IPCGet`, and `IPCDest` require the use of names to identify either sockets or nodes.

A *socket name* (the *`socketname`* parameter) may be a maximum of 16 characters long and may consist of any ASCII character. Upper and lower case characters are not considered distinct. For example, the socket names “john” and “JOHN” are equivalent. Be careful with trailing blanks after a socket name; “john ” and “john” are not equivalent.

Nodename Parameter

A *node name* (the `nodename` parameter) refers typically to a remote node and has a hierarchical structure as follows:

```
node[.domain[.organization]]
```

The NS-ARPA node name syntax is described previously in “Node Names” in the “Introduction” section of this manual.

Note The socket name and node name parameters must be represented as arrays of ASCII characters in Pascal/1000 and an array of integers in FORTRAN. Do *not* use the Pascal string type or the FORTRAN 77 character string type definitions to describe these parameters.

Cross-System NetIPC

Network interprocess communication between an HP 1000 and other types This section explains the NetIPC calls that need to be considered for a *cross-system* application. Cross-system means that two different types of computer systems are communicating with one another. Cross-system NetIPC is supported between the HP 1000 and other HP computer systems (HP 9000, HP 3000, and PC). The NetIPC programs running on these pairs of computer systems will be able to send and receive data.

Note NetIPC on the HP 3000 refers to both MPE V and MPE XL versions unless otherwise stated.

This section does *not* explain all the NetIPC calls on the other HP computers. For this information, refer to the appropriate manual for that system.

Before reading this "Cross-System NetIPC" section, you must have a good understanding of the NetIPC calls. Review the remaining sections on the calls before the reading this section. For an example of programs that will communicate with similar programs included as samples with HP 9000 or HP 3000, refer to the "Client-Server Program Examples" subsection later in this section.

Many NetIPC calls on HP 1000, HP 9000, HP 3000, and the PC are different even if they have the same name. To understand the details and differences of each call on another system, you must read the corresponding NetIPC documentation for that system.

The "Porting NetIPC Programs" appendix in this manual describes programming considerations when porting HP 1000 NetIPC programs to run under HP 9000 and vice versa. This appendix also summarizes the differences among the NetIPC calls between HP 1000 and HP 9000. Porting refers to the process of moving a set of programs from one type of computer system to another. For example, if you have NetIPC programs currently executing on an HP 1000 system and you are migrating to an HP 9000, you may want to port those programs to the HP 9000. When porting programs, you will encounter programming language differences as well as NetIPC differences. Refer to the language manuals for information on differences among the programming languages.

Local NetIPC Calls

Not all of the NetIPC calls affect cross-system NetIPC communication. There are two categories of calls when considering cross-system NetIPC communication—local and remote. Calls made for the local process do not directly affect the remote process. The local NetIPC calls are used to set up or prepare the local node for interprocess communication with the remote node. That is, the resulting impact of the local calls is only to the local node. There is no information or action that affects the remote node. This is true whether or not the remote node is another HP 1000 or not. Table 5-4 lists the NetIPC calls affecting the local process.

Table 5-4. NetIPC Calls Affecting The Local Process

HP 1000	HP 9000	HP 3000	PC
Addopt	addopt()	ADDOPT	AddOpt
Adrof	Not implemented	Not implemented	Not implemented
Not implemented	Not implemented	Not implemented	ConvertNetworkLong
Not implemented	Not implemented	Not implemented	ConvertNetworkShort
InitOpt	initopt()	INITOPT	InitOpt
Not implemented	Not implemented	IPCCHECK	Not implemented
IPCControl	ipccontrol()	IPCCONTROL	IPCControl
IPCCreate	ipccreate()	IPCCREATE	IPCCreate
Not implemented	Not implemented	IPCERRMSG	Not implemented
IPCGet	Not implemented	IPCGET	Not implemented
IPCGive	Not implemented	IPCGIVE	Not implemented
IPCName	ipcname()	IPCNAME	Not implemented
IPCNamerase	ipcnamerase()	IPCNAMERASE	Not implemented
IPCSelect	ipcselect()	Not implemented	Not implemented
Not implemented	Not implemented	Not implemented	IPCWait
Not implemented	optoverhead()	OPTOVERHEAD	OptOverhead
ReadOpt	readopt()	READOPT (NetIPC-3000/V only)	ReadOpt

The calls listed in Table 5-4 affect local processes only and will therefore have no adverse affects if used in a program communicating with an unlike system (for example, an HP 3000 program communicating with an HP 1000 program). However, keep in mind that the calls (even those of the same name) differ from system type to system type. The following are some local call differences to be aware of:

- *Maximum number of sockets*. The maximum number of socket descriptors owned by an HP 3000 process at any given time is 64; on the HP 1000 the maximum is 32; on the HP 9000 the maximum is 60 (including file descriptors); on the PC the maximum is 21. This number includes both call socket and virtual circuit socket descriptors.

- *IPCControl parameters*. The `IPCControl` call supports different set of request codes on different system types. Refer to the NetIPC documentation for a particular system (this manual is for the HP 1000 only) for a full description of the request codes available on that system.

- *Path report descriptors*. On the HP 9000 and HP 3000, path report descriptors are called destination descriptors. Both types of descriptors are used in the same way. They contain addressing information that is used by a NetIPC process to direct requests to a certain call socket at a certain node.

- *Manipulation of descriptors*. The HP 9000 implementation of NetIPC allow you to manipulate call socket and destination descriptors with the `IPCName` and `IPCNamerase` calls. The `IPCName` and `IPCNamerase` calls on the HP 1000 manipulate only call socket descriptors. The HP 1000 also allows you to manipulate any socket descriptors (call socket, VC socket, and path report descriptors) with the `IPCGive` and `IPCGet` calls. On the HP 3000, `IPCGIVE` and `IPCGET` calls can be used to manipulate call socket and VC socket descriptors, but cannot be used to manipulate destination descriptors.
- *Asynchronous I/O*. The HP 9000 and HP 1000 NetIPC implementations utilize the `IPCSelect` call to perform asynchronous I/O. The HP 3000 NetIPC implementation utilizes the MPE intrinsics, `IOWAIT` and `IODONTWAIT`. PC NetIPC uses `IPCWait`.
- *Call sockets*. On the PC, call sockets are called source sockets and call socket descriptors are called source socket descriptors. Both sets of terms are used in the same way.

Note There are many additional differences between NetIPC calls for the HP 1000 and other types of HP computers. In addition, NetIPC calls with the same names may return different error codes depending. Refer to the system's corresponding NetIPC manual for information to determine the differences and for a list of error codes.

Remote NetIPC Calls

Unlike local NetIPC calls, remote NetIPC calls affect the peer process at the remote node. Table 5-5 lists the NetIPC calls that affect the remote process.

Table 5-5. NetIPC Calls Affecting the Remote Process

HP 1000	HP 9000	HP 3000	PC
IPCConnect	ipcconnect	IPCCONNECT	IPCConnect
IPCDest	ipcdest	IPCDEST	IPCDest
IPCLookUp	ipclookup	IPCLOOKUP	Not implemented
IPCRecv	ipcrecv	IPCRECV	IPCRecv
IPCRecvCn	ipcrecvcn	IPCRECVCN	IPCRecvCn
IPCSend	ipcsend	IPCSEND	IPCSend
IPCShutDown	ipcshutdown	IPCSHUTDOWN	IPCShutDown

HP 1000 to HP 9000 NetIPC

The NetIPC calls affecting cross-system communication with the remote process have the following differences: different send and receive sizes, range of permitted TCP protocol addresses for users, checksumming, and socket sharing. Table 5-6 lists the NetIPC calls affecting the remote process and summarizes the cross-system considerations.

Table 5-6. Cross-System NetIPC Calls (HP 1000—HP 9000)

NetIPC Call	Cross-System Considerations for HP 1000—HP 9000
`IPCConnect`	*Checksumming*—When the `ipcconnect()` call is executed on the HP 9000 node, then checksumming is always enabled for the HP 9000-to-HP 1000 connection. *Send and Receive sizes*—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.
`IPCCreate` `IPCDest`	*TCP Protocol Address*—The HP 1000 and HP 9000 implementations of `IPCCreate` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process.
`IPCLookup`	No differences that affect cross-system operations.
`IPCRecv`	*Receive size*—The HP 1000 receive size range is 1 to 8,000 bytes. The HP 9000 receive size range is 1 to 32,767 bytes. Although the range sizes that can be specified in the *`dlen`* parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.
`IPCRecvCn`	*Checksumming*—When the `ipcrecvcn()` call is executed on the HP 9000 node, checksumming is always enabled for the HP 9000-to-HP 1000 connection. *Send and Receive sizes*—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.
`IPCSend`	*Send size*—The HP 1000 send size range is 1 to 8,000 bytes. The HP 9000 send size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.
`IPCShutDown`	*Socket Shut Down*—The shutdown procedure for both HP 1000 and HP 9000 processes is identical except for shared sockets on HP 9000. Shared sockets are destroyed only when the descriptor being released is the sole descriptor for that socket. Therefore, the HP 9000 process may take longer to close the connection than expected.

Note There are many additional differences between remote NetIPC calls for the HP 9000 and HP 1000 systems. However, these differences should not affect the local node only. Refer to the "Porting NetIPC Programs" appendix in this manual for a summary of differences between HP 9000 and HP 1000 NetIPC implementations.

HP 1000 to HP 3000 NetIPC

The NetIPC calls affecting cross-system communication with the remote process have the following differences: different send and receive sizes, range of permitted TCP protocol addresses for users, and checksumming. Table 5-7 lists the NetIPC calls affecting the remote process and summarizes the cross-system considerations.

Table 5-7. Cross-System NetIPC Calls (HP 1000—HP 3000)

NetIPC Call	Cross-System Considerations for HP 1000—HP 3000
`IPCConnect`	*Checksumming*—TCP checksumming will be enabled for both sides of the connection if it is enabled by either side. On both the HP 1000 and HP 3000, checksumming can be enabled by setting bit 22 in the *`flags`* parameter. On the HP 3000, this bit can be used to override the checksumming decision made during the network transport configuration for this particular process. *Send and Receive sizes*—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 3000 send and receive size range is 1 to 30,000 bytes. Although the ranges are different, you must specify a send size within the correct range for the respective system; otherwise, an error will occur. For example, if the HP 3000 sends 16,000 bytes, the HP 1000 node can call `IPCRecv` twice, receiving 8000 bytes the first time and the second 8000 bytes the second time. Two processes should already understand how much and what kind of data they expect to send and receive whether or not they are cross-system processes. Note that the default send and receive sizes differ on the HP 1000 and HP 3000. On the HP 1000, the default send and receive size is 100 bytes. On the HP 3000, the default send and receive size is less than or equal to 1024 bytes.
`IPCCreate` `IPCDest`	*TCP Protocol Address*—The recommended range of TCP addresses for user applications is from 30767 to 32767 decimal (74057 to 77777 octal) for both the HP 3000 and HP 1000. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process. However, on NS3000/XL Release 1.1 these calls are not available to a non-privileged user.
`IPCLookup`	No differences that affect cross-system operations.
`IPCRecv`	*Receive size* (*`dlen`* *parameter*)—Range for the HP 3000 is 1 to 30,000 bytes. Range for the HP 1000 is 1 to 8,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

NetIPC Call	Cross-System Considerations for HP 1000—HP 3000
	Data wait flag—The HP 1000 `IPCRecv` call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the *`dlen`* parameter has been received. This flag is not available on the HP 3000, meaning that the call may complete before all the data is received. However, the HP 3000 `IPCRecv` supports other flags such as the “more data” and “destroy data” flags. Refer to the description of `IPCRecv` in the NetIPC3000 manual for detailed information.
`IPCRecvCn`	*Send and Receive sizes*—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 3000 send and receive size range is 1 to 30,000 bytes. Although the ranges are different, you must specify a send size within the correct range for the respective system; otherwise, an error will occur. For example, if the HP 3000 sends 16,000 bytes, the HP 1000 node can call `IPCRecv` twice, receiving 8000 bytes the first time and the second 8000 bytes the second time. Two processes should already understand how much and what kind of data they expect to send and receive whether or not they are cross-system processes. Note that the default send and receive sizes differ on the HP 1000 and HP 3000. On the HP 1000, the default send and receive size is 100 bytes. On the HP 3000, the default send and receive size is less than or equal to 1024 bytes.
`IPCSend`	*Send size*—The HP 1000 send size range is 1 to 8,000 bytes. The HP 3000 send size range is 1 to 30,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system. *Urgent Data*—The HP 3000 supports an “urgent data” option in the `opt` parameter. If this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 1000.
`IPCShutDown`	*Socket Shut Down*—The shutdown procedure for both HP 1000 and HP 3000 processes is the same, except that the *`graceful release`* flag is not available on the HP 1000. Do *not* set the graceful release flag (`flags 17`) on the HP 3000. Otherwise, the HP 1000 will not perform a normal shutdown. If the HP 3000 process does set the graceful release flag, the HP 1000 `IPCRecv` call will get a NetIPC error 68 (no more data) instead of a NetIPC error 64 (connection aborted by peer). The HP 1000 process should handle the error 68 as if it were an error 64. After receiving a NetIPC error 68, subsequent `IPCRecv` calls will get a NetIPC error 109 (remote connection has already graceful released the socket), because there is no more data available.

HP 1000 to PC NetIPC

The NetIPC calls affecting cross-system communication with the remote process have the following differences: different send and receive sizes, range of permitted TCP protocol addresses for users, and checksumming. Table 5-8 lists the NetIPC calls affecting the remote process and summarize the cross-system considerations.

Table 5-8. Cross-System NetIPC Calls (HP 1000—PC)

NetIPC Call	Cross-System Considerations for HP 1000—PC
`IPCConnect`	*Checksumming*—With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 1000 requires it, the TCP checksum is in effect on both sides of the connection. *Send and Receive sizes*—The HP 1000 send and receive size range is 1 to 8,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system; otherwise, an error will occur. For example, if a PC sends a 60,000 byte buffer, the HP 1000 process may get all the data by posting seven IPCRecv functions of up to 8,000 bytes until all the data has been received.
`IPCCreate` `IPCDest`	*TCP Protocol Address*—The HP 1000 and PC implementations of `IPCCreate` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process.
`IPCRecv`	*Receive size*—The HP 1000 receive size range is 1 to 8,000 bytes. The HP 1000 enables you to specify the maximum receive size of the data buffer through the *`opt`* array in the `IPCConnect` call. This determines what the maximum value for *`dlen`* can be for any `IPCRecv` call. PC NetIPC has no option array defined for `IPCConnect`. This does not affect cross-system communication. The maximum receive size of the data in the buffer on the HP 1000 will determine the receive size buffer on the PC. *Data wait flag*—The HP 1000 `IPCRecv` call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the *`dlen`* parameter has been received. This flag is not available on the PC, meaning that the call may complete before all the data is received.
`IPCRecvCn`	*Checksumming*—With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 1000 requires it, the TCP checksum will be in effect on both sides of the connection. *Send and Receive sizes*—The HP 1000 send and receive size range is 1 to 8,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system. For example, if a PC sends 16,000 bytes, the HP 1000 computer can call `IPCRecv` twice, receiving 8,000 bytes the first time and the second 8,000 bytes the second time.
`IPCSend`	*Send size*—The HP 1000 send size range is 1 to 8,000 bytes. The HP 1000 enables you to specify the maximum send size of the data buffer through the *`opt`* array in the `IPCConnect` call. This determines what the maximum value for *`dlen`* can be for any `IPCSend` call. PC NetIPC has no option array defined for `IPCConnect`. This does not affect cross-system communication. The maximum send size of the data in the buffer on the HP 1000 will determine the send size buffer on the PC.

Loading NetIPC Programs

HP 1000 NetIPC programs should be compiled and linked as CDS programs. Refer to the *RTE-A Programmer's Reference Manual* and *RTE-A Link Manual* for more information on CDS programs. After the program is linked, an RTE executable file (type 6) is ready to be scheduled.

Process Scheduling

NetIPC itself does not include a call to schedule a peer process. The method used to schedule a remote NetIPC process depends on the types of systems involved. The following paragraphs discuss these methods.

Remote HP 1000 NetIPC Process

There are at least six different ways (listed below) to schedule a remote HP 1000 NetIPC process from another HP 1000 node. A remote HP 1000 NetIPC process must be ready to execute by being an RTE type 6 file.

- *Internet Network Services Daemon (INETD)*. INETD can accept incoming connections for user written NETIPC server programs and schedule a copy of the server program for each connection. Refer to the INETD section in the *NS-ARPA/1000 Generation and Initialization Manual*.

- *Remote Process Management (RPM)*. The `RPMCreate` call programmatically schedules a program. Refer to the section "Remote Process Management" later in this manual. RPM is an NS Common Service.

- *Program-to-Program communication (PTOP)*. The `POPEN` call programmatically schedules a program. PTOP is a DS/1000-IV Compatible Service and is described in the *DS/1000-IV Compatible Services Reference Manual*.

- *Distributed EXEC (DEXEC)*. One of the DEXEC scheduling calls, such as DEXEC 9, 10, 12, 23, 24, programmatically schedules a program. DEXEC is a DS/1000-IV Compatible Service and is described in the *DS/1000-IV Compatible Services Reference Manual*.

- *REMAT*. The REMAT `QU` (queue schedule a program without wait) command interactively schedules a program. REMAT is a DS/1000-IV Compatible Service and is described in the *DS/1000-IV Compatible Services Reference Manual*.

- *TELNET virtual terminal*. Logon remotely with TELNET and use the RTE `XQ` (schedule a program without wait) command to interactively schedule a program. TELNET is an ARPA Service. Refer to the "TELNET" section in this manual.

- *RTE WELCOME file*. The WELCOME file can have RTE run commands to schedule programs after system boot up. Refer to the *RTE-A System Generation and Installation Manual* for information about booting up the RTE system and about the WELCOME file.

You cannot use any of the above NS-ARPA and DS/1000-IV compatible services to schedule a remote HP 1000 process from a non-HP 1000 node. These services are not provided with cross-system support.

Remote HP 1000 processes that are to work with non-HP 1000 processes can be manually started or can be programs that are started at system start up.

- To manually start up a NetIPC program, simply logon to the HP 1000 system and run the NetIPC program with the RTE `XQ` (run program without wait) command.
- To have the NetIPC program execute at system start up, put the RTE `XQ` command in the WELCOME file.

The `XQ` command is explained in the *RTE-A User's Manual*.

Remote HP 9000 NetIPC Process

Remote HP 9000 processes can be manually started or can be scheduled by daemons that are started at system start up. In HP-UX a daemon is a process that runs continuously and usually performs system administrative tasks. Although a daemon runs continuously, it performs actions upon an event happening or at designated times.

To manually start up a NetIPC program, simply logon to the HP 9000 system and run the NetIPC program. HP recommends that you write a NetIPC daemon to schedule your NetIPC programs. You can start the daemon at system start up by placing it in your `/etc/netlinkrc` file. Refer to the HP 9000 LAN software installation documentation for more information about this file and system start up.

Remote HP 3000 NetIPC Process

To manually start up an HP 3000 NetIPC program, log on to the HP 3000 and run the NetIPC program with the `RUN` command.

You can schedule the program to start at a particular time by writing a job file to execute the program, and then including time and date parameters in the `:STREAM` command that executes the job file.

Remote PC NetIPC Process

To manually start up a PC NetIPC program, enter the NetIPC program name at the MS-DOS* prompt.

To execute from within MS-Windows, copy the NetIPC program files to your Windows directory and double click with the mouse on the executable file.

*MS-DOS is a U.S. registered trademark of Microsoft Corporation.

NetIPC Syntax Conventions

The syntax provided in the following pages for each NetIPC call is meant to illustrate a Pascal procedure call statement. Parameters that are either output, or both input and output, are underlined in the syntax diagram. All other parameters are input parameters. Please refer to the sample programs for examples of Pascal and FORTRAN variable declarations.

All NetIPC call parameters are required. You may pass a zero in some parameters in order to obtain a default value.

Note Do *not* use the Pascal/1000 `IMPORT` statement in place of the `EXTERNAL` statement for NetIPC procedures. It is not supported with NetIPC.

IPCCONNECT

Requests a connection to another process.

Syntax

```
IPCCONNECT(calldesc,pathdesc,flags,opt,vcdesc,result)
```

Parameters

calldesc — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Call socket descriptor. Refers to a call socket owned by the calling process.

pathdesc — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Path report descriptor. Refers to the path report which indicates the location of the destination call socket (this is the call socket to which the connection request will be sent). A path report descriptor can be obtained by calling `IPCLookUp` or `IPCGet`.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. Refer to “Flags Parameter” for more information on the structure of this parameter. The following option is defined for this call:

- `flags [22]`—CHECKSUMMING (input). When set, this flag causes TCP to enable checksumming. However, *not* setting this bit does not ensure that checksumming will not occur. TCP checksum will always be performed if: the peer process calls `IPCRecvCn` with the checksumming bit set. TCP checksum is performed in addition to data link checksum. If TCP performs checksumming, increased overhead is required and real-time integrity cannot be guaranteed.

opt — *Byte array (Pascal); Integer array (FORTRAN), by reference*. An array of options and associated information. Refer to “Opt Parameter” for information on the structure and use of this parameter. The following options are defined for this call:

- maximum send size (*optioncode* = 3, *datalength* = 2). A two-byte integer that specifies the maximum number of bytes you expect to send with a single `IPCSend` call on this connection. *Range*: 1 to 8,000 bytes. *Default*: 100 bytes. If this option is not specified, `IPCSend` will return errors if a call attempts to send greater than 100 bytes.

- maximum receive size (*optioncode* = 4, *datalength* = 2). A two-byte integer that specifies the maximum number of bytes you expect to receive with a single IPCRecv call on this connection. *Range*: 1 to 8,000 bytes. *Default*: 100 bytes. If this option is not specified, IPCRecv will return errors if a call attempts to receive greater than 100 bytes.

vcdesc — *32-bit integer, by reference*. VC socket descriptor. Refers to a VC socket that is the endpoint of the virtual circuit connection at this node. May be used in subsequent NetIPC calls to reference the connection.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

The IPCConnect call is used to initiate a virtual circuit on which data may be sent and received. Compared to the telephone system, IPCConnect is similar to dialing a telephone but not waiting for an answer. Pursuing this analogy, the *pathdesc* parameter is similar to a telephone number and the *calldesc* parameter references a call socket or "telephone." When IPCConnect is called from your process, it uses its "telephone" (call socket descriptor) and "telephone number" (path report descriptor) to "dial," or initiate a connection with, another process. Socket descriptors can be obtained with either an IPCCreate or IPCGet call. Path report descriptors can be obtained with IPCLookUp and IPCGet.

IPCConnect could be said to "not wait for an answer" when it is called because it reports only whether a virtual circuit has been *initiated*, not whether it was successfully established. If the connection is successfully initiated, IPCConnect will return a VC socket descriptor in its *vcdesc* parameter. This VC socket descriptor refers to a VC socket that is the endpoint of the virtual circuit at the local node. The VC socket descriptor identifies the connection in much the same way that switch buttons are used to identify conversations on a multi-extension telephone.

Establishing a virtual circuit with NetIPC calls is a two-step process:

- First, IPCConnect is called to request a connection;
- Second, IPCRecv is called to find out if a connection initiated with IPCConnect was successfully established.

Compared to the telephone system, this process is similar to dialing a telephone and then checking to see if someone answered. Using this two-step process, processes can initiate several connections simultaneously without waiting for each one to complete.

IPCConnect's *opt* parameter enables you to specify the maximum number of bytes you expect to send and receive on the connection. The default for both sending and receiving is 100 bytes.

A process may own a maximum of 32 call socket, VC socket, and path report descriptors. IPCConnect will return an error if a process attempts to exceed this limit.

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

Checksumming—When the `ipcconnect()` call is executed on the HP 9000 node, then checksumming is always enabled for the HP 9000-to-HP 1000 connection.

Send and Receive sizes—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 9000 send and receive size range is 1 to 32,000 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

Checksumming—TCP checksumming will be enabled for both sides of the connection if it is enabled by either side. On both the HP 1000 and HP 3000, checksumming can be enabled by setting bit 22 in the *`flags`* parameter. On the HP 3000, this bit can be used to override the checksumming decision made during network transport configuration for this particular process.

Send and Receive sizes—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 3000 send and receive size range is 1 to 30,000 bytes. Although the ranges are different, you must specify a buffer size within the correct range for the respective system; otherwise, an error will occur.

Note that the default send and receive sizes differ on the HP 1000 and HP 3000. On the HP 1000, the default send and receive size is 100 bytes. On the HP 3000, the default send and receive size is less than or equal to 1024 bytes.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and the PC:

Checksumming—With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 1000 requires it, the TCP checksum is in effect on both sides of the connection.

Send and Receive sizes—The HP 1000 send and receive size range is 1 to 8,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system; otherwise, an error will occur. For example, if a PC sends a 60,000 byte buffer, the HP 1000 process may get all the data by posting seven IPCRecv calls of up to 8,000 bytes until all the data has been received.

IPCCONTROL

Performs special operations on sockets.

Syntax

```
IPCCONTROL(descriptor,request,wrtdata,wlen,readdata,rlen,
          flags,result)
```

Parameters

descriptor — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The descriptor that refers to the socket to be manipulated. May be a call socket descriptor or VC socket descriptor depending on the request code specified in the request parameter.

request — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Request code. Defines which operation is to be performed. May be one of the following:

- 1 = Place the socket referenced in the *descriptor* parameter in asynchronous mode. For IPCSend and IPCRecv calls, this is the VC socket described by the VC socket descriptor in the *vcdesc* parameter. For IPCRecvCn, it is the call socket described by the call socket descriptor in the *calldesc* parameter. (Refer to "Synchronous and Asynchronous Socket Modes" at the beginning of this section for more information on asynchronous I/O.)

- 2 = Place the socket referenced in the *descriptor* parameter in synchronous mode. For IPCSend and IPCRecv calls this is the VC socket described by the VC socket descriptor in the *vcdesc* parameter. For IPCRecvCn, it is the call socket described by the call socket descriptor in the *calldesc* parameter. (Refer to "Synchronous and Asynchronous Socket Modes" at the beginning of this section for more information on synchronous I/O.)

- 3 = Change the referenced socket's synchronous timeout. The default timeout value is 60 seconds. For IPCSend and IPCRecv calls, this is the VC socket described by the VC socket descriptor in the *vcdesc* parameter. For IPCRecvCn, it is the call socket described by the call socket descriptor in the *calldesc* parameter. The timeout value is given in tenths of seconds. (For example, a value of 1200 would indicate 120 seconds.) The new timeout value must be placed in the *wrtdata* parameter. The timeout value must be in the range of zero to 32767. Negative values have no meaning and will result in error. A value of zero sets the timeout to infinity. The timeout will not be reset if the referenced socket is switched to asynchronous mode and then back to synchronous mode.

- 1000 = Change the read threshold of the VC socket referenced in *descriptor* parameter. (Read thresholds are one byte by default.) The *descriptor* parameter must reference a VC socket descriptor. The new read threshold value must be placed in the *wrtdata* parameter. Refer to "Asynchronous and Synchronous Socket Modes" at the beginning of this section for more information on read thresholds.
- 1001 = Change the write threshold of the VC socket referenced by the *descriptor* parameter. (Write thresholds are one byte by default.) The *descriptor* parameter must reference a VC socket descriptor. The new write threshold value must be placed in the *wrtdata* parameter. Refer to "Asynchronous and Synchronous Socket Modes" at the beginning of this section for more information on write thresholds.

wrtdata — *16-bit integer, by reference*. A data buffer or data vector used to pass timeout and threshold information. If a *request* of 3, 1000, or 1001 is specified, the *wrtdata* and *wlen* parameters are required. Refer to "Data Parameter" for more information on data buffers and data vectors.

wlen — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Length in bytes of the *wrtdata* parameter. Must be set to 2 bytes.

readdata — *Array, by reference*. This parameter is reserved for future use.

rlen (input/output) — *32-bit integer, by reference*. This parameter is reserved for future use.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). Refer to "Flags Parameter" for more information on the structure of this parameter.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

The `IPCControl` call is used to manipulate sockets in special ways. The type of request is specified by placing a certain request code in the *request*s parameter. Although all of the request types require the *descriptor*, *request*s and *result* parameters, some of the parameters are meaningless for certain requests. If request code 3, 1000 or 1001 is specified, the *wrtdata* and *wlen* parameters are required.

Refer to "Synchronous and Asynchronous Socket Modes" at the beginning of this section for a detailed discussion of synchronous mode, asynchronous mode, synchronous timeouts and read and write thresholds.

IPCCREATE

Creates a call socket.

Syntax

```
IPCCREATE(socketkind,protocol,flags,opt,calldesc,result)
```

Parameters

socketkind — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Indicates the type of socket to be created. Must be 3 to indicate a call socket. (Other values are reserved for future use.)

Default: If zero is specified, a call socket will be created.

protocol — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Indicates the protocol module that the calling process wishes to access. Must be 4 to indicate Transmission Control Protocol (TCP). (Other values are reserved for future use.)

Default: If zero is specified, TCP will always be chosen for call sockets.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). Refer to "Flags Parameter" for more information on the structure of this parameter.

opt — *Byte array (Pascal); Integer array (FORTRAN), by reference*. An array of options and associated information. Refer to "Opt Parameter" for more information on the structure and use of this parameter. The following options are defined for this call:

- maximum connection requests backlog (*optioncode* = 6, *datalength* = 2). A two-byte integer that specifies the maximum number of unreceived connection requests that may be queued to a call socket. The value can be from 0 to 10. *Default*: Three requests. (NOTE: A queue limit of three may be too few if many processes attempt to initiate connections to the call socket simultaneously. If this occurs, some connection requests may be ignored.)
- protocol address (*optioncode* = 128, *datalength* = 2). A two-byte integer that specifies a TCP protocol address to be used by the newly created call socket. The valid range for IPC address is 1 to 32767. If this option is not specified, NetIPC will dynamically allocate an address. *Recommended Range*: The recommended range of TCP addresses for user applications is from 30767 to 32767 decimal.

calldesc *32-bit integer, by reference*. Call socket descriptor. Refers to the newly created call socket.

result *32-bit integer, by reference*. The returned error code; zero if no error.

Discussion

IPCCreate is used to create a call socket or "telephone" which will be used by subsequent NetIPC calls to establish a virtual circuit connection between two or more processes. When invoked successfully, IPCCreate returns a call socket descriptor, or "telephone number," that refers to the newly created call socket. Invoking IPCCreate is a prerequisite for establishing a connection.

A process may own a maximum of 32 call socket, VC socket, and path report descriptors. IPCCreate will return an error if a process attempts to exceed this limit.

An option code of 128 can be used to create a call socket with a specific protocol address. If this protocol address is known to the process's peer, the peer process can call IPCDest with this address (also known as a "well-known address"). The call is made in IPCDest's *protoaddr* parameter so that it may obtain a destination descriptor that references this call socket. Refer to the explanation of IPCDest for more information.

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

TCP Protocol Address—The HP 1000 and HP 9000 implementations of IPCCreate support different ranges of permitted TCP protocol addresses that can be specified in the *opt* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The IPCDest call uses the TCP protocol address specified in IPCCreate on the remote process.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

TCP Protocol Address—The recommended range of TCP addresses for user applications is from 30767 to 32767 decimal (74057 to 77777 octal) for both the HP 3000 and HP 1000. The IPCDest call uses the TCP protocol address specified in IPCCreate on the remote process. However, on NS3000/XL Release 1.1, a non-privileged user cannot specify a TCP address.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and the PC:

TCP Protocol Address—The HP 1000 and PC implementations of IPCCreate support different ranges of permitted TCP protocol addresses that can be specified in the *opt* parameter. The PC supports any value. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The IPCDest call uses the TCP protocol address specified in IPCCreate on the remote process.

IPCDEST

Creates a path report descriptor.

Syntax

```
IPCDEST(socketkind,nodename,nodelen,protocol,protoaddr,protolen,
        flags,opt,pathdesc,result)
```

Parameters

socketkind — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Defines the type of socket. Must be 3 to specify a call socket. Other values are reserved for future use.

nodename — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. A variable length array of ASCII characters identifying the node on which the path report descriptor is to be created. The syntax of the node name is *node*[.*domain*[.*organization*]], which is further described in "Node Names" of the "Introduction" section and in "Nodename Parameter" in this section.

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If the *nodelen* parameter is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the *nodename* parameter. If this parameter is set to zero, the *nodename* parameter is ignored and the node name defaults to the local node. A fully-qualified node name length may be 50 bytes long.

protocol — *32-bit integer, by value in Pascal, by reference in FORTRAN*. Defines the Transport Layer protocol to be used. Must be 4 to indicate the Transmission Control Protocol (TCP). Other values are reserved for future use.

protoaddr — integer array, by reference. A data buffer that contains a TCP protocol address. *Recommended Range*: The recommended range of TCP addresses for user applications is from 30767 to 32767 decimal.

protolen — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the protocol address. TCP protocol addresses are two bytes long.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). Refer to "Flags Parameter" for information on the structure of this parameter.

opt — Refer to "Opt Parameter" for information on the structure and use of this parameter. No options are defined for this call.

pathdesc — *32-bit integer, by reference*. Path report descriptor. Describes the location of named call socket. May be used in a subsequent IPCConnect call to establish a connection to another process.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

The IPCDest call creates a path report descriptor that the calling process can use to establish a connection to another process. This call is similar in function to IPCLookup because it returns a path report descriptor. However, because IPCDest allows you to specify a protocol address, it allows you to obtain a path report descriptor for a call socket with a *particular protocol address*. A call socket can be created with a particular protocol address by using the IPCCreate call with the PROTOCOL ADDRESS option.

The IPCDest call does not verify that the remote endpoint described by the input parameters exists. This evaluation is delayed until the path report descriptor is used in a subsequent IPCConnect call. Figure 5-13 shows how a a connection can be established using these two calls.

PROCESS A	PROCESS B
IPCCreate Create call socket "A" with protocol address "x" by using the **PROTOCOL ADDRESS** option.	IPCCreate Create call socket "B"
	IPCDest Obtain a path report descriptor for socket "A" by specifying protocol address "x" in the *protoaddr* parameter
	IPCConnect Request a connection to Process A by using the path report descriptor obtained by *IPCDest*.
IPCRecvCn Receive Process B's connection request	IPCRecv Confirm Virtual Circuit connection status.

Figure 5-13. Connection Established with IPCDest and IPCConnect

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

TCP Protocol Address—The HP 1000 and HP 9000 implementations of `IPCCreate` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

TCP Protocol Address—The recommended range of TCP addresses for user applications is from 30767 to 32767 decimal (74057 to 77777 octal) for both the HP 3000 and HP 1000. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process. However, on NS3000/XL Release 1.1, a non-privileged user cannot specify a TCP address.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and the PC:

TCP Protocol Address—The HP 1000 and PC implementations of `IPCCreate` support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. The PC supports any value. However, both implementations recommend that users specify TCP addresses in the range of 30767 to 32767 decimal for cross-system use. The `IPCDest` call uses the TCP protocol address specified in `IPCCreate` on the remote process.

IPCGET

Receives a descriptor that has been given away via `IPCGive`.

Syntax

IPCGET(*givenname*,*nlen*,*flags*,<u>*descriptor*</u>,<u>*result*</u>)

Parameters

givenname — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. An array containing the ASCII-coded socket name that was assigned to the descriptor when it was given away via a call to `IPCGive`. Upper and lower case characters are not considered distinct. Refer to "Socketname Parameter" for a detailed discussion of naming.

nlen — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The length in characters of *givenname*. Maximum length is 16 bytes.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). If `IPCGet` is called repeatedly, this field must be cleared before each successive call. Refer to "Flags Parameters" for more information on the structure of this parameter.

descriptor — *32-bit integer, by reference*. The descriptor that was given away via a call to `IPCGive`. May be a call socket descriptor, path report descriptor or VC socket descriptor.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

`IPCGet` is used to obtain ownership of a call socket descriptor, path report descriptor or VC socket descriptor that was given away by another process with an `IPCGive` call. The process that calls `IPCGet` must match the name assigned to the descriptor by `IPCGive`. The name can be obtained, and thus matched, using one of several methods. For example, the name could be:

- coded into both processes
- passed to the processes when they are scheduled
- placed in a well-known memory or file location
- sent inside data

A process may own a maximum of 32 call socket, VC socket, and path report descriptors. `IPCGet` will return an error if a process attempts to exceed this limit.

IPCGIVE

Gives up a descriptor so that another process may obtain it.

Syntax

```
IPCGIVE(descriptor,givenname,nlen,flags,result)
```

Parameters

descriptor
: *32-bit integer, by value in Pascal, by reference in FORTRAN*. The descriptor to be given up. May be a call socket descriptor, VC socket descriptor or path report descriptor.

givenname (input/output)
: *Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. An array containing the ASCII-coded socket name to be temporarily assigned to the specified descriptor. Upper and lower case characters are not considered distinct. NetIPC can also return a randomly generated, eight-character name to this parameter (see *nlen*). Refer to "Socketname Parameter" for a detailed discussion of naming.

nlen
: *32-bit integer, by value in Pascal, by reference in FORTRAN*. The length in characters of *givenname*. Maximum length is 16 bytes.

  Default: If zero is specified, NetIPC will generate a random eight byte name and return it in the *givenname* parameter. (The length eight is not returned through *nlen*.)

flags
: *32-bit integer, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). Refer to "Flags Parameter" for more information on the structure of this parameter.

result
: *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

A process can invoke `IPCGive` to give a call socket descriptor, VC socket descriptor, or path report descriptor that it owns to another process at the same node. For example, Process A at node X can give a VC socket descriptor to Process B, also at node X, so that Process B may use a connection Process A has previously established with Process C at node Z. Because Process B was "given" the endpoint of a previously established connection, it does not need to create its own call socket and engage in the NetIPC connection dialogue in order to exchange data with Process C.

When a process calls `IPCGive`, the referenced descriptor is given to NetIPC. The calling process releases ownership of the descriptor and loses all access to it. Another process can get the

released descriptor from NetIPC by calling `IPCGet` and supplying the proper name. If the calling process terminates before the descriptor is retrieved by another process via `IPCGet`, the descriptor is destroyed.

The name associated with the descriptor by `IPCGive` can be user-defined or randomly-generated by NetIPC. The name associated with a descriptor must be unique to your node (i.e., the same name cannot be simultaneously associated with two descriptors). For example, a descriptor associated with the name "John" will work on the first call to `IPCGive`, but a subsequent call with "John" will result in an error. A name can be used or reused only if it is not currently being used.

Data sent to a connection that has been given away is still queued on that connection. It is recommended that descriptors not be given away while they are in awkward states. For example, it may be confusing to give away a VC socket descriptor that refers to the endpoint of a virtual circuit that has been initiated with `IPCConnect`, but has not been fully established with `IPCRecv`.

You can associate a permanent name with a call socket descriptor by calling `IPCName`. Unlike `IPCGive`, this call is used when the calling process wants to retain access to the call socket, but would like another process to be able to reference it. Refer to the discussion of `IPCName` for more information.

IPCLOOKUP

Obtains a path report descriptor.

Syntax

```
IPCLOOKUP(socketname,nlen,nodename,nodelen,flags,pathdesc,protocol,
          socketkind,result)
```

Parameters

socketname — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference.* An array containing the ASCII-coded name of the call socket to be "looked up." Upper and lower case characters are not considered distinct. Refer to "Socketname Parameter" for a detailed discussion of naming.

nlen — *32-bit integer, by value in Pascal, by reference in FORTRAN.* The length of the socket name in characters. Maximum length is 16 characters.

nodename — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference.* A variable length array of ASCII characters identifying the node where the socket specified in the *socketname* parameter resides. The syntax of the node name is *node*[.*domain*[.*organization*]], which is further described in "Node Names" of the "Introduction" section and in "Nodename Parameter" in this section.

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If the *nodelen* parameter is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen — *32-bit integer, by value in Pascal, by reference in FORTRAN.* The length in bytes of the *nodename* parameter. If this parameter is zero (0), the *nodename* parameter is ignored and the node name defaults to the local node. A fully-qualified node name length may be 50 bytes long.

flags — *32-bit integer, by reference.* A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). Refer to "Flags Parameter" for more information on the structure of this parameter.

pathdesc — *32-bit integer, by reference.* Path report descriptor. Refers to the path report descriptor which indicates the location of the named call socket. May be used in subsequent NetIPC calls (`IPCConnect`, `IPCName`, `IPCGive`, etc.).

protocol — *32-bit integer, by reference.* Identifies the protocol module with which the "looked up" socket is associated. May be used in an `IPCCreate` call to create a call socket with the appropriate protocol binding.

socketkind — *32-bit integer, by reference*. Identifies the socket's type.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

The IPCLookUp call is used to gain access to a named call socket (refer to the IPCName call). When supplied with valid socket and node names, it looks up the call socket in the socket registry at the node specified in the *nodename* parameter and returns a path report descriptor that can be used by subsequent NetIPC calls to locate the call socket. When used in an IPCConnect call, for example, a path report descriptor can provide the information necessary to direct a connection request to the proper node and call socket and thus initiate a connection.

Compared to the telephone system, invoking IPCLookUp is analogous to calling directory assistance: the NetIPC process "calls the operator" (IPCLookUp) and gives him/her a "name" (*socketname*) and a "location" (*nodename* parameter). The node name can be a "city" (*node*), "state" (*domain*), and a "country" (*organization name*), just the "city" and the "state," only the "city," or no location at all. The omitted parts, or all, of the location will be defaulted. Once the name is found, the operator returns a "telephone number" (*pathdesc*) to the caller.

A process may own a maximum of 32 call socket, VC socket, and path report descriptors. IPCLookUp will return an error if a process attempts to exceed this limit.

IPCDest also obtains a path report descriptor by specifying a particular protocol address instead of a name. The advantage of using IPCLookUp is that names might be easier to remember and use. With IPCDest, the address must be unique, and other processes must cooperate and not use that same address.

Race Conditions

When a process attempts to look up a socket name in the appropriate socket registry, the name must be there or a "name not found" error (error code 37) will be returned to the calling process. When two processes are running concurrently, it may be difficult to ensure that a socket name is placed in the socket registry prior to being "looked up" by another process. This problem is referred to as a *race condition* because the two processes are "racing" to see which one will access the socket registry first. There are a few ways to avoid a race situation:

- The process that calls IPCLookUp can test for a "name not found" error (error code 37) in the call's *result* parameter. If this error is returned, the process can try again by entering a loop and repeating the IPCLookUp call for a specified number of times.
- The process that calls IPCLookUp can call EXEC 12 (time schedule a program) before calling IPCLookUp.
- The process that calls IPCName can name its call socket and then schedule the process that calls IPCLookUp. Remote Process Management (RPM), DEXEC, and PTOP include calls that allow you to programmatically schedule a process at a remote NS-ARPA/1000 node.

IPCNAME

Associates a name with a call socket descriptor.

Syntax

```
IPCNAME(descriptor,socketname,nlen,result)
```

Parameters

descriptor	*32-bit integer, by value in Pascal, by reference in FORTRAN*. The call socket descriptor to be named.
socketname (input/output)	*Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. An array containing the ASCII-coded socket name to be associated with the descriptor. Upper and lower case characters are considered equivalent. NetIPC can also return a randomly-generated name in this parameter (see `nlen`). Refer to "Socketname Parameter" for a detailed discussion of naming.
nlen	*32-bit integer, by value in Pascal, by reference in FORTRAN*. The length in characters of *socketname*. Maximum length is 16 characters. *Default*: If zero is specified, NetIPC will return a random, eight-byte name in the *socketname* parameter. The eight-byte length is not returned in the *nlen* parameter.
result	*32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

`IPCName` associates a name with the descriptor and adds this information to the local node's socket registry. Like a telephone directory that associates names with telephone numbers, the socket registry associates names with protocol addresses. NetIPC must be provided with these addresses before it can establish a connection to the corresponding call socket.

The name a process associates with its call socket descriptor must be known to its peer process so that the peer process may look up the name with an `IPCLookUp` call. This may be accomplished by hard-coding the name into both processes or by passing the name from one process to another.

The name associated with a descriptor can be user-defined or randomly generated by NetIPC and must be unique to your node (that is, it cannot be simultaneously associated with two descriptors.) For example, if a descriptor is assigned the name "Liz" with a call to `IPCName`, a subsequent call to `IPCName` with "Liz" will result in an error. You can ensure that the name you assign to a descriptor is unique by using the random name generating feature of `IPCName`. A name can be reused only if it is not currently being used. A descriptor, however, may be listed under multiple names.

Under most circumstances, `IPCName` should be called with a name and the call socket descriptor that refers to a call socket owned by the calling process. If the call completes successfully, the call socket will be listed in the socket registry at the local node. `IPCLookUp` can be called to "look up" the socket name in the local node's socket registry.

`IPCName` always enters its listings into the local node's socket registry. `IPCLookUp`, by contrast, can look up socket names at both the local node and at a remote node.

Note You cannot use `IPCName` to name VC sockets.

IPCNAMERASE

Deletes a name associated with a call socket descriptor.

Syntax

```
IPCNAMERASE(socketname,nlen,result)
```

Parameters

socketname — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. An array containing an ASCII-coded name that was previously associated with a call socket descriptor via `IPCName`. Upper and lower case characters are considered equivalent. Refer to “Socketname Parameter” for a detailed description of naming.

nlen — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the specified name. Maximum length is 16 bytes.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

`IPCNamErase` can be called to remove listings from the local node’s socket registry. Only the owner of a call socket descriptor may remove its name from the local socket registry. (A NetIPC error code 38 is returned in the *result* parameter if the calling process attempts to erase the name of a socket it does not own.)

If a call socket descriptor is destroyed via `IPCShutDown`, or if its owner terminates, then any listings for it that exist at the local socket registry are automatically purged.

Establishes a virtual circuit connection by receiving a response to a connection request, or receives data on a previously established connection.

Syntax

```
IPCRECV(vcdesc,data,dlen,flags,opt,result)
```

Parameters

vcdesc — *32-bit integer, by value in Pascal, by reference in FORTRAN*. VC socket descriptor. Refers to a VC socket that: (1) is the endpoint of a virtual circuit connection that has not yet been established, or (2) is the endpoint of a previously established virtual circuit on which data will be received.

data — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. A data buffer that will hold the received data, or a data vector describing the location where the data is to be placed. Refer to "Data Parameter" for information on the structure and use of this parameter.

dlen (input/output) — *32-bit integer, by reference*. When the *data* parameter is a data buffer, *dlen* is the maximum number of bytes you are willing to receive. When the *data* parameter is a data vector, *dlen* refers to the length of the data vector in bytes. As a return parameter (output), *dlen* indicates how many bytes were actually received.

If `IPCRecv` is used to establish a connection (not to receive data), the *dlen* parameter is meaningless on input and a value of 0 is returned on output.

If the DATA_WAIT flag (see `flags [21]` below) is zero, then *dlen* returns with the length of whatever data there is. If the DATA_WAIT flag is set to one, then either *dlen* returns with the same amount requested or a "WOULD BLOCK" error occurs. Refer to "Synchronous vs. Asynchronous I/O" later in this subsection for more detailed explanations.

flags (input/output) — *32-bit integer, by reference*. A 32-bit map of special request bits. Refer to "Flags Parameter" for more information on the structure and use of this parameter. The first `IPCRecv` call establishes a virtual connection and *flags* has no meaning. For subsequent `IPCRecv` calls, *flags* will then be invoked for the established connection. *Flags* must be initialized each time it is used by *any* NetIPC call. The following flags are defined for this call:

- `flags [21]`—DATA_WAIT (input). When this flag is set, `IPCRecv` waits until all the data that it requested in the *`dlen`* parameter has been received. If this bit is set to zero, `IPCRecv` may complete receiving less data than it requested in *`dlen`*. Refer to "Synchronous vs. Asynchronous I/O" later in this subsection for more detailed explanations.

Note User programs written prior to software Revision 5.0 that wait on the `IPCRecv` call until *`dlen`* amount of data has been received *must* change to set the DATA_WAIT flag to continue operating as they did before.

- `flags [31]`—PREVIEW (input). When set, this flag allows you to preview the data queued on the connection. Data is placed in the *`data`* parameter but not dequeued from the connection. Because the data is not dequeued, another `IPCRecv` call is needed to delete the same data.
- `flags [32]`—VECTORED (input). When set, this flag indicates that the *`data`* parameter is a data vector and not a data buffer.

`opt` — *`Byte array (Pascal); Integer array (FORTRAN), by reference`*. An array of options and associated information. Refer to "Opt Parameter" for information on the structure and use of this parameter. The following option is defined for this call:

- data offset (*`optioncode`* = `8`, *`datalength`* = `2`). A two-byte integer that defines a byte offset from the beginning of a data buffer where NetIPC is to begin placing the data. This option is valid only if the data parameter is a *data buffer* and not data vector.

`result` — *`32-bit integer, by reference`*. The error code returned; zero if no error.

Discussion

`IPCRecv` has two functions:

- establish a virtual circuit connection that was initiated with `IPCConnect`
- receive data on a previously established virtual circuit connection

Establishing a Connection

When `IPCRecv` is successfully called to establish a connection, no data is returned in the *`data`* parameter and zero is returned in the *`result`* parameter. If the call is unsuccessful, a non-zero value is returned in *`result`*. The call could be unsuccessful for the following reasons:

- *Timed-Out Error Received*. The synchronous timer expired before the connection could be established. The connection is still pending and `IPCRecv` should be called again to establish the connection. The timeout can be adjusted by calling `IPCControl`. Refer to the discussion of `IPCControl` for more information.

- *"Would Block" Error Received*. The VC socket referenced by `IPCRecv` is in asynchronous mode and the call could not be satisfied. The connection is still pending and `IPCRecv` should be called again to establish the connection. For more information on trying to avoid this error, refer to the following discussion titled "Synchronous vs. Asynchronous I/O".

- *Connection Establishment Failed*. If the connection could not be established for a reason other than those listed above, the referenced VC socket should be shut down by calling `IPCShutDown`.

Receiving Data

When `IPCRecv` is called to receive data queued on an established connection, the following alternatives are available:

- *Normal reading*. The requested data is dequeued from the connection and placed into the user's buffer.

- *Preview reading*. This alternative is specified by setting the PREVIEW bit (`flags [31]`) of the *`flags`* parameter. When this bit is set, the requested data is placed into the calling process's buffer but not dequeued from the connection. Consequently, the next `IPCRecv` call will read the same data. Because PREVIEW enables a process to determine what a data buffer contains before actually reading it, it is especially useful to set this bit when the receiving process must assemble messages from the byte streams that it receives. For example, if the sending process places the length of its "message" in the first two bytes of its send buffer, the receiving process can use the PREVIEW option to extract the length information from the data received. When the buffer is received again with a subsequent `IPCRecv` call, the receiving process can specify this length information in the *`dlen`* parameter and thus reassemble the "message."

- *Vectored or "scattered" reading*. The calling process may pass a data vector argument that describes one or more locations. Received data will be placed into these locations. This alternative can be used with both the normal and preview reads described above and is specified by setting the VECTORED bit (`flags [32]`) of the *`flags`* parameter. Refer to the "Type Coercion" subsection earlier in this section for an example of a single data parameter representing either vectored or unvectored data.

An IPCRecv request is considered *satisfiable* if the following condition is true:

- *Enough data is queued on the connection to satisfy the request*. If the specified data length (or aggregate vector total) is not large enough to hold all of the data on a connection, then only the amount of data requested will be returned to the calling process.

