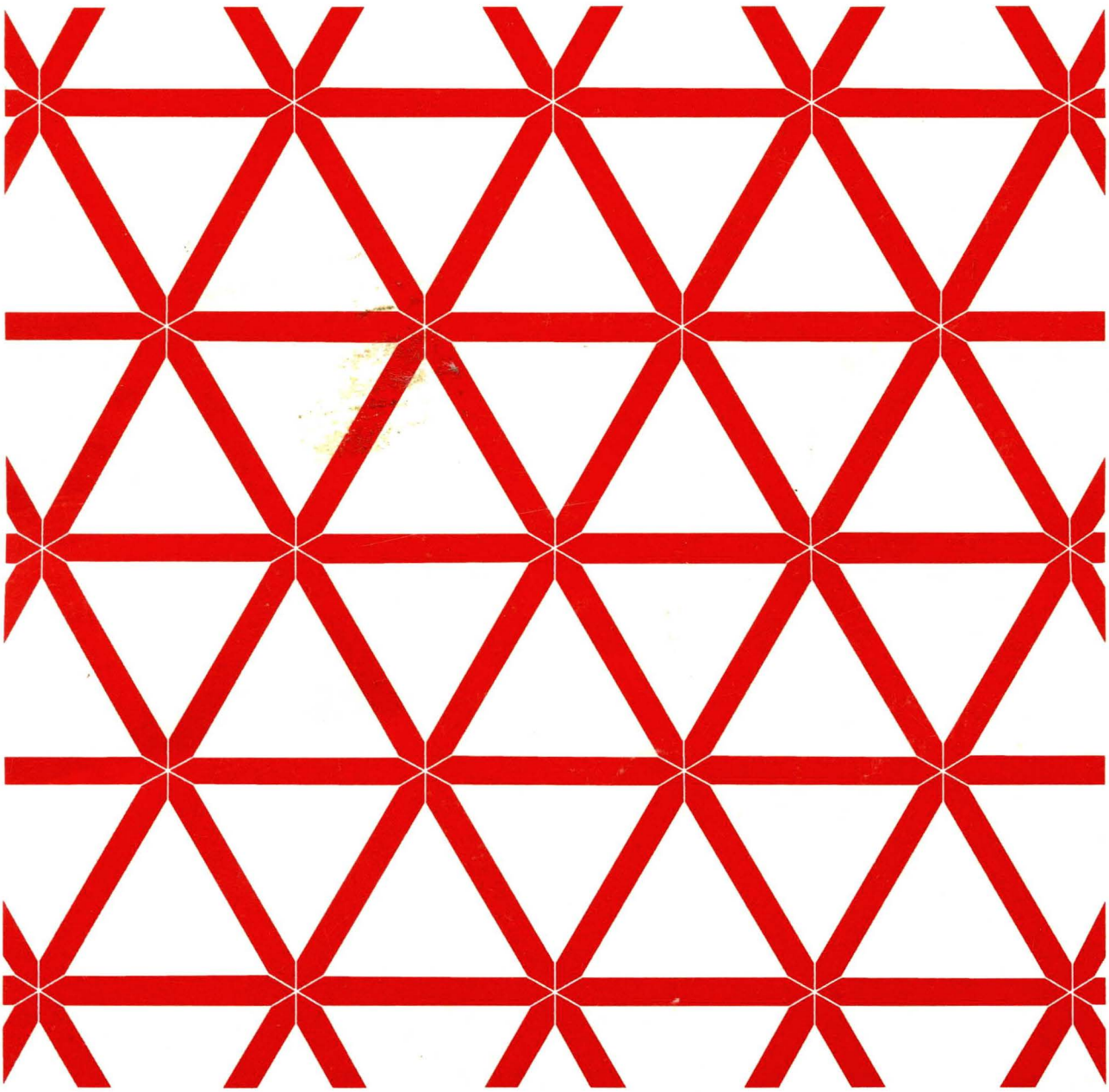
IBM

SC09-1261-00

Transmission Control Protocol/ Internet Protocol for MVS

Programmer's Reference



Transmission Control Protocol/ Internet Protocol for MVS

SC09-1261-00

Programmer's Reference

First Edition (June 1989)