Synchronous vs. Asynchronous I/O

The IPCRecv call functions differently depending on whether the socket referenced is in synchronous or asynchronous mode, and whether or not the DATA_WAIT bit (bit 21) is set in the *flags* parameter. The following paragraphs and Table 5-9 and Table 5-10 describe these differences. When a socket is created, it is placed in synchronous mode by default. You can place a socket in asynchronous mode by calling IPCControl. Refer to the discussion of IPCControl earlier in this section for more information.

Note The “amount requested” by an IPCRecv call refers to the number of bytes specified by the *dlen* parameter or the amount specified in the data vector if the VECTORED flag is set.

- *Synchronous I/O, DATA_WAIT.* If the socket referenced by IPCRecv is in synchronous mode and the DATA_WAIT bit (bit 21) is set, the calling process will block until one of the following actions occur:
  - The amount of data queued on the connection is equal to or greater than the amount requested.
  - The call times out.
  - The connection goes down.

  If the data queued on the connection is less than *dlen* bytes, IPCRecv will suspend the calling process and the synchronous timer will be set. If the timer expires before enough data arrives to satisfy the request, the calling process will resume and a timeout error (code 59) will be returned indicating that a timeout occurred. The synchronous timeout can be adjusted by calling IPCControl. Refer to the discussion of IPCControl for more information.

- *Synchronous I/O, DATA_WAIT set to zero*. If the socket referenced by IPCRecv is in synchronous mode and the DATA_WAIT bit (bit 21) is set to zero, the calling process will block until one of the following actions occur:
  - *Some amount of data* is queued on the connection. The amount of data queued may or may not be the amount requested, and may be as little as one byte.
  - The call times out.
  - The connection goes down.

If no data is queued on the connection within the synchronous timeout period, the calling process will resume and a timeout error (code 59) will be returned indicating that a timeout occurred.

- *Asynchronous I/O, DATA_WAIT set*. If the socket referenced by `IPCRecv` is in asynchronous mode and the DATA_WAIT bit is set, the calling process will block until one of the following actions occur:

  - The amount of data queued on the connection is less than the amount requested. A “WOULD BLOCK” error (code 56) is returned to the calling process.

  - A remote abort error occurs.

  The calling process is *not* suspended awaiting the arrival of data. You can perform a read select on the referenced socket by invoking `IPCSelect`. `IPCSelect` determines whether or not a socket is readable prior to calling `IPCRecv` to receive data. Refer to the discussion of `IPCSelect` later in this section for more information.

- *Asynchronous I/O, DATA_WAIT set to zero*. If the socket referenced by `IPCRecv` is in asynchronous mode and the `DATA_WAIT` bit is set to zero, as little as one byte of data will satisfy the `IPCRecv` request.

  - If *no* data is queued to the connection, a “WOULD BLOCK” error (code 56) is returned to the calling process.

  - A remote abort error occurs.

One helpful way to remember the difference between DATA_WAIT set and set to zero is:

- If DATA_WAIT is set, you are willing to wait for the exact amount of data you requested.

- If DATA_WAIT is set to zero, you will take any amount of data.

For more discussion of asynchronous and synchronous I/O, refer to “Synchronous and Asynchronous Socket Modes” at the beginning of this section. Examples follow in Table 5-9 and Table 5-10.

Table 5-9. Synchronous I/O Example

NetIPC Call to Send/Receive Data	Synchronous DATA_WAIT is Set to Zero	Synchronous DATA_WAIT is Set to One
1. Program A requests 200 bytes: `IPCRecv(...200...)`	Program A receives 0 bytes and waits for Program B to send data with an `IPCSend`.	Program A receives 0 bytes and waits for Program B to send data with an `IPCSend`.
2. Program B on the remote node sends 100 bytes: `IPCSend(...100...)`	Program B sends 100 bytes. Program A receives the 100 bytes and the `IPCRecv` completes.	Program B sends 100 bytes. Program A still waits for 200 bytes.
3. Program B on the remote node sends another 100 bytes: `IPCSend(...100...)`	Program B sends another 100 bytes. Program A is not doing an `IPCRecv` call, so it does not receive any data.	Program B sends another 100 bytes (200 total). Program A receives the 200 bytes and completes.
4. Program A requests 100 bytes: `IPCRecv(...100...)`	Program A receives the 100 bytes and completes.	Program A receives 0 bytes and waits for Program B to send data with and `IPCSend`.

Note that steps 1 through 4 below are the same for both DATA_WAIT set to one and set to zero.

Table 5-10. Asynchronous I/O Example

NetIPC Call to Send/Receive Data	Asynchronous DATA_WAIT is Set to Zero	Asynchronous DATA_WAIT is Set to One
1. Program A requests 200 bytes: `IPCRecv(...200...)`	Program A receives 0 bytes and completes with a WOULD BLOCK error.	Program A receives 0 bytes and completes with a WOULD BLOCK error.
2. Program B on the remote node sends 100 bytes: `IPCSend(...100...)`	Program B sends 100 bytes. Program A is no longer doing an `IPCRecv` call, so does not wait and does not receive any data.	Program B sends 100 bytes. Program A is no longer doing an `IPCRecv` call, so does not wait and does not receive any data.
3. Program B on the remote node sends another 100 bytes: `IPCSend(...100...)`	Program B sends another 100 bytes (200 bytes total). Program A is not doing an `IPCRecv` call, so it does not receive any data.	Program B sends another 100 bytes (200 bytes total). Program A is not doing an `IPCRecv` call, so it does not receive any data.
4. Program A requests 100 bytes: `IPCRecv(...100...)`	Program A receives 100 bytes and completes.	Program A receives 100 bytes and completes.
5. Program B sends another 100 bytes: `IPCSend(...100...)`	Program B sends another 100 bytes (200 total). Program A is not doing an `IPCRecv` call, so it does not receive any data yet.	Program B sends another 100 bytes (200 total). Program A is not doing an `IPCRecv` call, so it does not receive any data yet.
6. Program A requests 300 bytes: `IPCRecv(...300...)`	Program A receives 200 bytes and completes.	Program A wants 300 bytes and completes with a WOULD BLOCK error. No data is passed to Program A.

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

Receive size (`dlen` *parameter*)—The HP 1000 receive size range is 1 to 8,000 bytes. The HP 9000 receive size range is 1 to 32,767 bytes. Although the range sizes that can be specified in the `dlen` parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

Receive size (`dlen` *paramete*r)—The HP 3000 receive size range is 1 to 30,000 bytes. The HP 1000 receive size range is 1 to 8,000 bytes. Although the range sizes that can be specified in the `dlen` parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

Data wait flag—The HP 1000 `IPCRecv` call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the `dlen` parameter has been received. This flag is not available on the HP 3000, meaning that the call may complete before all the data is received. However, the HP 3000 `IPCRecv` supports other flags such as the “more data” and “destroy data” flags. Refer to the description of `IPCRecv` in the NetIPC3000 manual for detailed information.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and the PC:

Receive size (`dlen` *parameter*)—The HP 1000 receive size range is 1 to 8,000 bytes. The HP 1000 enables you to specify the maximum receive size of the data buffer through the `opt` array in the `IPCConnect` call. This determines what the maximum value for `dlen` can be for any `IPCRecv` call. PC NetIPC has no option array defined for `IPCConnect.` This does not affect cross-system communication. The maximum receive size of the data in the buffer on the HP 1000 will determine the receive size buffer on the PC.

Data wait flag—The HP 1000 `IPCRecv` call supports a “DATA_WAIT” flag. This flag, when set, specifies that the call will not complete until the amount of data specified by the `dlen` parameter has been received. This flag is not available on the PC, meaning that the call may complete before all the data is received.

IPCRECVCN

Receives a connection request on a call socket.

Syntax

```
IPCRECVCN(calldesc,vcdesc,flags,opt,result)
```

Parameters

calldesc — `32-bit integer, by value in Pascal, by reference in FORTRAN`. Socket descriptor. Refers to a call socket owned by the calling process.

vcdesc — `32-bit integer, by reference`. VC socket descriptor. Refers to a VC socket that is the endpoint of the newly-established virtual circuit connection.

flags — `32-bit integer, by reference`. A 32-bit map of special request bits. Refer to "Flags Parameter" for more information on the structure of this parameter. The following flags are defined for this call:

- `flags [22]`—CHECKSUMMING (input). When set, this flag causes TCP to enable checksumming. However, *not* setting this bit does not ensure that checksumming will not occur. TCP checksum will always be performed if: the peer process calls `IPCConnect` with the checksumming bit set. TCP checksum is performed in addition to data link checksum. If TCP performs checksumming, increased overhead is required and real-time integrity cannot be guaranteed.

opt — `Byte array (Pascal); Integer array (FORTRAN), by reference`. An array of options and associated information. Refer to "Opt Parameter" for information on the structure and use of this parameter. The following options are defined for this call:

- maximum send size (*optioncode* = 3, *datalength* = 2). A two-byte integer that specifies the maximum number of bytes you expect to send with a single call to `IPCSend` on this connection. *Range*: 1 to 8,000 bytes. *Default*: 100 bytes. If this option is not specified, `IPCSend` will return an error if a call attempts to send greater than 100 bytes.

- maximum receive size (*optioncode* = 4, *datalength* = 2). A two-byte integer that specifies the maximum number of bytes you expect to receive with a single call to `IPCRecv` on this connection. *Range*: 1 to 8,000 bytes. *Default*: 100 bytes. If this option is not specified, `IPCRecv` will return errors if a call attempts to receive greater than 100 bytes.

result	*32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

Compared to the telephone system, IPCRecvCn is analogous to answering a telephone because processes must call IPCRecvCn to receive connection requests. Consider the following example: Process A calls IPCConnect with a path report descriptor that refers to a path report which indicates the location of a call socket owned by Process B. This causes Process B's "telephone" (call socket) to "ring" (receive a connection request). In order to answer its "telephone," Process B calls IPCRecvCn. A process's call socket (or "telephone") is considered to be "ringing" when it has one or more queued connection requests. Process A must still call IPCRecv to determine whether the connection was successfully established (that is, whether Process B "answered its telephone").

When IPCRecvCn is invoked successfully against a call socket that has queued connection requests, it returns a VC socket descriptor to the calling process. This VC socket descriptor can be used to specify the virtual circuit connection a process intends to send on, receive on, give away, or shut down with subsequent NetIPC calls.

A process may own a maximum of 32 call socket, VC socket, and path report descriptors. IPCRecvCn will return an error if a process attempts to exceed this limit.

Synchronous vs. Asynchronous I/O

IPCRecvCn functions differently depending on whether the call socket referenced is in synchronous or asynchronous mode. When a socket is created, it is placed in synchronous mode by default. You can place a socket in asynchronous mode by calling IPCControl. Refer to the discussion of IPCControl for more information. The following paragraphs describe these differences:

- *Synchronous I/O*. IPCRecvCn will block when invoked against a call socket that has no queued connection requests if the socket is in synchronous mode. The calling process will resume execution when a connection request arrives, or after the synchronous timeout interval has expired. IPCRecvCn calls will not block indefinitely against a given call socket unless the referenced socket's synchronous timeout interval has been set to infinity via a call to IPCControl. The default synchronous timeout is 60 seconds.

- *Asynchronous I/O*. IPCRecvCn will never block against sockets in asynchronous mode. When IPCRecvCn is invoked against an asynchronous call socket that has no queued connection requests, a "would block" error (error code 56) is returned to the calling process. When IPCRecvCn is used in this way, the calling process does not wait to receive a connection request. In order to determine when connection requests are present, a process can perform an exception select on the referenced call socket by calling IPCSelect. (Refer to the discussion of IPCSelect for more information.)

For a detailed discussion of synchronous and asynchronous I/O, refer to "Synchronous and Asynchronous Socket Modes" at the beginning of this section.

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

Checksumming—When the `ipcrecvcn()` call is executed on the HP 9000 node, then checksumming is always enabled for the HP 9000-to-HP 1000 connection.

Send and Receive sizes—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 9000 send and receive size range is 1 to 32,767 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

Send and Receive sizes—The HP 1000 send and receive size range is 1 to 8,000 bytes. The HP 3000 send and receive size range is 1 to 30,000 bytes. Although the ranges are different, you must specify a buffer size within the correct range for the respective system; otherwise, an error will occur. For example, if the HP 3000 sends 16,000 bytes, the HP 1000 node can call `IPCRecv` twice, receiving 8,000 bytes the first time and the second 8,000 bytes the second time. Two processes should already understand how much and what kind of data they expect to send and receive whether or not they are cross-system processes. Note that the default send and receive sizes differ on the HP 1000 and HP 3000. On the HP 1000, the default send and receive size is 100 bytes. On the HP 3000, the default send and receive size is less than or equal to 1024 bytes.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and the PC:

Checksumming—With PC NetIPC, the TCP checksum option cannot be turned on. But if the HP 1000 requires it, the TCP checksum will be in effect on both sides of the connection.

Send and Receive sizes—The HP 1000 send and receive size range is 1 to 8,000 bytes. The PC send and receive size range is 1 to 65,535 bytes. Although the ranges are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range. For example, if a PC sends 16,000 bytes, the HP 1000 computer can call `IPCRecv` twice, receiving 8,000 bytes the first time and the second 8,000 bytes the second time.

Determines the status of a call socket or VC socket.

Syntax

```
IPCSELECT(sdbound,readmap,writemap,exceptionmap,timeout,result)
```

Parameters

sdbound (input/output)
: `32-bit integer, by reference`. Specifies the upper ordinal bound on the range of descriptors specified in the *readmap*, *writemap* and *exceptionmap* parameters. An `IPCSelect` call will be most efficient if this parameter is set to the maximum ordinal value of the sockets specified in these parameters. Because a NetIPC process may have concurrent access to a maximum of 32 descriptors, *sdbound* may be given a maximum value of 32. As an output parameter, *sdbound* contains the upper ordinal boundary of all of the descriptors that met the select criteria. If none of the criteria were met, *sdbound* will be set to zero.

readmap (input/output)
: `32-bit integer, by reference`. A bit map indexed by VC socket descriptors. When *readmap* is an input parameter, this map should have bits set for all of the VC sockets from which you would like to receive data. As an output parameter, *readmap* is a bit map describing all of the read-selected VC sockets that are readable. (Refer to the discussion below for more information on the structure and use of this parameter.)

writemap (input/output)
: `32-bit integer, by reference`. A bit map indexed by either call socket descriptors or VC socket descriptors. When *writemap* is an input parameter, this map should have bits set for all of the call sockets on which you would like to initiate connections, or all of the VC sockets to which you would like to send data. As an output parameter, *writemap* is a bit map describing all of the write-selected sockets that are writeable. (Refer to the discussion below for more information on the structure and use of this parameter.)

exceptionmap (input/output)
: `32-bit integer, by reference`. A bit map indexed by either call socket descriptors or VC socket descriptors. When *exceptionmap* is an input parameter, this map should have bits for all of the sockets for which notification of exceptional conditions is desired. As an output parameter, *exceptionmap* is a bit map describing all of the exception-selected sockets that are exceptional. For call sockets, an exceptional condition is present if a connection request is queued to the socket; for VC sockets, an exceptional condition is present if the connection referenced by the socket has been aborted. (Refer to the discussion below for more information on the structure and use of this parameter.)

timeout — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The number of tenths of seconds the calling process is willing to wait for some event to occur which would cause `IPCSelect`'s report to change. This timeout is put into effect only when none of the sockets referenced can immediately satisfy the select criteria (that is, none are readable, writeable or exceptional). If this value is set to zero, the call will not block. If it is set to -1, the timeout will be set to infinity (that is, the call will block).

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

`IPCSelect` permits a process to detect, and/or wait for, the occurrence of any of several events across any of several sockets. Compared to the telephone system, invoking `IPCSelect` is analogous to performing powerful "switchboard-like" operations because it enables a process to act as a "switchboard operator" by monitoring the sockets, or "telephones," that it owns. A process should call `IPCSelect` with map bits set for descriptors that it owns. If a process attempts to perform a select on a descriptor that it does not own, or on a path report descriptor, an error will be returned.

`IPCSelect` reports three types of information:

- Whether any of the referenced VC sockets are *readable*. A VC socket is considered readable if it can immediately satisfy an `IPCRecv` request (with the DATA_WAIT flag set) for a number of bytes *equal to or less than* its read threshold. Each VC socket has an associated read threshold which, when the socket is first created, is set to one byte. This value can be modified by calling `IPCControl`. (For more information on setting read thresholds, refer to "Synchronous and Asynchronous Socket Modes" at the beginning of this section.) Read selecting on call sockets has no meaning. Although doing so will not produce an error, this practice should be avoided.

- Whether any of the referenced VC sockets are *writeable*. A VC socket is considered writeable if it can immediately accommodate an `IPCSend` request that involves a number of bytes *equal to or less than* the socket's write threshold. Each VC socket has an associated write threshold which, when the socket is first created, is set to one byte. This value can be modified by calling `IPCControl`. (For more information on setting write thresholds, refer to "Synchronous and Asynchronous Socket Modes" at the beginning of this section.)

- Whether any of the referenced call or VC sockets are *exceptional*. A VC socket is considered exceptional if it has a problem associated with it (for example, the connection it references was aborted). A call socket is considered exceptional if it has a connection request queued on it, or if it can no longer be supported by NetIPC (for example, the node's network manager has shut the node down). In addition, path report descriptors and non-existent sockets will select as exceptional.

The following are examples of read selecting, write selecting, and exception selecting using `IPCSelect`.

Examples

Detecting Connection Requests. By setting bits in the *exceptionmap* parameter, a process can determine if incoming connection requests are queued to certain call sockets. Consider the following example: Process A must determine whether certain call sockets have received connection requests. To do this, Process A calls IPCSelect with the *exceptionmap* bits set to correspond to these sockets. Assuming that the timeout interval is long enough (set by the *timeout* parameter), IPCSelect will complete after at least one connection request has arrived and has been queued on one of the sockets specified in *exceptionmap*. When the call completes, only those bits that correspond to sockets that have queued connection requests remain set; the other bits will have been cleared.

Performing a Read Select. By setting bits in the *readmap* parameter, a process can determine whether certain VC sockets are readable. Consider the following example: Process A must determine which of its VC sockets have data queued to them. To do this Process A performs a *read select* on those sockets by setting bits in the *readmap* parameter to correspond with the desired VC sockets. Upon completion of the call, only the bits that represent readable sockets will remain set; the other bits will have been cleared. Process A can call IPCSelect with a zero-length timeout to determine the status of a socket immediately, or with a non-zero timeout if it is willing to wait for some data to arrive.

Performing a Write Select. By setting bits in the *writemap* parameter, a process can determine whether certain VC sockets are writeable. Consider the following example: Process A must determine which of its VC sockets can accommodate a new IPCSend request, and which of its call sockets can accommodate a new IPCConnect request. To do this, it can perform a *write select* on these sockets by setting bits in the *writemap* parameter to correspond with the desired VC and call sockets. Upon completion of the call, only the bits that represent writeable sockets will remain set; the other bits will have been cleared. Process A can call IPCSelect with a zero-length timeout to determine the status of a socket immediately, or with a non-zero timeout if it is willing to wait before sending data on the connection.

Exception Selecting. By setting bits in the *exceptionmap* parameter, a process can determine whether certain connections have been aborted. VC sockets that reference aborted connections will always *exception select* as "true" (their bits will be set when the call completes). Exception selecting on VC sockets can also be useful when the connection associated with the socket is not fully established. Consider the following example: Process B has successfully created a VC socket via a call to IPCConnect, but will not know whether the connection associated with the socket is established until it calls IPCRecv. If Process B calls IPCRecv before the connection is established, or before it becomes known that the connection cannot be established, it will block if the VC socket is in synchronous mode, or return a "would block" error (error code 56) if the VC socket is in asynchronous mode. Process B can avoid blocking, in the synchronous case, or polling, in the asynchronous case, by performing an exception select on the new VC socket. The socket will select as true if the connection has been established (a call to IPCRecv will be successful), or if there is a problem associated with it (a call to IPCRecv will be unsuccessful).

IPCSelect Call Bit Map Parameters

Note NetIPC calls assume that the bits in the *readmap*, *writemap* and *exceptionmap* parameters are numbered from left to right with the most significant bit considered to be bit one and the least significant bit considered to be bit 32. However, in FORTRAN 77, the *numbering* of these bits is from 31 to 0:

MSB	
1 2 3 4 5 6 ... 32	`IPCSelect` map parameters
1 2 3 4 5 6 ... 32	Pascal
31 ... 3 2 1 0	FORTRAN

In Pascal/1000, it is useful to represent the *readmap*, *writemap* and *exceptionmap* parameters as type `bit_map_type`:

```
type bit_map_type = packed array [1..32] of boolean;
```

To set a bit in any of these parameters to correspond to a specific call socket or VC socket, use the appropriate *calldesc* or *vcdesc* value as a subscript and assign the value `TRUE`. For example:

```
read_map [vcdesc]       := TRUE;
write_map [calldesc]    := TRUE;
exception_map [vcdesc]  := TRUE;
```

In FORTRAN 77, the *readmap*, *writemap* and *exceptionmap* parameters must be declared as 32-bit integers (`INTEGER*4`). The simplest way to set a bit in one of these parameters is to use the FORTRAN 77 library function `ibset(a,b)`. The *readmap*, *writemap* or *exceptionmap* parameter is passed in the first argument (*a*) and the bit position you want to set is passed in the second argument (*b*).

In the following FORTRAN 77 example, a bit is set in the *readmap* parameter:

```
C    The vcdesc value is subtracted from 32 so that the proper
C    bit is set. This maps ibset's bit numbering convention into
C    NetIPC's.

     readmap = ibset(readmap,(32-vcdesc))
```

Multiple bits can be set by repeating the ibset function.

In Pascal/1000, type coercion can be used to quickly clear all of the bits in a bit map. This can be done as follows:

```
type bit_map_type = packed array [1..32] of boolean;

read_map = record
        case boolean of
            true : (bits : bit_map_type);
           false : (int  : integer);
        end;
```

To invoke the `bit_map_type`, specify `read_map.bits`; to invoke the integer type, specify `read_map.int`. If the value 0 is assigned to `read_map.int`, all 32 bits will be cleared.

IPCSEND

Sends data on a virtual circuit connection.

Syntax

```
IPCSEND(vcdesc,data,dlen,flags,opt,result)
```

Parameters

vcdesc — *32-bit integer, by value in Pascal, by reference in FORTRAN*. VC socket descriptor. Refers to the VC socket endpoint of the connection through which the data will be sent. A VC socket descriptor can be obtained by calling `IPCConnect`, `IPCRecvCn` and `IPCGet`.

data — *Packed array of characters (Pascal); Integer array (FORTRAN), by reference*. A buffer that will hold the data to be sent, or a data vector describing where the data to be sent is located. Refer to “Data Parameter” for more information on the structure and use of this parameter.

dlen — *32-bit integer, by value in Pascal, by reference in FORTRAN*. If *data* is a data buffer, *dlen* is the length of the data in the buffer. If *data* is a data vector, *dlen* is the length of the data vector.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. Refer to “Flags Parameters” for more information on the structure and use of this parameter. The following flags are defined for this call:

- `flags [27]`—HIGH THROUGHPUT (input). Indicates that you would prefer high throughput to low delay. Refer to the following “Discussion” subsection for more information.
- `flags [32]`—VECTORED (input). Indicates that the data parameter refers to a data vector and not to a data buffer.

opt — *Byte array (Pascal); Integer array (FORTRAN), by reference*. An array of options and associated information. Refer to “Opt Parameter” for more information on the structure and use of this parameter. The following option is defined for this call:

- data offset (*optioncode* = `8`, *datalength* = `2`). A two-byte integer that indicates a byte offset from the beginning of the data buffer where the data to be sent actually begins. Only valid if the data parameter is a data buffer.

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

IPCSend is used to send data on an established connection. The data may be sent as a single contiguous buffer or as a scattered data vector. If the data is vectored, NetIPC will gather all the referenced data before sending it.

If flags [27] is set, the Transmission Control Protocol (TCP) may not immediately transmit the data indicated by the *data* parameter. Instead, it may wait until it has received an amount of data that can be transmitted with the greatest efficiency. Several transmissions of small amounts of data consume more resources than one large transmission. If flags [27] is *not* set (set to zero), TCP will attempt to transmit the data immediately, regardless of efficiency considerations. If your process will be sending large amounts of data, HP recommends that you set flags [27]. If flags [27] is set and you submit only a small amount of data (less than a few hundred bytes), then TCP may hold onto the data for a considerable period of time before transmitting it. HP also recommends that you do *not* set flags [27] when sending the last transmission on a connection.

Synchronous vs. Asynchronous I/O

IPCSend functions differently depending on whether the VC socket referenced is in synchronous or asynchronous mode. The following paragraphs describe these differences:

- *Synchronous I/O*. Send requests issued against VC sockets in synchronous mode may block. IPCSend will block if it can not immediately obtain the buffer space needed to accommodate the data. The call will resume after the required buffer space becomes available, or if the synchronous timer expires. Timeouts usually occur when the process on the receiving end of the connection stops receiving the data sent to it. (The length of the synchronous timeout interval can be adjusted via IPCControl. Refer to the discussion of this call for more information.)

- *Asynchronous I/O*. Send requests issued against sockets in asynchronous mode will never block. If the buffer space needed to accommodate the data is not immediately available, a “would block” error (error code 56) is returned. After receiving this error, the process can try the call again later, or determine when the socket is writeable by calling IPCSelect. (Refer to the discussion of IPCSelect for more information on writeable sockets.)

For a detailed discussion of synchronous and asynchronous I/O, refer to “Synchronous and Asynchronous Socket Modes” at the beginning of this section.

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

Send size (*dlen parameter*)—The HP 1000 send size range is 1 to 8,000 bytes. The HP 9000 send size range is 1 to 32,767 bytes. Although the range sizes that can be specified in the *dlen* parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

Send size (`dlen` *parameter*)—The HP 1000 send size range is 1 to 8,000 bytes. The HP 3000 send size range is 1 to 30,000 bytes. Although the range sizes that can be specified in the `dlen` parameter are different, cross-system communication is not affected. Just be sure to specify a buffer size within the correct range for the respective system.

Urgent Data—The HP 3000 supports an "urgent data" option in the `opt` parameter. If this bit is set by the HP 3000 program, it will be ignored by the receiving process on the HP 1000.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and the PC:

Send size (`dlen` *parameter*)—The HP 1000 send size range is 1 to 8,000 bytes. The HP 1000 enables you to specify the maximum send size of the data buffer through the `opt` array in the `IPCConnect` call. This determines what the maximum value for `dlen` can be for any `IPCSend` call. PC NetIPC has no option array defined for `IPCConnect.` This does not affect cross-system communication. The maximum send size of the data in the buffer on the HP 1000 will determine the send size buffer on the PC.

IPCSHUTDOWN

Releases a descriptor and any resources associated with it.

Syntax

```
IPCSHUTDOWN(descriptor,flags,opt,result)
```

Parameters

descriptor — *32-bit integer, by value in Pascal, by reference in FORTRAN*. The descriptor to be released. May be a call socket descriptor, VC socket descriptor, or path report descriptor.

flags — *32-bit integer, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. All bits must be clear (set to zero). Refer to "Flags Parameter" for more information on the structure of this parameter.

opt — *Byte array (Pascal); Integer array (FORTRAN), by reference*. An array of options and associated information. This parameter is reserved for future use. You must initialize the *opt* parameter to contain zero arguments. (See `InitOpt` later in this section for more information.)

result — *32-bit integer, by reference*. The error code returned; zero if no error.

Discussion

`IPCShutDown` is called to release a descriptor and any resources associated with it. The descriptor referenced may be a call socket descriptor, VC socket descriptor, or path report descriptor. How `IPCShutDown` functions depends on which type of descriptor is being used. If the descriptor is a:

- *call socket descriptor*, the call socket referenced by the descriptor is destroyed along with any names associated with it. The process that had access to the call socket may no longer use it, and all connection requests queued to the socket are aborted. Since system resources are used when a call socket is created, you may want to release your call sockets when they are no longer needed. A call socket is needed as long as a process is expecting to receive a connection request on that socket. After the connection request is received via `IPCRecvCn`, and as long as no other connection requests are expected, the call socket descriptor can be released. Similarly, a process that requests a connection can release its call socket any time after its call to `IPCConnect`, as long as it is not expecting to receive a connection request on that socket. Using `IPCShutDown` to release a call socket descriptor does not affect any VC sockets.

- *path report descriptor*, the addressing information stored in conjunction with the descriptor is destroyed along with any names associated with it in the local socket registry. Because path report descriptors also require system resources, you may want to release them when they are no longer needed.

- *VC socket descriptor*, the VC socket descriptor is released and the referenced connection is aborted and is no longer available for sending or receiving data. Because `IPCShutDown` takes effect very quickly, all of the data that is in transit on the connection, including any data that has already been queued on the destination VC socket, may be destroyed when the connection is shut down. Obtaining confirmation from the receiver is the only way a sender will know that data sent was actually received. Shutting down a VC socket does not affect any call sockets.

Although NetIPC guarantees that data will be delivered reliably, this guarantee is contingent upon a functioning network; data may not be received if nodes crash, network links fail, or peer processes abort. If the process calling `IPCShutDown` sends important data to its peer process just prior to shutting that process down, it is recommended that the calling process receive a confirmation from the peer process before calling `IPCShutDown` to ensure that the data was received.

For more information on `IPCShutdown`, refer to “Shutting Down a Connection” at the beginning of this section.

Cross-System Considerations

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 9000:

Socket Shut Down—The shutdown procedure for both HP 1000 and HP 9000 processes is identical except for shared sockets on HP 9000. Shared sockets are destroyed only when the descriptor being released is the sole descriptor for that socket. Therefore, the HP 9000 process may take longer to close the connection than expected.

The following information summarizes cross-system NetIPC programming considerations for HP 1000 and HP 3000:

Socket Shut Down—The shutdown procedure for both HP 1000 and HP 3000 processes is the same, except that the *`graceful release`* flag is not available on the HP 1000. Do *not* set the graceful release flag (`flags 17`) on the HP 3000. Otherwise, the HP 1000 will not perform a normal shutdown. If the HP 3000 process does set the graceful release flag, the HP 1000 `IPCRecv` call will get a NetIPC error 68 (no more data) instead of a NetIPC error 64 (Connection aborted by peer). The HP 1000 process should handle the error 68 as if it were an error 64. After receiving a NetIPC error 68, subsequent `IPCRecv` calls will get a NetIPC error 109 (remote connection has already graceful released the socket), because there is no more data available.

There are no cross-system considerations for HP 1000 and the PC.

Special NetIPC Calls

The following calls, with the exception of `AdrOf`, are used to manipulate the *`opt`* parameter found in many NetIPC and RPM calls. `AdrOf` is provided so that you can obtain the byte address of any byte within a data object. Byte addresses are used to specify data vectors. For more information on the *`opt`* parameter and its structure, refer to “Opt Parameter” at the beginning of this section.

ADDOPT

Adds an argument and its associated data to the *opt* parameter.

Syntax

```
ADDOPT(opt,argnum,optioncode,datalength,data,error)
```

Parameters

opt — *Byte array (Pascal); Integer array (FORTRAN), by reference*. The *opt* parameter to which you want to add an argument. Refer to "Opt Parameter" for information on the structure and use of this parameter.

argnum — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The number of the argument to be added. The first argument is number zero.

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The option code of the argument to be added. These codes are described in each NetIPC call *opt* parameter description.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the data to be included. This information is provided in each NetIPC call *opt* parameter description.

data — `Packed array of characters (Pascal); Integer array (FORTRAN), by reference`. An array containing the data associated with the argument.

error — *16-bit integer, by reference*. The error code returned; zero if no error.

Discussion

The `AddOpt` call adds an argument and its associated data to an *opt* parameter. The parameter must be initialized by `InitOpt` before arguments can be added.

The following Pascal/1000 program fragment illustrates the use of `InitOpt` and `AddOpt` to initialize and add two arguments to the option parameter of an `IPCConnect` call. In this example, the *opt* parameter is used to specify a maximum send size and maximum receive size of 1000 bytes. (Maximum send size indicates the maximum number of bytes that you expect to send with a single `IPCSend` call and maximum receive size indicates the maximum number of bytes you expect to receive with a single `IPCRecv` call.)

ADDOPT

```
{InitOpt initializes the opt parameter to contain two arguments -- one
for the maximum send size and one for the maximum receive size.}

INITOPT(opt,2,error_return);

{Addopt is called to add the maximum send size.  The data parameter
contains the value 1000. Note that the first argument is number zero.}

ADDOPT(opt,0,3,2,data,error_return);

{Addopt is called once more to add the maximum receive size.  The data
parameter contains the value 1000.}

ADDOPT(opt,1,4,2,data,error_return);

{IPCConnect can now be called with the opt parameter.}

IPCCONNECT(calldesc,pathdesc,flags,opt,vcdesc,result);
```

Obtains the byte address of any byte within a data object.

Syntax

```
ADROF(firstobjword,offset,byteaddress)
```

Parameters

firstobjword — *16-bit integer, by reference*. The name of the first (16-bit) word of the data object.

offset — *16-bit integer, by value in Pascal, by reference in FORTRAN*. An offset from the beginning of the data object. May be positive or negative. (The first byte of a data object resides at offset zero.)

byteaddress — *16-bit integer, by reference*. The byte address of the byte that is *offset* bytes away from the first object word.

Discussion

`AdrOf` enables you to obtain a byte address that can be placed in the byte address field of a data vector. This call is necessary because most high level languages on RTE-A systems do not support the referencing of byte addresses.

The following program fragment shows `AdrOf` being used to prepare a data vector.

```
TYPE
    byte = 0..255;
    int16 = -32768..32767;
    vector_type = array [1..10] of int16;

VAR
    big_array : RECORD

                  CASE BOOLEAN OF
                  true   :  (bytes : packed array [0..999] of byte);
                  false  :  (words : array [0..499] of int16);
                  END; {case}

                END; {big_array}

header        : RECORD
                  length   : int16;
                  msg_kind : int16;
                  options  : int16;
                END; {header}

    vector     : vector_type;
    vector_len : integer;
```

```
BEGIN
.
.
.
{Prepare a data vector that describes the header record,
bytes 51 through 60 of big_array, and also bytes 500 through 999
of big_array.}

ADROF(header.length,0,vector[1]);

vector[2] := 6;

ADROF(big_array.words[0],51,vector[3]);

vector[4] := 10;

ADROF(big_array.words[0],500,vector[5]);

vector[6] := 500;
vector_len := 12;
.
.
.
flags[31] := TRUE;

IPCSEND(vcdesc,vector,vector_len,flags,opt,result);
```

INITOPT

Initializes the *opt* parameter so that arguments can be added.

Syntax

```
INITOPT(opt,optnumarguments,error)
```

Parameters

opt — *Byte array (Pascal); Integer array (FORTRAN), by reference*. The *opt* parameter to be initialized. Refer to "Opt Parameter" for information on the structure and use of this parameter.

optnumarguments — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The number of arguments that will be placed in the *opt* parameter. If this parameter is zero, the *opt* parameter will be initialized to contain zero arguments.

error — *16-bit integer, by reference*. The error code returned; zero if no error.

Discussion

`InitOpt` must be called to initialize the *opt* parameter prior to adding arguments to it with `AddOpt`. The *optnumarguments* parameter specifies how many arguments can be placed in the *opt* parameter. For example, if zero is specified, no arguments can be added to the *opt* parameter; if three is specified, three arguments must be added.

In the following program fragment, the same *opt* parameter is used in two different `IPCConnect` calls. The first call requests a connection with the default maximum send and receive sizes (100 bytes), so its option parameter is initialized to contain zero arguments. The second `IPCConnect` call requests a connection with a maximum send and receive size of 1000 bytes. Thus, its option parameter must be initialized to contain two arguments, the first to contain the maximum send size, and the second to contain the maximum receive size.

```
{InitOpt initializes the opt parameter used in the first IPCCONNECT
call to contain zero entries. This will cause the maximum send
and receive sizes to default to 100 bytes.}

INITOPT(opt,0,error_return);

{IPCConnect can now be called using the opt parameter.}

IPCCONNECT(calldesc,pathdesc,flags,opt,vcdesc,result);

{InitOpt reinitializes the opt parameter to be used in the second
IPCConnect call. This call specifies the maximum send and receive
sizes, so it must be initialized to contain two arguments.}

INITOPT(opt,2,error_return);
```

```
{The AddOpt call is used to add the maximum send size argument
as the first argument to the opt parameter.  The maximum
send has an option code of 3. The data parameter contains
the value 1000. (Note that the first argument is argument number zero.)}

ADDOPT(opt,0,3,2,data,error_return);

{The AddOpt call is used again to add the maximum receive size
as the second argument to the opt parameter.  The maximum receive
size has an option code of 4. The data parameter contains the
value 1000.}

ADDOPT(opt,1,4,2,data,error_return);

{IPCConnect can now be called using the opt parameter.}

IPCCONNECT(calldesc_two,pathdesc_two,flags,opt,vcdesc_two,result);
```

READOPT

Obtains the option code and argument data associated with an *opt* parameter argument.

Syntax

```
READOPT(opt,argnum,optioncode,datalength,data,error)
```

Parameters

opt	*Byte array (Pascal); Integer array (FORTRAN), by reference*. The *opt* parameter to be read. Refer to “Opt Parameter” for information on the structure and use of this parameter.
argnum	*16-bit integer, by value in Pascal, by reference in FORTRAN*. The number of the argument to be obtained. The first argument is number zero.
optioncode	*16-bit integer, by reference*. The option code associated with the argument. These codes are described in each NetIPC call *opt* parameter description.
datalength (input/output)	*16-bit integer, by reference*. The length of the data buffer into which the argument should be read. If the data buffer is not large enough to accommodate the argument data, an error will be returned. On output, this parameter contains the length of the data actually read. (The length of the data associated with a particular option code is provided in each NetIPC call *opt* parameter description.)
data	*Array, by reference*. An array which will contain the data read from the argument.
error	*16-bit integer, by reference*. The error code returned; zero if no error.

Client-Server Program Examples

The two pairs of NetIPC example programs that follow are referred to as servers and clients. One server-client pair is in Pascal and the other pair is the equivalent programs in FORTRAN. This server-client pair is a fairly typical model for an application having multiple nodes (the clients) requesting information from a database or file on a single system (the server). The server program handles incoming requests from multiple clients on a first-come, first-served basis.

The following steps provide a general description of the interaction between the server and client programs:

1. The server waits for the client to request service.
2. A client establishes a connection with the server. Then that client asks for information by sending a user name to the server.
3. The server searches for the proper information by opening a data file. Each record in the data file contains a user name field and an information field. The server searches for the user name. If found, the server sends the accompanying information in reply to the client.
4. The client receives its information and may request more.
5. Once the client has received all that it needs, it shuts down the connection.
6. After the server is notified that the connection has been shut down, it cleans up its internal data structures (tables).

The maximum number of clients that the server can handle at one time is system dependent. This number is based upon the maximum number of incoming call requests for a socket. On HP 1000, the maximum number is 31. On HP 9000, the maximum number is 59. On HP 3000, the maximum number is 63.

Detailed explanations of the client and server programs, what NetIPC calls were used, and why are discussed in the following paragraphs.

Server Program

You must start up the server yourself if this is a cross-system application. Otherwise, you can use RPM to execute a program on a remote NS-ARPA/1000 node.

- The server starts up first creating a call socket with a well-known TCP port address (`IPCCreate`). A well-known port address means that the socket's address is known by the client program. The call socket is where the client will direct its initial connection request.
- After the call socket is created, the server waits for incoming calls from its clients. The server waits by setting its synchronous timeout to infinity (`IPCControl`). The server will wait indefinitely for a client to make a request.

- The server constantly checks for a connection request from a client (`IPCSelect`). How the server does this is further explained in "Explanation of Server Using IPCSelect" below. When a client sends a connection request, the server accepts the incoming connection request (`IPCRecvCn`). At this point the server does *not* know which client is requesting the connection. If required, client identification information could be provided in the first data message sent by the client.

- Once the server accepts the connection request, the virtual circuit is then established with the client. The server then waits for the client to request information. The client requests information from the server with an `IPCSend` call. The server responds with an `IPCRecv` call. The client and server then have an interactive dialogue of `IPCSend` and `IPCRecv` calls:

  The client sends a message containing a name (`IPCSend`). The server reads the message (`IPCRecv`). Variable length messages can be handled by manipulating the send and receive buffer sizes in the `IPCSend` and `IPCRecv` calls.

  The server looks up the name in the data file. If the name is found, the server retrieves the accompanying information and sends it to the client (`IPCSend`). If the name is not found, the server returns an error string to the client (`IPCSend`).

- After handling the client's request for information, the server checks for any more clients requesting a connection or information.

- After the client has received its information data, it shuts down (`IPCShutDown`) the connection established with the server. The server receives an error on the virtual circuit as notification that the connection has gone down. The server does not do any error recovery and assumes that the client knows that the transaction is complete.

  The server never shuts down its call socket, because it is always ready to accept incoming requests from clients. The call socket is shut down automatically by NS-ARPA when the program terminates.

Explanation of Server Using IPCSelect

The server uses `IPCSelect` to facilitate handling several clients. The server has two bit maps—the read map for receiving clients' requests for information and the exception map for handling connection requests on the call socket and shutdown notification on virtual circuits.

Each bit represents a socket that the server has in use.

The read map is cleared initially. The exception map has one bit set for the server's call socket.

The server suspends on the `IPCSelect` call. The timeout for the `IPCSelect` has been set to infinity to wait for a client's request. (In other applications, the server could have a timeout and could do other processing while waiting for a client's request.)

The server responds to any one of these three cases:

- A client requests a new connection. The server's call socket's bit in the exception map is set to true.

- A client requests data (such a client has already established virtual circuit). The server's virtual circuit socket's bit in the read map is set to true.

- A virtual circuit goes down. The server's virtual circuit socket's bit in the exception map is set to true.

When a client requests to establish a connection, the server's exception map is set and the server responds with an `IPCRecvCn` to the client to establish a connection. If the server has all its bits set in the exception map, it has reached its limit and cannot handle any more client requests.

Once the connection is established, the server has a virtual circuit to "watch" for incoming data. The server sets a bit in the read map for this virtual circuit.

The server also sets a bit in the exception map for the virtual circuit to be notified if the connection goes down. The bit for the call socket remains set in the exception map. The server is maintaining two variables for the maps—one to keep track of what bits should be set each time and one that is actually used for the `IPCSelect`.

After setting its maps, the server waits for another client request with an `IPCSelect` call. Other client requests are handled by the server in a similar way as the connection request. If a client requests data, the server's read map is set (TRUE) which triggers the server that there is data on a particular virtual circuit. After checking its read map, the server reads the data (`IPCRecv`) which is a name to be used as a "key" for information retrieval. If the name matches the server's data file, the server returns corresponding information to the client (`IPCSend`). If the name does not match, the server returns an error string.

After the `IPCSend` call completes, the server resumes monitoring client requests with the `IPCSelect` call.

When a virtual circuit goes down, the error notification is also made with a bit in the exception map being set (TRUE). The server responds by issuing an `IPCShutDown` for the virtual circuit and clears the bits maps for the corresponding socket descriptor. Then the server resumes monitoring other client requests with the `IPCSelect` call.

Client Program

You must start up the client yourself if this is a cross-system application. Otherwise, you can use RPM to execute a program on a remote NS-ARPA/1000 node.

- The client program prompts you for the name of the system on which the server program resides. The client then creates a call socket (`IPCCreate`).

- The client creates a path report descriptor for the socket with `IPCDest` using the well-known address.

- A connection request to the server (`IPCConnect`) is received by the server's call to `IPCRecvCn`.

- The timeout on the virtual circuit socket is set to infinity (`IPCControl`). This causes the client to wait for the server's response indefinitely unless the client receives notification that the connection is down.

- The client asks the user for the name for which information is requested and sends the request to the server.

- The client then suspends on an `IPCRecv` call waiting for the server to reply. As explained in the "Server Program" above, the server receives the request for information, retrieves it, and then sends information back to the client (`IPCSend`). The client will then obtain the information (`IPCRecv`).
- The client then prompts the user for another name and the above procedures repeat. You terminate the program by entering `EOT` for the name. This invokes the client to shut down the connection (`IPCShutDown`). The shut down does not need response from the server, because no data is pending. The server assumes that the client knows when it has received all the information or data that it needs.

Cross-System NetIPC Program Examples

The following HP 1000 NetIPC programs will work together on the HP 1000. In addition, each program will work as a cross-system application with a corresponding NetIPC program written for either the HP 9000 or HP 3000. Corresponding programs have been included in the other NetIPC manuals for HP 9000 and HP 3000. These programs are not intended to be portable to other computers.

Programming examples are included with NS-ARPA/1000 software. The example files are as follows:

Pascal client	`/NS1000/EXAMPLES/client.pas`
Pascal server	`/NS1000/EXAMPLES/server.pas`
FORTRAN 77 client	`/NS1000/EXAMPLES/client.ftn`
FORTRAN 77 server	`/NS1000/EXAMPLES/server.ftn`
data file	`/NS1000/EXAMPLES/datafile`

Pascal/1000 Client NetIPC Program

```
$PASCAL '91790-18263 REV.5010 <880420.0920>'
$CDS$

{----------------------------------------------------------------------}
{}
{        NAME: CLIENT
{      SOURCE: 91790-18263
{       RELOC: NONE
{        PGMR: LMS
{}
{----------------------------------------------------------------------}
{
{
{ PURPOSE:
{    To show the operation of the IpcSelect() call.
{}


PROGRAM client ( input, output );

LABEL
   89,
   99;

CONST

   BUFFERLEN = 20;
   CALL_SOCKET = 3;
   CHANGE_TIMEOUT = 3;
   FOREVER = TRUE;
   INFINITE_SELECT = -1;
   INFOBUFLEN = 60;
   INTEGER_LEN = 2;
   INT16_LEN = 2;
   LENGTH_OF_DATA = 20;
   MAX_BUFF_SIZE = 1000;
   MAX_RCV_SIZE = 4;
   MAX_SEND_SIZE = 3;
   MAX_SOCKETS = 32;
   PROTO_ADDR = 31767;
   TCP = 4;
   ZERO = 0;
```

```
TYPE
{}
{ WARNING: If this program is ported to the 800 you need to delete
{ this type declaration.  HP-PA Pascal pre-defines this type.
{}
ShortInt = -32768..32767;

{}
{ WARNING: the bits entry of this record is not portable.
{ The declaration is 1..32 on the 1000, and 0..63 on the 800.
{}
BitMapType = RECORD
    CASE Integer OF
    1: ( bits     : PACKED ARRAY[1..32] OF Boolean );
    2: ( longint  : Integer );
    3: ( ints     : ARRAY[1..2] OF ShortInt );
    END;

byte = 0..255;
byte_array_type = packed array [1..8] of byte;
buffer_type = packed array [1..BUFFERLEN] of char;
InfoBufType = packed array [1..INFOBUFLEN] of char;
name_of_call_array_type = packed array [1..10] of char;
name_array_type = packed array [1..7] of char;

VAR
buffer_len                        : Integer;
call_name                         : name_of_call_array_type;
call_sd                           : Integer;
control_value                     : ShortInt;
data_buf                          : InfoBufType;
dummy_len                         : Integer;
dummy_parm                        : Integer;
error_return                      : Integer;
flags_array                       : integer;
node_name                         : Buffer_Type;
node_name_len                     : Integer;
opt_data                          : ShortInt;
opt_num_arguments                 : ShortInt;
option                            : byte_array_type;
protoaddr                         : ShortInt;
protocol_kind                     : Integer;
req_name_len                      : Integer;
requested_name                    : Buffer_Type;
short_error                       : ShortInt;
socket_kind                       : Integer;
temp_position                     : ShortInt;
timeout                           : Integer;
timeout_len                       : Integer;
vc_sd                             : Integer;
```

```
$TITLE 'IPC Procedures', PAGE $
PROCEDURE ADDOPT
  (VAR opt       : byte_array_type;
   argnum      : ShortInt;
   optcode     : ShortInt;
   data_len    : ShortInt;
   VAR data    : ShortInt;
   VAR error   : ShortInt);
   EXTERNAL;

PROCEDURE INITOPT
  (VAR opt       : byte_array_type;
       num_args  : ShortInt;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE IPCConnect
  (    call_sd   : Integer;
       pathdesc  : Integer;
   VAR flags     : Integer;
   VAR opt       : Byte_array_type;
   VAR vc_sd     : Integer;
   VAR error     : Integer);
   EXTERNAL;

PROCEDURE IPCControl
  (    socket    : integer;
       request   : integer;
   VAR wrtdata   : ShortInt;
       wrtlen    : Integer;
   VAR data      : Integer;
   VAR datalen   : Integer;
   VAR flags     : Integer;
   VAR result    : Integer );
   EXTERNAL;

PROCEDURE IPCCREATE
  (    socket    : integer;
       protocol  : integer;
   VAR flags     : integer;
   VAR opt       : byte_array_type;
   VAR csd       : integer;
   VAR result    : integer);
   EXTERNAL;

PROCEDURE IPCNAME
  (    descriptor : integer;
   VAR name       : name_array_type;
       nlen       : integer;
   VAR result     : integer);
   EXTERNAL;
```

```
PROCEDURE IPCDEST
   (    sock_kind : Integer;
    VAR node_name : Buffer_Type;
        name_len  : Integer;
        protocol  : Integer;
    VAR protoaddr : ShortInt;
        proto_len : Integer;
    VAR flags     : integer;
    VAR opt       : byte_array_type;
    VAR pathdesc  : Integer;
    VAR result    : Integer);
   EXTERNAL;

PROCEDURE IPCRECVCN
   (    csd    : integer;
    VAR vcsd   : integer;
    VAR flags  : integer;
    VAR opt    : byte_array_type;
    VAR result : integer);
   EXTERNAL;

PROCEDURE IPCRECV
  (    csd    :integer;
   VAR data  : InfoBufType;
   VAR dlen  : integer;
   VAR flags : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSelect
  (VAR sbound : Integer;
   VAR rmap   : BitMapType;
   VAR wmap   : BitMapType;
   VAR xmap   : BitMapType;
       timeout: Integer;
   VAR result : Integer );
   EXTERNAL;

PROCEDURE IPCSEND
  (    vcsd    : integer;
   VAR data    : buffer_type;
       dlen    : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSHUTDOWN
  (    vcsd   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;
```

```
$ TITLE 'Internal Procedures', PAGE $

PROCEDURE GetLen
(VAR  buffer      :    Buffer_Type;
 VAR  current_pos :    ShortInt;
 VAR  length      :    Integer );
   FORWARD;
   { Get the length of a string.  Return the next position }

PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);
   FORWARD;

PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);
   FORWARD;

PROCEDURE SetUp;
   FORWARD;
   { Create a call socket, connect to server using IPCDest }

PROCEDURE ShutdownSockets;
   FORWARD;
   { Shut down the call and vc sockets }


$ TITLE 'Error_Routine', PAGE $
PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);

   BEGIN    { Error_Routine }

   writeln('Client: Error occurred in ', where,' call.' );
   writeln('Client: The error code is: ', what:5,
                  '.  The local descriptor is: ', sd:4 );

   GOTO 89;

   END;    { Error_Routine }

$ TITLE 'GetLen', PAGE $
PROCEDURE GetLen
(VAR  buffer      :   Buffer_Type;
 VAR  current_pos :    ShortInt;
 VAR  length      :    Integer );

{ Get the length of a string.  Return the next position }

VAR
   orig_pos :    ShortInt;
```

```
   BEGIN    { GetLen }
   {}
   { Find the first blank in the string.  Return the difference
   { between the blank position, and the initial value of current_pos
   {}

   orig_pos := current_pos;

   WHILE buffer[current_pos] <> ' ' DO
      current_pos := current_pos + 1;

   { set the length value for the caller }
   length := current_pos - orig_pos;

   { increment beyond the space, for the next time }
   current_pos := current_pos + 1;

   END;     { GetLen }
$ TITLE 'Initialize_Option ', PAGE $

PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);

VAR
  opt_num_arguments : ShortInt;
  result            : ShortInt;

BEGIN   {Initialize_Option}

  opt_num_arguments := 0;
  INITOPT( opt_parameter,opt_num_arguments,result );
  IF result <> ZERO THEN
    BEGIN      { error on initopt }
    call_name := 'INITOPT   ';
    Error_Routine( call_name, result, 0 );
    END;       { error on initopt }

END;    {Initialize_Option}

$ TITLE 'SetUp', PAGE $
PROCEDURE SetUp;
{ Create a call socket using a well-known address }

VAR
   pathdesc     : Integer;

   BEGIN    { SetUp }

   { Prepare to create a call socket }
   socket_kind := CALL_SOCKET;
   protocol_kind := TCP;
```

```
{ clear the flags and option arrays }
flags_array := 0;
Initialize_Option( option );

{}
{A call socket is created by calling IPCCREATE.  The value returned
{in the call_sd parameter will be used in the following calls.
{}

IPCCREATE( socket_kind, protocol_kind, flags_array, option,
          call_sd, error_return );

IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCREATE ';
   Error_Routine( call_name,error_return, call_sd );
   END;

{}
{ The server is waiting on a well-known address.  Get the path
{ descriptor for the socket from the remote node.
{}
flags_array := 0;
protoaddr := PROTO_ADDR;

IPCDest( socket_kind, node_name, node_name_len, protocol_kind,
   protoaddr, INTEGER_LEN, flags_array, option,
   pathdesc, error_return );
IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCDEST   ';
   Error_Routine( call_name,error_return, pathdesc );
   END;

flags_array := 0;

{ Now connect to the server }
IPCConnect( call_sd, pathdesc, flags_array, option,
                         vc_sd, error_return );
IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCONNECT';
   Error_Routine( call_name,error_return, pathdesc );
   END;

{ Set the timeout to infinity with IPCControl for later calls }
flags_array := 0;
control_value := 0;
timeout_len := 2;

IPCControl( vc_sd, CHANGE_TIMEOUT, control_value, timeout_len,
   dummy_parm, dummy_len, flags_array, error_return );
```

```
   IF error_return <> ZERO THEN
      BEGIN
      call_name := 'IPCCONTROL';
      Error_Routine( call_name,error_return, vc_sd );
      END;

   flags_array := 0;
   Initialize_Option( option );

   {}
   { Verify the server received the connect req.  Wait for the
   { server to do an IPCRecvCn.
   {}
   IPCRecv( vc_sd, data_buf, buffer_len, flags_array,
                                   option, error_return );
   IF error_return <> ZERO THEN
      BEGIN
      call_name := 'IPCRECV   ';
      Error_Routine( call_name,error_return, vc_sd );
      END;

   END;     { SetUp }

$ TITLE 'ShutdownSockets', PAGE $
PROCEDURE ShutdownSockets;

VAR
   result    :             Integer;

   BEGIN    { ShutdownSockets }
   {}
   { We are terminating this program.  Clean up the allocated
   { sockets.
   {}
   flags_array := 0;
   Initialize_Option( option );

   IPCShutdown( vc_sd, flags_array, option, result );
   { Don't worry about errors here, since there isn't much we can do. }

   IPCShutdown( call_sd, flags_array, option, result );
   { Don't worry about errors here, since there isn't much we can do. }

   END;     { ShutdownSockets }

$TITLE 'Client MAIN', PAGE $
BEGIN { Client }

node_name_len := 0;
requested_name := '';

{ Ask the user for the NS node name of the remote node }
Prompt( 'Client: Enter the remote node name: ' );
Readln( node_name );
```

```
temp_position := 1;
GetLen( node_name, temp_position, node_name_len );

{ Create a call socket and connect to the server }
SetUp;

WHILE requested_name <> 'EOT' DO
   BEGIN    { loop for name }

   { Ask the user for a name to be retrieved }
   Prompt( 'Client: Enter name for data retrieval: ' );
   Readln( requested_name );
   req_name_len := BUFFERLEN;
   flags_array := 0;

   IF requested_name <> 'EOT' THEN
      BEGIN    { continue processing }

      { Ask for the name the user requested }
      IPCSend( vc_sd, requested_name, req_name_len, flags_array, option,
            error_return );

      { Block waiting for the response back from the server. }
      buffer_len := INFOBUFLEN;
      flags_array := 0;

      IPCRecv( vc_sd, data_buf, buffer_len,  flags_array, option,
            error_return );
        IF error_return <> ZERO THEN
          BEGIN      { error on initopt }
          call_name := 'IPCRECV   ';
          Error_Routine( call_name, error_return, vc_sd );
          END;       { error on initopt }

      { Print out the data received }
      Writeln( 'Client data is: ', data_buf );

      END;      { continue processing }
   END;     { loop for name }
89:

{ Clean up the call and vc sockets }
ShutDownSockets;

99:


END.  { Client }
```

Pascal/1000 Server NetIPC Program

```
$PASCAL '91790-18264 REV.5010 <880420.0919>'
$CDS$

{----------------------------------------------------------------------}
{}
{       NAME: SERVER
{     SOURCE: 91790-18264
{      RELOC: NONE
{       PGMR: LMS
{}
{----------------------------------------------------------------------}
{
{
{ PURPOSE:
{    To show the operation of the IpcSelect() call.
{
{ REVISION HISTORY
{}

PROGRAM server ( input, output );

LABEL
  99;

CONST

   ADDR_OPT_CODE = 128;
   BUFFERLEN = 20;
   CALL_SOCKET = 3;
   CHANGE_BACKLOG = 6;
   CHANGE_TIMEOUT = 3;
   FOREVER = TRUE;
   INFINITE_SELECT = -1;
   INFOBUFLEN = 60;
   INT16_LEN = 2;
   LENGTH_OF_DATA = 20;
   MAX_BACKLOG = 5;
   MAX_BUFF_SIZE = 1000;
   MAX_RCV_SIZE = 4;
   MAX_SEND_SIZE = 3;
   MAX_SOCKETS = 32;
   PROTO_ADDR = 31767;
   TCP = 4;
   ZERO = 0;
```

```
TYPE
  {}
  { WARNING: If this program is ported to the 800 you need to delete
  { this type declaration. HP-PA Pascal pre-defines this type.
  {}
      ShortInt = -32768..32767;

  {}
  { WARNING: The bits entry of this record is not portable.
  { The declaration is 1..32 on the 1000, and 0..63 on the 840.
  {}
  BitMapType = RECORD
      CASE Integer OF
      1: ( bits     : PACKED ARRAY[1..32] OF Boolean );
      2: ( longint  : Packed Array[1..2] OF Integer );
      3: ( ints     : ARRAY[1..4] OF ShortInt );
      END;

  byte = 0..255;
  byte_array_type = packed array [1..40] of byte;
  buffer_type = packed array [1..BUFFERLEN] of char;
  InfoBufType = packed array [1..INFOBUFLEN] of char;
  name_of_call_array_type = packed array [1..10] of char;
  name_array_type = packed array [1..7] of char;

VAR
  call_name                         : name_of_call_array_type;
  call_sd                           : integer;
  control_value                     : ShortInt;
  curr_rmap                         : BitMapType;
  curr_wmap                         : BitMapType;
  curr_xmap                         : BitMapType;
  dummy_parm                        : Integer;
  dummy_len                         : Integer;
  error_return                      : Integer;
  flags_array                       : integer;
  map_offset                        : ShortInt;
  opt_data                          : ShortInt;
  opt_num_arguments                 : ShortInt;
  option                            : byte_array_type;
  protoaddr                         : ShortInt;
  protocol_kind                     : Integer;
  rmap                              : BitMapType;
  sbound                            : Integer;
  short_error                       : ShortInt;
  socket_kind                       : Integer;
  timeout                           : Integer;
  timeout_len                       : Integer;
  vc_count                          : Integer;
  xmap                              : BitMapType;
```

```
$TITLE 'IPC Procedures', PAGE $
PROCEDURE ADDOPT
  (VAR opt       : byte_array_type;
       argnum    : ShortInt;
       optcode   : ShortInt;
       data_len  : ShortInt;
   VAR data      : ShortInt;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE INITOPT
  (VAR opt       : byte_array_type;
       num_args  : ShortInt;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE READOPT
  (VAR opt       : byte_array_type;
       argnum    : ShortInt;
   VAR optcode   : ShortInt;
   VAR data_len  : ShortInt;
   VAR data      : Integer;
   VAR error     : ShortInt);
   EXTERNAL;

PROCEDURE IPCControl
  (    socket    : integer;
       request   : integer;
   VAR wrtdata   : ShortInt;
       wrtlen    : Integer;
   VAR data      : Integer;
   VAR datalen   : Integer;
   VAR flags     : Integer;
   VAR result    : Integer );
   EXTERNAL;

PROCEDURE IPCCREATE
  (    socket    : integer;
       protocol  : integer;
   VAR flags     : integer;
   VAR opt       : byte_array_type;
   VAR csd       : integer;
   VAR result    : integer);
   EXTERNAL;

PROCEDURE IPCNAME
  (    descriptor : integer;
   VAR name       : name_array_type;
       nlen       : integer;
   VAR result     : integer);
   EXTERNAL;
```

```
PROCEDURE IPCRECVCN
   (    csd    : integer;
    VAR vcsd   : integer;
    VAR flags  : integer;
    VAR opt    : byte_array_type;
    VAR result : integer);
    EXTERNAL;

PROCEDURE IPCRECV
  (    csd    :integer;
   VAR data  : buffer_type;
   VAR dlen  : integer;
   VAR flags : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSelect
  (VAR sbound : Integer;
   VAR rmap   : BitMapType;
   VAR wmap   : BitMapType;
   VAR xmap   : BitMapType;
       timeout: Integer;
   VAR result : Integer );
   EXTERNAL;

PROCEDURE IPCSEND
  (    vcsd    : integer;
   VAR data    : InfoBufType;
       dlen   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

PROCEDURE IPCSHUTDOWN
  (    vcsd   : integer;
   VAR flags  : integer;
   VAR opt    : byte_array_type;
   VAR result : integer);
   EXTERNAL;

$ TITLE 'Internal Procedures', PAGE $

PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);
   FORWARD;

PROCEDURE HandleNewRequest;
   FORWARD;
   { A new client wants to talk to us, complete the vc establishment }
```

```
PROCEDURE Initialize_Option
  (VAR opt_parameter : byte_array_type);
   FORWARD;

PROCEDURE ProcessRead
   (    map_offset   : ShortInt );
   FORWARD;
   { Process the read that is waiting on a particular vc }

PROCEDURE ReadData
(VAR client_buf    : Buffer_Type;
 VAR output_buf    : InfoBufType );
   FORWARD;
  { Read the data from the file, prepare for the IPCSend call. }

PROCEDURE SetUp;
   FORWARD;
   { Create a call socket using a well-known address }

PROCEDURE ShutdownVC
   (    map_offset   : ShortInt );
   FORWARD;
   { Shut down a vc that the client no longer needs }

$ TITLE 'Error_Routine', PAGE $
PROCEDURE Error_Routine
  (VAR where  : name_of_call_array_type;
       what   : integer;
       sd     : integer);

   BEGIN    { Error_Routine }

   writeln('Server: Error occurred in ', where,' call.' );
   writeln('Server: The error code is: ', what:5,
                                    '.  The local descriptor is: ', sd:4 );

   GOTO 99;

   END;    { Error_Routine }

$ TITLE 'HandleNewRequest', PAGE $
PROCEDURE HandleNewRequest;
{ A new client wants to talk to us, complete the vc establishment }
VAR
   result    : Integer;
   vc_sd    : Integer;

   BEGIN    { HandleNewRequest }

   Initialize_Option( option );
   flags_array := 0;
```

```
   { Accept the connection for this new vc. }
   IPCRecvCn( call_sd, vc_sd, flags_array, option, result );
   IF result <> ZERO THEN
    BEGIN      { error on ipcrecvcn }
    call_name := 'IPCRECVCN ';
    Error_Routine(call_name,result, vc_sd );
    END;       { error on ipcrecvcn }

   { Increment the total number of active vcs for the server }
   vc_count := vc_count + 1;

   { Now set the read and exception maps for this new vc }
   rmap.bits[vc_sd] := TRUE;
   xmap.bits[vc_sd] := TRUE;

   { Set the timeout to infinity with IPCControl for later calls }
   flags_array := 0;
   control_value := 0;
   timeout_len := 2;

   IPCControl( vc_sd, CHANGE_TIMEOUT, control_value, timeout_len,
      dummy_parm, dummy_len, flags_array, error_return );

   IF error_return <> ZERO THEN
      BEGIN
      call_name := 'IPCCONTROL';
      Error_Routine( call_name,error_return, vc_sd );
      END;

   {}
   { Check if we have reached the maximum number of sockets.
   { If so, disallow any new requests by clearing the exception
   { map for the call socket.
   {}
   IF vc_count = MAX_SOCKETS -1 THEN
      BEGIN    { reached socket limit }

      xmap.bits[call_sd] := FALSE;
      END;     { reached socket limit }

   END;     { HandleNewRequest }

$ TITLE 'Initialize_Option ', PAGE $

PROCEDURE Initialize_Option
(VAR opt_parameter : byte_array_type);

VAR
   opt_num_arguments : ShortInt;
   result            : ShortInt;
```

```
BEGIN

  opt_num_arguments := 0;
  INITOPT( opt_parameter,opt_num_arguments,result );
  IF result <> ZERO THEN
    BEGIN      { error on initopt }
    call_name := 'INITOPT   ';
    Error_Routine( call_name, result, 0 );
    END;       { error on initopt }

END; {Initialize_Option}

$ TITLE 'ProcessRead', PAGE $
PROCEDURE ProcessRead
  (    map_offset   : ShortInt );
{ Process the read that is waiting on a particular vc }
VAR
   buffer_len      : Integer;
   client_buf      : Buffer_type;
   data_buf        : InfoBufType;
   result          : Integer;
   vc_sd           : Integer;

   BEGIN    { ProcessRead }
   { There is a pending read on a vc.  Do an IPCRecv on the vc }
   flags_array := 0;
   Initialize_Option( option );

   vc_sd := map_offset;

   { Get the name this client wants data for }
   buffer_len := BUFFERLEN;

   IPCRecv( vc_sd, client_buf, buffer_len,
      flags_array, option, result );
   IF result <> ZERO THEN
   BEGIN       { error on ipcrecv }
   call_name := 'IPCRECV ';
   Error_Routine( call_name,result,vc_sd );
   END;        { error on ipcrecv }

   { Get the data we need from the file to send to the client }
    ReadData( client_buf, data_buf );
    buffer_len := INFOBUFLEN;

   IPCSend( vc_sd, data_buf, buffer_len, flags_array,
      option, result );
   IF result <> ZERO THEN
      BEGIN       { error on ipcsend }
      call_name := 'IPCSEND ';
      Error_Routine( call_name,result,vc_sd );
      END;        { error on ipcsend }
```

```
   END;     { ProcessRead }

$ TITLE 'ReadData', PAGE $
PROCEDURE ReadData
(VAR client_buf    : Buffer_Type;
 VAR output_buf    : InfoBufType );
  { Read the data from the file, prepare for the IPCSend call. }

CONST
   LAST_REC    = 4;

VAR
   current_rec    : ShortInt;
   datafile       : TEXT;
   info_buf       : InfoBufType;
   infofile       : Buffer_Type;
   found          : Boolean;
   name_buf       : Buffer_Type;

   BEGIN    { ReadData }

   {}
   { Open the file named "datafile".  Search until the last record
   { is found, or we match the user name the client wants.
   { If there is a match, retrieve the remaining data from the
   { file, and prepare to send it back.
   {
   { If there is no match, return "name not found" to the client.
   {}

   found := FALSE;
   current_rec := 1;
   infofile := 'datafile';

    RESET( datafile, infofile );

    WHILE ( NOT found ) AND ( current_rec <= LAST_REC ) DO
       BEGIN    { search the file }

       READLN( datafile, name_buf, info_buf );

       IF client_buf = name_buf THEN
          BEGIN    { found a match }
          {}
          { We found the name the client requested in the file.
          { Set the flag to fall out of the while loop, and
          { get the buffer to be sent to the client.
          {}
          writeln( 'Server: ', client_buf, ' information found.' );

          found := TRUE;
          output_buf := info_buf;

          END;     { found a match }
```

```
        { increment to test the next record in the file }
        current_rec := current_rec +1;

        END;     { search the file }

   {}
   { We've fallen out of the WHILE loop because there is a match,
   { or we reached the end of the file.  Find out which one it is.
   {}

   IF NOT found THEN
      BEGIN    { didn't find the requested name }
      {}
      { We didn't find the data in the file.  Put an error
      { message in the data buffer.
      {}
      writeln ('Server: ', client_buf, ' not in file.' );

      output_buf :=
    'SERVER did not find the requested name in the datafile.    ';

      END;     { didn't find the requested name }


   END;     { ReadData }

$ TITLE 'SetUp', PAGE $
PROCEDURE SetUp;
{ Create a call socket using a well-known address }

   BEGIN    { SetUp }

   { Set up the opt array for the two parms we will use }
   opt_num_arguments := 2;
   InitOpt( option, opt_num_arguments, short_error );
   IF short_error <> ZERO THEN
      BEGIN    { error on initopt }
      call_name := 'InitOpt';
      error_return := short_error;
      Error_Routine( call_name,error_return,call_sd );
      END;     { error on initopt }

   { Now add the option for the well-known address for the IPCCreate Call }
   protoaddr := PROTO_ADDR;

   AddOpt( option, 0, ADDR_OPT_CODE, INT16_LEN, protoaddr, short_error );
   IF short_error <> ZERO THEN
      BEGIN    { error on AddOpt }
      call_name := 'AddOpt';
      error_return := short_error;
      Error_Routine( call_name,error_return,call_sd );
      END;     { error on AddOpt }
```

```
   { Change the backlog queue to the maximum }
   opt_data := MAX_BACKLOG;
   AddOpt( option, 1, CHANGE_BACKLOG, INT16_LEN, opt_data, short_error );
   IF short_error <> ZERO THEN
      BEGIN    { error on AddOpt }
      call_name := 'AddOpt';
      error_return := short_error;
      Error_Routine( call_name,error_return,call_sd );
      END;     { error on AddOpt }

   { Prepare to create a call socket }
   socket_kind := CALL_SOCKET;
   protocol_kind := TCP;

   { clear the flags array }
   flags_array := 0;

   {}
   {A call socket is created by calling IPCCREATE.  The value returned
   {in the call_sd parameter will be used in the following calls.
   {}

   IPCCREATE( socket_kind, protocol_kind, flags_array, option,
              call_sd, error_return );

   IF error_return <> ZERO THEN
      BEGIN
      call_name := 'IPCCREATE ';
      Error_Routine( call_name,error_return,call_sd );
      END;

{ Set the call_sd timeout to infinity with IPCControl for later calls }
flags_array := 0;
control_value := 0;
timeout_len := 2;

IPCControl( call_sd, CHANGE_TIMEOUT, control_value, timeout_len,
   dummy_parm, dummy_len, flags_array, error_return );

IF error_return <> ZERO THEN
   BEGIN
   call_name := 'IPCCONTROL';
   Error_Routine( call_name,error_return,call_sd );
   END;

   { Now set IPCSelect's bit map for the call socket }
   xmap.bits[call_sd] := TRUE;

   END;     { SetUp }

$ TITLE 'ShutdownVC', PAGE $
PROCEDURE ShutdownVC
   (    map_offset   : ShortInt );
{ Shut down a vc that the client no longer needs }
```

```
VAR
   result    :             Integer;
   vc_sd     :             Integer;

   BEGIN    { ShutdownVC }
   {}
   { The client shut down the vc, or it has gone down due to a
   { Networking problem.  Either way, merely accept the shutdown.
   {}
   flags_array := 0;
   Initialize_Option( option );

   vc_sd := map_offset;

   IPCShutdown( vc_sd, flags_array, option, result );
   { Don't worry about errors here, since there isn't much we can do. }

   { Decrement the number of active vcs }
   vc_count := vc_count -1;

   { Clear the read map and exception map bits for this vc }
   rmap.bits[map_offset] := FALSE;
   xmap.bits[map_offset] := FALSE;

   {}
   { Always set the exception map for the call socket.  That way
   { we'll be sure to re-enable new requests if we were at the
   { limit before this vc was shut down.
   {}
   xmap.bits[call_sd] := TRUE;

   END;     { ShutdownVC }

$TITLE 'Server MAIN', PAGE $
BEGIN { Server }

{ Create a call socket with a well known address for the clients to use. }
SetUp;

{}
{ Loop forever waiting to serve clients.  If any new clients request
{ service, the exception map will be set on the call socket.  If
{ a client asks for information, the read map will be set on the
{ vc socket for that client.  When the client has received the data,
{ it will shut down the vc, and the vc socket will have the exception
{ map set.  Handle each one of these cases in this loop.
{
{ If any other situations occur, exit out of the loop, and let the
{ NS clean up routines de-allocate the sockets for this server.
{}

WHILE FOREVER = TRUE DO
   BEGIN    { Forever Do }
```

```
{}
{Set the bit masks to check for all the vcs that we own.
{ The rmap & xmap variables are maintained by ProcessNewRequest
{ and ShutdownVC.
{}
curr_rmap := rmap;
curr_xmap := xmap;


sbound := MAX_SOCKETS;
timeout := INFINITE_SELECT;

{}
{ Do an exceptional select on the call socket, and on all vcs
{ we own.    Do a read select on all the vc sockets.
{}

IPCSelect( sbound, curr_rmap, curr_wmap, curr_xmap,
   timeout, error_return );
IF error_return <> ZERO THEN
   BEGIN       { Select Error }
   call_name := 'IPCSELECT ';
   Error_Routine( call_name,error_return,call_sd );
   END;        { Select Error }

{ See if there are any clients requesting information }
IF ( curr_rmap.longint[1] <> 0 ) OR
                ( curr_rmap.longint[2] <> 0 ) THEN
   BEGIN    { Process read on VC sockets }

   { We have someone to service.  Find out who it is. }
   FOR map_offset := 1 TO MAX_SOCKETS DO
      BEGIN    { check all vcs }

      IF curr_rmap.bits[map_offset] = TRUE THEN
         BEGIN    { have read on a vc }

         {}
         { We know the client who needs service,
         { Do an IPCRecv, get the necessary data,
         { and do an IPCSend to send it back.
         {}
         ProcessRead( map_offset );

         END;     { have read on a vc }
      END;     { check all vcs }
   END;     { Process read on VC sockets }

{ See if any clients have sent a message to the call socket }
IF curr_xmap.bits[call_sd] = TRUE THEN
   BEGIN    { new request on the call socket }

      {}
      { We have a new client, go do an IPCRecvCn, and set the
      { bit masks to accept reads and exceptions on the new vc.
```

```
      {}
      HandleNewRequest;


      { Clear the call socket xmap bit to simplify the test for the vcs }
      curr_xmap.bits[call_sd] := FALSE;

      END;     { new request on the call socket }

   {}
   { If we get an exception on a vc socket, shut it down. The client
   { knows to shut down a socket once it has received the data it needs.
   {}
   IF ( curr_xmap.longint[1] <> 0 ) OR ( curr_xmap.longint[2] <> 0 ) THEN
      BEGIN    { check for errors on vc sockets }

      { One vc had an exception, find out which one }
      FOR map_offset := 1 TO MAX_SOCKETS DO
         BEGIN       { check all vcs }

         IF curr_xmap.bits[map_offset] = TRUE THEN
            BEGIN    { shut down the vc }

            {}
            { Do an IPCShutdown on the vc, and clear
            { its bit in both the read and exception maps.
            {}
            ShutdownVC( map_offset );

            END;     { shut down the vc }
         END;        { check all vcs }
      END;     { check for errors on vc sockets }

   END;     { Forever Do }

99:

{}
{ We have some problem, the NS cleanup routine will shut down
{ all the sockets we own once the program has terminated.
{}

END.  { Server }
```

NetIPC Program Data Example

```
Mickey Mouse        Lives at Disneyland, and goes steady with Minnie.
Donald Duck         Has three nephews: Huey, Dewey, and Louie.
Snow White          & the 7 dwarfs: Sleepy, Dopey, Grumpy, Sneezy, Bashful,etc
Peter Pan           Loves to fly around the sky with his friend Tinkerbell.

{-------------------------------------------------------------------}
datafile 91790-17084 rev.5010 <880419.1522>
{       NAME: DATAFILE
{     SOURCE: 91790-17084
{      RELOC: NONE
{       PGMR: LMS
{
{   This is the data file used by SERVER.FTN and SERVER.PAS
{   The data portion of this file (from the first line to the
{   comments at the end) MUST remain in its present format!
{
{   All names and places listed above are trademarks of Walt Disney
{   Productions.
{-------------------------------------------------------------------}
```

FORTRAN 77 Client NetIPC Program

```
FTN77,L
$cds on
$files(1,1)

      PROGRAM client(4,99),91790-18265 REV.5010 <880901.1020>
C
C       NAME: CLIENT
C     SOURCE: 91790-18265
C      RELOC: NONE
C       PGMR: KB
C
C     This program is the peer process to server.  It requests a
C     connection with server and exchanges messages in synchronous mode.
C     The server can be located at an HP 9000 Series 800, HP 1000,
C     or HP 3000 node.

      IMPLICIT None

C   VARIABLE DECLARATIONS:

C     The variable declarations for each NetIPC call are separated for
C     clarity.  However, declarations for variables which have been
C     declared for previous calls are commented out.   The purpose of
C     this is to demonstrate the complete set of declarations needed for
C     each NetIPC call.
C
C     Two exclamation points (!!) next to a variable name indicates that
C     its value may be changed by the call.
C
C   INITOPT:
      INTEGER*2 options(14) !! INITOPT initializes the options parameter so
      INTEGER*2 optnumargs   ! that 'optnumargs' arguments can be added.
      INTEGER*2 error       !! REFER TO 'OPT PARAMETERS' SECTION OF THE
                             ! NETIPC REFERENCE MANUAL FOR IMPORTANT INFO
                             ! REGARDING options BYTE ARRAY LENGTH!!!
C   IPCCREATE:
      INTEGER*4 socketkind   ! IPCCREATE creates a call socket descriptor
      INTEGER*4 protocol     ! using the options prepared by ADDOPT.
      INTEGER*4 flags        ! 'Calldesc' will reference the socket.
C     INTEGER*2 options(14)  ! 'Socketkind' must be 3 to mean call socket.
      INTEGER*4 calldesc    !! Protocol of 4 specifies the TCP protocol.
      INTEGER*4 resultcode  !! Other socket kinds, protocols, and use of the
                             ! flags parameter are reserved for future use.
C
C   IPCDEST:
C     INTEGER*4 socketkind   ! IPCDEST obtains a path report descriptor,
      INTEGER*2 nodename(48) ! pathdesc, by specifying a protocol address
      INTEGER*4 nodelen      ! in the protoaddr parameter.  This is an
C     INTEGER*4 protocol     ! alternative to using IPCLOOKUP and IPCNAME
      INTEGER*2 protoaddr    ! but IPCDEST is a local call.  That is, the
      INTEGER*4 protolen     ! IPCDEST does not verify that the remote end-
C     INTEGER*4 flags        ! point described by the input parameters
```

```
C     INTEGER*2 options(14)  ! exists.  This checking is done when the
      INTEGER*4 pathdesc    !! pathdesc is used for the first time.  See
C     INTEGER*4 resultcode  !! the IPCDEST section of the manual for more.
C
C   IPCCONNECT:
C     INTEGER*4 calldesc     ! IPCCONNECT initiates a virtual circuit on
C     INTEGER*4 pathdesc     ! which data may be sent and received.
C     INTEGER*4 flags        ! IPCCONNECT takes a call desc and a path desc,
C     INTEGER*2 options(14)  ! and creates a virtual circuit referenced by
      INTEGER*4 vcdesc      !! vcdesc.
C     INTEGER*4 resultcode  !!
C
C   IPCCONTROL:
      INTEGER*4 descriptor   ! IPCCONTROL performs specialized requests
      INTEGER*4 request      ! on sockets.  The socket and request are
      INTEGER*2 wrtdata      ! specified by 'descriptor' and 'request',
      INTEGER*4 wlen         ! respectively.  Use of the other parameters
      INTEGER*2 readdata    !! will vary depending on 'request'.  Please
      INTEGER*4 rlen        !! refer to the IPCCONTROL section of the
C     INTEGER*4 flags        ! Reference Manual for further IMPORTANT
C     INTEGER*4 resultcode  !! information.
C
C   IPCSEND:
C     INTEGER*4 vcdesc       ! IPCSEND sends data buffer 'senddata' of
      INTEGER*2 senddata(10) ! length 'dlength' over the VC connection.
      INTEGER*4 dlength      ! Here, we will use 20 byte data buffers.
C     INTEGER*4 flags        ! Refer to the reference manual for discussion
C     INTEGER*2 options(14)  ! of the flags and options parameters with
C     INTEGER*4 resultcode   ! this call.
C
C   IPCRECV:
C     INTEGER*4 vcdesc       ! IPCRECV receives data from the VC connection
      INTEGER*2 recvdata(30) ! and stores it in 'recvdata'.  Here, we use
C     INTEGER*4 dlength      ! 60 byte data buffers.  Refer to the reference
C     INTEGER*4 flags        ! for discussion of the flags and options
C     INTEGER*2 options(14)  ! parameters with this call.
C     INTEGER*4 resultcode   !
C
      CHARACTER BLANK*1, EOT*3
      CHARACTER buffer*20

      CHARACTER*11 this_call

C   FOR GETST CALL:
      INTEGER*2 maxlen
      INTEGER*2 tlog

      INTEGER*2 i            ! counter

      EQUIVALENCE (buffer, senddata)
```

```
      DATA BLANK/' '/, EOT/'EOT'/
      DATA socketkind/3/,protocol/4/,flags/0/

      maxlen = -48
      CALL GETST(nodename,maxlen,tlog)
      IF (tlog .EQ. 0) THEN
          WRITE (1,*) 'client : Usage:  ru,client nodename'
          STOP
      ENDIF

C     INITOPT is called to initialize the option parameter used in the
C     IPCCREATE, IPCLOOKUP, IPCCONNECT, IPCRECV, IPCSEND and
C     IPCSHUTDOWN calls.  By setting optnumargs to zero, the
C     option parameter is initialized to contain zero entries.

      optnumargs = 0
      error = 0
      this_call = 'INITOPT    '
      CALL INITOPT(options,optnumargs,error)
      IF(resultcode .NE. 0) CALL OPT_ERROR(error, this_call)

C     A call socket is created by calling IPCCREATE.  The value returned
C     in the calldesc parameter will be used in the following IPCCONNECT
C     call.  The flags parameter is not used in this program so flags
C     is made a double integer and assigned the value zero to ensure that
C     all the bits are clear.

      resultcode = 0
      this_call = 'IPCCREATE  '
      CALL IPCCREATE(socketkind,protocol,flags,options,calldesc,
     >               resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)

C     IPCDest obtains the path_report_des by using a well known TCP
C     address that the server has previously assigned to its call socket
C     using ADDOPT and IPCCREATE.  Both the client and the server must
C     use the same address.  Using IPCDest in this way is one way
C     to obtain a path report descriptor.  Another way would be to
C     assign a character name to the call socket in the server program
C     using IPCName, and then to use IPCLookup in the client program
C     to find the get the path report descriptor.  The two methods are
C     equivalent although both client and server must use the same one.

      nodelen = tlog
      protoaddr = 31767      !"well-known" address
      protolen = 2
      flags = 0
      resultcode = 0
      this_call = 'IPCDEST    '
      CALL IPCDEST(socketkind,nodename,nodelen,protocol,protoaddr,
     >             protolen,flags,options,pathdesc,resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)
```

```
C     The calldesc returned by IPCCREATE and the pathdesc returned by
C     IPCLOOKUP are used in IPCCONNECT to request a connection with the
C     server.
C     The vcdesc returned by IPCCONNECT is used in subsequent
C     calls to reference the connection.  Once this call has completed
C     successfully there is no longer a need for the call socket.  Thus,
C     you may release the call socket descriptor by calling IPCSHUTDOWN
C     to return the resources to the system, or by calling IPCGive to
C     give the call socket descriptor to another process that uses IPCGet.
C     Releasing the call socket descriptor has no effect upon the VC
C     socket or its descriptor.

      flags = 0
      resultcode = 0
      this_call = 'IPCCONNECT '
      CALL IPCCONNECT(calldesc,pathdesc,flags,options,vcdesc,resultcode)
      IF(resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)

C     IPCControl is called in order to set the timeout to infinity.  The
C     request of 3 means we are setting the timeout.  A wrtdata of 0 means
C     timeout is infinity.

      descriptor = vcdesc
      request = 3   ! timeout
      wrtdata = 0   ! infinite
      wlen = 2
      flags = 0
      resultcode = 0
      this_call = 'IPCCONTROL '
      CALL IPCCONTROL (vcdesc,request,wrtdata,wlen,readdata,rlen,
     >                 flags, resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)

C     IPCRECV is called to determine if the connection has been
C     established.

      flags = 0
      dlength = 60
      resultcode = 0
      this_call = 'IPCRECV    '
      CALL IPCRECV(vcdesc,recvdata,dlength,flags,options,resultcode)
      IF(resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)

C     Loop forever till user types in 'EOT' in response to ENTER NAME:
C     Client will then terminate itself and let the networking code
C     clean up which will notify server via the exceptional condition
C     on the appropriate VC socket.

      DO WHILE (.TRUE.)
88        WRITE(1,*) 'ENTER NAME: _'
          READ (1, '(A20)') buffer
          IF (buffer .EQ. EOT) STOP
          IF (buffer .EQ. BLANK) THEN
              WRITE(1,*) 'Type EOT to terminate.'
```

```
              GO TO 88
          END IF

C     Data is sent to server on the newly established connection.

          dlength = 20
          flags = 0
          resultcode = 0
          this_call = 'IPCSEND    '
          CALL IPCSEND(vcdesc,senddata,dlength,flags,options,
     >                 resultcode)
          IF(resultcode .NE. 0) CALL RESULT_ERROR(resultcode,
     >                                            this_call)

C   The client calls IPCRECV to receive 60 bytes of data from server.

          dlength = 60
          flags = 0                      ! Clear flags first, then
          flags = IBSET(flags,(31-20)) ! set DATA WAIT flag [20] so that
                                         ! we are sure to get 60 bytes back.
          resultcode = 0
          this_call = 'IPCRECV#2  '
          CALL IPCRECV(vcdesc,recvdata,dlength,flags,options,
     >                 resultcode)
          IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,
     >                                             this_call)
          WRITE(1,*) buffer,'_'
          WRITE(1,'(30A2)') (recvdata(i), i=1,30)
      END DO

      END  ! end of MAIN program
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                            C
C  SUBROUTINE OPT_ERROR                                                      C
C     OPT_ERROR is called when ADDOPT or INITOPT returns with an error       C
C     condition (error .NE. 0).                                              C
C                                                                            C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE OPT_ERROR (code, which_call)

      INTEGER*2 code           ! error code returned from call

      CHARACTER*11 which_call  ! call which produced error

      WRITE(1,'("ERROR CODE RETURNED FROM ",A11)') which_call
      WRITE(1,'("THE ERROR CODE WAS ",I4)') code

      STOP

      END ! end of OPT_ERROR

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                            C
C  SUBROUTINE RESULT_ERROR                                                   C
C     RESULT_ERROR is called when a NETIPC call returns with an error        C
C     condition (resultcode .NE. 0).                                         C
C                                                                            C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE RESULT_ERROR (code, which_call)

      INTEGER*4 code           ! error code returned from call

      CHARACTER*11 which_call  ! call which produced error

      WRITE(1,'("ERROR CODE RETURNED FROM ",A11)') which_call
      WRITE(1,'("THE ERROR CODE WAS ",I4)') code

      STOP

      END ! end of RESULT_ERROR
```

FORTRAN 77 Server NetIPC Program

```
FTN77,L
$cds on
$files(1,1)

      PROGRAM server(4,99),91790-18266 REV.5010 <880420.1507>

C
C       NAME: SERVER
C     SOURCE: 91790-18266
C      RELOC: NONE
C       PGMR: KB
C

C     This program is the peer process to requester.  It establishes a
C     connection with client upon client's request and exchanges messages
C     in synchronous mode.  The requestor can be located at an HP 9000,
C     Series 800, HP 1000 or HP 3000 node.

      IMPLICIT None

C   VARIABLE DECLARATIONS:
C     The variable declarations for each NetIPC call are separated for
C     clarity.  However, declarations for variables which have been
C     declared for previous calls are commented out.   The purpose of
C     this is to demonstrate the complete set of declaration needed for
C     each NetIPC call.
C
C   INITOPT:
    INTEGER*2 options(24)     ! INITOPT initializes the options parameter so
    INTEGER*2 optnumargs      ! that 'optnumargs' arguments can be added.
    INTEGER*2 error           ! REFER TO 'OPT PARAMETERS' SECTION OF THE
                              ! NETIPC REFERENCE MANUAL FOR IMPORTANT INFO
                              ! REGARDING options BYTE ARRAY LENGTH!!!

C   ADDOPT:
C     INTEGER*2 options(24)   ! ADDOPT adds an option to the entity 'options'.
      INTEGER*2 argnum        ! 'Options' can hold more than one option, so
      INTEGER*2 optioncode    ! 'argnum' specifies which one is being added.
      INTEGER*2 dlength       ! 'Optioncode' says which type of option is to
      INTEGER*2 data          ! be added, and 'data' says what that option
C     INTEGER*2 error         ! is to be.
C
C   IPCCREATE:
      INTEGER*4 socketkind    ! IPCCREATE creates a call socket descriptor
      INTEGER*4 protocol      ! using the options prepared by ADDOPT.
      INTEGER*4 flags         ! 'Calldesc' will reference the socket.
C     INTEGER*2 options(24)   ! 'Socketkind' must be 3 to mean call socket.
      INTEGER*4 calldesc      ! Protocol of 4 specifies the TCP protocol.
      INTEGER*4 resultcode    ! Other socket kinds, protocols, and use of the
                              ! flags parameter are reserved for future use.
```

```
C
C   IPCCONTROL:
      INTEGER*4 descriptor   ! IPCCONTROL performs specialized requests
      INTEGER*4 request      ! on sockets.  The socket and request are
      INTEGER*2 wrtdata      ! specified by 'descriptor' and 'request',
      INTEGER*4 wlen         ! respectively.  Use of the other parameters
      INTEGER*2 readdata     ! will vary depending on 'request'.  Please
      INTEGER*4 rlen         ! refer to the IPCCONTROL section of the
C     INTEGER*4 flags        ! Reference Manual for further IMPORTANT
C     INTEGER*4 resultcode   ! information.
C
C   IPCSELECT:
      INTEGER*4 sdbound      ! IPCSELECT determines the status of a call
      INTEGER*4 readmap      ! socket or virtual circuit socket.  The 'map'
      INTEGER*4 writemap     ! parameters are bit maps which indicate
      INTEGER*4 exceptionmap ! whether sockets are readable, writeable, or
      INTEGER*4 timeout      ! exceptional.  PLEASE REFER TO THE REFERENCE
C     INTEGER*4 resultcode   ! MANUAL FOR DISCUSSION OF THESE TOPICS!
C
C   IPCSHUTDOWN:
C     INTEGER*4 descriptor   ! IPCSHUTDOWN shuts down a socket and returns
C     INTEGER*4 flags        ! its resources to the system.  PLEASE REFER
C     INTEGER*2 options(24)  ! TO 'SHUTTING DOWN A CONNECTION' IN THE
C     INTEGER*4 resultcode   ! REFERENCE MANUAL!
C
C   OTHER VARIABLES & CONSTANTS:
      INTEGER*2 MAX_BACKLOG  ! Maximum number of pending connection requests

      INTEGER*4 MAX_DESC     ! Maximum number of sockets allowed at one time.

      INTEGER*2 active_vc    ! Number of active virtual circuit connections.

      CHARACTER*8 FNAME      ! Name of datafile from which to read data

      CHARACTER*11 this_call ! Name of last NetIPC call used.  This is for
                             ! identifying where errors occur.

      INTEGER*4 current_rmap ! current readmap

      COMMON MAX_DESC

C  CONSTANT VALUES ASSIGNED:
      DATA socketkind/3/,protocol/4/,flags/0/
      DATA MAX_BACKLOG/5/
      DATA FNAME/'DATAFILE'/
      DATA MAX_DESC/32/
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                               C
C BEGIN MAIN PROGRAM SERVER:                                                     C
C                                                                               C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

C Open database file needed to service clients:
      error = 0
      this_call = 'OPEN       '
      OPEN (77,FILE=FNAME,IOSTAT=error,USE='EXCLUSIVE',STATUS='UNKNOWN')
      IF (error .NE. 0) CALL OPT_ERROR(error,this_call)

C Initialize options to contain 2 parameters:

      optnumargs = 2      ! We will assign 2 args: TCP address & max backlog.
      error = 0           ! Reset the error code.
      this_call = 'INITOPT    '
      call INITOPT(options, optnumargs, error)
      IF (error .NE. 0) CALL OPT_ERROR(error, this_call)

C Assign an option to ADDOPT to give a TCP address to the call socket
C during the IPCCREATE call.

      argnum = 0         ! This is the first argument we are assigning.
      optioncode = 128   ! Option 128 means "assign a TCP address".
      dlength = 2        ! A TCP address is two bytes long.
      data = 31767       ! We will assign TCP address of 31767.
      error = 0          ! Reset the error code.
      this_call = 'ADDOPT     '
      CALL ADDOPT(options, argnum, optioncode, dlength, data, error)
      IF (error .NE. 0) CALL OPT_ERROR(error, this_call)

C Assign an option to ADDOPT to set maximum request backlog to 5
C during the IPCCREATE call.

      argnum = 1         ! This is the second argument we are assigning.
      optioncode = 6     ! Option 6 means "set the maximum backlog".
      dlength = 2        ! Data is two bytes long.
      data = 5           ! We will set maximum backlog to 5.
      error = 0          ! Reset the error code.
      this_call = 'ADDOPT     '
      CALL ADDOPT(options, argnum, optioncode, dlength, data, error)
      IF (error .NE. 0) CALL OPT_ERROR(error, this_call)

C Create a call socket by calling IPCCREATE.  In the variable declarations
C and initializations, we specified a call socket with the TCP protocol and
C we set 'flags' to zero since it is not used here.

      resultcode = 0    ! Reset the result code.
      this_call = 'IPCCREATE  '
      CALL IPCCREATE(socketkind,protocol,flags,options,calldesc,
     >               resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)
```

```
C Call IPCCONTROL to set the synchronous timeout to infinity.
C This is so that the server will wait indefinitely for a client
C to make a request.

      descriptor = calldesc   ! We will set the call socket timeout.
      request = 3             ! Request of 3 is "set synchronous timeout".
      wrtdata = 0             ! Set timeout to 0, which means infinity, not 0.
      wlen = 2                ! Wrtdata is 2 bytes long.
C     readdata                ! Reserved for future use: value unimportant.
C     rlen                    ! Reserved for future use: value unimportant.
      flags = 0               ! Reserved for future use but must be zero.
      resultcode = 0          ! Reset the result code.
      this_call = 'IPCCONTROL '
      CALL IPCCONTROL(descriptor,request,wrtdata,wlen,readdata,rlen,
     >                flags, resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)

C In preparation to start monitoring the call sockets for incoming requests,
C set the bit mask to receive connection(s) on the newly allocated call
C socket.  PLEASE SEE THE IPCSELECT SECTION OF THE REFERENCE MANUAL FOR
C IMPORTANT INFORMATION ABOUT IPCSELECT BIT MASKS.

      exceptionmap = 0
      exceptionmap = ibset(exceptionmap, (MAX_DESC - calldesc))

C Check the call socket descriptors to see which ones are exceptional.
C Exceptional call sockets already have connection requests queued on
C them.  The other sockets are readable.  IPCSELECT has an infinite
C timeout, so it will block until a connection request is received
C or until data is received over a VC connection.  After IPCSELECT
C returns, we will process the data received, or the connection request.

      sdbound = MAX_DESC    ! Maximum number of descriptors.
      timeout = -1          ! Infinite timeout so IPCSELECT will block.
      writemap = 0          ! Clear writemap bits.
      readmap = 0           ! Clear readmap bits to receive connection
                            ! on newly established VC connections.
      current_rmap = 0      ! Clear the current readmap.
      active_vc = 1         ! The first active VC will be number 1.

      DO WHILE (.TRUE.)     ! Forever loop.
         resultcode = 0     ! Reset result code.
         this_call = 'IPCSELECT  '
         CALL IPCSELECT(sdbound,readmap,writemap,exceptionmap,timeout,
     >                  resultcode)
         IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,this_call)
C        A bit in the readmap is set when there is VC data to read.
         IF (readmap .NE. 0) CALL PROCESS_DATA(readmap,exceptionmap)
```

```
C        A bit in the exceptionmap is set when a connection request is
C        detected, or when a VC shutdown is detected.
         IF (exceptionmap .NE. 0) CALL PROCESS_XMAP (calldesc,
     >            exceptionmap, active_vc, current_rmap)
         sdbound = MAX_DESC ! Reset sdbound since IPCSELECT changes it.
         writemap = 0       ! Reset writemap since IPCSELECT changes it.
         readmap = current_rmap
         exceptionmap = current_rmap
         IF (active_vc .LT. MAX_DESC) THEN
             exceptionmap = IBSET(exceptionmap, (MAX_DESC - calldesc))
         END IF
      END DO

       END   ! END OF MAIN PROGRAM

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  SUBROUTINE PROCESS_XMAP                                                     C
C     PROCESS_XMAP is called from MAIN when a bit of the exceptionmap is      C
C     set.  A bit in the exceptionmap will be set when either of the          C
C     following occurs:                                                       C
C              1.  There is a connection request.  In this case, the          C
C                  bit which is set corresponds to a call socket.             C
C              2.  A virtual circuit connection shuts down.  In this          C
C                  case, the bit which is set corresponds to a VC socket.     C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE PROCESS_XMAP (calldesc, exceptionmap,
     >                         active_vc, current_rmap)

      IMPLICIT none

C   INITOPT:
      INTEGER*2 options(24)  ! INITOPT initializes the options parameter so
      INTEGER*2 optnumargs   ! that 'optnumargs' arguments can be added.
      INTEGER*2 error        ! REFER TO 'OPT PARAMETERS' SECTION OF THE
                             ! NETIPC REFERENCE MANUAL FOR IMPORTANT INFO
                             ! REGARDING options BYTE ARRAY LENGTH!!!
                             ! The rule is: array_len=4+8*optnumargs+data
C   IPCRECVCN:
      INTEGER*4 calldesc     ! IPCRECVCN receives a connection request on
      INTEGER*4 vcdesc       ! the call socket specified by calldesc,
      INTEGER*4 flags        ! returns a virtual circuit descriptor, vcdesc
C     INTEGER*2 options(24)  ! which refers to the VC created.  Here, options
      INTEGER*4 resultcode   ! can be specified to alter the max send and
                             ! receive sizes for the VC connection.

      INTEGER*2 active_vc    ! the ordinal number of the active VC.

      INTEGER*4 MAX_DESC     ! the maximum number of descriptors

      INTEGER*4 VC_DESC      ! the current descriptor
```

```
      CHARACTER*11 this_call ! name of IPC call that may have an error.

      INTEGER*4 current_rmap ! current readmap

      INTEGER*4 exceptionmap ! has bits set for aborted connections and
                             ! connection requests

      COMMON MAX_DESC

C Reset the options array to 0 so IPCRECVCN and IPCSHUTDOWN don't complain.

      optnumargs = 0
      error = 0
      this_call = 'INITOPT    '
      CALL INITOPT(options, optnumargs, error)
      IF (error .NE. 0) CALL OPT_ERROR(error, this_call)

C Find out if there are any connection requests by examining the
C exceptionmap.

      IF (BTEST(exceptionmap, (MAX_DESC - calldesc))) THEN
C        A new connection has been requested.
         flags = 0      ! reset flags array
         resultcode = 0
         this_call = 'IPCRECVCN  '
         CALL IPCRECVCN(calldesc, vcdesc, flags, options, resultcode)
         IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode, this_call)
         active_vc = active_vc + 1
C       Now set the read bit for the VC and clear the calldesc exception bit.
        current_rmap = IBSET(current_rmap, (MAX_DESC - vcdesc))
        exceptionmap = IBCLR(exceptionmap, (MAX_DESC - calldesc))
     END IF

C Find out if VC sockets are exceptional (connections aborted).
C If so, shut down socket and update readmap for next IPCSELECT call.

      VC_DESC = 0
      exceptionmap = IBCLR(exceptionmap, (MAX_DESC - calldesc))
      DO WHILE ((exceptionmap .NE. 0) .AND. (VC_DESC .LT. MAX_DESC))
         IF (BTEST(exceptionmap, (MAX_DESC - VC_DESC))) THEN
            flags = 0
            resultcode = 0
            this_call = 'IPCSHUTDOWN'
            CALL IPCSHUTDOWN(VC_DESC, flags, options, resultcode)
            IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,
     >                                               this_call)
            active_vc = active_vc - 1
            exceptionmap = IBCLR(exceptionmap, (MAX_DESC - VC_DESC))
            current_rmap = IBCLR(current_rmap, (MAX_DESC - VC_DESC))
        END IF
        VC_DESC = VC_DESC + 1
      END DO
      END  ! end of subroutine PROCESS_XMAP
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  SUBROUTINE PROCESS_DATA                                                     C
C     PROCESS_DATA is called from MAIN when a bit of the readmap is set.       C
C     A bit in the read map is set when a VC socket becomes readable (i.e.     C
C     has data queued on it).                                                  C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE PROCESS_DATA (readmap,exceptionmap)

      IMPLICIT none

C   IPCSEND:
      INTEGER*4 vcdesc       ! IPCSEND sends data buffer 'senddata' of
      INTEGER*4 senddata(15) ! length 'dlength' over the VC connection.
      INTEGER*4 dlength      ! Here, we will use 60 byte data buffers.
      INTEGER*4 flags        ! Refer to the reference manual for discussion
      INTEGER*2 options(24)  ! of the flags and options parameters with
      INTEGER*4 resultcode   ! this call.
C
C   IPCRECV:
C     INTEGER*4 vcdesc       ! IPCRECV receives data from the VC connection
      INTEGER*4 recvdata(5)  ! and stores it in 'recvdata'.  Here, we use
C     INTEGER*4 dlength      ! 20 byte data buffers.  Refer to the reference
C     INTEGER*4 flags        ! for discussion of the flags and options
C     INTEGER*2 options(24)  ! parameters with this call.
C     INTEGER*4 resultcode   !
C
C   INITOPT:
C    INTEGER*2 options(24)   ! INITOPT initializes the options parameter so
     INTEGER*2 optnumargs    ! that 'optnumargs' arguments can be added.
     INTEGER*2 error         ! REFER TO 'OPT PARAMETERS' SECTION OF THE
                             ! NETIPC REFERENCE MANUAL FOR IMPORTANT INFO
                             ! REGARDING options BYTE ARRAY LENGTH!!!

     INTEGER*2 index        ! counter

     INTEGER*4 MAX_DESC     ! the maximum number of descriptors

     INTEGER*4 VC_DESC      ! the current VC descriptor

     INTEGER*4 readmap      ! read map bit array, indicates readability.
     INTEGER*4 exceptionmap ! exceptionmap bit array, indicates errors.

     CHARACTER*11 this_call ! name of IPC call that may have an error.

     CHARACTER*20 name_requested  ! client's request

     CHARACTER*20 name_in_file    ! name in data file

     INTEGER*4 eof_message(15)    ! name not found in data file message.

     COMMON MAX_DESC
```

```
      EQUIVALENCE (recvdata, name_requested)

      DATA eof_message/'does not appear in data file'/

C Reset the options array to 0 so IPCRECVCN and IPCSHUTDOWN don't complain.

      optnumargs = 0
      error = 0
      this_call = 'INITOPT    '
      CALL INITOPT(options, optnumargs, error)
      IF (error .NE. 0) CALL OPT_ERROR(error, this_call)

      VC_DESC = 0
      DO WHILE ((readmap .NE. 0) .AND. (VC_DESC .LT. MAX_DESC))
         IF (BTEST(readmap, (MAX_DESC - VC_DESC)) .AND.
     >       .NOT. BTEST(exceptionmap, (MAX_DESC - VC_DESC))) THEN
C A socket is readable, and there is no error (exception). Get the request.
            flags = 0                   ! Clear flag, then
            flags = IBSET(flags,31-20)  ! Set DATA WAIT flag [20] to be
                                        ! sure to get 20 bytes back.
            dlength = 20
            vcdesc = VC_DESC
            resultcode = 0
            this_call = 'IPCRECV    '
            CALL IPCRECV(vcdesc, recvdata, dlength, flags, options,
     >                   resultcode)
            IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,
     >                                               this_call)
            readmap = IBCLR(readmap, (MAX_DESC - VC_DESC))
C Now try to fill the request.
            REWIND (77)        ! Start at top of data file.
            DO WHILE (.TRUE.)  ! Check each line.
               read(77, '(A20, 15A4)', end=98)  ! at EOF go to 98
     >             (name_in_file, (senddata(index), index=1,15))
               IF (name_requested .EQ. name_in_file) THEN
                  flags = 0
                  dlength = 60
                  resultcode = 0
                  this_call = 'IPCSEND    '
                  CALL IPCSEND(vcdesc, senddata, dlength, flags,
     >                         options, resultcode)
                  IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,
     >                                                     this_call)
                  GOTO 88
               END IF
            END DO

C Did not find the name in file, send eof_message and try next request.
98          flags = 0
            dlength = 60
            resultcode = 0
            this_call = 'IPCSEND    '
            CALL IPCSEND(vcdesc, eof_message, dlength, flags, options,
     >                   resultcode)
```

```
            IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,
     >                                               this_call)
         END IF
88       VC_DESC = VC_DESC + 1
      END DO
      END ! end of subroutine PROCESS_DATA

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  SUBROUTINE OPT_ERROR                                                        C
C     OPT_ERROR is called when ADDOPT or INITOPT returns with an error         C
C     condition (error .NE. 0).                                                C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE OPT_ERROR (code, which_call)

      INTEGER*2 code            ! error code returned from call

      CHARACTER*11 which_call  ! call which produced error

      WRITE(1,'("SERVER: ERROR CODE RETURNED FROM ",A11)') which_call
      WRITE(1,'("THE ERROR CODE WAS ",I4)') code

      STOP
      END ! end of OPT_ERROR

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  SUBROUTINE RESULT_ERROR                                                     C
C     RESULT_ERROR is called when a NETIPC call returns with an error          C
C     condition (resultcode .NE. 0).                                           C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE RESULT_ERROR (code, which_call)

      INTEGER*4 code            ! error code returned from call

      CHARACTER*11 which_call  ! call which produced error

      WRITE(1,'("SERVER: ERROR CODE RETURNED FROM ",A11)') which_call
      WRITE(1,'("THE ERROR CODE WAS ",I4)') code

      STOP
      END ! end of RESULT_ERROR
```

NS-ARPA/1000 NetIPC Program Examples

The following programs illustrate the use of NetIPC calls in Pascal/1000 and FORTRAN 77. DEXEC calls are used to start up the remote NetIPC program. You can replace the DEXEC calls with RPM calls.