This edition applies to the Transmission Control Protocol/Internet Protocol for MVS Licensed Program (Program Number 5685-061), and to all subsequent releases and modifications until otherwise indicated in new editions. Changes are periodically made to the information herein; any such changes will be reported in subsequent revisions.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

Note to US Government Users: Documentation related to restricted rights. Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to:

IBM Canada Ltd.,
Information Development,
Department 849,
1150 Eglinton Avenue East,
North York, Ontario, Canada.
M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

This book describes the programming interfaces for the IBM Transmission Control Protocol/Internet Protocol for MVS (TCP/IP for MVS) program. These interfaces allow you to develop application programs that use the services of TCP/IP to enable MVS and non-MVS hosts to communicate across an internetwork (internet). However, this book also provides the following types of information, which are explicitly identified where they occur:

Product-Sensitive Programming Interfaces

Installation exits and other "product-sensitive" interfaces are provided to allow the customer installation to perform tasks such as product tailoring, monitoring, modification, or diagnosis. They are dependent on the detailed design or implementation of TCP/IP for MVS. Such interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new TCP/IP for MVS releases or versions, or as a result of maintenance. The product-sensitive interfaces provided by TCP/IP for MVS are as follows:

All product-sensitive programming interfaces for TCP/IP for MVS are described in this book, *IBM Transmission Control Protocol/Internet Protocol for MVS: Programmer's Reference*, SC09-1261.

Internal Product Information

Internal product information is provided as additional guidance on writing application programs that use the layers of the TCP/IP protocol suite. This internal information should never be used as programming interface information.

Identified here are those interfaces provided by TCP/IP for MVS by which a customer-written program is to request or receive functions or services of TCP/IP for MVS:

- Chapter 2, TCP/UDP/IP API (Pascal Language)
- Chapter 3, C Socket Application Program Interface
- Chapter 4, X-Windows Interface
- Chapter 5, Remote Procedure Calls
- Appendix A, VMCF Interface
- Appendix B, Interface to SMTP Address Space

  Note: The above application programming interfaces (API) are available for use by only one task per address space.
- Appendix D, Network File System Server Exit Routines.

Macros

This section lists the macros which are intended to be used as, or as part of, a programming interface for customers.

Distribution Library Name	Macro Name
TCPIP.COMMMAC	auth_destroy
TCPIP.COMMMAC	clnt_call
TCPIP.COMMMAC	clnt_freeres
TCPIP.COMMMAC	clnt_geterr
TCPIP.COMMMAC	svc_destroy
TCPIP.COMMMAC	svc_freeargs
TCPIP.COMMMAC	svc_getcaller
TCPIP.COMMMAC	xdr_inline
TCPIP.COMMMAC	xdr_enum

Note: Some programming interfaces include the use or specification of certain fields within an otherwise internal control block. You should use only those fields which are documented as part of the programming interface.

Throughout this book, the abbreviation MVS refers to the following IBM products:

- IBM Multiple Virtual Storage/System Product Version 1 Release 3.5 (MVS/370), or later
- IBM Multiple Virtual Storage/System Product Version 2 Release 1.3 (MVS/XA™), or later
- IBM Multiple Virtual Storage/System Product Version 3 Release 1.0 (MVS/ESA™), or later.

This book is intended for use by an experienced programmer familiar with MVS, the IBM Operating System/Multiple Virtual Storage (OS/MVS) commands, and the TCP/IP protocols.

The glossary at the end of this book defines the most commonly used terms in a TCP/IP internet environment.

MVS/XA and MVS/ESA are trademarks of the International Business Machines Corporation.