Pascal/1000 Example 1

```
$PASCAL '91790-16236 REV.5240 <900208.1421>'
$CDS ON$
$CODE_CONSTANTS OFF$
$RUN_STRING 80$

{}
{       NAME: IPC1
{     SOURCE: 91790-18236
{      RELOC: 91790-16236
{       PGMR: VH
{
{------------------------------------------------------------------------
{ Modifications:
{
{   Date    Rev. Pgmr  Description
{   ------  ---- ----  --------------------------------------------------
{   900208  5020  BEB  Changed the error code for shutdown from 65 to 64.
{   910514  5240  VH   Cleaned & Modified to take inputs from runstrings.
{
{------------------------------------------------------------------------
{}

PROGRAM ipc1;
{}
{ DESCRIPTION:
{   This program illustrates the use of IPCCreate, IPCName, IPCControl,
{      IPCRcvcn, IPCRecv, IPCSend, and IPCShutdown. Using IPCCreate, the
{      program creates a socket and names it <my_socket_name> as retrieved
{      from the runstring using IPCName. IPCCreate automatically creates
{      a socket in synchronous mode with a default timeout of 60 seconds.
{      The program then calls IPCRecvcn to wait to receive a connection
{      request from a peer program.
{
{   The peer program will be invoked with the name of the node where this
{      program is running and its socket's name. Once the connection is
{      established, the peer program sends a data message and a shutdown
{      message. This program uses IPCRecv to receive the messages and upon
{      receiving the shutdown message it calls IPCSend to send the shutdown
{      message back to its peer. IPCShutdown is then called to shutdown the
{      connection.
{
{ USAGE:
{   ipc1 <my_socket_name>
{
```

```
{        my_socket_name: the name of the socket created by ipc1.
{}

LABEL
  99;

{---------------------------------------------------------------}
{                    Constant Definitions                        }
{---------------------------------------------------------------}
CONST
  PROGNAME ='ipc1';
  ZERO = 0;
  LENGTH_OF_DATA = 20;
  TIMEOUT     = 59;
  SHUTDOWN    = 64;            { Connection Shutdown }
  CALL_SOCKET = 3;             { IPC call socket     }
  TCP = 4;                     { TCP Protocol number }
  MAX_SOCKET_NAME = 16;        { Maximum length of socket name }

{---------------------------------------------------------------}
{                       Type Definitions                         }
{---------------------------------------------------------------}
TYPE
  Int16 = -32768..32767;
  Byte  = 0..255;
  OptionType      = PACKED Array [1..8] of Byte;
  BufferType      = PACKED Array [1..LENGTH_OF_DATA] of Char;
  ReadDataType    = PACKED Array [1..3] of Int16;
  CallNameType    = PACKED Array [1..15] of Char;
  EnvStringType   = PACKED Array [1..80] of Char;

{---------------------------------------------------------------}
{                    Variable Declarations                       }
{---------------------------------------------------------------}
VAR
  screen : Text;                     {* Output pointer.      *}
  option : OptionType;               {* NetIPC Options Array *}
  socket_kind,                       {* CALL or VC socket.   *}
  protocol_kind,                     {* Transport protocol.  *}
  call_socket_descriptor,            {* CALL socket's descri.*}
  result,                            {* Result from IPC.     *}
  VC_socket_descriptor,              {* VC socket's descript.*}
  flags   : integer;                 {* NetIPC flags.        *}
  socket_name : EnvStringType;       {* Socket name.         *}
  socket_len  : integer;             {* Socket Length.       *}
  call_name   : CallNameType;        {* Used for error report*}

{---------------------------------------------------------------}
{                   External Declarations                        }
{---------------------------------------------------------------}
PROCEDURE InitOpt
  (VAR opt       : OptionType;
```

```
       num_args  : Int16;
   VAR error     : Int16);EXTERNAL;

PROCEDURE IPCCREATE
  (VAR socket    : integer;
   VAR protocol  : integer;
   VAR flags     : integer;
   VAR opt       : OptionType;
   VAR csd       : integer;
   VAR result    : integer);EXTERNAL;

PROCEDURE IPCNAME
  (VAR descriptor : integer;
   VAR name       : EnvStringType;
   VAR nlen       : integer;
   VAR result     : integer);EXTERNAL;

PROCEDURE IPCRECVCN
   (VAR csd    : integer;
    VAR vcsd   : integer;
    VAR flags  : integer;
    VAR opt    : OptionType;
    VAR result : integer);EXTERNAL;

PROCEDURE IPCRECV
  (VAR csd    :integer;
   VAR data  : BufferType;
   VAR dlen  : integer;
   VAR flags : integer;
   VAR opt   : OptionType;
   VAR result: integer);EXTERNAL;

PROCEDURE IPCSEND
  (VAR vcsd   : integer;
   VAR data   : BufferType;
   VAR dlen   : integer;
   VAR flags  : integer;
   VAR opt    : OptionType;
   VAR result : integer);EXTERNAL;

PROCEDURE IPCSHUTDOWN
  (VAR vcsd   : integer
   VAR flags  : integer
   VAR opt    : OptionType
   VAR result : integer);EXTERNAL;

FUNCTION GetRunString $ALIAS 'Pas.Parameters'$
  (    pos    : Int16;
   VAR evnstr : EnvStringType;
       len    : Int16
  ): Int16; EXTERNAL;

{----------------------------------------------------------------}
{                     InitOption                                 }
```

```
{-------------------------------------------------------------}
{                                                             }
{ Description:                                                }
{     This routine initializes the option parameter to contain }
{       zero entries. It is always a good idea to initialize   }
{       the option parameter before using it in any NetIPC call}
{-------------------------------------------------------------}
PROCEDURE InitOption
  (VAR opt_parameter : OptionType);
VAR
  result            : Int16;

BEGIN {* initialize option *}
  {}
  { initialize option parameter to contain zero entry.
  {}
  INITOPT(opt_parameter,0,result);
  IF (result <> ZERO) THEN
    BEGIN {* can't initialize *}
         writeln(screen,'An error occurred in your InitOpt call.');
         writeln(screen,'The error returned was ',result:3);
         GOTO 99;
    END;  {* can't initialize *}
END;  {* initialize option *}

{-------------------------------------------------------------}
{                         ReportError                         }
{-------------------------------------------------------------}
{                                                             }
{ Description:                                                }
{     This routine lets the user know if any error occur during}
{       any of the NetIPC calls. It will check for a connection}
{       aborted by peer error in which case ipc1 must shut down}
{       its VC connection.                                    }
{-------------------------------------------------------------}
PROCEDURE ReportError
  (VAR where  : CallNameType;
   VAR what   : integer;
   VAR vcsd   : integer);
VAR
  opt          : OptionType;
  call         : CallNameType;

BEGIN {* report error *}
  {}
  { If the error code received is a shutdown error, then the connection
  { has been aborted by ipc2.
  {}
  IF (what = SHUTDOWN) THEN
    BEGIN {* peer aborted connection *}
      InitOption(opt);
      {}
      { We use IPCShutdown to shut down the VC socket.
      {}
```

```
      flags := 0;
      IPCSHUTDOWN(vcsd, flags, opt, result);
      writeln(screen,PROGNAME,': connection aborted by peer.');
      IF (result <> ZERO) THEN
         BEGIN {* can't shut down *}
            writeln(screen,PROGNAME,': cannot shutdown connection.');
         END;  {* can't shut down *}
       GOTO 99;
    END    {* peer aborted connection *}
  ELSE IF (what = TIMEOUT) THEN
    BEGIN {* a timeout has occurred *}
       writeln(screen,PROGNAME,': a timeout has occurred in ',where);
       GOTO 99;
    END   {* a timeout has occurred *}
  ELSE
    BEGIN {* other error *}
       writeln(screen,PROGNAME,': An error occurred in your ',where,' call.');
       writeln(screen,PROGNAME,': The error code returned was ',what);
      GOTO 99;
    END;  {* other error *}
END; {* report error *}


{---------------------------------------------------------------}
{                  ReceiveMsg                                   }
{---------------------------------------------------------------}
{                                                               }
{ Description:                                                  }
{     This routine uses IPCRecv to receive msgs from its peer   }
{       ipc2. The message received will be displayed on the    }
{       screen, except the shutdown message which indicates     }
{       it is time to shut down the connection.                 }
{                                                               }
{     When the shutdown message is received, IPCSend is used    }
{       to send a similar shutdown message back to ipc2 to      }
{       synchronize the shutdown of the connection. IPCShutdown}
{       is then called to shut down the connection which will   }
{       in effect release all resources used for connection,    }
{       including CALL and VC socket descriptors to the system.}
{---------------------------------------------------------------}
PROCEDURE ReceiveMsg
   (VAR vcsd : integer;
    VAR opt  : OptionType);
VAR
  shut_down_message     : BufferType;
  receive_buffer        : BufferType;
  receive_buffer_length : integer;
  flags                 : integer;
  call                  : CallNameType;

BEGIN {* receive message *}
{}
{ The shut_down_message parameter is initialized to contain a shut
{ down "message" that is identical to that sent by ipc2.
```

```
{}
shut_down_message := 'I want to shut down.';
receive_buffer_length := LENGTH_OF_DATA;

{}
{ Use IPCRecv to receive messages from peer.
{}
flags := 0;
IPCRECV(vcsd, receive_buffer, receive_buffer_length, flags, opt, result);
IF (result <> ZERO) THEN
  BEGIN {* can't receive *}
     call := 'IPCRECV   ';
     ReportError(call, result, vcsd);
  END; {* can't receive *}
writeln(screen,PROGNAME,': waiting to receive data from peer (IPCRecv).');

{}
{ If the shutdown message is received, our peer want to shut down the
{ connection. In this case, we will send a shutdown message back and
{ shut down the connection from our side. Use IPCSend to send the msg.
{}
IF (receive_buffer = shut_down_message) THEN
   BEGIN {* time to shut down connection *}
      {}
      { IPCSend will send msg. to the peer.
      {}
      writeln(screen,PROGNAME,': received shutdown msg. from peer.');

      flags := 0;
      IPCSEND(vcsd, shut_down_message, receive_buffer_length, flags,
                                                         opt,result);
      IF (result <> ZERO) THEN
         BEGIN {* can't send *}
           call := 'IPCSEND   ';
           ReportError(call, result,vcsd);
         END;  {* can't send *}
      writeln(screen,PROGNAME,': sending shutdown msg. to peer (IPCSend).');

      {}
      { Now we shut down the connection. Resources will be returned
      { to the system.
      {}
      flags := 0;
      IPCSHUTDOWN(vcsd, flags, opt, result);
      IF (result <> ZERO) THEN
          BEGIN {* can't shut down! *}
             call := 'IPCSHUTDOWN ';
             ReportError(call, result, vcsd);
          END; {* can't shut down! *}
       writeln(screen,PROGNAME,': shutting down connection (IPCShutdown).');
      {}
    END  {* time to shut down connection *}
ELSE
   BEGIN {* not a shutdown message *}
```

```
      {}
      { Display data on screen and call receive_msg again. We will
      { get out when we receive a shutdown message or we get an error
      { from IPCRecv.
      {}
       writeln(screen,PROGNAME,':received data from peer.');
       writeln(screen,receive_buffer);
       ReceiveMsg(vcsd, opt);
    END; {* not a shutdown message *}
END; {* receive message *}

{----------------------------------------------------------------}
{                        Main Program                            }
{----------------------------------------------------------------}
BEGIN {* main program *}
{}
{ Get the name of my socket from the runstring.
{}
Rewrite(screen,'1');
socket_len := GetRunString(1,socket_name,MAX_SOCKET_NAME);
IF (socket_len <= 0) OR (socket_len > MAX_SOCKET_NAME) THEN
   BEGIN {* terminate with usage *}
      writeln(screen,'Usage: ',PROGNAME,'  <my_socket_name>');
      GOTO 99;
   END;  {* terminate with usage *}

{}
{ The InitOption procedure uses the NetIPC call InitOpt to initialize the    }
{ option parameter used by the IPCCreate, IPCRecvcn, IPCRecv and IPCShutdown }
{ calls.
{}
InitOption(option);

{}
{ Set the socket kind to call socket for IPCCreate call and specify
{ TCP as the underlying transport protocol.
{}
socket_kind   := CALL_SOCKET;
protocol_kind := TCP;

{}
{ Always initialize flag parameter before using it.
{}
flags := 0;

{}
{ A call socket is created by calling IPCCreate. The value returned
{ in the call_socket_descriptor parameter will be used in the following
{ IPCName call.
{}
IPCCREATE(socket_kind, protocol_kind, flags, option,
                                 call_socket_descriptor, result);
IF (result <> ZERO) THEN
   BEGIN  {* can't create call socket *}
       call_name := 'IPCCREATE ';
```

```
       ReportError(call_name, result, call_socket_descriptor);
   END;   {* can't create call socket *}

{}
{ Be nice to user!
{}
writeln(screen,PROGNAME,': a CALL socket has been created (IPCCreate).');

{}
{ IPCNAME is called to assign a name to the call socket created by
{ IPCCREATE. This name must be given to the peer program ipc2 during
{ its invocation.
{}
IPCNAME(call_socket_descriptor, socket_name, socket_len, result);
IF (result <> ZERO) THEN
  BEGIN {* can't name socket *}
      call_name := 'IPCNAME   ';
      ReportError(call_name, result, call_socket_descriptor);
  END;

{}
{ Be nice.
{}
writeln(screen,PROGNAME,': the CALL socket has been named (IPCName).');

{}
{ Since the call socket is in synchronous mode (by default), the following
{ IPCRecvcn call will be blocked if the peer program ipc2 is not running at
{ this time. ipc2 is supposed to look up the socket name of ipc1 and then
{ issue an IPCConnect request. If that is the case then IPCRecvcn will
{ receive this connect request and return a VC socket descriptor which
{ is needed to send and receive data. At this point, you may optionally
{ release the CALL socket descriptor by calling the IPCShutdown if you
{ wish to return resources to the system. Doing so will not affect the
{ newly-created VC socket.
{}
write(screen,PROGNAME,': waiting for connection request from peer');
writeln(screen,' (IPCRecvcn).');
flags := 0;
IPCRECVCN(call_socket_descriptor, vc_socket_descriptor, flags,
                                                option,result);
IF (result <> ZERO) THEN
   BEGIN {* can't receive connection *}
       call_name := 'IPCRecvcn ';
       ReportError(call_name, result, call_socket_descriptor);
   END;  {* can't receive connection *}

{}
{ Be nice
{}
writeln(screen,PROGNAME,': A connection has been received from peer.');

{}
{ Connection is now established between ipc1 and its peer program ipc2.
{ The routine Receive_Msg below will wait to receive a msg from ipc2.
```

```
{ When it receives a shutdown message from ipc2, ipc1 will send a
{ similar shutdown message and performs a IPCShutdown on its VC connection.
{}
ReceiveMsg(vc_socket_descriptor, option);

99:
END. {* main program *}
```

Pascal/1000 Example 2

```
$PASCAL '91790-16241 REV.5240 <860303.1238>'
$CDS ON$
$CODE_CONSTANTS OFF$
$RUN_STRING 80$

{}
{       NAME: IPC2
{     SOURCE: 91790-18241
{      RELOC: 91790-16241
{       PGMR: VH
{}
{------------------------------------------------------------------------
{ Modifications:
{
{   Date    Rev. Pgmr  Description
{   ------  ---- ----  -----------------------------------------------
{   910514  5240  VH   Cleaned & Modified to take inputs from runstrings.
{
{------------------------------------------------------------------------
{}
PROGRAM ipc2;
{}
{ DESCRIPTION:
{   This program illustrates the use of IPCCreate, IPCLookup, IPCConnect,
{      IPCRcv, IPCSend, and IPCShutdown. This program is a peer of ipc1.
{      ipc2 creates a call socket using IPCCreate then uses the name of
{      the node where its peer ipc1 is running and ipc1's socket name to
{      lookup ipc1's call socket by calling IPCLookup. IPCConnect is then
{      called to make a VC connection to ipc1. IPCSend and IPCRecv are
{      then used to exchange data msg. between itself and ipc1. IPCShutdown
{      is used to shut down the connection afterward.
{
{ USAGE:
{   ipc2 <peer_node_name> <peer_socket_name>
{
{   peer_node_name: the name of the node where ipc1 is running.
{
{   peer_socket_name: the name of ipc1's call socket.
{
{}
LABEL
  99;

{----------------------------------------------------------------}
{                     Constant Definitions                        }
{----------------------------------------------------------------}
CONST
  PROGNAME ='ipc2';
  ZERO = 0;
  LENGTH_OF_DATA = 20;
  MAX_RETRY = 500;              { Used during lookup  }
  NAME_NOT_FOUND = 37;          { Error from IPCLookup}
```

```
  CALL_SOCKET = 3;              { IPC call socket     }
  TCP = 4;                      { TCP Protocol number }
  MAX_SOCKET_NAME = 16;         { Maximum length of socket name }
  MAX_NODE_NAME = 50;           { Maximum length of node name   }
                                { nodeName. Domain.Organization }
                                { 16 chars.16 chars.16 chars    }

{---------------------------------------------------------------}
{ Type Definitions                                              }
{---------------------------------------------------------------}
TYPE
  int16 = -32768..32767;
  Byte = 0..255;
  OptionType      = PACKED Array [1..8] of Byte;
  BufferType      = PACKED Array [1..LENGTH_OF_DATA] of Char;
  EnvStringType   = PACKED Array [1..80] of Char;
  name_array_type = packed array [1..7] of char;
  CallNameType    = PACKED Array [1..15] of Char;

{---------------------------------------------------------------}
{                      Variable Declarations                    }
{---------------------------------------------------------------}
VAR
  screen : Text;                         {* Output pointer.      *}
  option : OptionType;                   {* NetIPC Options Array *}
  socket_kind,                           {* CALL or VC socket.   *}
  protocol_kind,                         {* Transport protocol.  *}
  call_socket_descriptor,                {* CALL socket's descri.*}
  vc_socket_descriptor,                  {* VC socket's descript.*}
  path_report_descriptor,                {* Path report's descri.*}
  protocol_returned,                     {* Protocol from lookup.*}
  data_length,                           {* Length of data sent. *}
  counter,                               {* Used for termination.*}
  flags,                                 {* NetIPC's flags.      *}
  result      : integer;                 {* Result from IPC.     *}
  socket_name : EnvStringType;           {* Peer's socket name.  *}
  socket_len  : integer;                 {* Peer's socket len.   *}
  peer_node   : EnvStringType;           {* Peer's node name.    *}
  peer_node_len: integer;                {* Peer's node len.     *}
  call_name   : CallNameType;            {* Used in reporting err*}
  data_buffer : BufferType;              {* Data buffer.         *}
  bad_error,lookup_OK : boolean;

{---------------------------------------------------------------}
{                     External Declarations                     }
{---------------------------------------------------------------}
PROCEDURE INITOPT
  (VAR opt       : OptionType;
       num_args  : int16;
   VAR error     : int16);EXTERNAL;

PROCEDURE IPCCREATE
  (VAR socket   : integer;
   VAR protocol : integer;
```

```
   VAR flags    : integer;
   VAR opt      : OptionType;
   VAR csd      : integer;
   VAR result   : integer);EXTERNAL;

PROCEDURE IPCLOOKUP
  (VAR name     : EnvStringType;
   VAR name_len : integer;
   VAR location : EnvStringType;
   VAR loc_len  : integer;
   VAR flags    : integer;
   VAR prd      : integer;
   VAR protocol : integer;
   VAR socket   : integer;
   VAR result   : integer);EXTERNAL;

PROCEDURE IPCCONNECT
  (VAR csd      : integer;
   VAR prd      : integer;
   VAR flags    : integer;
   VAR opt      : OptionType;
   VAR vcsd     : integer;
   VAR result   : integer);EXTERNAL;

PROCEDURE IPCRECV
  (VAR vcsd   : integer;
   VAR data   : BufferType;
   VAR dlen   : integer;
   VAR flags  : integer;
   VAR opt    : OptionType;
   VAR result : integer);EXTERNAL;

PROCEDURE IPCSEND
 (VAR vcsd    : integer;
  VAR data    : BufferType;
  VAR dlen    : integer;
  VAR flags   : integer;
  VAR opt     : OptionType;
  VAR result  : integer);EXTERNAL;

PROCEDURE IPCSHUTDOWN
  (VAR vcsd     : integer;
   VAR flags    : integer;
   VAR opt      : OptionType;
   VAR result   : integer);EXTERNAL;

FUNCTION GetRunString $ALIAS 'Pas.Parameters'$
  (    pos    : Int16;
   VAR evnstr : EnvStringType;
       len    : Int16
  ): Int16; EXTERNAL;
```

```
{----------------------------------------------------------------}
{                         ReportError                            }
{----------------------------------------------------------------}
{                                                                }
{ Description:                                                   }
{     This routine lets the user know if any error occurred      }
{       during any of the NetIPC calls.                          }
{----------------------------------------------------------------}
PROCEDURE ReportError
  (VAR where : CallNametype;
   VAR what  : integer);

BEGIN {* report error *}
  {}
  writeln(screen,PROGNAME,': an error occurred in your ',where,' call.');
  writeln(screen,PROGNAME,': the error code returned was ',what);
  writeln(screen,PROGNAME,': the count was ',counter);
  GOTO 99;
  {}
END;  {* report error *}

{----------------------------------------------------------------}
{                    InitOption                                  }
{----------------------------------------------------------------}
{                                                                }
{ Description:                                                   }
{     This routine initializes the option parameter to contain }
{       zero entries. It is always a good idea to initialize   }
{       the option parameter before using it in any NetIPC call}
{----------------------------------------------------------------}
PROCEDURE InitOption
  (VAR opt_parameter : OptionType);
VAR
  result            : int16;

BEGIN {* initialize option *}
  {}
  { initialize option parameter to contain zero entry.
  {}
  INITOPT(opt_parameter,0,result);
  IF (result <> ZERO) THEN
    BEGIN {* can't initialize *}
         writeln(screen,PROGNAME,': an error occurred in your InitOpt call.');
         writeln(screen,PROGNAME,': the error returned was ',result:3);
         GOTO 99;
    END;  {* can't initialize *}
END; {initialize_option}

{----------------------------------------------------------------}
{                         SendMessage                            }
{----------------------------------------------------------------}
PROCEDURE SendMessage
  (VAR vcsd  : integer;
   VAR opt   : OptionType);
```

```
VAR
  shut_down_message    : BufferType;
  send_buffer          : BufferType;
  receive_buffer       : BufferType;
  send_buffer_length   : integer;
  flags                : integer;
  error                : integer;
  call                 : CallNameType;

BEGIN {* send message *}
   {}
   { Send message to our peer using IPCSend.
   {}
   send_buffer := 'Here is the message.';
   send_buffer_length := length_of_data;
   flags := 0;
   IPCSEND(vcsd,send_buffer,send_buffer_length,flags,opt,error);
   IF (error <> ZERO) THEN
      BEGIN {* can't send *}
        call := 'IPCSEND   ';
        ReportError(call,error);
      END;  {* can't send *}
   writeln(screen,PROGNAME,': sent data message to peer (IPCSend).');

   {}
   { Now we send a shutdown message to peer using IPCSend.
   {}
   shut_down_message := 'I want to shut down.';
   IPCSEND(vcsd,shut_down_message,send_buffer_length,flags,opt,error);
   IF (error <> ZERO) THEN
      BEGIN {* can't send *}
        call := 'SHUTDOWN ';
        ReportError(call,error);
      END;  {* can't send *}
    writeln(screen,PROGNAME,': sent shutdown message to peer (IPCSend).');

   {}
   { Wait for our peer to send back a shutdown message.
   { Use IPCRecv to receive this message.
   {}
   flags := 0;
   IPCRECV(vcsd,receive_buffer,send_buffer_length,flags,opt,error);
   IF (error <> ZERO) THEN
      BEGIN {* can't receive *}
         call := 'IPCRECV   ';
         ReportError(call,error);
      END;  {* can't receive *}

    {}
    {If the receive_buffer contains the shutdown "message," we will
    { call IPCShutdown to shut down the connection.
    {}
    IF (receive_buffer = shut_down_message) THEN
       BEGIN {* shut down connection *}
```

```
          write(screen,PROGNAME,': receiving shutdown message from peer');
          writeln(screen,' (IPCRecv).');
          flags := 0;
          IPCSHUTDOWN(vcsd,flags,opt,error);
          IF (error <> ZERO) THEN
             BEGIN {* can't shut down *}
                call := 'SHUTDOWN  ';
                ReportError(call,error);
             END;  {* can't shut down *}
         writeln(screen,PROGNAME,': shutting down connection (IPCShutdown).');
      END  {* shut down connection *}
    ELSE
      {}
      { we should not get here because our peer will only send
      { us the shutdown message. If we get here, then the receive
      { buffer is corrupted.
      {}
      BEGIN {* shouldn't be here *}
         write(screen,PROGNAME,': expecting shutdown message from peer.');
         writeln(screen,PROGNAME,': but received something else.');
         writeln(screen,PROGNAME,': will shut down connection.');
         flags := 0;
         IPCSHUTDOWN(vcsd,flags,opt,error);
         IF (error <> ZERO) THEN
            BEGIN {* can't shut down *}
               call := 'SHUTDOWN  ';
               ReportError(call,error);
            END; {* can't shut down *}
          writeln(screen,PROGNAME,': shutting down connection.');
        END;  {* shouldn't be here *}

END; {send_routine}

{---------------------------------------------------------------}
{                      Main Program                             }
{---------------------------------------------------------------}
BEGIN {main program}
{}
Rewrite(screen,'1');
{}
{ Get the names of my peer's socket and node from the runstring.
{}
socket_len := GetRunString(1,socket_name,MAX_SOCKET_NAME);
IF (socket_len <= 0) OR (socket_len > MAX_SOCKET_NAME) THEN
   BEGIN {* terminate with usage *}
      writeln(screen,'Usage: ',PROGNAME,' <peer_socket_name> <peer_node_name>');
      GOTO 99;
   END;  {* terminate with usage *}

peer_node_len := GetRunString(2,peer_node,MAX_NODE_NAME);
IF (peer_node_len <= 0) OR (peer_node_len > MAX_SOCKET_NAME) THEN
   BEGIN {* terminate with usage *}
      writeln(screen,'Usage: ',PROGNAME,' <peer_socket_name> <peer_node_name>');
```

```
      GOTO 99;
   END;  {* terminate with usage *}

{}
{ The InitOption procedure uses the NetIPC call InitOpt to initialize the    }
{ option parameter used by the IPCCreate, IPCRecvcn, IPCRecv and IPCShutdown }
{ calls.
{}
InitOption(option);

{}
{ Set the socket kind to call socket for IPCCreate call and specify
{ TCP as the underlying transport protocol.
{}
socket_kind := CALL_SOCKET;
protocol_kind := TCP;

{}
{ The flags parameter is not used in this program so flags is
{ made type integer and assigned the value zero to ensure that all
{ the bits are clear.
{}
flags := ZERO;

{}
{ A CALL socket is created by calling IPCCreate. The value returned in
{ the call_socket_descriptor parameter will be used in the following
{ IPCConnect call to ipc1.
{}
IPCCREATE(socket_kind,protocol_kind,flags,option,call_socket_descriptor,result);
IF (result <> ZERO) THEN
  BEGIN {* can't create *}
     call_name := 'IPCCREATE ';
     ReportError(call_name,result);
  END; {* can't create *}
writeln(screen,PROGNAME,': a CALL socket has been created (IPCCreate).');

{}
{ With the peer's node name and socket name, we call IPCLookup to
{ to gain access to the peer's call socket. If successful, a path
{ descriptor will be returned which we need for IPCConnect.
{ IPCLOOKUP searches the socket registry at the given node for our
{ peer's name. This call returns a path_report_descriptor that is used in
{ the following IPCCONNECT call to request a connection with the peer.
{
{ Because it is possible for IPCLookupP to search for the socket name before
{ our peer places it in its node's socket registry, we will try to look up the
{ name several times before aborting.}
{}
lookup_OK := FALSE;
bad_error := FALSE;
counter   := ZERO;
```

```
writeln(screen,PROGNAME,': looking up peer socket...(IPCLookup).');
REPEAT {* we give up if we cannot find our peer's socket after MAX_RETRY *}
  {}
  IPCLOOKUP(socket_name,socket_len,peer_node,peer_node_len,flags,
            path_report_descriptor,protocol_returned,socket_kind,result);

  IF (result = ZERO) THEN
     BEGIN {* found it. *}
        lookup_OK := TRUE;
     END   {* found it. *}
  ELSE
    BEGIN {* can't find it yet *}
       IF (result = NAME_NOT_FOUND) THEN
          BEGIN {* retry if possible *}
             counter := counter + 1;
          END   {* retry if possible *}
       ELSE
         BEGIN {* other error - need to abort *}
             bad_error := TRUE;
          END;  {* other error - need to abort *}
    END;  {* can't find it yet *}

UNTIL (lookup_OK) OR (counter = MAX_RETRY) OR (bad_error);

{}
{ At this point, we either lookup successfully or else we abort.
{}
IF (bad_error) OR (counter = MAX_RETRY) THEN
  BEGIN {* abort *}
     call_name := 'IPCLOOKUP ';
     ReportError(call_name,result);
  END;  {* abort *}

{}
{ The call_socket_descriptor returned by IPCCREATE and the
{ path_report_descriptor returned by IPCLOOKUP are used in
{ IPCCONNECT to request a connection with ipc1. The VC_socket_descriptor
{ returned by IPCCONNECT is used in subsequent calls to reference the
{ connection. Once this call completes successfully, you may optionally
{ release the call socket descriptor by calling IPCSHUTDOWN in order to return
{ resources to the system. Doing so will not affect the newly-created
{ VC socket descriptor.
{}
flags := 0;
IPCCONNECT(call_socket_descriptor,path_report_descriptor,flags,option,
                                        vc_socket_descriptor,result);
IF (result <> ZERO) THEN
  BEGIN {* can't connect *}
     call_name := 'IPCCONNECT';
     ReportError(call_name,result);
  END; {* can't connect *}
writeln(screen,PROGNAME,': trying to connect to peer VC socket (IPCConnect).');
```

```
{}
{ IPCRECV is called here to determine if the connection has been established.
{}
flags := 0;
data_length := LENGTH_OF_DATA
IPCRECV(vc_socket_descriptor,data_buffer,data_length,flags,option,result);
IF (result <> ZERO) THEN
   BEGIN {* something's wrong. *}
      call_name := 'IPCRECV  ';
      ReportError(call_name,result);
   END;   {* something's wrong. *}

writeln(screen,PROGNAME,': connection to peer is established (IPCRecv).');
{}
{ The send routine below will send a message to the peer and then a shutdown
{ message using IPCSend. IPCRecv is then called to receive the shutdown msg.
{ back from the peer. Once the shutdown message is received, IPCShutdown is
{ called to close the connection.
{}
SendMessage(vc_socket_descriptor,option);

99:

END. {* main program *}
```

FORTRAN 77 Example 1

```
FTN77,L
$CDS ON

      PROGRAM IPC3(4,99),91790-16237 REV.5240 <900208.1422>

C
C       NAME: IPC3
C     SOURCE: 91790-18237
C      RELOC: 91790-16237
C       PGMR: VH
C
C---------------------------------------------------------------------
C Modifications:
C
C   Date    Rev. Pgmr  Description
C   ------  ---- ----  -----------------------------------------------
C   900208  5020  BEB  Changed the error code for shutdown from 65 to 64.
C
C   910521  5240  VH   Modified to take inputs from runstring.
C---------------------------------------------------------------------
C
C   This program illustrates the use of IPCCreate, IPCName, IPCControl,
C      IPCRcvcn, IPCRecv, IPCSend, and IPCShutdown. Using IPCCreate, the
C      program creates a socket and names it <my_socket_name> as retrieved
C      from the runstring using IPCName. IPCCreate automatically creates
C      a socket in synchronous mode with a default timeout of 60 seconds.
C      The program then calls IPCRecvcn to wait to receive a connection
C      request from a peer program.
C
C   The peer program will be invoked with the name of the node where this
C      program is running and its socket's name. Once the connection is
C      established, the peer program sends a data message and a shutdown
C      message. This program uses IPCRecv to receive the messages and upon
C      receiving the shutdown message it calls IPCSend to send the shutdown
C      message back to its peer. IPCShutdown is then called to shut down the
C      connection.
C
C USAGE:
C   ipc3 <my_socket_name>
C
C        my_socket_name: the name of the socket created by ipc1.
C
C ---------------------------------------------------------------------
      IMPLICIT NONE

      INTEGER*2  RHPAR                ! function to get runstring
      INTEGER*2  socket_name(8)       ! my socket name (max is 16).
      INTEGER*2  index
      INTEGER*2  opt_num_arguments
      INTEGER*2  receive_buffer(10)      ! data buffer.
      INTEGER*2  option(2)               ! NetIPC option array.
      INTEGER*2  shut_down_message(10)   ! shutdown connection msg.
```

```
      INTEGER*2  length                  ! len of socket name
      INTEGER*2  here                    ! where the error occurs.
      INTEGER*2  init_result

      INTEGER*4  socket_kind             ! kind of socket (CALL or VC)
      INTEGER*4  protocol_kind           ! always TCP.
      INTEGER*4  call_socket_descriptor ! CALL socket descriptor
      INTEGER*4  vc_socket_descriptor   ! VC socket descriptor
      INTEGER*4  result                  ! result from NetIPC calls.
      INTEGER*4  socket_length           ! len of socket name
      INTEGER*4  msg_buffer_length       ! len of msg. buffer
      INTEGER*4  flags                   ! NetIPC flags


      DATA        shut_down_message/20HI want to shut down./

C     ----------------------------------------
C     Get the socket name from the runstring.
C     ----------------------------------------
      socket_length = RHPAR(1,socket_name,16)
      IF (socket_length .EQ. 0) THEN
         write(1,'("Usage: ipc3 <my_socket_name>")')
         goto 99
      ENDIF

C     --------------------------------------------------------------
C     The INITOPT call initializes the option parameter used by the
C     IPCCREATE, IPCRECVCN, IPCRECV and IPCSHUTDOWN calls. By setting
C     the opt_num_arguments parameter to zero, the option parameter is
C     initialized to contain zero entries. (An example of adding entries
C     to an option parameter is included in the discussion of ADDOPT in
C     this section.
C     --------------------------------------------------------------

      opt_num_arguments = 0
      CALL INITOPT(option,opt_num_arguments,init_result)
      here = 1
      IF (init_result .NE. 0) GO TO 99

C     -------------------------------------------------------------

C     socket_kind is set to 3 and protocol_kind is set to 4 to
C     specify a call socket and the TCP protocol for the following
C     IPCCREATE call.
C     -------------------------------------------------------------
      socket_kind = 3
      protocol_kind = 4

C     --------------------------------------------------------------
C     The flags parameter is not used in this program, so flags
C     is made a double integer and assigned the value zero to ensure
C     that all the bits are clear.
C     --------------------------------------------------------------
      flags = 0
```

```
C     -------------------------------------------------------------------
C     A call socket is created by calling IPCCREATE. The value returned
C     in the call_socket_descriptor parameter will be used in the follow
C     IPCNAME call.
C     ------------------------------------------------------------------
      write(1,'("ipc3: creating CALL socket (IPCCreate).")')
      CALL IPCCREATE(socket_kind,protocol_kind,flags,option,
     +                        call_socket_descriptor,result)
      here = 2 IF (result .NE. 0) GO TO 99

C     ------------------------------------------------------------
C     IPCNAME is called to assign a name to the newly-created call
C     socket. This name should be known to the peer ipc4.
C     -------------------------------------------------------------
      write(1,'("ipc3: naming CALL socket (IPCName).")')
      CALL IPCNAME(call_socket_descriptor,socket_name,socket_length,
     +                                                     result)
      here = 3
      IF (result .NE. 0) GO TO 99

C     ----------------------------------------------------------------
C     Since the call socket is in synchronous mode (by default), the
C     following IPCRecvcn call will be blocked if the peer program ipc4
C     is not running at this time. ipc4 is supposed to look up the
C     socket name of ipc3 and then issue an IPCConnect request. If
C     that is the case then IPCRecvcn will receive this connect request
C     and return a VC socket descriptor which is needed to send and
C     receive data. At this point, you may optionally release the CALL
C     socket descriptor by calling the IPCShutdown if you wish to return
C     resources to the system. Doing so will not affect the newly-created
C     VC socket.
C     ------------------------------------------------------------------
      write(1,'("ipc3: waiting connection from peer (IPCRecvcn).")')
      flags = 0
      CALL IPCRECVCN(call_socket_descriptor,vc_socket_descriptor,
     +                                      flags,option,result)
      here = 5 IF (result .NE. 0) GO TO 99

C     ------------------------------------------------------------------

C     Connection is now established between ipc3 and its peer program
C     ipc4. Now we wait to receive a msg from ipc4. When a shutdown
C     message is received, ipc3 will send a similar shutdown message
C     and performs a IPCShutdown on its VC connection.
C     ------------------------------------------------------------------
10    flags = 0
      msg_buffer_length = 20
      CALL IPCRECV(vc_socket_descriptor,receive_buffer,
     +             msg_buffer_length,flags,option,result)
      here = 6
      IF (result .NE. 0) GO TO 99
```

```
C     ------------------------------------------------------------------
C     The receive buffer is compared to the shutdown “message.”
C     If the shutdown “message” is received, ipc1 sends a shut
C     down “message” back to ipc4 so that ipc4 will know that its
C     data has been received. We then do the IPCShutdown.
C     -----------------------------------------------------------------
      IF (receive_buffer .EQ. shut_down_message) THEN
         flags = 0
         write(1,’(“ipc3: received shutdown message (IPCRecv).”)’)
         write(1,’(“ipc3: sending shutdown message (IPCSend).”)’)
         CALL IPCSEND(vc_socket_descriptor,shut_down_message,
     +               msg_buffer_length,flags,option,result)
         here = 7
         IF (result .NE. 0) GO TO 99

         write(1,’(“ipc3: shutting down connection (IPCShutdown).”)’)
         CALL IPCSHUTDOWN(vc_socket_descriptor,flags,option,result)
         IF (result .NE. 0) THEN
            write(1,’(“ipc3: cannot shutdown connection.”)’)
            write(1,’(“ipc3: error code returned: ”,14)’) result
         ENDIF
         GO TO 99
      ELSE
C        -------------------------------------------------------
C        If the shutdown “message” was not received, ipc3 will
C        simply receive the data and print it. It then returns to
C        the previous IPCRECV call receive subsequent data until
C        the shutdown “message” is received.
C         -------------------------------------------------------
         write(1,’(“ipc3: received data message (IPCRecv).”)’)
         WRITE(1,’(10A2)’)(receive_buffer(index),index = 1,10)
         GO TO 10
      ENDIF

99    IF (result .NE. 0) THEN
         WRITE(1,’(“ipc3: result error code: ”,I4)’) result
         WRITE(1,’(“ipc3: at location: ”,I4)’) here
      ENDIF
100   STOP

      END
```

FORTRAN 77 Example 2

```
FTN77,L
$CDS ON

      PROGRAM IPC4(4,99),91790-16238 REV.5240 <860303.1239>

C
C       NAME: IPC4
C     SOURCE: 91790-18238
C      RELOC: 91790-16238
C       PGMR: VH
C
C----------------------------------------------------------------------
C Modifications:
C
C   Date    Rev. Pgmr  Description
C   ------  ---- ----  ---------------------------------------------------
C   910521  5240  VH   Modified to take inputs from runstring.
C----------------------------------------------------------------------
C
C   This program illustrates the use of IPCCreate, IPCLookup, IPCConnect,
C      IPCRcv, IPCSend, and IPCShutdown. This program is a peer of ipc3.
C      ipc4 creates a call socket using IPCCreate then uses the name of
C      the node where its peer ipc3 is running and ipc3's socket name to
C      lookup ipc3's call socket by calling IPCLookup. IPCConnect is then
C      called to make a VC connection to ipc3. IPCSend and IPCRecv are
C      then used to exchange data msg. between itself and ipc3.
C      IPCShutdown is used to shut down the connection afterward.
C
C USAGE:
C   ipc4 <peer_node_name> <peer_socket_name>
C
C   peer_node_name: the name of the node where ipc3 is running.
C
C   peer_socket_name: the name of ipc3's call socket.
C
C ------------------------------------------------------------------------

      IMPLICIT NONE

      INTEGER*2 socket_name(8)           ! peer socket name (max is 16).
      INTEGER*2 node_name(25)            ! peer node name (max is 50).
      INTEGER*2 RHPAR                    ! function to get runstring
      INTEGER*2 send_buffer(10)          ! netipc send buffer.
      INTEGER*2 option(2)                ! netipc option
      INTEGER*2 receive_buffer(10)       ! netipc receive buffer.
      INTEGER*2 data_buffer(25)          ! data buffer
      INTEGER*2 shut_down_message(10)    ! shutdown message.
      INTEGER*2 opt_num_arguments, counter
      INTEGER*2 init_result,here,index
```

```
      INTEGER*4 socket_kind              ! CALL or VC socket type
      INTEGER*4 protocol_kind            ! TCP protocol
      INTEGER*4 call_socket_descriptor   ! CALL socket descriptor
      INTEGER*4 vc_socket_descriptor     ! VC socket descriptor
      INTEGER*4 result                   ! netipc result
      INTEGER*4 socket_length            ! socket name's length
      INTEGER*4 node_length              ! node name's length
      INTEGER*4 msg_buffer_length        ! msg buffer's length
      INTEGER*4 data_length              ! data length
      INTEGER*4 path_report_descriptor, protocol_returned,flags

      DATA   send_buffer/20HHere is the message./
      DATA   shut_down_message/20HI want to shut down./

C     ----------------------------------------
C     Get the socket name from the runstring.
C     ----------------------------------------
      socket_length = RHPAR(1,socket_name,16)
      IF (socket_length .EQ. 0) THEN
         write(1,'("Usage: ipc4 <peer_socket_name> <peer_node_name>")')
         goto 100
      ENDIF

C     ----------------------------------------
C     Get the node name from the runstring.
C     ----------------------------------------
      node_length = RHPAR(2,node_name,50)
      IF (node_length .EQ. 0) THEN
         write(1,'("Usage: ipc4 <peer_socket_name> <peer_node_name>")')
         goto 100
      ENDIF

C     ----------------------------------------------------------------
C     INITOPT is called to initialize the option parameter used in the
C     IPCCREATE, IPCLOOKUP, IPCCONNECT, IPCRECV, IPCSEND and
C     IPCSHUTDOWN calls. By setting opt_num_arguments to zero, the
C     option parameter is initialized to contain zero entries.
C     (An example of adding options to an option parameter is included
C     in the discussion of ADDOPT in this section.
C     ----------------------------------------------------------------
      opt_num_arguments = 0
      CALL INITOPT(option,opt_num_arguments,init_result)

      here = 1
      IF (init_result .NE. 0) GO TO 99

C     ----------------------------------------------------------------
C     socket_kind is set to 3 and protocol_kind is set to 4 to specify
C     a call socket and the TCP protocol for the following IPCCREATE
C     call.
C     -----------------------------------------------------------------
      socket_kind = 3
      protocol_kind = 4
```

```
C     ----------------------------------------------------------------
C     The flags parameter is not used in this program so flags is made
C     a double integer and assigned the value zero to ensure that all
C     the bits are clear.
C     -----------------------------------------------------------------
      flags = 0

C     ------------------------------------------------------------------
C     A call socket is created by calling IPCCREATE. The value returned
C     in the call_socket_descriptor parameter will be used in the following
C     IPCCONNECT call.
C     -----------------------------------------------------------------
      write(1,'("ipc4: creating CALL socket (IPCCreate).")')
      CALL IPCCREATE(socket_kind,protocol_kind,flags,option,
     +                        call_socket_descriptor,result)
      here = 2
      IF (result  .NE. 0) GO TO 99

C     ----------------------------------------------------------------
C     With the peer's node name and socket name, we call IPCLookup to
C     to gain access to the peer's call socket. If successful, a path
C     descriptor will be returned which we need for IPCConnect.
C     IPCLOOKUP searches the socket registry at the given node for our
C     peer's name. This call returns a path_report_descriptor that is
C     used in the following IPCCONNECT call to request a connection with
C     the peer.
C
C     Because it is possible for IPCLookup to search for the socket
C     name before our peer places it in its node's socket registry, we
C     will try to look up the name several times before aborting.
C     -----------------------------------------------------------------
      counter = 0
      flags = 0
      write(1,'("ipc4: looking up peer CALL socket (IPCLookup).")')

21    CALL IPCLOOKUP(socket_name,socket_length,node_name,node_length,
     +    flags,path_report_descriptor,protocol_returned,socket_kind,
     +    result)

      counter = counter + 1
      here = 4

      IF (result .EQ. 0) GO TO 28

      IF (result .NE. 37) GO TO 99

      IF (counter .LE. 500) THEN
         GO TO 21
      ELSE
         GO TO 99
      ENDIF

C     -----------------------------------------------------------------
C     The call_socket_descriptor returned by IPCCREATE and the
C     path_report_descriptor returned by IPCLOOKUP are used in
```

```
C     IPCCONNECT to request a connection with ipc3. The
C     VC_socket_descriptor returned by IPCCONNECT is used in subsequent
C     calls to reference the connection. Once this call has completed
C     successfully, you may optionally release the call socket descriptor
C     by calling IPCSHUTDOWN in order to return resources to the system.
C     Doing so will not affect the newly-created VC socket descriptor.
C     ------------------------------------------------------------------

28    flags = 0
      CALL IPCCONNECT(call_socket_descriptor,path_report_descriptor,
     +                     flags,option,vc_socket_descriptor,result)

      here = 5
      IF (result .NE. 0) GO TO 99

      flags = 0
      data_length = 20

C     -----------------------------------------------------------
C     IPCRECV is called to determine if the connection has been
C     established.
C     -----------------------------------------------------------
      CALL IPCRECV(vc_socket_descriptor,data_buffer,data_length,
     +                                       flags,option,result)

      here = 6
      IF (result .NE. 0) GO TO 99

C     ----------------------------------------------------------
C     Data is sent to ipc3 on the newly established connection.
C     ----------------------------------------------------------
      write(1,'("ipc4: sending data message (IPCSend).")')
      flags = 0
      msg_buffer_length = 20
      CALL IPCSEND(VC_socket_descriptor,send_buffer,
     +         msg_buffer_length,flags,option,result)

      here = 7
      IF (result .NE. 0) GO TO 99

C     --------------------------------------------------------------

C     After the data is sent, ipc4 initiates the shutdown dialogue
C     by sending a shutdown "message" to ipc3.
C     --------------------------------------------------------------
      write(1,'("ipc4: sending shutdown message (IPCSend).")')
      flags = 0
      CALL IPCSEND(vc_socket_descriptor,shut_down_message,
     +               msg_buffer_length,flags,option,result)

      here = 8
      IF (result .NE. 0) GO TO 99
```

```
C     -------------------------------------------------------------
C     After it receives the shutdown “message,” ipc3 will send its
C     own shutdown “message” to ipc4. IPCRECV is called to receive
C     this data.
C     -------------------------------------------------------------
      write(1,’(“ipc4: waiting to receive shutdown msg (IPCRecv).”)’)
      flags = 0
30    CALL IPCRECV(vc_socket_descriptor,receive_buffer,
     +           msg_buffer_length,flags,option,result)

      here = 9
      IF (result .NE. 0) GO TO 99

C     -----------------------------------------------------------------
C     If the receive_buffer contains the shutdown “message,” ipc4 will
C     call IPCSHUTDOWN to shut down its VC socket descriptor and
C     terminate the connection.
C     -----------------------------------------------------------------
      IF (receive_buffer .EQ. shut_down_message) THEN
          flags = 0
          write(1,’(“ipc4: shutting down connection (IPCShutdown).”)’)
          CALL IPCSHUTDOWN(vc_socket_descriptor,flags,option,result)
          here = 10
          IF (result .NE. 0) GO TO 99
          GO TO 100                        ! we’re done.

C     ---------------------------------------------------------------
C     Since the only data ipc4 receives from ipc3 is a shutdown message
C     it should never branch to the following ELSE statement. If this
C     process were the recipient of several IPCSEND calls, it should
C     call IPCRECV again.
C     ----------------------------------------------------------------
      ELSE
         WRITE(1,’(10A2)’)(receive_buffer(index),index=1,10)
         GO TO 30
      ENDIF

99    WRITE (1, ’(“ipc4: result error code: ”,I4)’) result
      WRITE (1, ’(“ipc4: at location: ”,I4)’) here

100   STOP

      END
```

6

Remote Process Management

Overview

Remote Process Management (RPM) is an NS Common Service that enables a process on one NS-ARPA/1000 node to schedule, control, and terminate a program on the local node or at a remote node. RPM calls are made programmatically. Because RPM includes many RTE-equivalent features that are documented using the term, *program*, it is used instead of *process* throughout this section.

Programs that use RPM calls can be categorized as either *parent* or *child* programs. The scheduling program is called the parent program. The scheduled program is called the child program. The child program must be an executable file on the node on which it is to be scheduled. Figure 6-1 shows a simple parent-child relationship. Different types of parent-child relationships are explained in “RPMCREATE” later in this section.

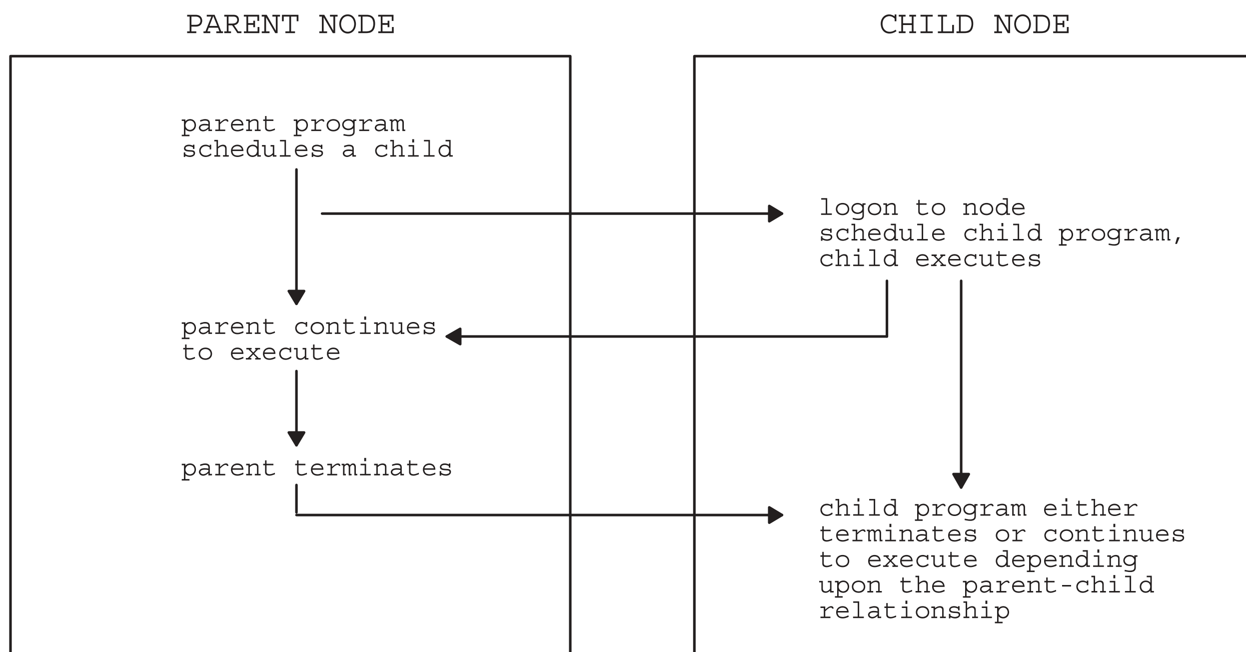


Figure 6-1. Parent-Child Relationship

Although the child program can exist on the local node, an advantage of RPM is the ability to schedule programs on a remote node. In this section, a node can be a local or a remote node, unless specifically stated otherwise.

Features of RPM

RPM facilitates scheduling remote NetIPC programs, and thus enhances the use of distributed Network Interprocess Communication. However, RPM is not limited to monitoring only NetIPC programs. You can use RPM to schedule any specified program having an executable file. Other features of RPM include the following:

- Scheduling a program in any session with a specified logon name.
- Scheduling several programs at the same node including scheduling several programs with wait (wait for child program to complete execution).
- Scheduling several programs to be within the same specified session.
- Scheduling a child program to be dependent upon the parent. If the parent program terminates, the child also terminates.

The form of remote processing offered by RPM is different than that provided by DEXEC. Because DEXEC is a DS/1000-IV Compatible Service, it can access NS-ARPA/1000 and DS/1000-IV nodes. RPM accesses only NS-ARPA/1000 nodes. RPM is intended to complement DEXEC. DEXEC is a distributed version of RTE EXEC calls, which contain functions not directly related to program scheduling and control, such as read/write and other I/O calls. In contrast, RPM provides functions for managing programs (some of which cannot be performed by EXEC calls alone) and does not have programmatic I/O requests.

Summary of RPM Calls

Application programs access RPM via four calls listed alphabetically in Table 6-1. A brief explanation of the RPM calls is presented here. More details are given in the descriptions of each RPM call later in this section.

Table 6-1. RPM Calls

Call	Description
RPMCONTROL (from a parent program)	Controls the execution of a child program at the remote or local node. The parent program can do one of the following operations: • Suspend a child program. • Resume a child program. • Set the break flag for a child program. • Change the program priority for a child program. • Get the program status of a child program.
RPMCREATE (from a parent program)	Schedules a child program and, if necessary, creates a session in which the program will run. The parent program can also do one or more of the following operations: • Wait for a child program to complete execution. • Have child programs share the same session. • Cause automatic termination of a child program when the parent program terminates. • Set the working directory for a child program. • Restore the child program from an executable file. • Pass a string to a child program. • Assign a memory partition for a child program. • Set the child program's scheduling priority. • Change the working set size for a child program. • Change the VMA space size for a child program. • Change the CDS code size for a child program. • Change the CDS data size for a child program. • Time schedule a child program. • Schedule a child program immediately. • Queue schedule a child program immediately.
RPMGETSTRING (from a child program)	Retrieves strings from the parent program.
RPMKILL (from a parent program)	Terminates a specified child program scheduled by an RPMCreate call.

RPM Programming Considerations

The following subsection explains the RPM program scheduling and program terminating considerations.

RPM parent programs must be compiled and linked as CDS programs. RPM child programs can be either CDS or non-CDS programs. If an RPM child program makes an RPM call, then it must be a CDS program. Refer to the *RTE-A Programmer's Reference Manual* and *RTE-A Link Manual* for more information on CDS programs.

A stack size of 5000 words is a recommended stack size for RPM programs. If a stack size error occurs when loading an RPM program, relink the program with a larger stack size. For example

```
CI> link parent.run
LINK> LK                     prepare program for changing
LINK> ST,6000                change stack size to 6000 words
LINK> EN
CI>
```

Alternately, the stack size can be specified in a .LOD file. Refer to the *RTE-A LINK User's Reference Manual* for more information.

Any program making an `RPMCreate` call becomes a parent program. The child program must be an executable file. `RPMCreate` must be issued before any other RPM calls can be made on or by that child program. `RPMCreate` returns a unique program descriptor to the parent program identifying the child program. This program descriptor is used in subsequent `RPMControl` and `RPMKill` calls from the parent program.

The parent program may also send a string to the child in the `RPMCreate` call. The child program uses the `RPMGetString` call to retrieve information from the parent program. The child program must be a CDS program if it makes the `RPMGetString` call or any RPM other call. Otherwise, the child may be either a CDS or a non-CDS program.

Once the child program has been scheduled, the parent can use an `RPMControl` call to send control requests to the child program. The `RPMControl` call can also be used to receive status information regarding the child program.

To terminate the child program, the parent should use the `RPMKill` call. A program may issue up to 31 `RPMCreate` calls. If more calls are needed, the parent must issue `RPMKill` calls for the already scheduled child programs. This is necessary even if they have already been terminated by another means to help RPM clean up and re-use memory. If there is not enough memory to create a child program, RPM generates an "Insufficient memory to create a child program" error (error code 10).

The child program can itself become a parent program by issuing an `RPMCreate` call. There is no upper limit to this hierarchy. As long as there are enough RTE resources available for scheduling programs (such as ID segments, sessions, memory partitions), a child can become a parent and schedule another child which then becomes a parent, and so on.

If a program (either a child or a parent) schedules another program using RTE EXEC calls, then that scheduled program is *not* considered to be an RPM child program. RPM does not monitor EXEC scheduled programs.

Here are a few considerations when creating and removing RPM programs:

- RPM programs that use the RTE `DTACH` and `ATACH` programming calls are not supported. Such programs separate themselves from RPM's control. RPM cannot monitor these programs.
- To ensure successful cleanup of NS-ARPA resources, do not remove the ID segment of the child program (such as using the RTE `OF,`*`program`*`,ID` command).
- It takes up to five seconds for RPM to cleanup after a child program terminates. Do not try to create the same child within that time.

RPM Syntax Conventions

The syntax provided in the following pages for each RPM call is meant to illustrate a Pascal procedure call statement. Parameters that are either output, or both input and output, are underlined in the syntax diagram. All other parameters are input parameters. Please refer to the sample programs for examples of Pascal and FORTRAN variable declarations.

All RPM call parameters are required. You may pass a zero in some parameters in order to obtain a default value.

The *flags*, *opt*, *result*, and *nodename* parameters are common parameters used in the RPM calls. They follow the same conventions as the NetIPC parameters. For quick reference, these parameters are briefly explained in the following paragraphs. For further information on these parameters, refer to "NetIPC Common Parameters" in the preceding section, "Network Interprocess Communication."

Use the `InitOpt`, `AddOpt`, and `ReadOpt` NetIPC calls to facilitate your use of the *opt* parameter. These NetIPC calls are explained in "Special NetIPC Calls" also in the section, "Network Interprocess Communication."

Flags Parameter

The *flags* parameter is a bit map of 32 *special request bits*. By setting bits in the *flags* parameter, you can invoke various services in the `RPMCreate` call. The `RPMControl` call also includes a *flags* parameter, but it is reserved for future use. However in this call, the *flags* parameter must be initialized to zero before the call can be used. The *flags* parameter must also be cleared after it is used, because a non-zero value may be returned in *flags*. This precaution should also be taken when programming with NetIPC calls.

Note NetIPC and RPM calls assume that the bits in the `flags` parameter are numbered from left to right with the most significant bit being bit *one* and the least significant bit being bit 32.

`MSB` `1 2 3 4 5 6 ...`	`32`	Pascal, NetIPC, and RPM
`MSB` `31 30 29 28 ...`	`0`	FORTRAN

The "Flags Parameter" subsection of the "Network Interprocess Communications" section explains how the *`flags`* parameter is declared and manipulated in Pascal and FORTRAN.

Opt Parameter

The *`opt`* parameter allows you to request optional services when invoking the `RPMCreate` call. The *`opt`* parameter is an array which enables a varying number of arguments to be specified.

Use the `InitOpt, AddOpt`, and `ReadOpt` NetIPC calls to facilitate your use of the *`opt`* parameter. These NetIPC calls are explained in "Special NetIPC Calls" in the section, "Network Interprocess Communication."

Result Parameter

Every RPM call has a *`result`* parameter. If an error occurs when a program makes an RPM call, an error code is returned in this parameter. The *NS-ARPA/1000 Error Message and Recover Manual* lists and explains the RPM error codes.

Nodename Parameter

A *node name* (the *`nodename`* parameter) refers to a node and has a hierarchical structure as follows:

`node``[.`*`domain`*`[.`*`organization`*`]]`

The NS-ARPA node name syntax is described previously in "Node Names" in the "Introduction" section of this manual.

RPMCONTROL

Controls the execution of a child program.

Syntax

```
RPMCONTROL (pd,nodename,nodelen,reqcode,wrtdata,wrtlen,
            readdata,readlen,flags,result)
```

Parameters

pd — *Byte array (Pascal); Word array (FORTRAN), by reference*. An array of 16 bytes containing the program descriptor of the child program to which control requests are sent. The program descriptor is a unique value returned from the `RPMCreate` call. Refer to the `RPMCreate` description in this section for more information about *pd*.

nodename — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A variable length array identifying the node on which the child program resides. The syntax of the node name is *node*[.*domain*[.*organization*]], which is further described in “Node Names” of the “Introduction” section and in “Nodename Parameter” of the “Network Interprocess Communication” section of this manual.

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If the *nodelen* parameter is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen — *32-bit non-negative integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the *nodename* parameter. If *nodelen* is zero (0), the *nodename* parameter is ignored and the node name defaults to the local node. The maximum length of a fully-qualified node name length is 50 bytes.

If *nodelen* is zero, it is assumed that the parent is either sending the `RPMControl` request to a dependent child program that it previously scheduled or to a child program on the parent’s node (which is the local node). Refer to “Dependent and Independent Child Programs” in the `RPMCreate` section for an explanation of dependent child programs.

reqcode — *32-bit non-negative integer, by value in Pascal, by reference in FORTRAN*. The request code for the control operation to be performed on the child program. The request codes are RTE-A specific, and you should refer to the *RTE-A User’s Manual* and *RTE-A Programmer’s Reference Manual* for detailed explanations of these RTE-A commands and calls. The request codes for `RPMControl` are as follows:

- 20001—Suspend execution of the child program. No data is required and none is returned. Therefore, *wrtlen* and *readlen* must be zero.

  This request is equivalent to the RTE-A SS (suspend program) command. As in RTE-A, if a child program is in a state that prevents it from being suspended, the program is not suspended until it is in the right state. No error is returned in *result* in this case (similarly as in RTE-A).

- 20002—Resume execution of the child program at the point it was suspended. No data is required and none is returned. Therefore, *wrtlen* and *readlen* must be zero.

  This request is equivalent to the RTE-A GO (resume program) command.

- 23120—Set the IFBRK break flag in the child program's ID segment. The child program must check this flag with the RTE-A IFBRK system call to respond to it. No data is required and none is returned. Therefore, *wrtlen* and *readlen* must be zero.

- 23030—Change the child program's priority. The priority number is a 16-bit integer from 1 to 32767 with the smaller number representing the higher priority. The priority number is placed in the *wrtdata* parameter. This request is equivalent to the RTE-A PR (change program priority) command, except that you cannot request the program priority.

- 23130—Get the child program's status. RPM invokes the RTE-A IDINFO call to obtain the status. Refer to the *RTE-A Programmer's Reference Manual* for more information and for a list of possible status values. The status is a 16-bit integer returned in the *readdata* parameter. The *readlen* parameter must be set to at least two bytes.

Due to a network time delay, the actual execution of any of the above requests may be delayed.

wrtdata

Byte array (Pascal); Word array (FORTRAN), by reference. A variable length array with data to be sent to the child program for the request. When a *reqcode* of 23030 is specified, the program priority is placed in *wrtdata*. The program priority is declared as a 16-bit integer, and the *wrtlen* parameter is two bytes.

wrtlen

32-bit non-negative integer, by value in Pascal, by reference in FORTRAN. Length in bytes of *wrtdata*.

Only *reqcode* 23030 (PR command) sends information from the calling parent program in *wrtdata*. The parameter *wrtlen* must be two. All other request codes must specify a zero for *wrtlen*.

readdata	*Byte array (Pascal); Word array (FORTRAN), by reference*. A variable length array with the data returned to the calling parent program. If *reqcode* of 23130 is used, then the program status is returned in *readdata*.
readlen (input/output)	*32-bit non-negative integer, by reference*. On input, *readlen* is the maximum number of bytes expected in the *readdata* parameter. On output, *readlen* is the actual number of bytes received in the *readdata* parameter. If *result* is non-zero (an error has occurred), *readlen* is set to zero, and no data is in *readdata*. Only *reqcode* 23130 (IDINFO call) receives information from a child program (program status) in *readdata*. The parameter *readlen* must be two. All other request codes must specify a zero in *readlen*.
flags	*32 bits, by reference*. A 32-bit map of special request bits. This parameter is reserved for future use. This parameter must contain all zeroes (cleared).
result	*32-bit non-negative integer, by reference*. The result of the RPMControl request; zero if no error. If *result* is not zero, an error has occurred. Errors are defined in the *NS-ARPA/1000 Error Message and Recover Manual*.

Discussion

RPMControl is used for controlling execution of a child program that was previously scheduled by an RPMCreate call.

The parent program need not be the original parent to call RPMControl to a specific child program. As long as the correct program descriptor is supplied in RPMControl, you can send control requests to any child program in any session on any specified node (*nodename* parameter).

RPMCREATE

Schedules a program and, if necessary, creates a session for that program to run in.

Syntax

```
RPMCREATE (progname,namelen,nodename,nodelen,login,loginlen,
           password,passwdlen,flags,opt,pd,result)
```

Parameters

progname — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A variable length array of ASCII characters containing the name of the child program to be scheduled. If the child program does not reside in the working directory, the full path name of the child program must be specified. The child program must be an executable file. Although RPMCreate may accept program names up to 256 characters, the child program name on an HP 1000 RTE-A system may not exceed 64 characters. The *progname* parameter is not case sensitive.

namelen — *32-bit positive integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the program name. This must always be a positive integer.

nodename — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A variable length array of ASCII characters identifying the node on which the child program resides. The syntax of the node name is *node*[.*domain*[.*organization*]], which is further described in "Node Names" of the "Introduction" section and in "Nodename Parameter" of the "Network Interprocess Communication" section of this manual.

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If the *nodelen* parameter is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen — *32-bit non-negative integer, by value in Pascal, by reference in FORTRAN*. Length in bytes of the *nodename* parameter. If *nodelen* is zero (0), the *nodename* parameter is ignored, and the child program is scheduled on the same node as the parent. A fully-qualified node name length may be 50 bytes long.

login — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A logon sequence for the (local or remote) node on which the child program is to be scheduled. *login* is an RTE-A logon without the password (defined in the *password* parameter described below). RPM needs the logon name to logon to the local or remote node.

loginlen — *32-bit non-negative integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the logon sequence. The maximum length for a logon on RTE-A is 16 bytes. If *loginlen* is zero (0), *passwdlen* must be zero.

When the *loginlen* and *passwdlen* are both zero and *nodename* is the local node, the child program is scheduled and attached to the *parent program's session*. Even if the session-sharing flag (flags[31]) is not set to disable session-sharing, the child program will session-share with the parent program in this case.

If *nodename* is NOT the local node and *loginlen* is zero, RPMCreate will return an error in *result*.

password — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A variable length array with the password for the RTE-A logon specified in *login*. If no password is required, the *passwdlen* parameter must be zero (0).

passwdlen — *32-bit non-negative integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the *password* parameter. If *passwdlen* is zero (0), *password* is ignored. The maximum password length in RTE-A is 14 bytes.

flags — *32 bits, by reference*. A 32-bit map of special request bits representing various functions. Refer to "Flags Parameter" in the "Network Interprocess Communication" section for explanations of the 32 special request bits and how to use them in Pascal/1000 and FORTRAN 77. The following flags are defined on input (bit 1 is the most significant bit); all other flags must be set to zero:

- flags[2] —wait for child (input). When set, this flag causes the calling parent program to wait until the child program terminates.

  The default is zero (0) for no waiting. The parent program resumes execution immediately after it is notified that the child program is successfully scheduled or an error occurs. Check the *result* parameter for an error.

- flags[31] —session-sharing (input). When set, this flag causes the child program to share a session with other child programs. The parent must set this bit for each child that is to share the same session. Refer to "Session-Sharing Among Child Programs" later in this section for more details on how child programs share sessions.

  The default is zero (0) for no session-sharing—the child program is scheduled in a new session.

Regardless of how `flags[31]` is set, session-sharing will occur on the local node in the parent program's session if *nodename* and *loginlen* are specified as follows:

- *nodename* specifies the local node or *nodelen* is zero.
- *loginlen* is zero.

- `flags[32]` —dependent (input). When set, this flag causes the scheduled child program to be dependent on the parent program. When the parent program terminates, the child program terminates automatically.

  The default is zero (0) making the child program independent. The scheduled child program continues executing on its own even after the parent program terminates. Refer to "Terminating Dependent and Independent Child Programs" later in this section for more information.

opt — *Byte array (Pascal), Word array (FORTRAN), by reference*. An array of options and associated information. The format of an *opt* array is the same as the NetIPC *opt*. Refer to "Opt Parameter" in the "Network Interprocess Communication" section of this manual for a detailed explanation. The options are equivalent to some RTE-A commands and calls dealing with program scheduling. Refer to the *RTE-A User's Manual* and *RTE-A Programmer's Reference Manual* for more information on the RTE-A commands and calls.

A detailed description of `RPMCreate` options is given later in this section under the subsection, "RPMCREATE Options." A list of `RPMCreate` options is presented in Table 6-2.

If no options are specified, the child program is assumed to reside in the current working directory of the session to which it logged on or in the `::programs` directory. RPM causes the child program to be restored with the clone name returned by `FmpRpProgram`. The child program is then scheduled with an `EXEC 10` (immediate schedule without wait) call with no parameters.

The total length of the *opt* array must be 996 bytes or less.

pd — *Byte array (Pascal), word array (FORTRAN), by reference*. An array of 16 bytes containing a unique program descriptor returned by RPM. This program descriptor is used to identify the scheduled child program. This value, randomly generated, is presumed to be unique across all nodes. A valid program descriptor is always a non-zero value. If `RPMCreate` is unsuccessful, *pd* is set to all zeroes.

The program descriptor is used in subsequent RPM calls to identify the child program.

result — *32-bit non-negative integer, by reference*. The result of the RPMCreate request; zero if no error. If *result* is not zero, an error has occurred. Errors are defined in the *NS-ARPA/1000 Error Message and Recover Manual*.

Table 6-2. RPMCreate Options

Numeric Code	Description	RTE-A Equivalent
Group 1:		
23000	Set working directory name	FmpSetWorkingDir
Group 2:		
23010	Restore program	RP command
Group 3:		
20000	Pass string	none
23020	Assign partition	AS command
23030	Set program priority	PR command
23040	Change working set size	WS command
23050	Change VMA space size	VS command
23060	Change CDS code size	CD command*
23070	Change CDS data size	DT command*
		*not exactly like RTE
Group 4:	If used, only one can be specified:	
23080	Time list scheduling	EXEC 12 call
23090	Immediate schedule w/o wait	EXEC 10 call
23100	Queue schedule w/o wait	EXEC 24 call
23110	Run program	FmpRunProgram

Discussion

RPMCreate enables the calling program to schedule another program. It must be the first RPM call made by a parent to schedule a child program. The child program must be an executable file on the local or at a remote system. In RTE, an executable file is a type 6 file. If a full path name is not given, the type 6 file must be in the search path of the child's session. Search paths are documented in the RTE PATH command in the *RTE-A User's Manual*.

By default, RPM creates a new session for the child program on the specified node (*nodename* and *nodelen* parameters). RPM uses the logon specified in the *login* parameter and the password in *password*. RPM schedules the child program to execute with an RTE EXEC 10 call, and the parent resumes executing itself.

RPMCreate returns a unique program descriptor to the parent program identifying the child program. This program descriptor is used in subsequent RPMControl and RPMKill calls from the parent program. The parent can use an RPMControl call to send control requests to the child program. The RPMControl call can also be used to receive status information regarding the child program. To terminate the child program, the parent should use the RPMKill call. The child can also be terminated automatically (without an RPMKill) when the parent terminates. Refer to "Terminating Dependent and Independent Child Programs" later in this section for more information.

A program may issue up to 31 RPMCreate calls. If more calls are needed, the parent must issue RPMKill calls for the already scheduled child programs. This is necessary even if they have already been terminated by another means to help RPM clean up and re-use memory. If there is not enough memory to create a child program, RPM generates an "Insufficient memory to create a child program" error (error code 10).

The parent program can also send a string to the child in the RPMCreate call with option 20000. The string is defined as part of the *opt* array parameter. The child program uses the RPMGetString call to retrieve information from the parent program. The child program must be a CDS program if it makes a RPMGetString call. If the child program is not using RPMGetString, it can be either a CDS or a non-CDS program.

The child program can itself become a parent program by issuing an RPMCreate call. There is no upper limit to this hierarchy. As long as there are resources available for scheduling programs (such as ID segments, sessions, memory partitions), a child can become a parent and schedule another child which then becomes a parent, and so on.

The child can be scheduled to share a session with other child programs. Refer to "Session-Sharing Among Child Programs" later in this section.

For more programming information, refer to the "RPM Programming Considerations" subsection earlier in this section.

Programs Scheduled by RPM Child Programs

RPM child programs should avoid other methods of scheduling programs, such as with EXEC and DEXEC. Such programs are *not* considered to be RPM child programs. RPM cannot monitor these non-RPM programs. Programs not under RPM's control may terminate without RPM's knowledge and also cannot be accessed by any RPM call. RPM programs that use the RTE DTACH and ATACH programming calls are also not supported for the above reasons.

If you want to schedule a program on the local node, use RPMCreate and specify the local node in the *nodename* and *nodelen* parameters. Then the child program will not be unexpectedly terminated.

Figure 6-2 illustrates two different cases of scheduling a program. In the first case, the child program (Child 1) has scheduled another child program (Child 2). Child 2 will continue to execute after Child 1 terminates. In the second case, the child program (Child 1) has scheduled a child program (Child 2) in its session using a call such as EXEC 10 or EXEC 24. This non-RPM program will be terminated by the system if Child 1 terminates and if there are no other RPM child programs running in that session. The session is logged off when there are no more child programs running in that child's session. The session logging off causes the non-RPM program to be automatically terminated.

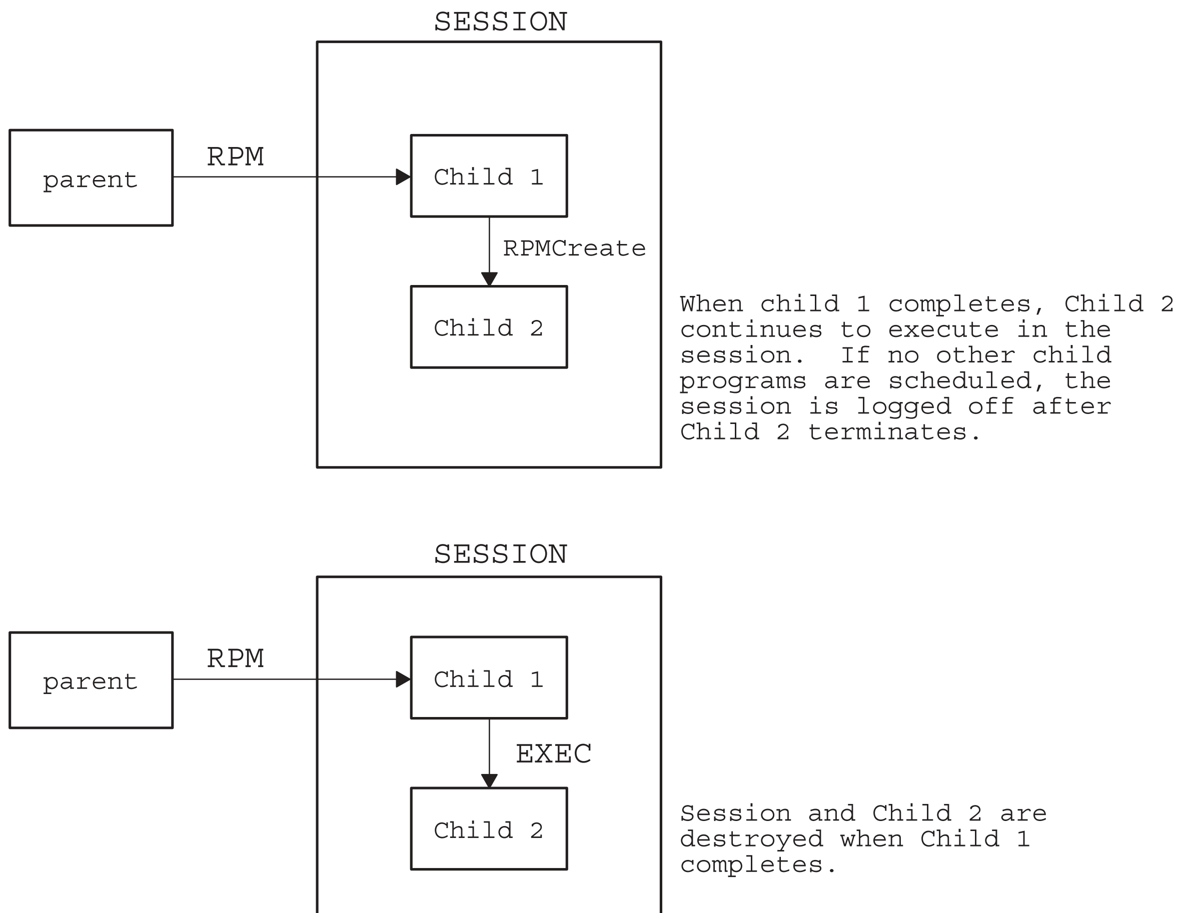


Figure 6-2. Example of Child Programs Scheduling Another Program

Terminating Dependent and Independent Child Programs

The dependent bit of the *flags* parameter determines whether the new child program will be dependent (bit is set) on its parent or independent (bit is clear; the default). Dependent mode ensures that the new child program will not become an "orphan" in the event of a program, system, or network link failure. A dependent child program will terminate under any one of the following conditions:

- RPMKill has been called. (RPMKill can terminate either an independent or dependent child program.)
- The parent program terminates before calling RPMKill.
- The remote communication line goes down. Either NS-ARPA has been shutdown or a transport problem has occurred.
- The system on which the parent is running fails.

- The ID segment of the RPM monitor has been removed (the RTE `OF,`*`program`*`,ID` command was entered).

If the child program is independent, it continues to execute until it terminates on its own or until it is explicitly terminated by `RPMKill`. Independent mode is less costly in terms of resources: the connection set up for the `RPMCreate` is not maintained after the child program is scheduled.

Session-Sharing Among Child Programs

If the session-sharing bit of the *`flags`* parameter is set, that child program is regarded as *shareable*. `RPMCreate` tries to schedule the child program in a session in which other child programs may be residing. However, RPM cannot always guarantee that the child program will actually share a session. In order for multiple child programs to reside in an RPM-created session (as distinguished from a parent's session), three criteria must be satisfied:

- All the programs must have been scheduled using `RPMCreate` with the session-sharing bit of the *`flags`* parameter set. Child programs scheduled with this bit set to zero cannot share sessions with programs which have this bit set. The former child programs are each scheduled in another dedicated session.

- All the programs must have been scheduled using `RPMCreate` from the same parent program or from different parent programs residing within the same session on the local node. Parent programs that are running in different local sessions will always schedule child programs in different sessions.

- All the programs must have the same logon string (*`login`* and *`loginlen`* parameters) and it cannot be the same as the parent's logon. All the necessary passwords (*`password`* and *`passwdlen`* parameters) must match.

Figure 6-3 on the following page shows examples of how child programs can share sessions.

The child program will share the *parent program's session* on the local node if *`nodename`* and *`loginlen`* are specified as follows:

- *`nodename`* specifies the local node or *`nodelen`* is zero.

- *`loginlen`* is zero.

In the above case, the session-sharing bit is either set or not set. Session 84 in Figure 6-3 shows an example of a child program in a parent's session. Table 6-3 summarizes where sessions are created.

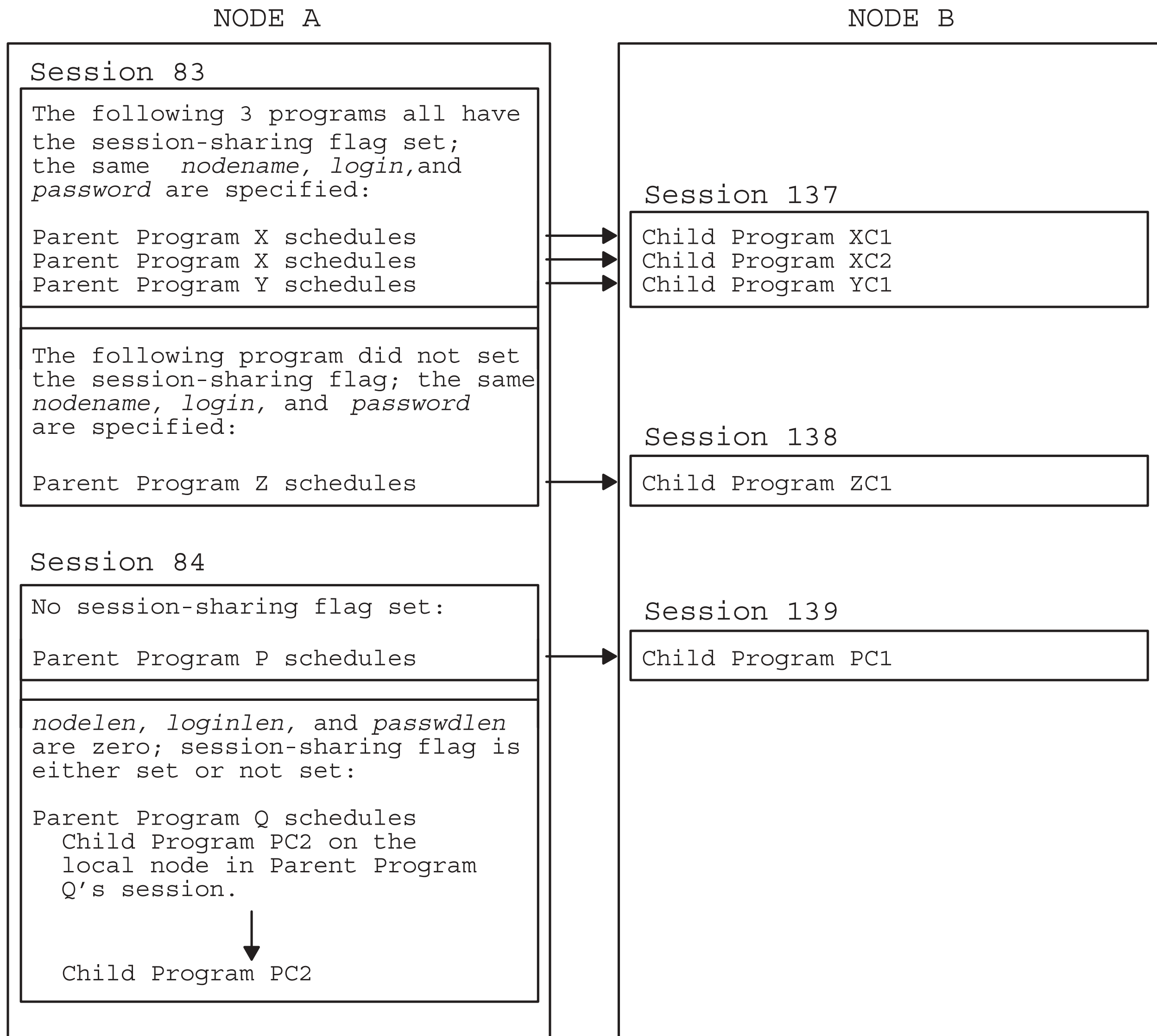


Figure 6-3. Parent-Child Relationships When Session-Sharing

Table 6-3. Where Sessions Are Created

Value of *nodelen*	Value of *loginlen*	Where Session is Created
nodelen = 0	*loginlen* = 0	Parent's node, parent's session.
nodelen = 0	*loginlen* not 0	Parent's node, new session.
nodelen not 0	*loginlen* = 0	Not permitted; error.
nodelen not 0	*loginlen* not 0	Remote node, new session; could be parent's node if *nodename* is the parent's node.

RPMCREATE Options

At the same time that a child program is scheduled, some equivalent RTE-A commands can be sent to the child program using the `RPMCreate` options. Under the *opt* parameter description of `RPMCreate`, Table 6-3 lists `RPMCreate` options. The format and contents of the *opt* parameter are described previously in “Opt Parameter” of the section, “Network Interprocess Communication.”

Here are some guidelines when specifying these options.

- *No Options Specified*. If no options are specified, the child program is assumed to reside in the current working directory of the specified session (from *login* and *loginlen*) or in the `::programs` directory. RPM causes the child program to be restored with the name returned by `FmpRpProgram`. The child program is then scheduled with an RTE `EXEC 10` (immediate schedule without wait) call with no parameters.

- *Option Groups*. `RPMCreate` request options are divided into four groups. Options in an earlier group, if specified, must be completely specified before options in a later group can be given. The Group 1 option must be specified before any Groups 2, 3, or 4 option is specified. The Group 2 option must be specified before any Group 3 or Group 4 option is specified. Group 3 options must be specified before any Group 4 option is specified. Only one Group 4 option can be specified. All other options after a Group 4 option is specified are ignored.

  For example, if a parent program wants to both set the working directory (Group 1 option 23000) and restore the child program (Group 2 option 23010), the parent must specify the Group 1 option (23000) first.

  If a Group 1, 2, or 3 option is specified more than once, no error occurs. Instead, the last version of that option takes affect. For example, if the priority option (23030) is specified twice, the second priority option is executed. An exception is the pass string option 20000, because more than one string can be passed with this option.

- *Group 4 Scheduling Options*. Group 4 options cause the child program to be scheduled. No other options (not even pass string option, 20000) can be specified after a Group 4 option.

  If no Group 4 option is specified, the `EXEC 10` (immediate schedule without wait) call is issued after the other options have been processed. No parameters are passed to the child program. Only one Group 4 option can be specified within an `RPMCreate` call.

- *Schedule With Wait*. The equivalent of the RTE-A `EXEC 9` (immediate schedule with wait) and `EXEC 23` (queue schedule with wait) calls can be achieved in RPM with options 23090 and 23100, respectively, and with the wait bit set in the *flags* parameter.

Refer to the *RTE-A User's Manual* and *RTE-A Programmer's Reference Manual* for more information on the equivalent RTE commands and calls.

Adding Options Into the Opt Array

The following subsections give detailed explanations of the four option groups. The options and related data are placed into the *opt* array by using the `AddOpt` call. This call is documented in "Special NetIPC Calls" in the "Network Interprocess Communication" section. The `AddOpt` call uses all the parameters listed below. However, the following subsections present only the *optioncode*, *datalength*, and *data* parameters, because these are the parameters that have specific values for each `RPMCreate` option.

Syntax

ADDOPT(*opt*,*argnum*,*optioncode*,*datalength*,*data*,*error*)

AddOpt Parameters

opt — *Byte array (PASCAL); Word array (FORTRAN), by reference*. The *opt* parameter to which you want to add an argument. Refer to "Opt Parameter" in the "Network Interprocess Communication" section for information on the structure and use of this parameter.

The total length of the *opt* array must be 996 bytes or less.

argnum — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The number of the argument to be added. The first argument number is zero.

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. An `RPMCreate` option code. These codes are explained in the subsequent subsections.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of the data to be included. This information is provided in each `RPMCreate` option description on the following pages.

data — *Array, by reference*. A variable length array of data to be passed to the child program. Null strings are valid.

error — *16-bit integer, by reference*. The error code returned; zero if no error. Error codes are documented in the *NS-ARPA/1000 Error Message and Recover Manual*.

ADDOPT Example

The following example shows a Pascal program fragment that adds the pass string option (option code 20000) and data to the *opt* parameter.

```
{InitOpt initializes the opt array to contain two options --
 both for passing strings.}

INITOPT(opt,2,result);

{AddOpt is called to add 5 bytes from the data0 array.
 The first argument of the opt array is the number zero.
 20000 is the option code for passing strings.}

ADDOPT(opt,0,20000,5,data0,result);

{AddOpt is called to add another 5 bytes from the data1 array.
 The second argument is the number one.}

ADDOPT(opt,1,20000,5,data1,result);

{RPMCreate can now be called with the opt parameter.}

RPMCREATE(progname,namelen,nodename,nodelen,login,loginlen,
          password,passwdlen,flags,opt,pd,result);
```

RPMCreate Option 20000—Pass String

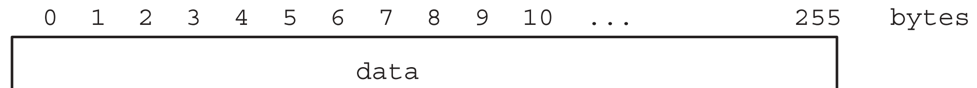
RTE-A System Equivalent: none.

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 20000 to indicate the "Pass String" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. A maximum of 256 bytes can be passed.

data — *Array, by reference*. A variable length array of data to be passed to the child program. Null strings are valid.



Discussion

The pass string option (20000) is a Group 2 option. This option can be specified more than once by the parent program for each string to be passed to the child program. Strings are queued to the child program in the order in which they are specified. The child program would issue an `RPMGetString` for each string it wants to obtain. For example, if three option 20000's were specified in *opt* with three strings, the child would issue three `RPMGetString` calls to retrieve the strings.

This option must be specified before any Group 4 option is specified.

All child programs invoking the `RPMGetString` call must be compiled as CDS programs.

RPMCreate Option 23000—Set Working Directory

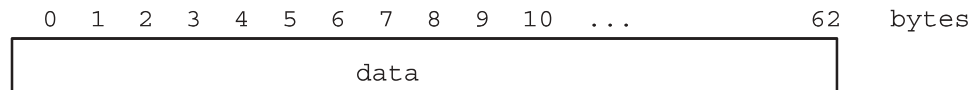
RTE-A FMP Equivalent: `FmpSetWorkingDir` call (documented in the *RTE-A Programmer's Reference Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23000 to indicate "Set Working Directory" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. The *data* is a working directory name. The *datalength* can be up to 63 bytes for an RTE working directory name.

data — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A packed array of characters specifying the working directory. The directory name must be fully-qualified. An exception would be if it is a subdirectory of the current working directory for the session created with the *login* parameter of `RPMCreate`. In this latter case, the current directory path can be omitted.



Discussion

The set working directory option (23000) is a Group 1 option that changes the working directory for the child program. If this option is specified more than once, no error occurs. Instead, the last version of this option takes affect. This option must be specified before any Group 2, 3, or 4 option is specified.

The working directory set by this option is used when no directory is specified in the *progname* parameter of `RPMCreate`. This working directory is searched first by the file system in the file search path. The new working directory can be a sub-directory.

If the set working directory option is not specified, the child program's working directory is the one currently associated with the session created with the *login* parameter. The child's search path is set to the current search path of that session. More information on search paths is documented in the *RTE-A User's Manual* under the `PATH` command (the "Modify UDSP" command).

A child program already running in that session may set the search path to override the default.

In shared sessions, the current working directory may have been changed by a previously scheduled child program in the same session. The parent program should always set the correct working directory when scheduling a child program in shared sessions. Also the working directory of a shared session may be changed by a new and unsuccessful `RPMCreate` call.

RPMCreate Option 23010—Restore Program

RTE-A FMP Equivalent: `FmpRpProgram` call (documented in the *RTE-A Programmer's Reference Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23010 to indicate the "Restore Program" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of `data` which is to be included in the `opt` array. Must be only one of the following values: 0, 6, or 7. No other values are allowed.

- If the length is 0, there is no `data`. The child program is restored under a system-assigned name. The child is always restored as a permanent ID segment.
- If the length is 6, it is assumed that the program name is specified in `data` and no cloning is to occur.
- If the length is 7, both the program name and letter `C` are specified. The letter `C` signifies to create a clone name.

data — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A six- or seven-byte array. The first six bytes is the name under which the child program should be restored. If the name is not specified, the program will be restored under a system-assigned name. The returned name is the first five characters of the child program name (minus the directory path and file type extension).

If the seventh byte contains the character `C`, a clone name is to be created. If the specified or assigned name from the first six bytes is already assigned, a clone name is created. For more information about cloning, refer to the *RTE-A User's Manual*.

0 1 2 3 4 5	6	bytes
child program name	C	

Discussion

The restore program option (23010) is a Group 2 option. If this option is specified more than once, no error occurs. Instead, the last version of this option takes affect. This option must be specified before any Group 3 or 4 option is specified.

The restore program option invokes the `FmpRpProgram` routine to restore a program from an executable file and create an ID segment for the program. If the restore program option is not specified, the child program is automatically restored. The child is always restored as a permanent ID segment. Therefore, the ID segment is *not* released as soon as the child program terminates. Within 5 seconds, the NS-ARPA/1000 cleanup routine, UPLIN, will release the ID segment.

RPMCreate Option 23020—Assign Partition

RTE-A System Equivalent: `AS` command (documented in the *RTE-A User's Manual*).

AddOpt Parameters

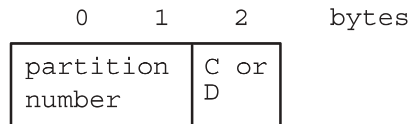
optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23020 to indicate the “Assign Partition” option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be only one of the following values: 2 or 3. No other values are allowed.

- If the length is 2, the 16-bit partition number is specified. The default is to assign the data section of the program to the reserved partition.
- If the length is 3, the 16-bit partition number should be followed by a `C` for code section or `D` for data section.

data — *Array, by reference*. A three-byte array. The first two bytes is a 16-bit integer specifying the reserved partition number in which the child program is to run.

The third byte is the character `C` or `D` to indicate either the code (`C`) or data (`D`) section. The code or data section of the program is assigned to the reserved partition. This argument applies only to CDS child programs. This argument can be in either upper or lower case.



Discussion

The assign partition option (23020) is a Group 3 option. This option causes the RTE `MESSS` command to be invoked to execute the RTE-A system `AS` command. This option causes a reserved memory partition to be assigned to the child program. The assign partition option can be specified only once, unlike the other Group 3 options. This option must be specified before any Group 4 option is specified.

The RTE-A system memory is divided at bootup time into dynamic and reserved partitions. Normally, when a program is run, it is assigned memory as required from the dynamic memory. Reserved partitions are partitions of fixed sizes that can be reserved for specific programs. Partition numbers are assigned sequentially from one as they are defined in the boot command file.

All child programs invoking any RPM call must be compiled as CDS programs. All child programs to be assigned to a code or a data partition must be CDS child programs. All other child programs may be either CDS or non-CDS.

RPMCreate Option 23030—Change Program Priority

RTE-A System equivalent: `PR` command (documented in the *RTE-A User's Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23030 to indicate the "Change Program Priority" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be a 2. No other values are allowed.

data — *16-bit integer, by reference*. An integer from 1 to 32767 specifying the child program priority.

```
   0      1     bytes
 ____________
|  priority  |
 ------------
```

Discussion

The set priority option (23030) is a Group 3 option. This option invokes the RTE `MESSS` call with the RTE-A system `PR` command. This option sets the priority of the child program. If this option is specified more than once, only the last one will take effect. This option must be specified before any Group 4 option is specified.

All programs running under RTE-A have a priority number which is recorded in the respective program ID segments. The priority number can be assigned when the program is written or when it is linked.

The priority number may be in the range of 1 to 32767, with smaller numbers representing higher priorities. Typical values for user application software would be in the range of 50 to 200. Higher priority real-time and system programs may be in the range of 1 to 40.

RPMCreate Option 23040—Modify Working Set Size

RTE-A System equivalent: `WS` command (documented in the *RTE-A User's Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23040 to indicate the "Modify Working Set Size" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be a 2. No other values are allowed.

data — *16-bit integer, by reference*. An integer from 2 to 1022 specifying the working set size in pages.

```
 0      1     bytes
+-----------+
| working   |
| set size  |
+-----------+
```

Discussion

The modify working set size option (23040) is a Group 3 option. This option invokes the RTE `MESSS` call with the RTE-A system `WS` command. This option sets the working set size for the child program. If this option is specified more than once, only the last one will take effect. This option must be specified before any Group 4 option is specified.

VMA programs are those that utilize an RTE-A feature which enables execution of programs requiring a very large amount of data storage. The data for a VMA program is contained in an area on disk called the Virtual Memory Area (VMA). The portion of data being processed is moved from disk to an area in memory called the Working Set (WS) so data is being transferred between VMA and WS as necessary during program execution.

RPMCreate Option 23050—Modify VMA Size

RTE-A System equivalent: `VS` command (documented in the *RTE-A User's Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23050 to indicate the "Modify VMA Size" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be a 2. No other values are allowed.

data — *16-bit integer, by reference*. An integer from 32 to 32767 specifying the virtual EMA size in pages.

```
  0      1     bytes
┌──────────────┐
│ virtual      │
│ EMA size     │
└──────────────┘
```

Discussion

The modify VMA size option (23050) is a Group 3 option. This option invokes the RTE `MESSS` call with the RTE-A system `VS` command. This option changes the VMA size requirements of a child program. If this option is specified more than once, only the last one will take effect. This option must be specified before any Group 4 option is specified.

RPMCreate Option 23060—Modify Code Partition Size

RTE-A System equivalent: `CD` command (documented in the *RTE-A User's Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23060 to indicate the "Modify Code Partition Size" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be a 2. No other values are allowed.

data — *16-bit integer, by reference*. A 16-bit integer specifying the maximum number of code segments permitted to remain in memory at once. This number must be less than or equal to the actual number of code segments for the program.

```
   0    1      bytes
+------------+
| code       |
| partition  |
| size       |
+------------+
```

Discussion

The modify code partition size option (23060) is a Group 3 option. This option invokes the RTE `MESSS` call with the RTE-A system `CD` command. This option sets the code partition size of the child program which is a CDS program. If this option is specified more than once, only the last one will take effect. This option must be specified before any Group 4 option is specified.

CDS programs have two areas of memory associated with them, one for code (the program itself) and one for data.

The modify code partition size option changes the size allocation of the code section. It changes the amount of memory the code will use by changing the number of code segments (pieces of the program) which will be kept in memory at one time; the other pieces will be kept on the disk. The RTE-A system will keep the most actively used pieces of the program in memory, leaving the others on the disk.

RPMCreate Option 23070—Modify Data Partition Size

RTE-A System equivalent: `DT` command (documented in the *RTE-A User's Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23070 to indicate the "Modify Data Partition Size" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be a 2. No other values are allowed.

data — *16-bit integer, by reference*. A 16-bit integer specifying the size of the data partition in pages.

0 1	bytes
data partition size	

Discussion

The modify data partition size option (23070) is a Group 3 option. This option invokes the RTE `MESSS` call with the RTE-A system `DT` command and is only used for CDS child programs.

This option changes the size allocation of the data section for a CDS child program.

If this option is specified more than once, only the last one will take effect. This option must be specified before any Group 4 option is specified.

RPMCreate Option 23080—Time Scheduling

RTE-A System Equivalent: `EXEC 12` call (documented in the *RTE-A Programmer's Reference Manual*).

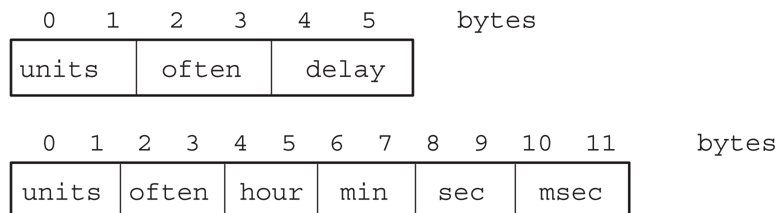
AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23080 to indicate the "Time Scheduling" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be 2, 4, 6, 8, 10, or 12 bytes. No other values are allowed. The exact length depends on the parameters specified.

- If the length is 2, only *units* is specified. The call is an Absolute Start Scheduling call to schedule the child program immediately.
- If the length is 4, the *units* and *often* values are specified. The call is an Absolute Start Scheduling call to schedule the child program immediately and how often it should be scheduled.
- If the length is 6, the values for *units*, *often*, and *delay/hour* are specified. If the *delay/hour* parameter is negative, the call is an Initial Offset Scheduling call. If the *delay/hour* parameter is non-negative, the call is an Absolute Start Scheduling call.
- If the length is 8, the values for *units*, *often*, *hour*, and *min* are specified. The call is an Scheduling Absolute Starting Time call.
- If the length is 10, the values for *units*, *often*, *hour*, *min*, and *sec* are specified. The call is an Scheduling Absolute Starting Time call.
- If the length is 12, the values for *units*, *often*, *hour*, *min*, *sec*, and *msec* are specified. The call is an Scheduling Absolute Starting Time call.

data — *Array, by reference*. A 2 to 12 byte array with the following contents:



units	A resolution code that specifies the time units. In conjunction with the parameter *often*, *units* specifies the time between each execution of the child program. *units* is one of the following values: 1 = tens of milliseconds 2 = seconds 3 = minutes 4 = hours
often	An integer value (0 to 4095) indicating the execution multiple or how often the program is to run.
delay	The initial offset. A negative number indicating the starting time of the first execution (not zero).

The following parameters collectively specify the starting time:

hour	The starting hour (0 to 23).
min	The starting minute (0 to 59).
sec	The starting second (0 to 59).
msec	The starting tens of milliseconds (0 to 99).

Discussion

The initial offset scheduling option (23080) is a Group 4 option. Only one Group 4 option can be specified. No other option can be specified after a Group 4 option.

This option invokes the RTE `EXEC 12` call (Initial Offset Scheduling/Absolute Start Scheduling), which schedules a program for execution at specified time intervals, starting either after an initial offset delay, or at a particular absolute start time. The values of the parameters for this call determine whether the child program should be scheduled using Initial Offset Scheduling and Absolute Start Scheduling.

The value of *delay* or *hour* determines whether the child will be scheduled with an absolute start time or an initial offset time. If this parameter is omitted, its value defaults to zero, and the child program will be scheduled at an absolute start time.

RPMCreate Option 23090—Program Scheduling (Immediate No Wait)

RTE-A System Equivalent: `EXEC 10` call (documented in the *RTE-A Programmer's Reference Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23090 to indicate the "Program Scheduling—Immediate No Wait" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be one of the following values: 0, 2, 4, 6, 8, 10, 13, or greater than 13 bytes. Any parameter that is not specified defaults to zero. The exact length depends on the parameters specified:

- If the length is 0, all parameters are omitted and their default value is zero.
- If the length is 2, only *pr1* is specified.
- If the length is 4, *pr1* and *pr2* are specified.
- If the length is 6, *pr1*, *pr2*, and *pr3* are specified.
- If the length is 8, *pr1*, *pr2*, *pr3*, and *pr4* are specified.
- If the length is 10, *pr1*, *pr2*, *pr3*, *pr4*, and *pr5* are specified.
- If the length is 13 or more, all seven parameters are specified.

data — *Array, by reference*. A variable length array with the following contents:

0 1	2 3	4 5	6 7	8 9	10 11 ... n	n + 1 bytes
pr1	pr2	pr3	pr4	pr5	bufr	bufln

pr1, *pr2*, *pr3*, *pr4*, *pr5* — Five optional integer parameters to be passed to the child program. If any of the parameters *pr1*, *pr2*, *pr3*, *pr4*, or *pr5* are omitted, the remaining parameters all default to 0.

bufr	A variable length buffer containing data to be sent to the child program. The child program can recover the buffer by using the RTE `GETST` subroutine or the RTE string passage `EXEC 14` call. Refer to the RTE manual for usage. NOTE: Any string that is retrieved with `GETST` must be structured so that two leading commas exist in the string. `GETST` discards the information preceding the two commas and returns the string following them.
bufln	The length of *bufr*. If a positive integer, *bufln* indicates the number of words. If a negative integer, *bufln* indicates the number of bytes in *bufr*. If the *bufr* parameter is specified, the last two bytes of data are *bufln*.

Discussion

The program scheduling option (23090) is a Group 4 option. Only one Group 4 option can be specified. No other option can be specified after a Group 4 option.

This option invokes the RTE `EXEC 10` call to immediately schedule a child program for execution without wait. The affect of an `EXEC 9` call, used to immediately schedule a program with wait, can be achieved by the parent program using option 23090 and setting the wait-for-child bit of the *flags* parameter.

There is no provision for the child to pass parameters or return status back to the parent.

Option 23090 is also similar to the RTE `XQ` command, run program without wait.

Note that an error is returned in *result* if an attempt is made to schedule a program which is already running.

ADDOPT Example

The following `AddOpt` example shows the parameters for the immediate no wait program scheduling option. Note that the total length of the *opt* array is the buffer length plus 12 bytes for the other six parameters (*pr1*, *pr2*, *pr3*, *pr4*, *pr5*, and *bufln*).

```
CONST
   MAX_BUFR_LENGTH = 256;
   IMMNOWAIT_OPT_LENGTH = MAX_BUFR_LENGTH + 12; { add 12 bytes to buffer size }
                                                { for parms 1 - 5 and bufLen  }
                                                { to get total length of opt  }
                                                { data                        }

TYPE
   ImmNoWaitOptType = RECORD
      CASE BYTE OF
      0 : (Bytes : RpmOptDataType);
      1 : (ImmNoWaitParm1 : Int16;
           ImmNoWaitParm2 : Int16;
           ImmNoWaitParm3 : Int16;
           ImmNoWaitParm4 : Int16;
           ImmNoWaitParm5 : Int16;
           ImmNoWaitBufr  : PACKED ARRAY [1..MAX_BUFR_LENGTH] OF CHAR;
           ImmNoWaitBufLength : Int16);     { use negative value for bytes }
    END; { ImmNoWaitOptType }

VAR
   ImmNoWaitOpt : ImmNoWaitOptType;

BEGIN
           :
           :
           :
WITH ImmNoWaitOpt DO
   BEGIN
   ImmNoWaitParm1      := 0;
   ImmNoWaitParm2      := 0;
   ImmNoWaitParm3      := 0;
   ImmNoWaitParm4      := 0;
   ImmNoWaitParm5      := 0;
   ImmNoWaitBufr       := 'ImmNoWait Buffer';
   ImmNoWaitBufLength  := -16;
   END;   { WITH ImmNoWaitOpt DO }

ADDOPT(RpmOpt,OptNum,23090,IMMNOWAIT_OPT_LENGTH,ImmNoWaitOpt.Bytes,error);
```

RPMCreate Option 23100—Queue Program Scheduling

RTE-A System Equivalent: `EXEC 24` call (documented in the *RTE-A Programmer's Reference Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23100 to indicate the "Queue Program Scheduling" option.

datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array. Must be one of the following values: 0, 2, 4, 6, 8, 10, 13, or greater than 13 bytes. Any parameter that is not specified defaults to zero. The exact length depends on the parameters specified:

- If the length is 0, all parameters are omitted and take their default value to be zero.
- If the length is 2, only *pr1* is specified.
- If the length is 4, *pr1* and *pr2* are specified.
- If the length is 6, *pr1*, *pr2*, and *pr3* are specified.
- If the length is 8, *pr1*, *pr2*, *pr3*, and *pr4* are specified.
- If the length is 10, *pr1*, *pr2*, *pr3*, *pr4*, and *pr5* are specified.
- If the length is 13 or more, all seven parameters are specified.

data — *Array, by reference*. A variable length array with the following contents:

0 1	2 3	4 5	6 7	8 9	10 11 ... n	n + 1 bytes
pr1	pr2	pr3	pr4	pr5	bufr	bufln

pr1, *pr2*, *pr3*, *pr4*, *pr5* — Five optional integer parameters to be passed to the child program. If any of the parameters *pr1*, *pr2*, *pr3*, *pr4*, or *pr5* are omitted, the remaining parameters all default to 0.

bufr	A variable length buffer containing data to be sent to the child program. The child program can recover the buffer by using the RTE `GETST` subroutine or the RTE string passage `EXEC 14` call. Refer to the RTE manual for usage. NOTE: Any string that is retrieved with `GETST` must be structured so that two leading commas exist in the string. `GETST` discards the information preceding the two commas and returns the string following them.
bufln	The length of *bufr*. If a positive integer, *bufln* indicates the number of words. If a negative integer, *bufln* indicates the number of bytes in *bufr*. If the *bufr* parameter is specified, the last two bytes of data are *bufln*.

Discussion

The queue program scheduling option (23100) is a Group 4 option. Only one Group 4 option can be specified. No other option can be specified after a Group 4 option.

This option invokes the RTE `EXEC 24` call to queue schedule a child program for execution without wait. If a program with the same name is already executing, RPM waits for that program to terminate before scheduling the child program. Otherwise, the child program is scheduled immediately.

The affect of a `EXEC 23` call, used to queue schedule a program with wait, can be achieved by the parent program using option 23100 and setting the wait-for-child bit of the *flags* parameter.

There is no provision for the child to pass parameters or return status back to the parent.

Caution Option 23100 should be used with extreme care and is recommended only for child programs which execute only for a very short duration. If this option is issued for a child program that is currently executing, RPM will suspend and will not be able to process other requests that arrive while waiting for the currently executing child program to terminate. If requests to RPM are frequent enough and RPM suspends for a long time, this may cause many requests to be rejected.

ADDOPT Example

The following `AddOpt` example shows the parameters for the queue program scheduling option. Note that the total length of the *opt* array is the buffer length plus 12 bytes for the other six parameters (*pr1*, *pr2*, *pr3*, *pr4*, *pr5*, and *bufln*).

```
CONST
   MAX_BUFR_LENGTH = 256;
   QUENOWAIT_OPT_LENGTH = MAX_BUFR_LENGTH + 12; { add 12 bytes to buffer size }
                                                { for parms 1 - 5 and bufLen  }
                                                { to get total length of opt  }
                                                { data                        }

TYPE
   QueNoWaitOptType = RECORD
      CASE BYTE OF
      0 : (Bytes : RpmOptDataType);
      1 : (QueNoWaitParm1 : Int16;
           QueNoWaitParm2 : Int16;
           QueNoWaitParm3 : Int16;
           QueNoWaitParm4 : Int16;
           QueNoWaitParm5 : Int16;
           QueNoWaitBufr  : PACKED ARRAY [1..MAX_BUFR_LENGTH] OF CHAR;
           QueNoWaitBufLength : Int16);     { use negative value for bytes }
   END;  { QueNoWaitOptType }

BEGIN
           :
           :
           :

WITH QueNoWaitOpt DO
   BEGIN
  QueNoWaitParm1      := 0;
  QueNoWaitParm2      := 0;
  QueNoWaitParm3      := 0;
  QueNoWaitParm4      := 0;
  QueNoWaitParm5      := 0;
  QueNoWaitBufr       := 'QueNoWait Buffer';
  QueNoWaitBufLength  := -16;
  END;   { WITH QueNoWaitOpt DO }

ADDOPT(RpmOpt,OptNum,23100,QUENOWAIT_OPT_LENGTH,QueNoWaitOpt.Bytes,error);
```

RPMCREATE

RPMCreate Option 23110—Program Scheduling

RTE-A FMP Equivalent: `FmpRunProgram` call (documented in the *RTE-A Programmer's Reference Manual*).

AddOpt Parameters

optioncode — *16-bit integer, by value in Pascal, by reference in FORTRAN*. 23110 to indicate the “Program Scheduling” option.

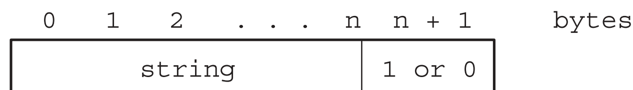
datalength — *16-bit integer, by value in Pascal, by reference in FORTRAN*. The length in bytes of *data* which is to be included in the *opt* array.

data — A variable length character string that contains the runstring. Note that the `XQ` command must be specified at the beginning of the runstring, or RPM will insert it. If the `RU` command is specified at the beginning of the runstring, RPM replaces it with an `XQ`. Also, the program name should be the same as the *progname* parameter of the `RPMCreate` call. An error is returned if this is not the case. This program name will be replaced by RPM by the name of the ID segment under which it is restored. The `IH` option of the RTE `RU` (Run Program) command is not permitted to follow the program name. Cloning can be inhibited by specifying option 23010 (Restore Program Option) beforehand in the *opt* array of the `RPMCreate` call.

The last two bytes of *data* is a 16-bit integer indicating how `FmpRunProgram` is to handle the string parameter. The possible values are as follows:

1 The string is converted to uppercase and each group of one or more consecutive blanks is converted to a comma.

0 The string is not altered.



Discussion

RPM always schedules the child program with the RTE `XQ` command to avoid being suspended while the child program is executing. To achieve the equivalent to an `RU` command, use option 23110 together with the wait-for-child bit of the *flags* parameter of the `RPMCreate` call.

RPMGETSTRING

Allows the child program to retrieve strings passed to it by the parent program.

Syntax

```
RPMGETSTRING (rpmstring,rpmstringlen,result)
```

Parameters

rpmstring — *Packed array of characters (Pascal); word array (FORTRAN), by reference.* A variable length array containing the string passed in the *opt* parameter of the RPMCreate call which scheduled this child program.

rpmstringlen (input/output) — *32-bit non-negative integer, by reference.* On input, *rpmstringlen* is the maximum byte length allowed for the *rpmstring*. On output, *rpmstringlen* indicates the actual length of the returned *rpmstring*. A string longer than what the buffer can accommodate will be truncated. On RTE-A the maximum string length retrieved with RPMGetString is 256 bytes.

If there is no string received in *rpmstring*, an error is returned in *result*.

result — *32-bit non-negative integer, by reference.* The result of the RPMGetString request; zero if no error. If *result* is not zero, an error has occurred. Errors are defined in the *NS-ARPA/1000 Error Message and Recovery Manual*.

Discussion

The RPMGetString call allows a program scheduled by an RPMCreate call to obtain the string passed to it by the parent program in the RPMCreate call. The RPMCreate call uses an *opt* parameter which contains an option code of 20000 followed by the string. Refer to the explanation of RPMCreate previously in this section.

The string obtained in this manner may contain any useful information. For example, it could contain the name of a (call) socket belonging to the parent, along with the name of the node on which the parent is executing. The scheduled child program can look up this socket name in order to acquire a destination descriptor for it. After creating a socket of its own, it can establish a connection to the parent program.

An RPM child program using RPMGetString must be compiled and linked as a CDS program.

If the *opt* parameter of `RPMCreate` contained more than one RPM string, issue as many `RPMGetString` calls as necessary to retrieve all the strings. For example:

Parent program:

```
ADDOPT (Opt, 0, 20000, Length1, RpmString1);
ADDOPT (Opt, 1, 20000, Length2, RpmString2);
ADDOPT (Opt, 2, 20000, Length3, RpmString3);
RPMCREATE (... Opt ...);
```

```
Child program:
```

```
RPMGETSTRING (RpmString1, Length1, Result);
RPMGETSTRING (RpmString2, Length2, Result);
RPMGETSTRING (RpmString3, Length3, Result);
```

RPMKILL

Terminates a specified child program scheduled by an `RPMCreate` call.

Syntax

```
RPMKILL (pd,nodename,nodelen,result)
```

Parameters

pd — *Byte array (Pascal); Word array (FORTRAN), by reference*. An array of 16 bytes containing the program descriptor returned by the `RPMCreate` call.

nodename — *Packed array of characters (Pascal); word array (FORTRAN), by reference*. A variable length array identifying the node on which the child program resides. The syntax of the node name is *node*[.*domain*[.*organization*]], which is further described in "Node Names" of the "Introduction" section and in "Nodename Parameter" of the "Network Interprocess Communication" section of this manual.

Default: You may omit the organization, organization and domain, or all parts of the node name. When organization or organization and domain are omitted, they will default to the local organization and/or domain. If the *nodelen* parameter is set to zero, *nodename* is ignored and the node name defaults to the local node.

nodelen — *32-bit non-negative integer, by value in Pascal, by reference in FORTRAN*. Length in bytes of the *nodename* parameter. If *nodelen* is zero (0), the *nodename* parameter is ignored. In this case, it is assumed that the parent is either terminating a dependent child program that it previously scheduled or the child program is on the local node.

result — *32-bit non-negative integer, by reference*. The result of the `RPMKill` request; zero if no error. If *result* is not zero, an error has occurred. Errors are defined in the *NS-ARPA/1000 Error Message and Recovery Manual*.

Discussion

The `RPMKill` call terminates a program that was scheduled by `RPMCreate`. The child program is specified by its program descriptor, *pd*. Any program on any node may call `RPMKill` to terminate an RPM program, as long as it has the correct program descriptor. `RPMKill` also terminates a child session if both of the following conditions are true:

- The session was programmatically created by RPM.
- There are no more child programs running in that session.

RPM Program Examples

There are two sets of RPM programs that follow, the first parent-child pair is in Pascal/1000 and the second pair is in FORTRAN 77. The Pascal programming examples are in your NS-ARPA/1000 software in `/NS1000/EXAMPLES/RPM1.PAS` (parent program) and in `/NS1000/EXAMPLES/RPM2.PAS` (child program). The FORTRAN 77 examples are in `/NS1000/EXAMPLES/PARENT.FTN` (parent program) and in `/NS1000/EXAMPLES/CHILD.FTN` (child program).

Each parent program uses the `RPMCreate` options to time schedule a child program. The `RPMControl` call is also used to obtain the child program's status. All the data declarations for `RPMCreate` options are included in this example for your convenience. Not all the data declarations were used for this parent program. For your own use, you can delete or comment out the unused data declarations.

Each child program uses `RPMGetString` to retrieve strings passed by the parent program. The child program must be a CDS program.

Pascal/1000 RPM Parent Program

```
$PASCAL '91790-18267 REV.5240 <870715.1002>'
$ STANDARD_LEVEL 'HP1000' $
$ CDS ON $
$ CODE_CONSTANTS OFF $
$ DEBUG on $
{}
{       NAME: RPM1
{     SOURCE: 91790-18267
{      RELOC: NONE
{       PGMR: VH
{}
{}
{ MODIFICATION HISTORY
{
{ Date    Programmer    Description
{ ----    ----------    -------------------------------------------------------
{ 052891  VH            Modified to take nodeName/Logon/Passwd/directory
{                           from the runstring.
{
{------------------------------------------------------------------------------}
{ This program calls RPMCREATE to create an RPM child and uses RPMCREATE      }
{ options to:  set the child's session working directory if specified by user }
{              RP the child                                                    }
{              send RPM data strings to the child                              }
{              time schedule the child                                         }
{ The program also calls RPMCONTROL to report the status of the child.        }
{ In addition, data declarations are included for all the RPMCREATE options.  }
{                                                                              }
{ Usage: rpm1 <nodename> <login> [password] [directory]                        }
{                                                                              }
{------------------------------------------------------------------------------}
```

```
program rpm1;

LABEL 999;

TYPE

{General-purpose types                                              }
   Byte         = 0..255;
   Int16        = -32768..32767;
   StringType   = PACKED ARRAY [1..256] OF CHAR;
   CallNameType = STRING[25];

CONST
   MAX_PASSWD_LENGTH   = 256;
   MAX_LOGIN_LENGTH    = 256;
   MAX_NODENAME_LENGTH = 50;
   MAX_PROGNAME_LENGTH = 64;

   MAX_OPT_BYTES       = 996;          { Max size for RPM protocol message   }
                                       { is 1024 bytes, including 28 bytes   }
                                       { for the RPM message header, so max  }
                                       { option array size is 1024 - 28 = 996}

   MAX_DATA_BUFFER_BYTES = 982;        { Used for RPMControl. 30 bytes for   }
                                       { RPM message header, 12 bytes for    }
                                       { opt array header and overhead, so   }
                                       { max is 1024 - 42 = 982              }

   MAX_BUFR_LENGTH = 256;              { (Bytes) Used when passing buffers   }
                                       { while scheduling. Can be increased, }
                                       { so long as                          }
                                       { total opt array size is <= 996 bytes}

   MAX_RPMSTRING_LENGTH = 256;         { Used when passing strings to child. }
                                       { Can be increased, so long as total  }
                                       { opt array size is <= 996 bytes      }

   OPT_DATA_BYTES = 2;                 { Dummy size for option data byte array}

   NO_ERROR = 0;
   MAX_RUN_STRING = 80;

TYPE
   EnvStringType         = PACKED ARRAY [1..MAX_RUN_STRING] of CHAR;
   NodeNameType          = PACKED ARRAY [1..MAX_NODENAME_LENGTH] OF CHAR;
   ProgramNameType       = PACKED ARRAY [1..MAX_PROGNAME_LENGTH] OF CHAR;
   ProgramDescriptorType = PACKED ARRAY [1..16] OF Byte;
   RpmOptType            = PACKED ARRAY [1..MAX_OPT_BYTES] OF Byte;
   RpmOptDataType        = PACKED ARRAY [1..OPT_DATA_BYTES] OF Byte;
   FlagsType = RECORD
      CASE Byte OF
      0 : (bits : PACKED ARRAY [0..31] OF BOOLEAN);
      1 : (int  : INTEGER);
   END;         { FlagsType }
```

```
   RpmDataBufferType = RECORD
      CASE Byte OF
      0 : (data_bytes : PACKED ARRAY [1..MAX_DATA_BUFFER_BYTES] OF Byte);
      1 : (data_words : PACKED ARRAY [1..MAX_DATA_BUFFER_BYTES DIV 2] OF
Int16);
      END;   { RpmDataBufferType}

   RpmStringType = PACKED ARRAY [1..MAX_RPMSTRING_LENGTH] OF CHAR;

VAR

   screen: Text;
   rlen : int16;
   rstring : EnvStringType;          { the runstring }
   Flags : FlagsType;
   RpmOptCode : Int16;
   RpmOpt : RpmOptType;              { will contain opt entries for RPMCreate}

   OptNum : Int16;

   result : INTEGER;                    { used for RPM calls                 }
   error  : Int16;                      { used for INITOPT and ADDOPT calls  }
   CallName : CallNameType;             { used by error reporting routines   }

{Variables for RPMCreate                                                     }

   NodeNameLength, ProgNameLength, LoginLength, PasswdLength : Int16;
   NodeName : NodeNameType;
   Password, Login : StringType;
   ProgramName : ProgramNameType;
   ProgramDescriptor : ProgramDescriptorType;

{Variables for RPMControl                                                   }

   Wrtlen, Readlen : INTEGER;
   WriteDataBuffer : RpmDataBufferType;
   ReadDataBuffer : RpmDataBufferType;

{-----------------------------------------------------------------------------}
{  RPMCreate Option Codes                                                     }
{-----------------------------------------------------------------------------}

CONST

   RPMOPT_WD = 23000;        {Group 1:  Set Working Directory                 }

   RPMOPT_RP = 23010;        {Group 2:  Restore Program                       }

   RPMOPT_AS = 23020;        {Group 3:  Assign Partition                      }
   RPMOPT_PR = 23030;        {          Set Program Priority                  }
   RPMOPT_WS = 23040;        {          Change Working Set Size               }
   RPMOPT_VS = 23050;        {          Change VMA Space Size                 }
   RPMOPT_CD = 23060;        {          Change CDS Code Size                  }
   RPMOPT_DT = 23070;        {          Change CDS Data Size                  }
   RPMOPT_RPMSTRING = 20000; {          Pass Parameter String                 }
```

```
   RPMOPT_TMLIST    = 23080; {Group 4: Time List Scheduling (EXEC 12)          }
   RPMOPT_IMMNOWAIT = 23090; {         Immediate Schedule W/out Wait (EXEC 10)}
   RPMOPT_QUENOWAIT = 23100; {         Queue Schedule Without Wait (EXEC 24)  }
   RPMOPT_XQ        = 23110; {         XQ (FmpRunProgram)                     }

{------------------------------------------------------------------------------}
{  WD (RPMCreate Option 23000) Declarations                                    }
{  WD option sets session's working directory                                  }
{------------------------------------------------------------------------------}

CONST
   MAX_PATH_LENGTH = 63;

TYPE

   WdOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (WdPath : PACKED ARRAY [1..MAX_PATH_LENGTH] OF CHAR);
   END;  { WdOptType }

VAR
   WdOpt : WdOptType;

{------------------------------------------------------------------------------}
{  RP (RPMCreate Option 23010) Declarations                                    }
{  RP option restores permanent                                                }
{------------------------------------------------------------------------------}

CONST
   MAX_ID_LENGTH = 6;
TYPE

   IDNameType = PACKED ARRAY [1..MAX_ID_LENGTH] OF CHAR;

   RpOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (RpIDName : IDNameType;
          RpClone : CHAR;);
   END;  { RpOptType }

VAR
   RpOpt : RpOptType;

{------------------------------------------------------------------------------}
{  AS (RPMCreate Option 23020) Declarations                                    }
{  AS option assigns program partition                                         }
{------------------------------------------------------------------------------}

TYPE
   AsOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (AsPartition : Int16;
```

```
         AsCodeOrData : CHAR;);
   END;  { AsOptType }

VAR
   AsOpt : AsOptType;

{-------------------------------------------------------------------------------}
{  PR (RPMCreate Option 23030) Declarations                                    }
{  PR option sets program priority                                             }
{-------------------------------------------------------------------------------}

TYPE
   PrOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (PrPriority : Int16); { range is 1..32767 }
   END;  { PrOptType }

VAR
   PrOpt : PrOptType;

{-------------------------------------------------------------------------------}
{  WS (RPMCreate Option 23040) Declarations                                    }
{  WS Option sets working set size                                             }
{-------------------------------------------------------------------------------}

TYPE
   WsOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (WsSize : Int16);  { range is 2..1022 }
   END;  { WsOptType }

VAR
   WsOpt : WsOptType;

{-------------------------------------------------------------------------------}
{  VS (RPMCreate Option 23050) Declarations                                    }
{  VS Option sets VMA size                                                     }
{-------------------------------------------------------------------------------}

TYPE
   VsOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (VsSize : Int16);             { Range for RPM is 32..32767 }
   END;  { VsOptType }

VAR
   VsOpt : VsOptType;

{-------------------------------------------------------------------------------}
{  CD (RPMCreate Option 23060) Declarations                                    }
{  CD Option sets max number of code segments allocated in memory              }
{-------------------------------------------------------------------------------}
```

```
TYPE
   CdOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (CdMaxCodeSegs : Int16);
   END;  { CdOptType }

VAR
   CdOpt : CdOptType;

{------------------------------------------------------------------------------}
{  DT (RPMCreate Option 23070) Declarations                                    }
{  DT Option sets max number of data segments allocated in memory              }
{------------------------------------------------------------------------------}

TYPE
   DtOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (DtMaxDataSegs : Int16);
   END;  { WsOptType }

VAR
   DtOpt : DtOptType;

{------------------------------------------------------------------------------}
{  RPMString (RPMCreate Option 20000) Declarations                             }
{  RPMString allows parent to pass string to child                             }
{------------------------------------------------------------------------------}

TYPE
   RPMStringOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (RPMString : RpmStringType);
   END;  { RPMStringOptType }

VAR
   RPMStringOpt : RPMStringOptType;

{------------------------------------------------------------------------------}
{  TIMELIST SCHEDULING (RPMCreate Option 23080) Declarations                   }
{  TIMELIST option allows time scheduling (EXEC 12)                            }
{------------------------------------------------------------------------------}

TYPE
   TimeListOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (TmListUnits : Int16;
           TmListOften : Int16;
           TmListDelay : Int16);                    { Range is -32768..-1 }
      2 : (TmListAbsUnits : Int16;
           TmListAbsOften : Int16;
           TmListAbsHour  : Int16;
```

```
          TmListAbsMin   : Int16;
          TmListAbsSec   : Int16;
          TmListAbsMsec  : Int16);
   END;  { TimeListOptType }

VAR
   TimeListOpt : TimeListOptType;

CONST
{Constants for TmListUnits }
   CENTISECONDS = 1;
   SECONDS      = 2;
   MINUTES      = 3;
   HOURS        = 4;

{-----------------------------------------------------------------------------}
{  IMMEDIATE SCHEDULING NOWAIT (RPMCreate Option 23090) Declarations          }
{  IMMEDIATE SCHEDULING option allows immediate scheduling nowait (EXEC 10)   }
{-----------------------------------------------------------------------------}

TYPE
   ImmNoWaitOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (ImmNoWaitParm1 : Int16;
           ImmNoWaitParm2 : Int16;
           ImmNoWaitParm3 : Int16;
           ImmNoWaitParm4 : Int16;
           ImmNoWaitParm5 : Int16;
           ImmNoWaitBufr  : PACKED ARRAY [1..MAX_BUFR_LENGTH] OF CHAR;
           ImmNoWaitBufLength : Int16);     { Use negative value for bytes }
   END;  { ImmNoWaitOptType }

VAR
   ImmNoWaitOpt : ImmNoWaitOptType;

{-----------------------------------------------------------------------------}
{  QUEUE SCHEDULING NOWAIT (RPMCreate Option 23100) Declarations              }
{  QUEUE SCHEDULING option allows queue scheduling nowait (EXEC 24)           }
{-----------------------------------------------------------------------------}

CONST
    QUE_NOWAIT_OPT_DATA_LENGTH = MAX_BUFR_LENGTH + 12;
TYPE
   QueNoWaitOptType = RECORD
      CASE BYTE OF
      0 : (Bytes  : RpmOptDataType);
      1 : (QueNoWaitParm1 : Int16;
           QueNoWaitParm2 : Int16;
           QueNoWaitParm3 : Int16;
           QueNoWaitParm4 : Int16;
           QueNoWaitParm5 : Int16;
           QueNoWaitBufr  : PACKED ARRAY [1..MAX_BUFR_LENGTH] OF CHAR;
           QueNoWaitBufLength : Int16);     { Use negative value for bytes }
   END;  { QueNoWaitOptType }
```

```
VAR
   QueNoWaitOpt : QueNoWaitOptType;

{----------------------------------------------------------------------------}
{  XQ (RPMCreate Option 23110) Declarations                                  }
{  XQ option is the equivalent to FmpRunProgram                              }
{----------------------------------------------------------------------------}

CONST
   MAX_RUNSTR_LENGTH = 256;

TYPE
   XQOptType = RECORD
      CASE BYTE OF
      0 : (Bytes   : RpmOptDataType);
      1 : (XQRunstr : PACKED ARRAY [1..MAX_RUNSTR_LENGTH] OF CHAR;
           XQAlter  : Int16);
   END;  { XQOptType }

VAR
   XQOpt : XQOptType;

{----------------------------------------------------------------------------}
{  RPMCONTROL Option Codes                                                   }
{----------------------------------------------------------------------------}

CONST
   RPMOPT_SUSPEND = 20001;
   RPMOPT_RESUME  = 20002;
   RPMOPT_BR      = 23120;
{  RPMOPT_PR      = 23030;   (Already declared as RPMCREATE option code)     }
   RPMOPT_IDINFO  = 23130;

{----------------------------------------------------------------------------}
{  RPM external procedure declarations                                       }
{----------------------------------------------------------------------------}

PROCEDURE RPMCONTROL
   (VAR pd              : ProgramDescriptorType;
    VAR nodename        : NodeNameType;
        nodenamelen     : INTEGER;
        reqcode         : INTEGER;
    VAR wrtdata         : RpmDataBufferType;
        wlen            : INTEGER;
    VAR readdata        : RpmDataBufferType;
    VAR rlen            : INTEGER;
    VAR flags           : FlagsType;
    VAR result          : INTEGER);         EXTERNAL;

PROCEDURE RPMCREATE
   (VAR progname        : ProgramNameType;
        namelen         : INTEGER;
    VAR nodename        : NodeNameType;
        nodenamelen     : INTEGER;
    VAR login           : StringType;
```

```
        loginlen       : INTEGER;
    VAR password       : StringType;
        passwdlen      : INTEGER;
    VAR flags          : FlagsType;
    VAR opt            : RpmOptType;
    VAR pd             : ProgramDescriptorType;
    VAR result         : INTEGER);        EXTERNAL;

PROCEDURE RPMGETSTRING
   (VAR rpmstring      : RpmStringType;
    VAR rpmstrlen      : INTEGER;
    VAR result         : INTEGER);         EXTERNAL;

PROCEDURE RPMKILL
   (VAR pd            : ProgramDescriptorType;
    VAR nodename      : NodeNameType;
        nodenamelen   : INTEGER;
    VAR result        : INTEGER);          EXTERNAL;

PROCEDURE INITOPT
   (VAR opt              : RpmOptType;
        optnumarguments : Int16;
    VAR error            : Int16);  EXTERNAL;

PROCEDURE ADDOPT
   (VAR opt              : RpmOptType;
        argnum           : Int16;
        optioncode       : Int16;
        datalength       : Int16;
    VAR data             : RpmOptDataType;
    VAR error            : Int16);  EXTERNAL;

FUNCTION GetRunString $ALIAS 'Pas.Parameters'$
   (    pos    : Int16;
    VAR envstr : EnvStringType;
        len    : Int16
   ): int16; External;

{-------------------------------------------------------------------------------}
{ RPM_ERROR procedure--used to report RPM call errors                           }
{-------------------------------------------------------------------------------}
PROCEDURE RPM_ERROR
   (VAR CallName   : CallNameType;
    VAR ResultCode : INTEGER);

BEGIN
   writeln(screen,'rpm1: an error occurred in your ',CallName,' call.');
   writeln(screen,'rpm1: the error code returned was: ',ResultCode);
   GOTO 999;
END;  { RPM_ERROR }
```

```
{-----------------------------------------------------------------------------}
{ OPT_ERROR procedure--used to report ADDOPT and INITOPT call errors          }
{-----------------------------------------------------------------------------}
PROCEDURE OPT_ERROR
   (VAR CallName  : CallNameType;
    VAR ErrorCode : Int16);

BEGIN
   writeln(screen,'rpm1: an error occurred in your ',CallName,' call.');
   writeln(screen,'rpm1: the error code returned was: ',ErrorCode);
   GOTO 999;
END;  { OPT_ERROR    }

{-----------------------------------------------------------------------------}
{                  ReportUsage                                                }
{-----------------------------------------------------------------------------}
Procedure ReportUsage;
BEGIN {* report usage *}
   writeln(screen,'Usage: rpm1 <nodeName> <loggin> [passwd] [directory]');
END;  {* report usage *}

{-----------------------------------------------------------------------------}
{ Main                                                                        }
{ Calls ADDOPT to set the child's session working directory, RP the child,    }
{ send 2 RPM data strings to the child and time schedule the child.           }
{ Calls RPMCREATE to create the child, and RPMCONTROL to get child's status.  }
{-----------------------------------------------------------------------------}
BEGIN {* main *}
{}
{ Get the runstring.
{}
rewrite(screen,'1');
rlen := GetRunString(1,rstring,MAX_RUN_STRING);
IF (rlen <= 0) THEN
   BEGIN {* no node name *}
      ReportUsage;
      GOTO 999;
   END;  {* no node name *}
strmove(rlen,rstring,1,NodeName,1);
NodeNameLength := rlen;

rlen := GetRunString(2,rstring,MAX_RUN_STRING);
IF (rlen <= 0) THEN
   BEGIN {* no loggin *}
      ReportUsage;
      GOTO 999;
   END;  {* no loggin *}
strmove(rlen,rstring,1,Login,1);
LoginLength := rlen;

{}
{ Allow if user does not have password - optional parameter.
{}
rlen := GetRunString(3,rstring,MAX_RUN_STRING);
IF (rlen > 0) THEN
```

```
   BEGIN {* password specified *}
      strmove(rlen,rstring,1,Password,1);
   END;  {* password specified *}
IF (rlen < 0) THEN
   PasswdLength := 0
ELSE
   PasswdLength := rlen;

{}
{ Get working directory if any - optional parameter.
{}
rlen := GetRunString(4,rstring,MAX_RUN_STRING);
IF (rlen > 0) THEN
   BEGIN {* directory specified. *}
      strmove(rlen,rstring,1,WdOpt.WdPath,1);
   END;  {* directory specified. *}
IF (rlen < 0) THEN
    rlen := 0;

Flags.int := 0;      { Flags not used, so clear array }
OptNum := 0;

{  Initialize RpmOpt array . }
IF (rlen > 0) THEN
   BEGIN {* need to add working directory also *}
      INITOPT( RpmOpt,5,error );
      IF error <> NO_ERROR THEN
         BEGIN
            CallName := 'INITOPT';
            OPT_ERROR( CallName,error );
         END;
       {}
       { Set child's session working directory
       {}
       ADDOPT( RpmOpt,OptNum,RPMOPT_WD,rlen,WdOpt.Bytes,error );
       OptNum := OptNum +1;
       IF error <> NO_ERROR THEN
          BEGIN
             CallName := 'ADDOPT--WD Opt';
             OPT_ERROR(CallName,error);
           END;
    END    {* need to add work directory also *}
ELSE
    BEGIN  {* only 4 options to add *}
       INITOPT( RpmOpt,4,error );
       IF error <> NO_ERROR THEN
          BEGIN
             CallName := 'INITOPT';
             OPT_ERROR( CallName,error );
          END;
    END;  {* only 4 options to add *}
{}
{  RP child with the ID 'CH1'
{}
```

```
RPOpt.RpIDName := 'CH1 ';
ADDOPT( RpmOpt,OptNum,RPMOPT_RP,6,RpOpt.Bytes,error );
OptNum := OptNum +1;
IF error <> NO_ERROR THEN
   BEGIN
      CallName := 'ADDOPT--RP Opt';
      OPT_ERROR(CallName,error);
   END;

{}
{ Send two data strings
{}
RpmStringOpt.RpmString:= 'String 1';
ADDOPT( RpmOpt,OptNum,RPMOPT_RPMSTRING,8,RpmStringOpt.Bytes,error );
OptNum := OptNum +1;
IF error <> NO_ERROR THEN
   BEGIN
      CallName := 'ADDOPT--RPM String Opt';
      OPT_ERROR(CallName,error);
   END;

RpmStringOpt.RpmString:= 'String 2';
ADDOPT( RpmOpt,OptNum,RPMOPT_RPMSTRING,8,RpmStringOpt.Bytes,error );
OptNum := OptNum +1;
IF error <> NO_ERROR THEN
   BEGIN
      CallName := 'ADDOPT--RPM String Opt';
      OPT_ERROR(CallName,error);
   END;

{}
{ Time schedule the child to run after 5 seconds.
{}
WITH TimeListOpt DO
   BEGIN
      TmListUnits := 2;    {* indicate unit in second *}
      TmListOften := 0;    {* execute once            *}
      TmListDelay := -5;   {* delay 5 seconds before executing *}
   END;     { WITH TimeListOpt }

ADDOPT( RpmOpt,OptNum,RPMOPT_TMLIST,6,TimeListOpt.Bytes,error );
OptNum := OptNum +1;
   IF error <> NO_ERROR THEN
      BEGIN
         CallName := 'ADDOPT--Time List Opt';
         OPT_ERROR(CallName,error);
      END;
write(screen,'rpm1: RPMCreate the child program...');
{ Set program name, node name, login and password for RPMCREATE }
ProgramName := 'RPM2';
ProgNameLength := 4;
RPMCREATE( ProgramName,ProgNameLength,NodeName,NodeNameLength,Login,
           LoginLength,Password,PasswdLength,Flags,
           RpmOpt,ProgramDescriptor,result );
```

```
   IF result <> NO_ERROR THEN
      BEGIN
         CallName := 'RPMCreate';
         RPM_ERROR( CallName,result );
      END;
writeln(screen,'[ok].');
{ Use RPMControl to check status }

Readlen := 2;                  { expect 2 bytes of data back for child status }

RPMCONTROL( ProgramDescriptor,NodeName,NodeNameLength,RPMOPT_IDINFO,
            WriteDataBuffer,Wrtlen,ReadDataBuffer,Readlen,Flags,result );

   IF (result <> NO_ERROR) THEN
      BEGIN  {* error *}
         CallName := 'RPMControl';
         RPM_ERROR(CallName,result );
      END    {* error *}
   ELSE
      BEGIN {* ok from child *}
         writeln (screen,'rpm1: status of the child program is: ',
                                 ReadDataBuffer.data_words [1]:3);
      END;  {* ok from child *}

999 :;

END. {* main *}
```

Pascal/1000 RPM Child Program

```
$ STANDARD_LEVEL 'HP1000' $
$ CDS ON $
$ CODE_CONSTANTS OFF $
{-----------------------------------------------------------------------------}
{ This RPM child program calls RPMGETSTRING to retrieve strings passed from   }
{ its parent.                                                                 }
{ It must be a CDS program.                                                   }
{-----------------------------------------------------------------------------}
program rpm2 (input,output);

LABEL 999;

TYPE

{General-purpose types                                                       }
   Byte = 0..255;
   Int16 = -32768..32767;
   StringType = PACKED ARRAY [1..256] OF CHAR;
   CallNameType = STRING[25];

CONST

   MAX_RPMSTRING_LENGTH = 256;           {Can be increased to meet application}
                                         {needs.  See upper bound note in     }
                                         {parent program.                     }

NO_ERROR = 0;

TYPE

   RpmStringType = PACKED ARRAY [1..MAX_RPMSTRING_LENGTH] OF CHAR;

VAR

   result : INTEGER;                      {used for error return              }
   CallName : CallNameType;

{-----------------------------------------------------------------------------}
{ RPM external procedure declaration                                         }
{-----------------------------------------------------------------------------}

PROCEDURE RPMGETSTRING
   (VAR rpmstring : RpmStringType;
    VAR rpmstrlen : INTEGER;
    VAR result   : INTEGER);         EXTERNAL;

{-----------------------------------------------------------------------------}
{ RPM_ERROR procedure--used to report RPM errors.                             }
{-----------------------------------------------------------------------------}
```

```
PROCEDURE RPM_ERROR
   (VAR CallName  : CallNameType;
    VAR ResultCode : INTEGER);

BEGIN
   writeln( 'An error occurred in your ',CallName,' call.');
   writeln( 'The error code returned was:  ',ResultCode);
   GOTO 999;
END;  { RPM_ERROR }

{-------------------------------------------------------------------------------}
{ GET_AND_PRINT_STRING procedure--calls RPMGETSTRING to retrieve RPM string    }
{ from parent, check for RPM error and print string.                            }
{-------------------------------------------------------------------------------}

PROCEDURE GET_AND_PRINT_STRING;
VAR
   RpmString : RpmStringType;
   RpmStrLen : INTEGER;
   index : Int16;
BEGIN  {GET_AND_PRINT_STRING}

RPMGETSTRING ( RpmString,RpmStrLen,result);
   IF result <> NO_ERROR THEN
      BEGIN
         CallName := 'RPMGetString';
         RPM_ERROR( CallName,result );
      END
   ELSE                                    { no error, so print RPM string  }
      BEGIN
         writeln('RPM String:');           { will be written to system      }
         FOR index := 1 TO RpmStrLen DO    { console since child runs under }
            write(RpmString[index]);       { programmatic session           }
         writeln
      END;
END;   {GET_AND_PRINT_STRING}

{-------------------------------------------------------------------------------}
{ Main                                                                          }
{-------------------------------------------------------------------------------}

BEGIN   {Main}

   GET_AND_PRINT_STRING;
   GET_AND_PRINT_STRING;

999 : writeln;

END.
```

FORTRAN 77 RPM Parent Program

```
FTN77,L,S
$cds on
$files(1,1)

      PROGRAM parent(4,99),91790-18270 REV.5240 <880901.1017>
C
C       NAME: PARENT
C     SOURCE: 91790-18270
C      RELOC: NONE
C       PGMR: KB, VH
C
C      This program is the peer process to child. It uses RPM calls to
C      schedule the child program at the same or a remote node. The parent
C      program calls RPMCREATE with three options:
C               1. set the child's working directory
C               2. RP the child
C               3. send RPM data strings to the child (2 strings are sent)
C      The program also calls RPMCONTROL to report the status of the child,
C      and RPMKILL to terminate the child.
C
C                     ****************************************
C                     *               IMPORTANT!             *
C                     *                                      *
C                     * Since this program uses RPM calls,   *
C                     * it will be necessary to increase the *
C                     * stack size upon loading. A stack     *
C                     * size of 6000 words is sufficient for *
C                     * this program to execute. See the     *
C                     * Link User's Manual for more info.    *
C                     * about changing stack size.           *
C                     ****************************************
C
      IMPLICIT None
C
C    VARIABLE DECLARATIONS:
C      The variable declarations for each RPM call are separated for clarity.
C      However, declarations for variables which have been declared for
C      previous calls are commented out. The purpose of this is to
C      demonstrate the complete set of declarations needed for each RPM call.
C
C      Two exclamation points (!!) next to a variable name indicated that
C      its value may be changed by the call.
C
C    INITOPT:
      INTEGER*2 options(498) !! INITOPT initializes the options parameter
      INTEGER*2 optnumargs    ! so that 'optnumargs' arguments can be added.
      INTEGER*2 error        !! This is the same call used with NetIPC,
                              ! however, if your program uses RPM and NetIPC
                              ! calls, you should initialize SEPARATE options
                              ! variables.
C
```

```
C   ADDOPT:
C     INTEGER*2 options(498) !! ADDOPT will be used to add the four options
      INTEGER*2 argnum        ! used by RPMCREATE. We have set the length of
      INTEGER*2 optioncode    ! 'options' to 996 bytes: maximum size for an
      INTEGER*2 dlength1      ! RPM message is 1024 and the header is 28
      INTEGER*2 dlength2      ! bytes. Thus 1024 - 28 = 996.
      INTEGER*2 dlength3      !
      INTEGER*2 dlength4      ! We declare a separate 'data' and 'dlength'
      INTEGER*2 data1(31)     ! variable for each ADDOPT we plan to use
      INTEGER*2 data2(10)     ! since 'data' values are character strings
      INTEGER*2 data3(25)     ! of varying lengths. 'Data' must be typed
      INTEGER*2 data4(25)     ! INTEGER, and we must use DATA statements
C     INTEGER*2 error         ! to assign character strings to INTEGERs.
C
C   RPMCREATE:
      INTEGER*2 progname(32)  ! RPMCREATE schedules a program at the node
      INTEGER*4 namelen       ! specified by nodename. If the node is not
      INTEGER*2 nodename(25)  ! the local node, but loginlen is zero, an
      INTEGER*4 nodelen       ! error will be returned by RPMCREATE. If
      INTEGER*2 login(8)      ! there is no password for a login, then
      INTEGER*4 loginlen      ! passwdlen must be zero or RPMCREATE will
      INTEGER*2 passwd(8)     ! return an error. RPMCREATE will process the
      INTEGER*4 passwdlen     ! options specified in 'options' then schedule
      INTEGER*4 flags        !! the child program to run as a dependent
C     INTEGER*2 options(498)  ! (default) program.
      INTEGER*4 pd(4)        !!
      INTEGER*4 resultcode   !!
C
C   RPMCONTROL:
C     INTEGER*4 pd(4)         ! RPMCONTROL will be used to request status
C     INTEGER*2 nodename(25)  ! information about the child program. The
C     INTEGER*2 namelen       ! read and write data buffers have been set
      INTEGER*4 reqcode       ! 982 bytes because the maximum message size
      INTEGER*2 wrtdata(491)  ! is 1024 bytes, the RPM header is 30 bytes,
      INTEGER*4 wrtlen        ! and the opt array header is 12 bytes long.
      INTEGER*2 readdata(491)!! Thus 1024 - (30 + 12) = 982.  However, we
      INTEGER*4 readlen      !! will only use RPMCONTROL to request status
C     INTEGER*4 flags        !! of child, thus, we don't need 'wrtdata' and
C     INTEGER*4 resultcode   !! we only need 16 bits of 'readdata'.
C                             !
C                             ! RPMCONTROL status codes are the same as
C                             ! RTE-A's IDINFO. Refer to RTE-A User/Pgmr
C                             ! Reference Manual for list of codes.
C
C   RPMKILL:
C     INTEGER*4 pd(4)         ! RPMKILL terminates the execution of the child
C     INTEGER*2 nodename(25)  ! program referenced by 'pd' at the node
C     INTEGER*4 nodelen       ! specified by nodename.
C     INTEGER*4 resultcode    !

C   FOR RHPAR to get the runstring
      INTEGER*2 RHPAR
      INTEGER*2 len
```

```
C   FOR ERROR ROUTINES:
      CHARACTER*40 this_call
      INTEGER*2 loopcount,tmp

      DATA progname/'child'/
      DATA namelen/5/
      DATA data2/'CHILD '/
      DATA dlength2/6/
      DATA data3/'This is the first string passed to child.'/
      DATA dlength3/41/
      DATA data4/'This is the second string passed to child.'/
      DATA dlength4/42/

C  Get name of node on which to schedule remote process:
      nodelen = RHPAR(1,nodename,50)
      IF (nodelen .EQ. 0) THEN
         CALL ReportUsage
      ENDIF

C  Get login:
      loginlen = RHPAR(2,login,16)
      IF (loginlen .EQ. 0) THEN
         CALL ReportUsage
      ENDIF

C  Get Password: - optional parameter
      passwdlen = RHPAR(3,passwd,16)

C  Get directory - optional parameter
      dlength1 = RHPAR(4,data1,62)

C  Call INITOPT to initialize the 'options' parameter:
      IF (dlength1 .EQ. 0) THEN
         optnumargs = 3
      ELSE
         optnumargs = 4
      ENDIF

      error = 0                   ! Reset error return code
      this_call = 'INITOPT'
      CALL INITOPT(options,optnumargs,error)
      IF (error .NE. 0) CALL OPT_ERROR(error,this_call)

C  Call ADDOPT to set the working directory (option 23000):
      argnum = -1                 ! First 'options' argument
      IF (dlength1 .NE. 0) THEN
        argnum = argnum + 1
        optioncode = 23000         ! Set working directory
        error = 0
        this_call = 'ADDOPT: working directory'
        CALL ADDOPT(options,argnum,optioncode,dlength1,data1,error)
        IF (error .NE. 0) CALL OPT_ERROR(error,this_call)
       ENDIF
```

```
C  Call ADDOPT to RP the child program (option 23010):

      argnum = argnum + 1         ! Second 'options' argument
      optioncode = 23010          ! RP program
      error = 0
      this_call = 'ADDOPT: RPed child program'
      CALL ADDOPT(options,argnum,optioncode,dlength2,data2,error)
      IF (error .NE. 0) CALL OPT_ERROR(error,this_call)

C  Call ADDOPT to add a string to pass to the child program:

      argnum = argnum + 1         ! Third 'options' argument
      optioncode = 20000          ! Pass string
      error = 0
      this_call = 'ADDOPT: add string1'
      CALL ADDOPT(options,argnum,optioncode,dlength3,data3,error)
      IF (error .NE. 0) CALL OPT_ERROR(error,this_call)

C  Call ADDOPT to pass another string to the child program (opt. 20000):
      argnum = argnum + 1         ! Fourth 'options' argument
      optioncode = 20000          ! Pass string
      error = 0
      this_call = 'ADDOPT: add string2'
      CALL ADDOPT(options,argnum,optioncode,dlength4,data4,error)
      IF (error .NE. 0) CALL OPT_ERROR(error,this_call)

C  Now we have the options variable prepared with the four options which
C  we wish to use. Note that we did not use an option from Group 4
C  (scheduling options). The default will be that the program is scheduled
C  immediately by the parent program.

C  Call RPMCREATE to schedule the child at the remote node. Note that
C  here we will not set any flags. This means that the following defaults
C  are used:
C          flags[1]=0  the parent program does not wait for the child.
C          flags[30]=0 the child will not share its session with another
C                      RPM program.
C          flags[31]=0 the child will be an INDEPENDENT child.
C  See the NS/1000 User/Programmer Reference Manual, section 9, for
C  further information.

      flags = 0
      this_call = 'RPMCREATE '
      CALL RPMCREATE(progname,namelen,nodename,nodelen,login,loginlen,
     >               passwd,passwdlen,flags,options,pd,resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,this_call)
      WRITE(1,'("Child program scheduled at node ",48A2)') nodename

C  Call RPMCONTROL to get the child program's status. A request code of
C  23130 requests the program's status. The status is returned as a 16 bit
C  integer to readdata:

      reqcode = 23130       ! Code for 'Request child status info'.
      wrtlen = 0            ! Must be set to zero for this request.
      readlen = 2           ! Must be set to two for status request.
```

```
      flags = 0             ! Not used, but must be cleared.
      resultcode = 0        ! Reset the result code to zero.
      this_call = 'RPMCONTROL '
      CALL RPMCONTROL(pd,nodename,nodelen,reqcode,wrtdata,wrtlen,
     >                readdata,readlen,flags,resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,this_call)
      WRITE(1,'("Status of the child program is: ",I8)') readdata(1)

C  Call RPMKILL to terminate the child program:
C  allow the child to execute before killing - wait for 4 seconds.
      WRITE(1,'("Waiting for child to finish executing (4 secs)...")')
      CALL EXEC (12,0,2,0,-4)
      CALL RPMKILL(pd,nodename,nodelen,resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,this_call)

      WRITE(1,'("Child killed. Parent completed successfully.")')
      END  ! of main program

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                         C
C  SUBROUTINE OPT_ERROR: called when ADDOPT or INITOPT returns with an    C
C                        error code .NE. 0                                C
C                                                                         C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE OPT_ERROR (code, which_call)
      INTEGER*2 code           ! error code returned from call
      CHARACTER*40 which_call  ! call which produced error

      WRITE(1,'("ERROR RETURNED FROM ",A11)') which_call
      WRITE(1,'("THE ERROR CODE WAS ",I4)') code

      STOP
      END ! of OPT_ERROR

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                        C
C  SUBROUTINE RESULT_ERROR: called when an RPM call returns with a      C
C                           resultcode .NE. 0                            C
C                                                                        C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE RESULT_ERROR (code, which_call)
      INTEGER*4 code              ! error code returned from call
      CHARACTER*11 which_call     ! call which produced error

      WRITE(1,'("ERROR RETURNED FROM ",A11)') which_call
      WRITE(1,'("THE ERROR CODE WAS ",I8)') code

      STOP
      END ! of RESULT_ERROR
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                         C
C  SUBROUTINE ReportUsage: called to report usage.                        C
C                                                                         C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE ReportUsage

      write(1,'("Usage: parent <node> <login> [passwd] [directory]")')

      STOP END
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

FORTRAN 77 RPM Child Program

```
FTN77,L,S
$cds on
$files(1,1)

      PROGRAM child(4,99),91790-18269 REV.5010 <880901.1018>
C
C       NAME: CHILD
C     SOURCE: 91790-18269
C      RELOC: NONE
C       PGMR: KB
C
C      This program is the peer process to parent.  It makes two calls to
C      RPMGETSTRING to receive the two strings sent by its parent.  The
C      child program then enters an endless loop, waiting to be killed by
C      the parent.  The endless loop is present simply to illustrate that
C      the parent's RPMKILL call actually terminates the execution of the
C      child.  The WRITE statements in this program send output to the
C      standard output device ('*'), which is most cases is the system
C      printer.  The WRITE statements may be changed to specify a specific LU.
C
C                    ****************************************
C                    *               IMPORTANT!             *
C                    *                                      *
C                    * Since this program uses RPM calls,   *
C                    * it will be necessary to increase the *
C                    * stack size upon loading.  A stack    *
C                    * size of 4000 words is sufficient for *
C                    * this program to execute.  See the    *
C                    * Link User's Manual for more info.    *
C                    * about changing stack size.           *
C                    ****************************************
C
      IMPLICIT None
C
C  RPMGETSTRING:
       INTEGER*2 rpmstring(40)     ! RPMGETSTRING retrieves ONE string
       INTEGER*4 rpmstringlen      ! passed to it be the parent's RPMCREATE
       INTEGER*4 resultcode        ! call.

       CHARACTER*12 this_call

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                      C
C   BEGIN MAIN PROGRAM:                                                C
C                                                                      C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      resultcode = 0               ! Reset the result code.
      rpmstringlen = 80            ! Receive 80 bytes.
      this_call = 'RPMGETSTRING'
      CALL RPMGETSTRING(rpmstring,rpmstringlen,resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,this_call)
```

```
      WRITE(*,'("RECEIVED STRING #1 FROM PARENT:")')
      WRITE(*,'(40A2)') rpmstring

      resultcode = 0              ! Reset the result code.
      rpmstringlen = 80           ! Receive 80 bytes.
      this_call = 'RPMGETSTRING'
      CALL RPMGETSTRING(rpmstring,rpmstringlen,resultcode)
      IF (resultcode .NE. 0) CALL RESULT_ERROR(resultcode,this_call)

      WRITE(*,'("RECEIVED STRING #2 FROM PARENT:")')
      WRITE(*,'(40A2)') rpmstring

99    GO TO 99            ! Endless loop.
      END

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                     C
C    SUBROUTINE RESULT_ERROR:                                         C
C                                                                     C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

      SUBROUTINE RESULT_ERROR (code, which_call)

      INTEGER*4 code          ! error code returned from call
      CHARACTER*12 which_call ! call which produced error

      WRITE(*,'("ERROR CODE RETURNED FROM ",A12)') which_call
      WRITE(*,'("THE ERROR CODE WAS ",I4)') code

      STOP
      END ! of RESULT_ERROR
```

A

NFT-FTP Comparison

The following paragraphs compare FTP and NFT.

System Type

FTP: Can only copy files between HP and non-HP systems that support ARPA FTP protocol. Can copy files from the local node to a remote node and from a remote node to the local node.

NFT: Can only copy files between NS systems. These systems may be HP 1000s, HP 3000s, HP 9000s, or PCs. Can copy files from the local node to a remote node, from a remote node to the local node, and from a remote node to another remote node.

Remote Logons

FTP: Can log on to specific accounts at ARPA and NS-ARPA nodes. A valid login name and password is needed to log onto HP 9000 systems.

NFT: Can log on to specific accounts at all NS nodes.

File Type

FTP: Can copy FMGR files as well as files in the hierarchical file system. Can only copy ASCII and binary files. The file type of the target file defaults to the type of the source file. This can be overridden by specifying a different file type in the target file descriptor for the target file. Unpredictable results will occur if binary file types are used for ASCII files and vice versa.

NFT: Can copy FMGR files as well as files in the hierarchical file system. The file type of the target file defaults to the type of the source file. This can be overridden by specifying a different file type in the file descriptor for the target file, or by specifying certain Interchange Format options.

File Specification

FTP: The source and target file specifications must both be files. File masks can be used in both file specifications.

NFT: The source and target file specifications must both be files. File masks can be used in both file specifications.

File Size

FTP: The size of the target file defaults to the size of the source file. This can be overridden by specifying a different file size in the target file descriptor. For binary files, if the destination file is larger than the source file, the destination file is padded with null characters.

NFT: The size of the target file defaults to the size of the source file. This can be overridden by specifying a different file size in the file descriptor for the target file or by specifying an Interchange Format option.

Record Length

FTP: The record size limit for files is determined by the operating system. The default ASCII file record size is 256 words, and the default binary file record size is 128 words.

NFT: If Interchange Format is used, the source file must not have records longer than 2200 words. If Transparent Format is used, there is no record size limit.

B

Porting NetIPC Programs

Overview

Network Interprocess Communication (NetIPC) is a service that enables processes on the same or different nodes to communicate using a series of programmatic calls. NetIPC is provided as part of the NS-ARPA/1000 product for the HP 1000 A-Series and the LAN/9000 product for the HP 9000.

The purpose of this appendix is to summarize differences and to provide information to help you successfully port NetIPC programs between NS-ARPA/1000 and LAN/9000 systems.

This appendix does *not* compare the programming language implementations at the different systems. For this information, you should refer to the appropriate language reference manuals.

When you are porting NetIPC programs, the following strategy may help:

1. Make sure that the NetIPC programs are executing correctly between homogeneous systems. That is, the programs should work between HP 1000 systems first.

2. Port the programs using the language reference manuals. Check carefully for compiler differences such as data types and lengths.

3. Then check the differences between NetIPC calls documented in this appendix. Check all the parameters; some are not implemented or have different values.

4. If your ported programs still do not work, consider both programming language and NetIPC differences.

NS-ARPA/1000 and LAN/9000

This section describes the differences between the NS-ARPA/1000 and LAN/9000 NetIPC implementations.

Path Report and Destination Descriptors

In NS-ARPA/1000 NetIPC, the descriptor returned by the socket registry software is called a *path report descriptor*; in LAN/9000, this descriptor is called a *destination descriptor*. Although path report descriptors and destination descriptors have slightly different meanings, their function is the same: both contain addressing information that is used by a NetIPC process to direct requests to a certain call socket at a certain node.

Socket Ownership

A LAN/9000 NetIPC process may own a maximum of 60 descriptors. This limit includes call socket, VC socket, and destination descriptors as well as HP-UX file descriptors and NetIPC and/or file descriptors inherited or otherwise opened by the process.

A NS-ARPA/1000 NetIPC process may own a maximum of 32 socket descriptors. This limit includes call socket, VC socket, and path report descriptors.

A LAN/9000 NetIPC process creates a call socket by calling `IPCCreate`; it creates a VC socket by calling `IPCConnect` or `IPCRecvCn`.

A NS-ARPA/1000 NetIPC process creates a call socket by calling `IPCCreate` or `IPCGet`; it creates a VC socket by calling `IPCConnect` or `IPCRecvCn`. An NS-ARPA/1000 NetIPC process may also gain access to a socket by calling `IPCGive`. Sockets are given away with the `IPCGive` call.

The `IPCGive` and `IPCGet` calls are not part of the LAN/9000 NetIPC implementation. Instead, LAN/9000 processes can also acquire access to sockets owned by other NetIPC processes by utilizing socket "sharing." On HP 9000 systems, NetIPC socket descriptors (call socket, VC socket, and destination), like HP-UX file descriptors, are copied to the "child" process when a process forks. As a result, more than one process can own a descriptor for the same socket. Programmers are responsible for regulating the use of shared sockets on LAN/9000 systems.

Socket Shut Down

The `IPCShutDown` call is used in both NS-ARPA/1000 and LAN/9000 NetIPC to release a descriptor and any resources associated with it. The shut down procedure for both NS-ARPA/1000 and LAN/9000 processes is identical with the following exception: the operation of the LAN/9000 implementation of `IPCShutDown` is affected by socket sharing.

When a LAN/9000 NetIPC process "shuts down" a VC socket descriptor that is shared by other processes, the descriptors owned by the other processes are not affected. The `IPCShutDown` call does not operate on the VC socket referred to by a VC socket descriptor unless the descriptor is

the *last* descriptor for that socket. A VC socket is destroyed along with its VC socket descriptor *only when the descriptor being released is the last descriptor for that socket*.

When shutting down a shared call socket descriptor, the call socket referred to by the descriptor is destroyed along with the descriptor and names associated with the descriptor *only if the descriptor being released is the last descriptor for that socket*. If another process, or processes, have descriptors for the same socket, these duplicate descriptors, and any names associated with the descriptors, are not affected.

When shutting down a shared destination descriptor, the addressing information stored in conjunction with the descriptor is destroyed along with the descriptor *only if the descriptor being released is the sole descriptor for that information*. If another process, or processes, have descriptors for the same information, these duplicate descriptors, and any names associated with the descriptors, are not affected.

Signals

Unlike NS-ARPA/1000 NetIPC calls, LAN/9000 NetIPC calls that would normally block may be interrupted by HP-UX signals. NetIPC calls that are interrupted by signals are optionally restartable. When a call is restarted after a signal, any timeouts (including the synchronous timeout) will be reset. As a result, signals that continuously interrupt/restart a NetIPC call at an interval shorter than the socket timeout will effectively void the timeout. Signals are explained in detail in the *HP-UX Reference Manual*.

TCP Checksum

The NS-ARPA/1000 `IPCConnect` and `IPCRecvCn` calls include a "checksumming" bit in their *`flags`* parameters. When set, this bit causes TCP to enable checksumming.

Unlike NS-ARPA/1000 NetIPC, the LAN/9000 `IPCConnect` and `IPCRecvCn` calls do not include "checksumming" bits. When a LAN/9000 NetIPC process calls `IPCConnect` or `IPCRecvCn`, TCP checksumming is *automatically enabled*.

TCP checksumming will *always* be performed if one or both NetIPC processes are LAN/9000 processes. If both processes are NS-ARPA/1000 NetIPC processes, TCP checksumming will be performed only if one or both processes call `IPCConnect` or `IPCRecvCn` with the "checksumming" bit set.

Remote Process Scheduling

NetIPC itself does not include a call to schedule a remote process. The method used to schedule a remote NetIPC process depends on the types of systems involved. For example, an NS-ARPA/1000 NetIPC process written to schedule an NS-ARPA/1000 peer process must be modified to utilize another scheduling method when it is ported to an LAN/9000 system.

Remote NS-ARPA/1000 Process

In order to schedule a remote NS-ARPA/1000 NetIPC process from an NS-ARPA/1000 node, you can use one of the following methods: the Remote Process Management (RPM) call `RPMCreate`, the Program-to-Program communication (PTOP) `POPEN` call, one of the DEXEC scheduling calls, the REMAT `QU` command, or the TELNET virtual terminal service.

You cannot use any of these services to schedule a remote NS-ARPA/1000 process from a LAN/9000 node because these services are only NS-ARPA/1000 services. The "Process Scheduling" section in the"Network Interprocess Communication" chapter earlier in this manual describes ways to schedule an NS-ARPA/1000 NetIPC process from a LAN/9000 node.

Remote LAN/9000 Process

Remote LAN/9000 processes can be manually started or can be scheduled by user-written daemons that are started at system start up. The "Process Scheduling" section in the"Network Interprocess Communication" chapter earlier in this manual describes ways to schedule a LAN/9000 NetIPC process from an NS-ARPA/1000 node.

Case Sensitivity

Because the HP-UX operating system is case-sensitive, LAN/9000 NetIPC call names must be typed using lower case characters. For example, the NetIPC call `IPCConnect` must be typed as `ipcconnect` on LAN/9000 systems.

NS-ARPA/1000 NetIPC call names are not case sensitive and may be typed using lower case or upper case characters, or a combination of both upper and lower case characters.

NetIPC Calls

For the purposes of the following discussion, the NS-ARPA/1000 and LAN/9000 NetIPC calls are divided into four categories:

- Calls that are *unique to NS-ARPA/1000 NetIPC*.
- Calls that are *unique to LAN/9000 NetIPC*.
- Calls that are *common* to both NS-ARPA/1000 and LAN/9000 NetIPC *and are implemented identically* on each system.
- Calls that are *common* to both NS-ARPA/1000 and LAN/9000 NetIPC *but are implemented differently* on each system.

Unique NetIPC Calls

The following calls are provided as part of the NS-ARPA/1000 NetIPC implementation only:

- *AdrOf*. This call obtains the byte address of any byte within a data object.
- *IPCGet*. This call allows a process to obtain ownership of a call socket, path report or VC socket descriptor that was given away by another process with an IPCGive call.
- *IPCGive*. This call allows a process to "give up" a call socket, VC socket or path report descriptor so that another process may obtain it.

The LAN/9000 NetIPC implementation includes one call that is not provided by NS-ARPA/1000 NetIPC:

- *OptOverHead*. This call is used to determine the number of bytes needed for the *opt* parameter.

Common NetIPC Calls

The following NetIPC calls are common to both the NS-ARPA/1000 and LAN/9000 NetIPC and are implemented identically.

Table B-1. Identical NetIPC Calls

AddOpt	IPCLookUp
InitOpt	IPCSend
IPCDest	ReadOpt

Call Comparison

Table B-2 lists the differences between the NetIPC calls that are common to both the NS-ARPA/1000 and LAN/9000 NetIPC implementations but that are implemented differently.

Table B-2. NS-ARPA/1000 and LAN/9000 Call Comparison

NetIPC Call	Differences Between Implementations
`IPCConnect`	The NS-ARPA/1000 implementation of `IPCConnect` defines a *`flags`* parameter bit that is not defined by the LAN/9000 implementation of the call: "checksumming" (bit 22). All LAN/9000 `IPCConnect` *`flags`* parameter bits must be clear (not set). NS-ARPA/1000 NetIPC processes can enable TCP checksumming by setting the "checksumming" bit. If this bit is not set, TCP checksum will not be performed for the connection unless the process's peer process calls `IPCRecvCn` with that call's "checksumming" bit set, or the peer process is a LAN/9000 NetIPC process. TCP checksumming is *always* enabled when the LAN/9000 implementation of `IPCConnect` is called. Refer to "TCP Checksum" earlier in this appendix for more information. The LAN/9000 implementation of `IPCConnect` allows a value of -1 to be assigned to the call's *`calldesc`* parameter. This value causes a call socket to be created and then destroyed after the call completes successfully. The NS-ARPA/1000 implementation of `IPCConnect` does not allow this value. The NS-ARPA/1000 and LAN/9000 implementations of `IPCConnect` implement different maximum send and receive sizes. The NS-ARPA/1000 maximum send and receive sizes are 8,000 bytes; the LAN/9000 maximum send and receive sizes are 32,000 bytes. The default size on both implementations is 100 bytes.
`IPCControl`	The LAN/9000 implementation of `IPCControl` includes four request codes that are not provided by the NS-ARPA/1000 implementation of the call: 4, 1002, 1003 and 9008. When request code 9008 is specified, the LAN/9000 implementation of `IPCControl` allows a value of -1 in the call's *`descriptor`* parameter; this is also not part of the NS-ARPA/1000 implementation of the call. Refer to the LAN/9000 NetIPC chapter for a description of these request codes. Unlike the NS-ARPA/1000 implementation of `IPCControl`, the operation of the LAN/9000 `IPCControl` call is affected by socket sharing. Refer to "Socket Ownership" earlier in this appendix for more information about socket sharing. Refer to the LAN/9000 NetIPC chapter for a complete description of how socket sharing affects the `IPCControl` call.
`IPCCreate`	The NS-ARPA/1000 and LAN/9000 implementations of IPCCreate support different ranges of permitted TCP protocol addresses that can be specified in the *`opt`* parameter. However, both implementations recommend that users specify TCP addresses in the range 30767 to 32767 decimal. The NS-ARPA/1000 and LAN/9000 implementations of `IPCCreate` also support different maximum connection request backlog defaults and ranges. The NS-ARPA/1000 implementation has a default of three connection requests and an allowable range of zero to five; the LAN/9000 implementation has a connection request default of one and an allowable range of 1 to 20.

NetIPC Call	Differences Between Implementations
`IPCName`	The LAN/9000 implementation of `IPCName` allows for the naming of destination (also known as path) descriptors. The NS-ARPA/1000 implementation of the call does not.
`IPCNamErase`	Unlike the NS-ARPA/1000 implementation of `IPCNamErase`, the operation of the LAN/9000 implementation of `IPCNamErase` is affected by socket sharing. Refer to “Socket Ownership” earlier in this appendix for more information about socket sharing. Unlike the LAN/9000 implementation of `IPCNamErase`, the operation of the NS-ARPA/1000 implementation of the call does not allow for erasing names assigned to path report (also known as destination) descriptors.
`IPCRecv`	The LAN/9000 implementation of `IPCRecv` defines bit 26 of the call’s *`flags`* parameter as “more data.” This bit is not implemented on NS-ARPA/1000. When this bit is set on an LAN/9000, it indicates that non-delimited data was received.
`IPCRecvCn`	The NS-ARPA/1000 implementation of `IPCRecv` includes a *`flags`* parameter bit that is not defined by the LAN/9000 implementation of the call: “checksumming” (bit 22). All LAN/9000 `IPCRecvCn` *`flags`* parameter bits must be clear (not set). NS-ARPA/1000 NetIPC processes can enable TCP checksumming by setting the “checksumming” bit. If this bit is not set, TCP checksum will not be performed for the connection unless the process’s peer process called `IPCConnect` with that call’s “checksumming” bit set, or the peer process is an LAN/9000 NetIPC process. TCP checksumming is *always* enabled when the LAN/9000 implementation of `IPCRecvCn` is called. Refer to “TCP Checksum” earlier in this appendix for more information.
`IPCSelect`	The LAN/9000 implementation of `IPCSelect` allows the *`sdbound`* parameter to have a maximum value of 60. The NS-ARPA/1000 implementation has an upper limit of 32.
`IPCShutDown`	Unlike the NS-ARPA/1000 of IPCShutDown, the operation of the LAN/9000 implementation of `IPCShutDown` is affected by socket sharing. Refer to “Socket Ownership” earlier in this appendix for more information.

Glossary

address

Used to indicate where nodes are located in a network. Addresses are usually numeric. In NS-ARPA/1000, nodes are assigned different types of addresses. Also see *Internetwork Protocol address*, *Router/1000 address*, and *LAN station address*.

ANH

See *Appropriate Next Hop*.

APLDR

The DS/1000-IV Compatible Services absolute program loader. Processes the REMAT `LO`, `IO` and `PL` commands, and FLOAD utility call. (The `LO` command and FLOAD call can be used to load programs into memory-based systems only.)

Application Layer

Layer 7 of the OSI model. Tasks include the user interface to remote services.

Appropriate Next Hop (ANH)

The next node to which IP is to route a message. Also see *Internetwork Protocol*.

ARP

Translates Internet (IP) addresses to physical addresses. Like Probe, ARP is not directly accessible to users.

ARPA

Advanced Research Projects Agency. ARPA services supported by NS-ARPA/1000 include TELNET (virtual terminal service) and FTP (File Transfer Protocol). Also see *TELNET* and *FTP*.

asynchronous mode

A mode of data exchange utilized by NetIPC processes. When NetIPC processes exchange data in asynchronous mode, send and receive requests do not cause the calling process to be suspended if a request cannot be immediately satisfied. Instead, a "would block" error is returned and the calling process is free to perform other tasks before retrying the request.

Berkeley Sockets

See *BSD IPC*.

Bisync

A type of communication link used by NS-ARPA/1000 to connect NS-ARPA/1000 to DS/3000 systems. Bisync links support only DS/1000-IV Compatible Services (RTE-MPE), and have no store-and-forward or rerouting capabilities. They can be hardwired or modem connections.

Bisync ID Sequences

Local and remote ID sequences can be assigned to limit access to the HP 1000 if you have an HP 3000 connected to the HP 1000 via a Bisync dial-up link. When the HP 1000 or an HP 3000 attempts to establish communication over the link, each machine sends its local ID sequence, which the other machine compares with its remote ID sequence.

BREVL

The NS-ARPA/1000 program that terminates event logging (EVMON).

broadcast bus network

A network in which nodes are connected by a linear run of cable. Messages are simultaneously transmitted to every node. Typically, the nodes will process only those messages addressed to them, and ignore all other messages. IEEE 802.3 networks are broadcast bus networks. Compare with *point-to-point network*.

BRTRC

The NS-ARPA/1000 program that terminates tracing (NSTRC).

BSD IPC

Berkeley Software Distribution Interprocess Communication (BSD IPC) provides industry standard libraries and tools for interprocess communication on HP, UNIX, and other systems that have BSD IPC.

buffer area

An area of DSAM that is dynamically mapped and contains Message Buffers (MBUFs), Clusters, and Message Accounts (MACCTs). Also see *Messages Buffers*, *Clusters*, and *Message Accounts*.

call socket

Used by NetIPC processes to create and connect virtual circuit sockets.

call socket descriptor

A descriptor that refers to a call socket. NetIPC processes obtain call socket descriptors by invoking the NetIPC calls `IPCCreate` or `IPCGet`.

CNSLM

An NS-ARPA/1000 slave monitor that reports MPE TELL and WARN messages. Used by the DS/1000-IV Compatible Services Transport.

clusters

Part of the DSAM buffer area. A cluster is 1024 bytes of contiguous physical memory. Also see *Message Buffers*.

communication link

The software and hardware that moves data from the driver and card of one machine to the driver of an adjacent machine. NS-ARPA/1000 supports the following communication links: IEEE 802.3, Ethernet, HDLC, X.25, and Bisync (to NS/3000 or DS/3000 only).

Connect Site Report (CSR)

Provides information on how to reach a given node within an internetwork and how to reach a given NetIPC socket within a node. CSRs are stored in the Socket Registry. Also see *Socket Registry*.

CONSM

An NS-ARPA/1000 module used by Network File Transfer. CONSM is required at the Consumer node. Also see *Consumer*.

Consumer

One of the three logical participants in the Three-Node Model utilized by the NS-ARPA/1000 User Service Network File Transfer (NFT). The Consumer is located on the same node as the target file, consumes the source file data and writes it into the target file. Also see *Three-Node Model*.

Converters

NS-ARPA/1000 monitors that are used to convert message formats for DS/3000 and DS/1000 services. If Converters are needed, they are scheduled by NS-ARPA/1000 and wait for class get completions.

copy descriptor

A parameter used by NFT to describe the source and target file names, nodes names, logons, and any NFT options that should be used when a file is copied using the DSCOPY command.

cross-system

A general term to mean that two different types of computer systems are communicating with one another. For example, cross-system NFT is supported between NS-ARPA/1000 and NS/3000 computer systems as well as between other NS systems. Refer to the *NS Cross-System NFT Reference Manual* for information about NFT. Cross-system NetIPC is also supported between NS-ARPA/1000 and NS/9000 Series 800.

CSR

See *Connect Site Report*.

datagram service

When a datagram service is utilized, the Network Layer (Layer 3 of the OSI architecture) delivers each message separately; no attempt is made to keep messages in order. Therefore, messages may arrive out of order, or not at all. Because there are no "set up," data transfer, or shutdown procedures, each datagram is sent independently and must contain destination information. The HP protocol PXP provides a datagram service. Also see *PXP*.

Data Link Layer

Layer 2 of the OSI model. Checks for and corrects transmission errors over the physical link. Also see *Open Systems Interconnection*.

data vector

A structure used by NetIPC calls that can describe several data objects. The description of each object consists of a byte address and a length. The byte address describes where the object is located and the length indicates how much data the object contains. Any kind of data object (arrays, portions of arrays, records, simple variables, etc.) can be described by a data vector.

DCN

See *Directly Connected Network*.

DCN information

Information configured for IP that indicates all the Router/1000 and LAN networks to which the local node belongs. Also see *Gateway information*, *NGT information*, and *Directory Connected Network*.

Directly Connected Network (DCN)

The local network. Also see *network*, *Router/1000 network*, and *IEEE 802.3 network*.

Distributed Executive (DEXEC)

A DS/1000-IV Compatible Service that allows you to control I/O devices located at remote HP 1000 nodes. DEXEC calls are the distributed equivalent of local RTE EXEC calls.

Distributed System Available Memory (DSAM)

A memory area specifically reserved for NS-ARPA/1000. Most NS-ARPA/1000 tables are stored in DSAM. NS-ARPA/1000 messages and messages for DS/1000-IV Compatible Services sent over non-Router/1000 links are stored in DSAM before transmission. DSAM is controlled by the NS-ARPA/1000 Memory Manager, and is implemented as a Shareable Extended Memory Access (SHEMA) partition. Also see *Memory Manager*, *Message Buffers*, *clusters*, and *buffer area*.

DLIST

The NS-ARPA/1000 remote directory list monitor. Lists contents of FMGR directories. Used by the DS/1000-IV Compatible Services.

DS File Transparency

A feature of the RTE operating system that allows you to access remote files. Also called Transparent File Access (TRFAS).

DS/1000-IV Compatible Links

See *HDLC*, *Bisync*, and *X.25*.

DS/1000-IV Compatible Services (RTE-MPE)

A term used to describe services that can be used for backward compatibility with DS/3000 or NS/3000 nodes over Bisync or X.25 links. These services include Program-to-Program Communication (PTOP), Remote File Access (RFA), RMOTE, and the Utility Subroutines.

DS/1000-IV Compatible Services (RTE-RTE)

A term used to describe services that can be used for backward compatibility with DS/1000-IV nodes. These services include Distributed Executive (DEXEC), Program-to-Program Communication (PTOP), REMAT, Remote Database Access (RDBA), Remote File Access (RFA), Transparent File Access (TRFAS), Utility Subroutines, Remote I/O Mapping, Remote System Download, and Remote Virtual Control Panel (VCP).

DS/1000-IV Compatible Transport

Transport used by the DS/1000-IV Compatible Services (RTE-RTE) and (RTE-MPE) and DS/1000-IV Compatible Links that are configured with RTR LIs. Each of these transports is responsible for delivery of data between the source and the destination. Also see *Router Link Interface*.

DSCOPY

An NS-ARPA/1000 module used by Network File Transfer. Establishes a VC connection with NFTMN at the Producer node. Also see *Producer* and *NFTMN*.

DSLIN

The NS-ARPA/1000 module used to establish PSI BISYNC connections to HP 3000s.

DSMOD

An NS-ARPA/1000 program that allows you to alter parameters for DS/1000-IV Compatible Services that are set by NSINIT during initialization. DSMOD allows the user to change the HP 3000 ID sequence, re-enable a link, display the Nodal Routing Vector (NRV), change the non-session password, schedule additional monitors, adjust timing, change the default session user name, and change the NRV.

DSRTR

The NS-ARPA/1000 transparent file access master. DSRTR is part of the RTE operating system.

DSTES

An NS-ARPA/1000 module that verifies the PTOP software to a DS/3000 node.

DSVCP

The DS Virtual Control Panel operator interface module for remote control of the A/L-Series front panel. DSCVP is a DS/1000-IV Compatible Service. Users must run the program DSVCP to access the control panel of a slave computer, and consequently access and alters its memory, and CPU and I/O registers. DSVCP can access the boot loader programs in the VCP and cause various programs to be downloaded, via the NS-ARPA link, or another loading device. DSVCP can be used over HDLC RTR links only.

Dynamic Rerouting

The capability to automatically reroute messages around inoperative HDLC links without user intervention. Dynamic rerouting is an option provided by the Router/1000 protocol.

Ethernet Local Area Network

Ethernet is a de-facto standard link level protocol. Ethernet defines a baseband, coaxial, bus media, and the Media Access Method CSMA/CD. IEEE 802.3 and Ethernet nodes can coexist on the same cable, but cannot communicate with each other.

event messages

Messages sent between protocol modules to indicate events and pass references to path records.

EVMON

The NS-ARPA/1000 Event Monitor. Logs disasters, errors, warnings, and internal state information.

exception select

Can be performed by using the NetIPC call `IPCSelect.` NetIPC processes can determine whether certain connections have been aborted by performing an exception select.

EXECM

The NS-ARPA/1000 remote EXEC slave monitor. Services remote EXEC (DEXEC) calls. These requests may come from other HP 1000 or HP 3000 nodes. Used by the DS/1000-IV Compatible Services. Also see *Distributed Executive*.

EXECW

The NS-ARPA/1000 remote "schedule with wait" monitor. Services remote EXEC (DEXEC) requests to schedule programs with wait. To run a program from REMAT or to execute the `LO` (load) and `PL` (program list) commands on memory based nodes, the remote node must have EXECW. Used by the DS/1000-IV Compatible Services.

FCL66

The NS-ARPA/1000 utility that is used to programmatically force cold load slave computers over an HDLC RTR link.

FMTRC

The NS-ARPA/1000 program that formats trace files produced by NSTRC. Also see *NSTRC*.

fully-qualified node name

An NS-ARPA/1000 node name that includes all three parts of the syntax (i.e., the node, domain, and organization). Also see *node name*.

FTP

File Transfer Protocol is an ARPA service that allows you to copy files from one node to another. The other computer must also support FTP.

gateway node

A node that is a member of two or more networks and allows communication between the networks to which it belongs.

Gateway Table (GT)

A table used by the Internetwork Protocol (IP) to determine which gateway to route through to reach a remote network. Also see *Internetwork Protocol* and *DCN information*.

General Pool

An area in DSAM that is managed by Memory Manager. Memory Manager allocates all memory from its General Pool. The General Pool is divided into sub-pools which are organized into different levels above the General Pool. Also see *Memory Manager* and *DSAM*.

global area

An area in DSAM that is always mapped in. Memory Manager stores information in the global area that it needs to access frequently or quickly. See *DSAM* and *Memory Manager*.

GRPM

The NS-ARPA/1000 transport monitor that acts as the RTE-RTE request/reply processor. GRPM is scheduled by ID*66 to allocate a class buffer in SAM to receive incoming messages. GRPM routes incoming message to the Slave Monitor's class numbers.

guardian node

A node that allows 91750 nodes to communicate with nodes on remote networks. Guardian nodes remove IP headers from messages sent over non-Router/1000 links before delivering them to the 91750 node. Guardian nodes also add IP headers to messages sent from 91750 nodes that are to be forwarded to remote networks.

HDLC

A type of communication link used by NS-ARPA/1000 to connect HP 1000s. Can have a RTR LI. Networks composed of HDLC links are point-to-point networks and can have arbitrary topologies. HDLC links can be hardwired or modem connections. Also see *point-to-point network*.

hierarchical

A point-to-point network topology. The hierarchical topology is sometimes used with supervisory-control applications, where large databases exist at one node, possibly along with control programs, that are accessed by nodes lower in the hierarchy. You can also use hierarchical topologies for distributed database applications.

IEEE 802.3 Link Interface

NS-ARPA/1000 link interface type that supports a single communication link type: IEEE 802.3 The LI software determines which IEEE 802.3 communication LU to use and sends the message to the IEEE 802.3 driver. At this point the control of the message moves from NS-ARPA/1000 to LAN/1000. Also see *communication link* and *Link Interface*.

IEEE 802.3 Local Area Network Link

A type of communication link used by NS-ARPA/1000 to join HP 1000s over a relatively small geographical area to form a LAN. Provides fast links and requires less hardware than point-to-point links for networks with several nodes. IEEE 802.3 networks are broadcast bus networks. IEEE 802.3 and Ethernet nodes can coexist on the same cable, but cannot communicate with each other. Also see *broadcast bus network* and *Ethernet*.

IEEE 802.3 network

A group of nodes connected to the same LAN bus. Also see *broadcast bus network*.

IEEE 802.3 protocol

A protocol used at the Subnet or Intranet Layer (Layer 3s of the NS-ARPA/1000 architecture). IEEE 802.3 defines some layer 3s functions for messages sent over IEEE 802.3 links. With IEEE 802.3 LAN networks, messages are transmitted to every node in the network and a node accepts only those messages that are addressed to it.

IFPM

An NS-ARPA/1000 transport monitor that acts as an interface between DS/1000-IV Compatible Services and IEEE 802.3 links for outbound messages. Implements the Interface Protocol (IFP).

Inbound Address List

A list of LAN addresses maintained by each node in a LAN network. A given node will receive only those messages that have a destination address that matches an address on its Inbound Address List. Each node's Inbound Address List contains the node's station address. In addition, a group of nodes may have the same multicast address in their Inbound Address List. Also see *multicast address*.

Inbound Proxy Address

Used by LAN nodes to receive Proxy Name Requests. The Inbound Proxy Address is added to the Inbound Address List of Probe Proxy Servers only.

Inbound Target Address

Added to every LAN node's Inbound Address List at initialization time. The Inbound Target Address allows the LAN nodes to receive Probe Name Requests.

INETD

Internet Network Services Daemon that monitors all incoming connection requests from TELNET, FTP, SMTP (Mail/1000), and Remote Spool Printing.

Initiator

One of the three logical participants in the Three-Node Model utilized by the NS-ARPA/1000 User Service Network File Transfer (NFT). The Initiator is located on the system where the copy request originates, receives the user request and initiates the copy process. Also see *Three-Node Model*.

INPRO

The inbound message processor for the NS-ARPA/1000 transport. Contains the inbound NS-ARPA protocol modules.

Interchange Format

One of the file copying formats used by the NS-ARPA/1000 User Service Network File Transfer (NFT). Interchange Format must be invoked explicitly using one or more of NFT's Interchange Format options. Interchange Format is useful when you want to change certain source file attributes, such as record length, in the target file.

Interface Protocol (IFP)

Provides the interface between NS Common Services Transport and DS/1000-IV Compatible Services Transport.

internetwork

Several networks that are joined, or concatenated, to form a network of networks.

internetwork communication

Communication between networks.

Internetwork Protocol (IP)

The NS-ARPA/1000 protocol based on the Defense Advanced Research Projects Agency's (DARPA) standard that is implemented at the Internet Layer (Layer 3i) of the NS-ARPA/1000 architecture. IP is primarily used to route messages between networks via gateways. It provides gateway-to-gateway routing, store-and-forward service between gateways, and message fragmentation and reassembly between source and destination networks. IP routines are contained in the INPRO and OUTPRO message processors. Also see *INPRO* and *OUTPRO*.

Internetwork Protocol (IP) address

An address used by the NS-ARPA/1000 Services and IP. An IP address consists of two parts: a network address, which identifies the network; and a node address, which identifies the node within a network. A network address is concatenated with a node address to form the IP address and uniquely identify a node within a network within an internetwork.

Intranet Layer

In NS-ARPA/1000, this is layer 3s. Layer 3s is part of the OSI Network Layer and handles intranetwork routing (routing within a network). NS-ARPA/1000 supports four Layer 3s protocols: Router/1000, IEEE 802.3, Ethernet, and X.25. Also see *Open Systems Interconnection*.

IOMAP

The NS-ARPA/1000 module that provides the user interface for Remote I/O mapping. Also see *Remote I/O Mapping*.

IP

See *Internetwork Protocol*.

LAN

Local Area Network. Also see *IEEE 802.3 Local Area Network Link* and *Ethernet Local Area Network*.

LAN station address

An address that HP assigns to each LAN interface card during manufacturing. The station address is used for addressing within the LAN (subnetwork addressing).

layers

Refers to the layers in the seven-layer Open Systems Interconnection (OSI) network architecture model developed by the International Standards Organization (ISO). In the NS-ARPA/1000 network architecture, different transmission and communications tasks are assigned to each layer, which is a logically distinct module. Also see *Open Systems Interconnection*.

LI

See *Link Interface*.

link

See *communication link*.

Link Interface (LI)

The software that determines which communication link type to use and then passes the message to the appropriate driver for the specific communication link. Communication links are bound to link interface types at NS-ARPA/1000 initialization time. NS-ARPA/1000 supports four LI types: Router (RTR), IEEE 802.3 (802), Ethernet (ETHER) and LAN. If both IEEE 802.3 and Ethernet are on the same LAN, then the LI type is declared `LAN`. Also see *Router Link Interface*, *IEEE 802.3 Link Interface*, *Ethernet Local Area Network*, and *communication link*.

local network

The network to which the local node belongs. Also referred to as the Directly Connected Network (DCN).

local node

Refers to the node where you are physically located and logged on, and at which you enter commands.

LOG3K

The NS-ARPA/1000 module that provides operator control over recording of DS messages to and from HP 3000s.

link type

See *communication link*.

LUMAP

The NS-ARPA/1000 DEXEC request module for Remote I/O Mapping. Also see *Remote I/O Mapping*.

LUQUE

The NS-ARPA/1000 module that provides class buffers for Remote I/O mapped data transfer. Also see *Remote I/O Mapping*.

#MAST

Handles request buffers for master routine subroutine calls. Adds a Router/1000 header to the request and passes it to GRPM or to the transmission LU. Used by the DS/1000-IV Compatible Services.

$MWB

The NS-ARPA/1000 module needed by APLDR to move words across maps. Provided by the RTE operating system. Also see *APLDR*.

master-slave protocol

The sequence of messages exchanged between master and slave Program-to-Program (PTOP) programs. Also see *Program-to-Program Communication*.

MATIC

An NS-ARPA/1000 module that provides timeout processing for Message Accounting. Used by the DS/1000-IV Compatible Services and Links.

Memory Manager

Manages DSAM, the NS-ARPA/1000 memory area. Memory Manager divides DSAM into three areas: global area, tables area, and buffer area. Also see *global area*, *tables area*, and *buffer area*.

Message Accounting (MA)

A datagram-oriented protocol that retransmits lost messages and suppresses duplicate messages. The DS/1000-IV Compatible Transport (RTE-RTE) uses Message Accounting.

Message Buffers (MBUFs)

Part of the DSAM buffer area. An MBUF is 128 bytes of contiguous physical memory.

monitor

In NS-ARPA/1000, a monitor is a software module that is scheduled at node initialization time, or by DSMOD, and remains scheduled until NS-ARPA is shut down. There are three types of NS-ARPA/1000 monitors: Transport Monitors, Slave Monitors, and Converters. Also see *Transport Monitors*, *Slave Monitors*, *Converters*, and *watch dogs*.

MMINIT

The Memory Manager initialization program. MMINIT sets up DSAM. MMINIT is scheduled by NSINIT at initialization. Also see *Memory Manager*, *DSAM*, and *NSINIT.*

multicast address

An address that may be included in the Inbound Address List of a node in a LAN network. A group of nodes may have the same multicast address in their Inbound Address List. Messages with that multicast address are received by all nodes in that group. Probe uses multicast addresses to send messages to HP nodes in a LAN. Two Probe multicast addresses are used at each LAN node: a Target Address and a Proxy Address. Also see *Inbound Address List*, *Probe*, *Target Address*, and *Proxy Address*.

MVCP3

The NS-ARPA/1000 module used to install the PTOP slave program `COPY3K.PUB.SYS` on an HP 3000 for use in implementing the RMOTE `MO` command. Also see *RMOTE*.

name

Used to identify objects, such as nodes and sockets. In NS-ARPA/1000, nodes are assigned node names. Also see *node names*.

NetIPC

See *Network Interprocess Communication*.

NetIPC root socket

NetIPC allocates one root socket for each NetIPC process. The root socket is used as an endpoint to lower level protocols and sets up a path for any call or VC sockets requested by the user.

NetIPC user record

NetIPC allocates one user record for each NetIPC process. NetIPC uses this record to keep track of that process's NetIPC sockets and other resources used for NetIPC.

network

A group of computer systems connected so that they can exchange information and share resources. More specifically, see *Router/1000 network* and *IEEE 802.3 network*.

network architecture

A structured, modular design for networks.

network boundary

The division between networks in an internetwork. Also see *internetwork*.

Network File Transfer (NFT)

An NS-ARPA/1000 User Service that allows you to copy files from one node to another interactively or programmatically. Cross-system NFT is also supported.

Network Interprocess Communication (NetIPC)

An NS-ARPA/1000 User Service that allows autonomous processes running concurrently at different nodes to exchange information in a peer-to-peer manner. Cross-system NetIPC is also supported.

Network Layer

Layer 3 of the OSI model. Tasks include determining the routes messages take to get from one node to another. In NS-ARPA/1000, this layer is split into two sub-layers, 3i and 3s. Layer 3i handles internetwork routing and 3s handles subnetwork or intranetwork routing, which is routing within the network.

network map

A diagram of the links and nodes in your network. A network map should include node information (system type and resources, peripherals, amount of memory, services supported) and link information (location of coaxial cables, terminators, repeaters, AUI cables, and MAUs). In addition, you should mark network boundaries, link interface types, card types, and the names and addresses that you assign.

For an internetwork, the network map shows how different networks are connected. A network map for an internetwork includes the following: the network names and types (and, if applicable, a unique IP address for each network), the gateway nodes, and the network boundaries.

Network security code

A code consisting of two alphanumeric characters that is required for the following tasks: shutting down NS-ARPA/1000 (via NSINIT); establishing Remote I/O maps; modifying timing parameters; changing the Nodal Routing Vector (via DSMOD).

Network User's security code

A code consisting of two alphanumeric characters that is required to execute the REMAT `SW` (switch) command; thus it is used to restrict REMAT access to remote nodes. Also see *REMAT*.

NFT

See *Network File Transfer*.

NFTMN

The Network File Transfer monitor program. NFTMN is scheduled at initialization time and remains scheduled, waiting for an `IPCRecvCn` call, until NS-ARPA is shut down. At the Producer node, NFTMN schedules PRODC; the NFTMN at the Consumer node schedules CONSM. Also see *Producer*, *Consumer*, *PRODC*, and *CONSM*.

Nodal Path Reports (NPRs)

Contains information stored in DSAM by the Nodal Registry software. Nodal Path Reports map node names to address information. NPRs are indexed according to node name, and each contain an IP address (or addresses, if the node belongs to multiple networks). If the node has LAN links, the NPR can also contain a LAN station address for each link. In addition, an NPR denotes which protocols and NS Common Services the node supports.

Nodal Registry

The NS-ARPA/1000 software that manages information that the transport and services use to establish connections with remote nodes.

Nodal Routing Vector (NRV)

Used by Router/1000 for subnet routing with Router/1000 networks and for address information for DS/1000-IV Compatible Services (RTE-RTE).

node

A computer system in a network.

node names

Each computer system, or node, in an NS-ARPA/1000 network has a node name. NS-ARPA/1000 node names contain three fields: a node, domain, and organization. Also see *fully-qualified node name*.

node number

See *Router/1000 address*.

NRINIT

The Nodal Registry initialization program.

NRLIST

The Nodal Registry List program. Lists the contents of the Nodal Registry.

NRV

See *Nodal Routing Vector*.

NS Common Services

A term used to refer to the services: Network File Transfer (NFT), Network Interprocess Communication (NetIPC), and Remote Process Management (RPM).

NSERRS.MSG

The name of the Network File Transfer message file. Must be on `/system`.

NSINF

The NS-ARPA/1000 information utility. Prints local address information, configured NS-ARPA/1000 resources, NS-ARPA/1000 network management utilities status and statistics, Memory Manager statistics, NS-ARPA/1000 program information, DS/1000-IV Compatible Services information, Nodal Routing Vector information, Remote Session information, and Message Accounting information.

NSINIT

NSINIT and its subordinate programs (NSPR1, NSPR2, NSPR3, NSPR4, NSPR5, NSPR6, and NSPARS) initialize and shut down the network.

NSINIT.MSG

The name of the NSINIT message file. Must be on `/system`.

NSTRC

The NS-ARPA/1000 trace utility. Records messages at the socket and network level.

Open Systems Interconnection (OSI)

A seven-layer network architecture model developed by the International Standards Organization (ISO). In the OSI model, transmission and communication tasks are assigned to logically distinct modules called layers. Each layer communicates with the layer directly above and below it, and through the layers below it to its peer in the remote computer. The OSI model defines seven layers: Application Layer, Presentation Layer, Transport Layer, Network Layer, Data Link Layer, and Physical Layer.

OPERM

An NS-ARPA/1000 slave monitor that provides remote RTE operator command capability.

OSI

See *Open Systems Interconnection*.

Outbound Proxy Address

Used by LAN nodes to send Proxy Name Requests.

Outbound Target Address

Used by LAN nodes to send Probe name and address requests (Name Requests and Virtual Address Requests).

OUTPRO

The outbound message processor for the NS-ARPA/1000 transport. Contains the outbound NS-ARPA protocol modules.

Packet Exchange Protocol (PXP)

An HP protocol that provides a low-overhead datagram service. PXP is a low-overhead request/reply protocol that is suited for querying data sources. PXP suppresses duplicate replies to a request but does not suppress responses to duplicate requests. PXP retransmits messages that are not acknowledged within a timeout interval. Socket Registry is the only NS-ARPA/1000 service that uses PXP; it is not user accessible. PXP is included in INPRO and OUTPRO.

path

The course within a machine that a message takes, typically through software protocol handlers.

path records

The data structures in which the NS-ARPA/1000 protocols keep address and other context information. The protocol modules also use path records to guide messages to the next appropriate protocol. Path records are logical records; the actual data structures that hold path record information are given different names by the protocols and usually contain other, protocol-specific, information.

path report

Created by the socket registry. Path reports are returned to NetIPC processes in the form of a path report descriptor. The path report contains addressing information that is used by the calling process to direct requests to a certain call socket at a certain node.

path report descriptor

A descriptor used by NetIPC processes to refer to a path report. A NetIPC process obtains a path report descriptor by invoking either the `IPCLookUp` or `IPCGet` call. Also see *path report*.

PCB

See *Protocol Control Block*, *Probe Control Block* or *PTOP Control Block*.

Physical Layer

Layer 1 of the OSI model. Transmits the electrical signals over the link.

point-to-point network

A network in which communication travels from one node (point) to another node over the links.

pool LU

Used for X.25 connections to HP 3000s (DS/1000-IV Compatible Services).

POOL Table

Contains entries for each concurrent remote user or program, including monitors, that access the local node to use DS/1000-IV Compatible Services (RTE-RTE).

PRDC1

An NS-ARPA/1000 module used by Network File Transfer. PRDC1 is scheduled by PRODC if the user specifies a file mask. PRDC1 is required at the Producer node if file masks are used. Also see *PRODC* and *Producer*.

Probe

A protocol that allows LAN nodes to query one another for Nodal Path Reports and other address information. The Probe protocol provides the following features: node name to Nodal Path Report mapping; IP address to LAN station address mapping; Nodal Registry updates. Also see *Nodal Path Reports*.

Probe addresses

See *Inbound Address List*, *multicast address*, *Target Address*, and *Proxy Address*.

Probe Control Block (PCB)

Required for each active Probe query.

Probe Proxy Server

Provides Nodal Path Reports (NPRs) for any node in an internetwork. If a LAN has a Probe Proxy Server, all nodes on that LAN can get the NPRs they need from the Probe Proxy Server; at the other nodes on the LAN, only the NPR for that node must be configured. Also see *Nodal Path Reports*.

Process Number List (PNL)

List containing TCBs used for transactions to HP 3000s (DS/1000-IV Compatible Services). Also see *Transaction Control Block*.

PRODC

An NS-ARPA/1000 module used by Network File Transfer. PRODC establishes a VC connection with the monitor NFTMN at the Consumer node (the node to which the file will be copied). Also see *NFTMN*, *Consumer*, *PRDC1*, and *Producer*.

Producer

One of the three logical participants in the Three-Node Model utilized by the NS-ARPA/1000 User Service Network File Transfer (NFT). The Producer is located on the same node as the source file, accesses that file and produces the data that is to be copied.

PROGL

The NS-ARPA/1000 slave monitor used for remote download from Communication Bootstrap Loader (CBL). Can simultaneously handle requests from up to 20 nodes at the same time. User supplied subroutines enable store-and-forward and/or LU to file conversion capabilities. Can be used over HDLC RTR links only. PROGL is used by the DS/1000-IV Compatible Services.

protocol

A set of rules for a particular communication task. A protocol handler or protocol module is a piece of software that implements a particular protocol.

Protocol Control Block (PCB)

Used by TCP and PXP to keep track of protocol-specific parameters such as segment sizes and time values.

protocol EMAs

A term used to describe EMA partitions that are used by the NSINIT subordinate programs (NSPR1, NSPR2, NSPR3, NSPR4, NSPR5, and NSPR6). Protocol EMAs are used to build protocol tables before moving the tables to DSAM.

Proxy Address

One of the two Probe multicast addresses that must be configured at each LAN node. Probe uses this address for two other addresses: an Inbound Proxy Address and an Outbound Proxy Address. Also see *Inbound Proxy Address* and *Outbound Proxy Address*.

protocol module

A group of software modules that implement a given protocol. For example, the TCP protocol module is the collective term for the group of software routines that implement the TCP protocol.

Presentation Layer

Layer 6 of the OSI architecture. Tasks include manipulation of user data such as text compression and encryption.

Program-to-Program Communication (PTOP)

An NS-ARPA/1000 User Service that enables a "master" program on one node to exchange information with and control the execution of a "slave" program on another node.

PTOP Control Block (PCB)

Parameter used in PTOP calls which serves as a control for the data link. Also see *Program-to-Program Communication*.

PTOPM

The NS-ARPA/1000 PTOP communication slave monitor. Handles programmatic `POPEN`, `PREAD`, `PWRIT`, `PCONT`, and `FINIS` requests and REMAT commands `SO` (slave off) and `SL` (slave list) on the slave side.

PXP

See *Packet Exchange Protocol*.

QCLM

An NS-ARPA/1000 transport monitor that acts as a communications error logger. QCLM prints most errors for the DS/1000-IV Compatible Services modules, the DS/1000-IV Compatible Transport, and the HDLC driver.

QUEUE

An NS-ARPA/1000 program that is scheduled by the interface driver to allocate a class buffer in SAM to receive incoming messages.

QUEX

An NS-ARPA/1000 transport monitor used for HP 3000 communication.

QUEZ

An NS-ARPA/1000 transport monitor used for HP 3000 I/O completions.

raw NPRs

Character strings that the Nodal Registry initialization program, NRINIT, uses to build Nodal Path Reports. A raw NPR contains a node name and an IP address or addresses. If the node has a LAN link, the raw NPR can also contain a LAN station address or addresses. Also see *Nodal Path Reports*.

RDBAM

The NS-ARPA/1000 remote database access slave monitor.

read select

A read select can be performed by using the NetIPC `IPCSelect` call. NetIPC processes can determine whether certain VC sockets are readable by performing a read select. A readable socket is one that can immediately satisfy a receive request for a number of bytes greater than or equal to its read threshold.

read threshold

Used by NetIPC processes that exchange data in asynchronous mode. A NetIPC process determines whether a VC socket is readable by examining the socket's read threshold. A VC socket is considered readable if it can immediately satisfy a receive request for a number of bytes greater than or equal to its read threshold.

READR

A LAN/1000 module for the Node Manager.

redundant links

Provide alternative routes between nodes in a Router/1000 network. If a node or link fails, the nodes can still communicate by rerouting around the failure. In a Router/1000 network with Dynamic Rerouting, the rerouting is automatic; otherwise you must explicitly configure new routes around the failure. Also see *Dynamic Rerouting*.

REMAT

A DS/1000-IV Compatible Service that allows you to send RTE commands, or special REMAT commands, to any HP 1000 computer in your network.

Remote Database Access

A DS/1000-IV Compatible Service that allows you to access an IMAGE database at a remote HP 1000.

Remote File Access (RFA)

A DS/1000-IV Compatible Service that enables you to perform I/O operations to files located at remote nodes.

Remote I/O Mapping

A DS/1000-IV Compatible Service that maps I/O requests from one node to another, allowing resource sharing.

Remote Process Management (RPM)

An NS-ARPA/1000 User Service that allows you to schedule, control, or terminate programs located at the same or different HP 1000 nodes in your network.

Remote System Download

A DS/1000-IV Compatible Service that allows you to download an operating system file to a remote HP 1000.

Remote Virtual Control Panel (VCP)

A DS/1000-IV Compatible Service that allows you to operate the VCP of a remote node. This feature is useful for applications that require a remote, terminal-less node (such as a harsh operating environment).

rerouting

The capability to reroute messages around inoperative links.

remote network

Any network in the internetwork to which the local node does not belong.

remote node

Refers to a node that is not physically located where you are, and which you communicate with via data communication.

resource sharing

The most significant feature of a network. Elements at each node are accessible from other nodes in the network. These elements may include disk files, printers, magnetic tapes, terminals, and other programs.

RFA

See *Remote File Access*.

RFAM

The NS-ARPA/1000 remote file access slave monitor. Used for FMGR files only and must reside at the node at which the file resides. Used by the DS/1000-IV Compatible Services.

RMOTE

A DS/1000-IV Compatible Service that creates an interactive session for you on a remote HP 3000 in your network, making your terminal appear to be directly connected to the other system.

ring

A point-to-point network topology. The ring is a string topology with an additional link between the end nodes. The store-and-forward delay is half that of a string topology because the maximum number of intervening nodes is halved. The ring topology is suited for data-sharing applications in which data stored at various nodes are accessible from all nodes. Ring networks are less vulnerable than string networks. If any one link fails, all the nodes can still communicate by rerouting around the failure. Also see *string*.

root socket

See *NetIPC root socket*.

route

The course through the network that a message takes from a source node to a destination node. A route can pass through intervening nodes.

Router Link Interface (RTR LI)

An NS-ARPA/1000 Link Interface that supports three communication link types: HDLC, X.25, and Data Link (master only). RTR LIs offer the following features: Dynamic Rerouting (HDLC only), low-overhead for DS/1000-IV Compatible Services, high-overhead for NS Common Services. The nodes linked by the RTR LI are members of the same network. Also see *Dynamic Rerouting* and *communication link*.

Router/1000

A protocol used at the Subnet or Intranet Layer (Layer 3s of the NS-ARPA/1000 architecture). Router/1000 provides store-and-forward and Dynamic Rerouting services for messages sent over HDLC links. Optionally provides rerouting and and Message Accounting. Also see *rerouting* and *Message Accounting*.

Router/1000 address

An address used by the DS/1000-IV Compatible Transport and Router/1000 software. Called "node number" on 91750 nodes. Must be unique to the internetwork.

Router/1000 header

Message header used for DS/1000-IV Compatible Services and Transport. In the 91750 product, this header is referred to as the DS message header.

Router/1000 network

A group of nodes that are connected by RTR LIs. There may be redundant, non-RTR LIs in a Router/1000 network, but for any two nodes in the network, a route must exist between them that consists only of RTR LIs.

RPCNV

The NS-ARPA/1000 to DS/3000 reply converter.

RPM

See *Remote Process Management*.

RQCNV

The NS-ARPA/1000 to DS/3000 request converter.

RTR LI

See *Router Link Interface*.

#SEND

An NS-ARPA/1000 module that is used with Dynamic Message Rerouting. #SEND sends update messages to neighboring nodes. #SEND is part of Router/1000.

#SLAV

Called by the DS/1000-IV slave monitors to send a reply and data, if any, back to the origin node. Used by the DS/1000-IV Compatible Services.

SAM

See *System Available Memory*.

SAP

Service Access Point. The only IEEE-defined SAP currently defined is 6 for IP (Internetwork Protocol).

SBUFs

See *socket buffers*.

Session Layer

Layer 5 of the OSI model. Tasks include connection establishment negotiation at remote nodes.

Slave Monitors

NS-ARPA/1000 monitors that service incoming requests for local resources from remote nodes.

SMB

See *System Memory Block*.

socket

Used to establish communication between NetIPC processes. Processes make use of sockets via the NetIPC calls to establish connections and exchange data. The NS-ARPA/1000 Transport Layer's Transmission Control Protocol (TCP) regulates the transmission of data to and from sockets.

socket buffers

Memory Manager allocates an inbound and an outbound SBUF for each NetIPC socket to hold queued inbound and outbound messages. Also see *Memory Manager*.

Socket Registry

Contains a listing of all the named call sockets that reside at a node. NetIPC processes reference call sockets created by other processes by passing a socket name and the corresponding node name to the socket registry software. The socket registry determines which socket is associated with the name and formats the address information to that socket into a path report which it returns to the inquiring process. Socket Registry is not directly accessible by the user.

star

A point-to-point network topology. The star topology is often used for centralized data collection, supervisory control, or in an application where the outlying nodes have little storage capacity. It is also used when a central node has a large database or control program that is accessed by the other nodes. In a star network, there is at most one intervening node between any two nodes. Star networks are vulnerable to failure of the central node. If the central node fails, no network communication is possible.

station address

See *LAN station address*.

store-and-forward

A method of forwarding messages in a network. In a store-and-forward network, a node can send a message to another node to which there is no direct link. Intermediate nodes can forward the message to the correct destination node. Messages can be stored and forwarded between several nodes.

stream mode

The mode of data transfer used by NetIPC processes. Data transferred between two NetIPC processes is treated as byte stream. When data arrives at a destination VC socket, it is simply appended to any data that may have already been sent to that socket. No attempt is made to preserve boundaries between data sent at different times.

string

A point-to-point network topology. The string topology requires one less communication link than there are computers in the network, and requires the fewest number of links. For communication between non-adjacent nodes, messages are stored and forwarded by intervening nodes. If a link fails, the nodes separated by the failure will not be able to communicate.

subnet mask

A mask that specifies the subnet number in an IP address. The node address portion of an IP address is divided into a subnet number and a node number. The bits in the subnet mask are set to 0 for the node number portion and 1 for the network address and subnet number portions.

subnetting

An optional addressing scheme that partitions the node address portion of an IP address into distinct subnetworks. The node address is divided into a subnet number and a node number. Subnetting allows you to use one network address for two or more physically distinct networks. Each network is then a subnetwork. The physical networks are connected via gateways.

subnetwork

A network that may be a member of an internetwork. Also see *Intranet Layer* and compare with *Internetwork Protocol* and *internetwork*.

subordinate programs

Programs that are part of NSINIT. Also see *NSINIT.*

synchronous mode

A mode of data exchange utilized by NetIPC processes. When NetIPC processes exchange data in synchronous mode, send and receive requests cause the calling process to be suspended until the request can be satisfied, a synchronous timeout occurs, or an error is detected.

SYSAT

The NS-ARPA/1000 System Attention Module required for Remote I/O Mapping. Sends a message to a remote system to set a program's break flag. Also used to send the System Attention request to a remote system. Used by the DS/1000-IV Compatible Services.

System Available Memory (SAM)

In NS-ARPA/1000, SAM is used as a buffer area between the NS-ARPA/1000 device drivers and the Transport. All inbound and outbound NS-ARPA messages pass through SAM.

System Memory Block (SMB)

A memory area in the system map that is specified at system generation time by the generator `MB` command. In NS-ARPA/1000, SMB is used for some tables provided for DS/1000-IV compatibility.

tables area

An area in DSAM that is dynamically mapped in as tables are requested. The tables area contains the dynamically-sized NS-ARPA tables used by protocols and services, which are referenced in the global area. Also see *DSAM*.

TCB

See *Transmission Control Block*.

TCP

See *Transmission Control Protocol*.

TELNET

An NS-ARPA/1000 User Service that allows you to have a virtual terminal connection to another computer in your network. The other computer must also support TELNET. TELNET stands for TELecommunications NETwork. TELNET communicates using the TELNET protocol which is a standard ARPA service.

Three-Node Model

The model utilized by the NS-ARPA/1000 User Service Network File Transfer (NFT). The Three-Node Model has three logical participants: the Initiator, the Producer, and the Consumer. All three participants are logically distinct. They may be three separate processes on three separate nodes, or any two, or all three, may reside on the same node. Also see *Initiator*, *Producer*, and *Consumer*.

Transaction Status Table (TST)

Used to keep track of all master requests from an HP 3000. Provides storage for information from the DS/3000 fixed format header and DS/1000-IV information generated by RQCNV.

Transmission Control Block (TCB)

A TCB is allocated for each program that uses DS/1000-IV Common Services. The TCB keeps track of requests until a reply is received or the request times out.

Transmission Control Protocol (TCP)

The Transport Layer (Layer 4) of the NS-ARPA/1000 architecture implements the Transmission Control Protocol (TCP), which is based on the DARPA standard. TCP is a stream-based (rather than a message-based) protocol that provides non-duplicated, in-sequence data delivery. TCP accepts arbitrarily long data buffers, segments them into packets and sends each packet separately. TCP keeps track of the bytes sent and retransmits them if they are not acknowledged within a timeout interval. TCP at the receiving node reassembles the packets, so that they are delivered to the user (NetIPC) in order (in-sequence delivery). All NS-ARPA/1000 services use TCP except Socket Registry and the DS/1000-IV Compatible Services. TCP is part of INPRO and OUTPRO.

Transparent File Access (TRFAS)

A feature of the RTE operating system that allows you to access remote files. Also called DS File Transparency.

Transparent Format

One of the file copying formats used by the NS-ARPA/1000 User Service Network File Transfer (NFT). Transparent Format is invoked by default when files are copied between NS-ARPA/1000 systems. It does not alter a file's attributes, but simply copies the file.

transport

A term used to collectively refer to layers 1 through 4 of the OSI model. Also see *Open Systems Interconnection*.

Transport Layer

Layer 4 of the OSI model. Responsible for end-to-end data integrity. End-to-end indicates that Layer 4 communicates with its peer only at the source and destination nodes, not at intermediate nodes. Layers 5 through 7 also provide end-to-end services, while Layers 1 through 3 are responsible for data integrity between each node. Also see *Open Systems Interconnection*.

Transport Monitors

NS-ARPA/1000 monitors that process inbound or outbound messages. Transport Monitors act as an interface between the user services and the communication device drivers.

TRC3K

The NS-ARPA/1000 module that formats the data recorded by LOG3K. Also see *LOG3K*.

TRFAS

See *Transparent File Access*.

TST

See *Transaction Status Table*.

UPLIN

An NS-ARPA/1000 module that cleans up NetIPC resources such as TCBs and user records. It is also the timeout and re-enable module that maintains a running time on all transactions, artificially terminates ("times out") any transaction that is not serviced within a user-specifiable time limit. It can restart any HP-supplied slave monitor that has been aborted, and logoff HP 3000 or HP 1000 sessions whose creating program has terminated with a session still outstanding. Also see *Transaction Control Block* and *user record*.

user record

See *NetIPC user record*.

Utility Subroutines

A set of NS-ARPA/1000 programmatic calls that enable you to perform special tasks such as downloading absolute or memory-image program files to memory-based nodes and programmatic remote logons to RTE-6/VM and RTE-IV systems.

VCPMN

The NS-ARPA/1000 Virtual Control Panel Monitor. This module monitors the Virtual Control Panel of a remote A/L-Series CPU. VCPMN intercepts and displays VCP messages on the system console of the neighbor node that is the master of the remote system, which is the slave. VCP is a DS/1000-IV Compatible Service.

virtual circuit

See *virtual circuit connection*.

virtual circuit connection

A connection between two NetIPC processes. Virtual circuit connections are the basis for interprocess communication. Once a virtual circuit is established, the NetIPC processes that share it may use it to exchange data. A virtual circuit connection has two major properties: it is a dedicated link, accessible only to the two processes that established the connection; and it provides reliable service, guaranteeing that data will not be corrupted, lost, duplicated or received out of order.

virtual circuit socket

Used by NetIPC processes to create a virtual circuit connection. Virtual circuit (VC) sockets are the endpoints of a virtual circuit connection.

virtual circuit socket descriptor

Refers to a virtual circuit (VC) socket. A VC socket is the endpoint of a virtual circuit connection between two processes. A VC socket descriptor is returned by the NetIPC calls `IPCRecvCn` and `IPCConnect` after an initial dialogue takes place over a connection formed by call sockets. A NetIPC process can also obtain a VC socket descriptor given away by another process by invoking the NetIPC call `IPCGet`.

watch dogs

NS-ARPA/1000 modules that keep track of internal timing and clean up system resources.

write select

A write select can be performed by using the NetIPC call `IPCSelect.` NetIPC processes can determine whether certain VC sockets are writeable by performing a write select. A writeable socket is one that can immediately satisfy a send request for a number of bytes less than or equal to its write threshold.

write threshold

Used by NetIPC processes that exchange data in asynchronous mode. A NetIPC process determines whether a VC socket is writeable by examining the socket's write threshold. A VC socket is considered writeable if it can immediately satisfy a send request for a number of bytes less than or equal to its write threshold.

X.25

A type of communication link used by NS-ARPA/1000 to provide connections to Packet Switching Networks (PSNs), also known as Value Added Networks (VANs). X.25 links are useful for long-distance communication, and can be more economical than leased lines in some applications. X.25 links have no store-and-forward or dynamic rerouting capabilities. X.25 links can have RTR LIs only and can be used for NS-ARPA and DS/1000-IV compatible services.

X.25 protocol

A protocol used at the Subnet or Intranet Layer (Layer 3s of the NS-ARPA/1000 architecture). X.25 defines some layer 3s functions for messages sent over X.25 links. X.25 networks define routing and store-and-forward features within Packet-Switching Networks.

Bibliography

NS and NS-ARPA/1000-Related Manuals:

NS-ARPA/1000 User/Programmer Reference Manual (91790-90020)
NS-ARPA/1000 Generation and Initialization Manual (91790-90030)
NS-ARPA/1000 Maintenance and Principles of Operation Manual (91790-90031)
NS-ARPA/1000 Quick Reference Guide (91790-90040)
NS-ARPA/1000 Error Message and Recovery Manual (91790-90045)
NS Message Formats Reference Manual (5958-8523)
NS Cross-System NFT Reference Manual (5958-8563)
NS-ARPA/1000 DS/1000-IV Compatible Services Manual (91790-90050)
NS-ARPA/1000 BSD IPC Programmer's Manual (91790-90060)
File Server Reference Guide for NS-ARPA/1000 and ARPA/1000 (91790-90054)

HP 9000 Manuals:

Using Network Services (B1012-90010)
Using ARPA Services (B1014-90006)

HP 3000 MPE V Manuals:

NS3000/V User/Programmer Reference Manual (32344-90001)
NetIPC 3000/V Programmer's Reference Manual (5958-8581)
NS3000/V Network Manager Reference Manual, Volume I (32344-90002)
NS3000/V Network Manager Reference Manual, Volume II (32344-90012)
NS3000/V Error Message and Recovery Manual (32344-90005)

HP 3000 MPE XL Manuals:

NS3000/XL Operations and Maintenance Reference Manual (36922-61005)
NetIPC 3000/XL Programmer's Reference Manual (36920-61005)
NS3000/XL Error Message Reference Manual (36923-61000)

DS and DS-Related Manuals:

DS/1000-IV User's Manual for RTE-A and RTE-6/VM (91750-90012)
DS/1000-IV Network Manager's Manual Generation and Initialization for RTE-A and RTE-6/VM (91750-90013)
DS/1000-IV Theory of Operation and Troubleshooting for RTE-A and RTE-6/VM (91750-90014)
DS/1000-IV Quick Reference Guide for RTE-A and RTE-6/VM (91750-90015)

X.25 Manuals:

DSN/X.25/1000 Reference Manual (91751-90002)
DSN/X.25/1000 Advanced Guide (91751-90003)
NS X.25 3000/V Link Guide (24405-90002)

RTE-A Manuals:

Getting Started with RTE-A (92077-90039)
RTE-A Driver Reference Manual (92077-90011)
RTE-A User's Manual (92077-90002)
RTE-A Print and Spooling Reference Manual (92077-90248)
RTE-A Backup and Disk Formatting Utilities Reference Manual (92077-90249)
RTE-A LINK User's Manual (92077-90035)
RTE-A Programmer's Reference Manual (92077-90007)
RTE-A System Generation and Installation Manual (92077-90034)

LAN/1000 Manuals:

HP 12076A LAN/1000 Link Local Area Network Interface Controller (LANIC) Installation Manual (12076-90001)
HP 12076A LAN/1000 Link Node Manager's Manual (12076-90002)
HP 12079A LAN/1000 Link Direct Driver Access Manual (12079-90001)

Index

Symbols

.. command, FTP, 3-19
! command, FTP, 3-17
? command
  FTP, 3-18, 3-42
  TELNET, 2-10
-g option, file name expansion, FTP, 3-2
-i option, interactive prompting, FTP, 3-2
-l option, log file, FTP, 3-2, 3-44
-n option, auto-logging, FTP, 3-2
-t option, command input file, FTP, 3-2, 3-74
-v option, verbose output, FTP, 3-2, 3-78
$CMNDO environment variable, 3-14
$CMNDO_LINK environment variable, 3-14
/ command, FTP, 3-20

A

AddOpt, 5-73, 6-5
  example, 6-20
  RPM, 6-19, 6-20
  RPM example, 6-34
AddOpt example, RPM, 6-37
AdrOf, 5-75
Advanced Research Projects Agency, ARPA, 1-3, 2-1
APPEND command, FTP, 3-22
Application layer, 1-2
ARPA, 1-3, 2-1
ASCII, DSCOPY option, 4-7
ASCII command, FTP, 3-23
assign partition, RPM, 6-24
asynchronous I/O, 5-14
  IPCRecv, 5-56
  IPCRecvCn, 5-61
  IPCSend, 5-68
  read and write thresholds, 5-14
ATACH, RPM, 6-5
AYT parameter, TELNET SEND command, 2-22

B

BELL command, FTP, 3-24
Berkeley Sockets, 1-4
BINARY, DSCOPY option, 4-7
BINARY command, FTP, 3-25
BISYNC, 1-7
block mode TELNET, 2-5
BREAK parameter, TELNET SEND command, 2-23
BSD IPC, 1-3, 1-4
BYE command, FTP, 3-27, 3-36, 3-62
byte addresses, 5-75

C

call socket descriptor, 5-3
CD command, FTP, 3-28
CDS
  RPM, 6-24, 6-39
  RPM programs, 6-3, 6-4
CDS program, RPM, 6-14
CDS programs, 1-8
chained TELNET sessions, 2-3, 2-20
checking the status of a connection, 5-8
checksumming, cross-system, 5-28, 5-29, 5-31
child program, RPM, 6-4, 6-21
CI programs, from FTP, 3-17
clients, maximum number, 5-80
CLOSE command
  FTP, 3-29
  TELNET, 2-11
CMNDO monitor, 3-14
command input file, FTP, 3-74
command stack, FTP, display with / command, 3-20
communication links, 1-1
  BISYNC, 1-7
  Ethernet, 1-7
  HDLC, 1-7
  IEEE 802.3, 1-7
  LAN, 1-7
  X.25, 1-7
computer network, 1-1
connections, 5-2
controlling programs, RPM, 6-9
copy descriptor, 4-6
creating a call socket, 5-4
cross-system
  checksumming, 5-28, 5-29, 5-31
  NetIPC, 5-25, 5-27, 5-29, 5-31
    HP 3000, 5-25, 5-26, 5-27, 5-29
    HP 9000, 5-25, 5-26, 5-27, 5-28
    PC, 5-25, 5-26, 5-27, 5-31
    program examples, 5-80, 5-83
  NetIPC calls, 5-26
  send and receive sizes, 5-28, 5-29, 5-31
  socket sharing, 5-28
  TCP protocol address, 5-28, 5-29, 5-31
cross-system NetIPC
  HP 3000, 5-1, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
  HP 9000, 5-1, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
  PC, 5-1, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
cross-sytem, NetIPC, 5-28

D

data interpretation, 4-4
Data Link layer, 1-2
data parameter, 5-17, 5-21
  byte address manipulation, 5-22
  data buffer, 5-23
  obtaining byte address, 5-75
  type coercion, 5-23
  vectored data, 5-23
data partition
  modify, 6-29
  RPM, 6-29
data vector, 5-21
DATA_WAIT, 5-56
DATA_WAIT flag, 5-53
DEBUG command, FTP, 3-30
DELETE command, FTP, 3-31
deleting directories, FTP
  with DELETE command, 3-31
  with MDELETE command, 3-47
deleting files, FTP
  with DELETE command, 3-31
  with MDELETE command, 3-47
dependent child, RPM, 6-12
dependent RPM programs
  remote process management, 6-15
  terminating, 6-15
descriptors, 5-3
  releasing, 5-10
  resources associated with, 5-10
detecting connection requests, 5-65
DEXEC, 1-4
  RPM, 6-2
DIR command, FTP, 3-32
directories
  accessing parent directory in FTP, 3-19
  creating remote directories in FTP, 3-51
  current working directory in FTP, 3-61
  deleting in FTP, 3-31
    multiple directories, 3-47
  listing directory in FTP, 3-45, 3-56
  listing in FTP, 3-32, 3-34, 3-48, 3-52
  removing in FTP, 3-67
  renaming in FTP, 3-66
  RTE-A, 1-10, 3-8
  setting working directory in FTP, 3-28, 3-43
distributed executive, 1-4
DL command, FTP, 3-34
domain, in node names, 1-8
DS/1000 compatible user services, 1-3
DS/1000-IV compatible services
  program-to-program communication, 1-4
  REMAT, 1-4
  remote file access, 1-4
  RMOTE, 1-4
  RTE-MPE, 1-3
  RTE-RTE, 1-3
DSCOPY, 4-5
  breakmode commands, 4-12
    abort, 4-12
    cancel, 4-12
    help, 4-12
    status, 4-12
  case sensitivity, 4-10
  CI return variable, 4-5
  coding in Pascal, 4-28
  commands, 4-15
  copy descriptor, 4-6
  examples, 4-13
  file masks, 4-11
  file names and logons, 4-11
  interrupting, 4-12
  line continuation, 4-10
  logons and VC+, 4-10
  P-globals, 4-5
  programmatic call, 4-28
  protection mode and update time, 4-10
  RTE file names and logons, 4-10
  used interactively, 4-5, 4-10
  used programmatically, 4-27
DSCOPY commands
  ? (HELP), 4-26
  +CLEAR, 4-16
  +DEFAULT, 4-17
  +ECHO command, 4-19
  +EX, 4-20
  +LL, 4-21
  +RU, 4-22
  +SHOW, 4-23
  +TRANSFER, 4-24
  +WD, 4-25
DSCOPY option
  ASCII, 4-7
  BINARY, 4-7
  FIXED, 4-7
  INTERCHANGE, 4-8
  MOVE, 4-9
  OVER, 4-9
  QUIET, 4-9
  REPLACE, 4-9
  RSIZE, 4-8
  SILENT, 4-9
  SIZE, 4-8
  STRIP, 4-8
  VARIABLE, 4-8
DSCOPYBUILD, 4-29
DTACH, RPM, 6-5

E

end-to-end communication, 1-2
environment variable
  $CMNDO, 3-14
  $CMNDO_LINK, 3-14
ESCAPE command, TELNET, 2-12

ESCAPE parameter, TELNET SEND command, 2-22
establishing a connection, 5-4, 5-53, 5-55
Ethernet, 1-7
example, RPM, 6-40
examples, remote process management, 6-42
exception selecting, 5-65
EXEC, RPM, 6-2, 6-4, 6-33
EXIT command
  FTP, 3-27, 3-36, 3-62
  TELNET, 2-14

F

file copying formats
  interchange format, 4-3
  transparent format, 4-3
file descriptor parameters, 3-10
file masks, 4-11
  used to copy groups of files, 4-11
  used to create target file names, 4-11
file name globbing, 3-39
file transfer, 3-1
  FTP
    local to remote, 3-60, 3-69
      multiple files, 3-54
    remote to local, 3-38, 3-64
      multiple files, 3-49
file transfer protocol, 3-1
files
  deleting in FTP, 3-31
    multiple files, 3-47
  file name globbing, 3-39
  from FMGR cartridge, 1-10, 3-11
  record length specification, 1-11, 3-9
  renaming in FTP, 3-66
  RTE-A, 1-10, 3-8
  size specification, 1-11, 3-9
  transferring in FTP, 3-38, 3-60, 3-64, 3-69
    multiple files, 3-49, 3-54
  type specification, 1-11, 3-9
FIXED, DSCOPY option, 4-7
flags parameter, 5-17, 6-5
  RPM, 6-5, 6-11, 6-16, 6-33
FMGR format
  namr, 1-11
  namr syntax, 1-11
FORM command, FTP, 3-37
FORTRAN 77, NetIPC, 5-83
FTP, 1-3, 3-1
  .. command, 3-19
  ! command, 3-17
  ? command, 3-18, 3-42
  / command, 3-20
  APPEND command, 3-22
  ASCII command, 3-23
  BELL command, 3-24
  BINARY command, 3-25
  BYE command, 3-27, 3-36, 3-62
  CD command, 3-28
  CLOSE command, 3-29
  closing connection, 3-29, 3-62
  command input file, 3-74
  command stack, display with / command, 3-20
  commands, 3-1, 3-15
    .., 3-19
    !, 3-17
    ?, 3-18, 3-42
    /, 3-20
    APPEND, 3-22
    ASCII, 3-23
    BELL, 3-24
    BINARY, 3-25
    BYE, 3-27, 3-36, 3-62
    CD, 3-28
    CLOSE, 3-29
    DEBUG, 3-30
    DELETE, 3-31
    DIR, 3-32
    DL, 3-34
    EXIT, 3-27, 3-36, 3-62
    FORM, 3-37
    GET, 3-38, 3-64
    GLOB, 3-39
    HASH, 3-41
    HELP, 3-42
    LCD, 3-43
    LL, 3-44
    LS, 3-45
    MDELETE, 3-47
    MDIR, 3-48
    MGET, 3-49
    MKDIR, 3-51
    MLS, 3-52
    MODE, 3-53
    MPUT, 3-54
    NLIST, 3-56
    OPEN, 3-58
    PROMPT, 3-59
    PUT, 3-60, 3-69
    PWD, 3-61
    QUIT, 3-27, 3-36, 3-62
    QUOTE, 3-63
    RECV, 3-38, 3-64
    REMOTEHELP, 3-65
    RENAME, 3-66
    RMDIR, 3-67
    RTEBIN, 3-68
    SEND, 3-60, 3-69
    SITE, 3-70
    STATUS, 3-71
    STRUCT, 3-72
    SYSTEM, 3-73
    TR, 3-74
    TYPE, 3-76
    USER, 3-77
    VERBOSE, 3-78
  DEBUG command, 3-30
  DELETE command, 3-31
  DIR command, 3-32

DL command, 3-34
EXIT command, 3-27, 3-36, 3-62
file transfer, 3-38
FORM command, 3-37
GET command, 3-38, 3-64
GLOB command, 3-39
HASH command, 3-41
HELP command, 3-42
help information, 3-8, 3-18, 3-42
invoking, 3-2
LCD command, 3-43
LL command, 3-44
log file, 3-44
LS command, 3-45
MDELETE command, 3-47
MDIR command, 3-48
MGET command, 3-49
MKDIR command, 3-51
MLS command, 3-52
MODE command, 3-53
MPUT command, 3-54
NLIST command, 3-56
OPEN command, 3-58
opening connection to remote host, 3-58
operation, 3-5
PROMPT command, 3-59
PUT command, 3-60, 3-69
PWD command, 3-61
QUIT command, 3-27, 3-36, 3-62
QUOTE command, 3-63
RECV command, 3-38, 3-64
REMOTEHELP command, 3-65
RENAME command, 3-66
RMDIR command, 3-67
RTEBIN command, 3-68
SEND command, 3-60, 3-69
SITE command, 3-70
STATUS command, 3-71
status information, 3-71
STRUCT command, 3-72
SYSTEM command, 3-73
terminating, 3-7, 3-27, 3-36
TR command, 3-74
transferring files, 3-11
TYPE command, 3-76
USER command, 3-77
VERBOSE command, 3-78
verbose output, 3-78
FTP sample session, 3-5

G

gathered write, 5-21
GET command, FTP, 3-38, 3-64
GLOB command, FTP, 3-39
globbing, FTP, 3-3

H

HASH command, FTP, 3-41
HDLC, 1-7
HELP command
FTP, 3-42
TELNET, 2-15
high throughput, 5-67
host names, 1-8
HP 1000 communication
file transfer, 3-1
FTP, 3-1
TELNET, 2-1
virtual terminal, 2-1
HP 3000, 5-25
NetIPC, 5-1, 5-25, 5-26, 5-27, 5-29, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
HP 9000, 5-25
NetIPC, 5-1, 5-25, 5-26, 5-27, 5-28, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
HP 9000 communication
file transfer, 3-1
FTP, 3-1
TELNET, 2-1
virtual terminal, 2-1

I

ID segment, RPM, 6-23, 6-25
IEEE 802.3, 1-7
independent RPM programs
remote process management, 6-15
terminating, 6-15
information utility, NSINF, 1-9
InitOpt, 5-77, 6-5
example, 6-20
RPM, 6-20
input file, FTP, 3-74
interactive network file transfer, 4-5
INTERCHANGE, DSCOPY option, 4-8
interchange format, 4-3
RTE-A type 6 files, 4-3, 4-7, 4-8
International Standards Organization, 1-1
interprocess communication
See also Network IPC
PTOP, 5-1
INTERRUPT parameter, TELNET SEND command, 2-22
INTERUPT command, TELNET, 2-16
IP, internet protocol, 1-9
IP addresses, 1-9
IP parameter, TELNET SEND command, 2-23
IPC. *See* Network IPC
IPCConnect, 5-35
cross-system, 5-37
IPCControl, 5-38
IPCCreate, 5-40
cross-system, 5-41
IPCDest, 5-42
cross-system, 5-44
IPCGet, 5-45

IPCGive, 5-46
IPCLookUp, 5-48
  race condition, 5-49
IPCName, 5-50
  naming path report descriptors, 5-51
  randomly generated names, 5-50
IPCNamErase, 5-52
IPCRecv, 5-53
  asynchronous I/O, 5-56
  cross-system, 5-59
  establishing a connection, 5-53, 5-55
  normal reading, 5-55
  preview reading, 5-55
  receiving data, 5-55
  scattered reading, 5-55
  synchronous I/O, 5-56
  waiting for data, 5-53
IPCRecvCn, 5-60
  cross-system, 5-62
  synchronous vs. asynchronous I/O, 5-61
IPCSelect, 5-63
  called in FORTRAN program, 5-66
  called in Pascal program, 5-66
  detecting connection requests, 5-65
  example, 5-81
  exception selecting, 5-65
  exceptional sockets, 5-64
  performing a read select, 5-65
  performing a write select, 5-65
  readable sockets, 5-64
  writeable sockets, 5-64
IPCSend, 5-67
  cross-system, 5-68
  high throughput, 5-67
  synchronous vs. asynchronous I/O, 5-68
IPCShutDown, 5-70
  cross-system, 5-71
  releasing a call socket, 5-70
  releasing a path report descriptor, 5-71
  releasing a VC socket descriptor, 5-71
ISO OSI model, 1-1

L

LAN, 1-7
layers, 1-1
LCD command, FTP, 3-43
levels, 1-1
LINK, command editing, 3-14
links, 1-1, 1-7
LL command, FTP, 3-44
loading, RPM, 6-4
local area network, 1-7
log file, FTP, 3-2, 3-44
login, RPM, 6-10
looking up a call socket name, 5-6
LS command, FTP, 3-45

M

MDELETE command, FTP, 3-47
MDIR command, FTP, 3-48
MGET command, FTP, 3-49
MIL-STD-1782, 1-3
MKDIR command, FTP, 3-51
MLS command, FTP, 3-52
MODE command
  FTP, 3-53
  TELNET, 2-18
modify code partition, RPM, 6-28
modify data partition, RPM, 6-29
modify VMA size, 6-27
MOVE, DSCOPY option, 4-9
MPE-V, 5-25
MPE-XL, 5-25
MPUT command, FTP, 3-54

N

naming a call socket, 5-5
namr, 1-11
  syntax, 1-11
NetIPC, 1-4
  See also Network IPC
  cross-system, 5-25, 5-27, 5-28, 5-29, 5-31
  cross-system calls, 5-26
  IPCControl, 5-38
  RPM, 6-2, 6-5, 6-12
NetIPC calls
  AddOpt, 5-73
  AdrOf, 5-75
  example of use, 5-80, 5-83, 5-122
  InitOpt, 5-77
  IPCCreate, 5-40
  IPCDest, 5-42
  IPCGet, 5-45
  IPCGive, 5-46
  IPCLookUp, 5-48
  IPCName, 5-50
  IPCNamErase, 5-52
  IPCRecv, 5-53
  IPCRecvCn, 5-60
  IPCSelect, 5-63
  IPCSend, 5-67
  IPCShutDown, 5-70
  ReadOpt, 5-79
  RPM, 6-5
  special, 5-72
NetIPC common parameters, 5-17
NetIPC syntax conventions, 5-34
network address, 1-9
network architecture, 1-1
network file transfer, 1-3, 1-4, 4-1
  ? (HELP) command, 4-26
  +CLEAR, 4-16
  +DEFAULT, 4-17
  +ECHO, 4-19
  +EX command, 4-20
  +LL command, 4-21

+RU command, 4-22
+SHOW command, 4-23
+TRANSFER command, 4-24
+WD command, 4-25
copy descriptor, 4-6
DS/1000-IV files, 4-14
DSCOPY commands, 4-15
DSCOPYBUILD, 4-29
features, 4-1
file copying formats, 4-3
file names and logons, 4-11
interactive, 4-5
optimizing performance, 4-13
programmatic, 4-27
programmatic examples, 4-32
RTE-A type 6 files, 4-3, 4-7, 4-8
running DSCOPY, 4-5
three-node model, 4-2
using file masks, 4-11
network interprocess communication, 1-3
Network IPC
asynchronous, 5-14
call socket descriptor, 5-3
call summary, 5-13
calls, 5-35
checking connection status, 5-8
client example, 5-82
common parameters, 5-17
connection dialogue, 5-4
connections, 5-2, 5-4
creating a call socket, 5-4
cross-system, 5-1, 5-25, 5-27, 5-28, 5-29, 5-31, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71, 5-83
HP 3000, 5-27
HP 9000, 5-27
PC, 5-27
data parameter, 5-21
descriptors, 5-3
detecting connection requests, 5-65
establishing a connection, 5-55
exception selecting, 5-65
flags parameter, 5-17
FORTRAN 77, 5-18, 5-83
high throughput, 5-67
HP 3000, 5-1, 5-25, 5-26, 5-29, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
HP 9000, 5-1, 5-25, 5-26, 5-28, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
IPCConnect, 5-35
looking up call socket name, 5-6
maximum number of clients, 5-80
maximum number of sockets, 5-26
naming a call socket, 5-5
opt parameter, 5-19
Pascal, 5-18, 5-34, 5-83
path report descriptor, 5-3
path reports, 5-3
PC, 5-1, 5-25, 5-26, 5-31, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
performing a read select, 5-65
performing a write select, 5-65
porting programs, 5-1, 5-25
program examples, 5-80, 5-83, 5-122
PTOP, 5-1
read and write thresholds, 5-14
receiving a connection request, 5-7
receiving data, 5-55
requesting a connection, 5-6
result parameter, 5-23
scheduling remote process, 5-4
sending and receiving data, 5-10
server example, 5-80, 5-81
shutting down a connection, 5-10
socket names, 5-3
socket registry, 5-3
sockets, 5-2
stream mode, 5-16
synchronous, 5-14
syntax conventions, 5-34
telephone analogy, 5-2
timing and time-outs, 5-11
VC socket descriptor, 5-3
network IPC, 1-4
Network layer, 1-2
network management services and features, 1-7
NFT, 1-4
See also network file transfer
NLIST command, FTP, 3-56
node address, 1-9
node names, 1-8, 1-9
domain, 1-8
organization, 1-8
syntax, 5-23
nodename
RPM, 6-7, 6-10
RPMKill, 6-41
nodename parameter, 5-17
RPM, 6-5
nodes, 1-1
NS common services, remote process management, 6-1
NS-ARPA common services, 1-3
BSD IPC, 1-3
FTP, 1-3
network file transfer, 1-3
network interprocess communication, 1-3
remote process management, 1-3
TELNET, 1-3
NS-ARPA/1000 user services, 1-3
NSINF, 1-9

O

OPEN command
FTP, 3-58
TELNET, 2-19
Open Systems Interconnection model, 1-1
opt array, RPM options, 6-13
opt parameter, 5-19
adding an argument, 5-73

initialization, 5-77
obtaining option code and data, 5-79
OPTARGUMENTS structure, 5-21
RPM, 6-5, 6-12
structure, 5-20
opt parameters, 5-17
option groups, RPM, 6-19
options
RPM, 6-13, 6-18
RPMCreate, 6-18
options groups, RPM, 6-18
organization, in node names, 1-8
OSI model, 1-1
Application layer, 1-2
Data Link layer, 1-2
Network layer, 1-2
Physical layer, 1-2
Presentation layer, 1-2
Session layer, 1-2
Transport layer, 1-2
OVER, DSCOPY option, 4-9

P

parameters, RPM, 6-5
parent directory, accessing in FTP, 3-19
parent program
RPM, 6-4, 6-14
session-sharing, 6-16
partition
reserved, 6-24
RPM, 6-24
Pascal, NetIPC, 5-83
passing strings, RPM, 6-21
path report descriptor, 5-3
path reports, 5-3
PC, 5-25
NetIPC, 5-1, 5-25, 5-26, 5-27, 5-31, 5-37, 5-41, 5-44, 5-59, 5-62, 5-68, 5-71
performing a read select, 5-65
performing a write select, 5-65
Physical layer, 1-2
porting NetIPC programs, 5-1, 5-25
Presentation layer, 1-2
process communication. *See* Network IPC
program descriptor, RPM, 6-7, 6-12, 6-14, 6-41
program name, RPM, 6-10
program priority, 1-8
RPM, 6-25
program scheduling, RPM, 6-33
program-to-program communication, 1-4
PROMPT command, FTP, 3-59
protocol
FTP, 1-3
TELNET, 1-3, 2-1
protocols, 1-1
PTOP, 1-4
interprocess communication, 5-1
PUT command, FTP, 3-60
PWD command, FTP, 3-61

Q

queue program scheduling, RPM, 6-36
QUIET, DSCOPY option, 4-9
QUIT command
FTP, 3-27, 3-36, 3-62
TELNET, 2-14, 2-20
QUOTE command, FTP, 3-63

R

race condition, 5-11, 5-49
read and write thresholds, 5-14
ReadOpt, 5-79, 6-5
receiving a connection request, 5-7
receiving data, 5-55
RECV command, FTP, 3-38, 3-64
REMAT, 1-4
remote file access, 1-4
remote process management, 1-3, 1-4, 5-17, 6-1
dependent programs, 6-15
example, 6-42
flags parameter, 6-5
independent programs, 6-15
REMOTEHELP command, FTP, 3-65
RENAME command, FTP, 3-66
REPLACE, DSCOPY option, 4-9
request code, RPM, 6-7
requesting a connection, 5-6
resource sharing, 1-1
restore program, 6-23
result parameter, 5-17, 5-23
RPM, 6-5
RFA, 1-4
RMDIR command, FTP, 3-67
RMOTE, 1-4
RPM, 1-4
See also remote process management
AddOpt, 6-5, 6-19, 6-20
AddOpt example, 6-34, 6-37, 6-40
assign partition, 6-24
ATACH, 6-5
CDS, 6-3, 6-4, 6-24, 6-39
CDS program, 6-14
child program, 6-1, 6-4, 6-21
controlling programs, 6-9
definition, 6-1
dependent child, 6-12
DEXEC, 6-2
DTACH, 6-5
EXEC, 6-2, 6-4, 6-33
flags parameter, 6-5, 6-11, 6-16, 6-33
ID segment, 6-23, 6-25
InitOpt, 6-5, 6-20
login, 6-10
modify code partition, 6-28
modify data partition, 6-29
NetIPC, 6-2, 6-5, 6-12
NetIPC calls, 6-5
nodename, 6-7, 6-10
nodename parameter, 6-5

- opt parameter, 6-12
  - option, 6-24
  - option groups, 6-18, 6-19
  - options, 6-13, 6-23
  - parameters, 6-5
  - parent program, 6-1, 6-4, 6-14
  - passing strings, 6-21
  - program descriptor, 6-7, 6-12, 6-14, 6-41
  - program name, 6-10
  - program priority, 6-25
  - program scheduling, 6-33
  - queue program scheduling, 6-36
  - ReadOpt, 6-5
  - request code, 6-7
  - restore program, 6-23
  - result parameter, 6-5
  - RPMControl, 6-3, 6-7
  - RPMCreate, 6-3, 6-22
  - RPMCreate options, 6-13
  - RPMGetString, 6-3, 6-21
  - RPMKill, 6-3, 6-41
  - RTE resources, 6-4
  - schedule with wait, 6-18
  - scheduling programs, 6-13
  - sending strings, 6-14
  - session, 6-13
  - session-sharing, 6-11, 6-16
  - stack size, 6-4
  - summary of calls, 6-3
  - syntax conventions, 6-5
  - terminate a program, 6-4, 6-14, 6-41
  - terminating programs, 6-14
  - time scheduling, 6-31
  - VMA size, 6-27
  - wait for child, 6-11
- RPMControl, 6-4
- RPMCreate, 6-4, 6-10
  - options, 6-13
- RPMCreate options, 6-18
- RPMGetString, 6-21, 6-39
- RPMKill, 6-4, 6-41
  - nodename, 6-41
- RSIZE, DSCOPY option, 4-8
- RTE resources, RPM, 6-4
- RTE-A type 6 files, 4-3, 4-7, 4-8
- RTEBIN command, FTP, 3-68
- RUN command, TELNET, 2-21

S

- scattered read, 5-21
- schedule with wait, RPM, 6-18
- scheduling program, RPM, 6-2
- scheduling programs, RPM, 6-13
- send and receive sizes, cross-system, 5-28, 5-29, 5-31
- SEND command
  - FTP, 3-60, 3-69
  - TELNET, 2-22
- sending and receiving data, 5-10
  - stream mode, 5-16
- session, RPM, 6-13
- Session layer, 1-2
- session-sharing
  - parent program, 6-16
  - RPM, 6-11, 6-16
- shutting down a connection, 5-10
- SILENT, DSCOPY option, 4-9
- SITE command, FTP, 3-70
- SIZE, DSCOPY option, 4-8
- socket modes, 5-14
- socket names, 5-3
  - syntax, 5-23
- socket registry, 5-3
- socket sharing, cross-system, 5-28
- socketname parameter, 5-17
- sockets
  - call, 5-5
  - description, 5-2
  - detecting connection requests, 5-65
  - exception selecting, 5-65
  - exceptional, 5-64
  - maximum number, 5-26
  - naming, 5-5
  - performing a read select, 5-65
  - performing a write select, 5-65
  - readable, 5-64
  - shutting down a call socket, 5-70
  - shutting down a path report descriptor, 5-71
  - shutting down a VC socket, 5-71
  - synchronous and asynchronous, 5-14
  - virtual circuit, 5-6
  - writeable, 5-64
- special NetIPC calls, 5-72
- stack size, RPM, 6-4
- STATUS command
  - FTP, 3-71
  - TELNET, 2-24
- stream mode, 5-16
- strings
  - passing, 6-39
  - RPM, 6-14, 6-39
- STRIP, DSCOPY option, 4-8
- STRUCT command, FTP, 3-72
- summary of NetIPC calls, 5-13
- supported connectivities, 1-5
- synchronous and asynchronous socket modes, 5-14
- synchronous I/O, 5-14
  - IPCRecv, 5-56
- synchronous time-out, 5-11
- syntax, RPM, 6-5
- SYSTEM command, FTP, 3-73

T

TCP. *See* transmission control protocol
TCP protocol address, cross-system, 5-28, 5-29, 5-31
TELNET, 1-3, 2-1
  ? command, 2-10
  block mode, 2-5
  CLOSE command, 2-11
  commands, 2-9
  connectivity considerations, 2-2
  ESCAPE command, 2-12
  EXIT command, 2-14
  HELP command, 2-15
  INTERUPT command, 2-16
  MODE command, 2-18
  OPEN command, 2-19
  QUIT command, 2-14, 2-20
  RUN command, 2-21
  SEND command, 2-22
  STATUS command, 2-24
  using, 2-6
TELNET commands, 2-9
  ?, 2-10
  CLOSE, 2-11
  ESCAPE, 2-12
  EXIT, 2-14
  HELP, 2-15
  INTERUPT, 2-16
  MODE, 2-18
  OPEN, 2-19
  QUIT, 2-14, 2-20
  RUN, 2-21
  SEND, 2-22
  STATUS, 2-24
TELNET operation, 2-8
TELNET protocol, 2-1
TELNET SEND commands
  AYT, 2-22
  BREAK, 2-22
  ESCAPE, 2-22
  INTERRUPT, 2-22
  IP, 2-22
terminal settings, 2-2
  DEC VAX computers, 2-3
terminate a program
  RPM, 6-4, 6-14, 6-41
  RPMKill, 6-4, 6-14
terminating programs, RPM, 6-14
terminating RPM programs
  dependent programs, 6-15
  independent programs, 6-15
terminiating FTP, temporarily, 3-17
three-node model, 4-2
  consumer, 4-2
  initiator, 4-2
  producer, 4-2
time scheduling, RPM, 6-31
timing and time-outs, 5-11
TR command, FTP, 3-74
  transfer file, 3-75
transfer file, FTP, 3-2, 3-74, 3-75
transmission control protocol, 5-2, 5-16
transparent format, 4-3
Transport layer, 1-2
TYPE command, FTP, 3-76

U

USER command, FTP, 3-77
user services
  distributed executive, 1-4
  FTP, 1-3
  network file transfer, 1-4
  network IPC, 1-4
  program-to-program communication, 1-4
  REMAT, 1-4
  remote file access, 1-4
  remote process management, 1-4
  RMOTE, 1-4
  TELNET, 1-3
  utility subroutines, 1-4
utility subroutines, 1-4

V

VARIABLE, DSCOPY option, 4-8
VC socket, 5-2, 5-6
VC socket descriptor, 5-3
vectored data, 5-22
VERBOSE command, FTP, 3-78
verbose output, FTP, 3-78
virtual circuit, 5-2
virtual circuit connection, 5-2, 5-6
  status, 5-8
virtual circuit socket, 5-2
virtual circuit socket descriptor, 5-3
virtual terminal, 1-3, 2-1
  TELNET, 1-3
VMA programs, RPM, 6-26
VMA size, RPM, 6-27

W

wait for child, RPM, 6-11
wild cards, FTP, 3-39
working directory
  FTP, setting with CD command, 3-28
  RPM, 6-22
  RPMCreate option, 6-22
working set size
  modify, 6-26
  RPM, 6-26
write thresholds, 5-14

X

X.25, 1-7