What This Book Contains

This book contains the following information:

- Chapter 1, "Computer Networks and TCP/IP Protocols" gives an overview of TCP/IP networks and the TCP/IP set of protocols.
- Chapter 2, "TCP/UDP/IP API (Pascal Language)" describes the functions of the Pascal Language Application Program Interface (API) used when writing application programs directly to the Transmission Control Protocol (TCP), User Datagram Protocol (UDP), and Internet Protocol (IP) boundaries.
- Chapter 3, "C Socket Application Program Interface" describes the functions of the C Language Socket API used when writing application programs directly to the TCP and UDP protocol boundaries.
- Chapter 4, "X-Windows Interface" describes the functions of the X-Windows Interface used when writing application programs directly to the X Window System protocol boundary.
- Chapter 5, "Remote Procedure Calls" describes the functions of the Remote Procedure Call (RPC) Interface used when writing application programs directly to the RPC protocol boundary.
- Appendix A, "VMCF Interface" describes how to communicate directly with TCP/IP using Virtual Machine Communication Facility (VMCF) calls.
- Appendix B, "Interface to SMTP Address Space" describes the interface to the Simple Mail Transfer Protocol (SMTP) address space.
- Appendix C, "Assembler Calls for the Pascal API" describes the assembler calls used with the Pascal API.
- Appendix D, "Network File System Server Exit Routines" describes the Network File System server exit routines.
- Appendix E, "Sample Programs" contains sample application programs illustrating the Pascal, C Socket and X-Windows APIs.
- Appendix F, "Related Protocol Specifications" lists the publications that are used as the protocol specifications.

How to Read the Syntax Diagrams

This book presents the command descriptions using "railroad track" syntax diagrams. The structure of these diagrams is defined below.

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

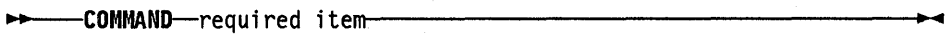
  The ►►— symbol indicates the beginning of a statement.

  The —► symbol indicates that the statement syntax is continued on the next line.

  The ►— symbol indicates that a statement is continued from the previous line.

  The —►◄ symbol indicates the end of a statement.

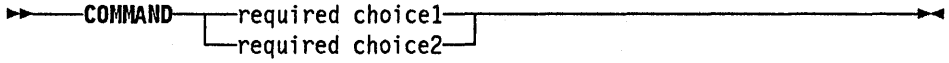
- Required items appear on the horizontal line (the main path).



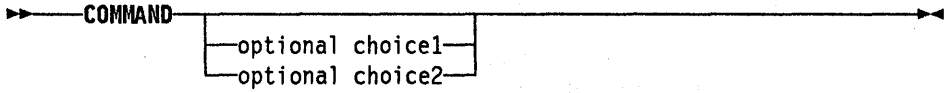
- Optional items appear below the main path.

- If you can choose from two or more items, they appear vertically, in a stack.

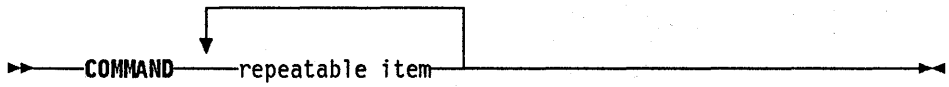
  If you *must* choose one of the items, one item of the stack appears on the main path.



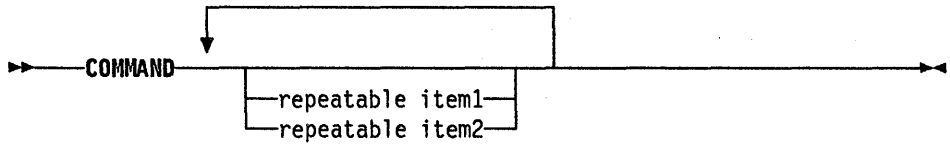
  If all items are optional, the entire stack appears below the main path.



- An arrow returning to the left above the main line indicates an item that can be repeated.



- A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

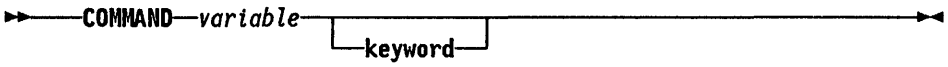


- Commands are shown in **BOLD UPPERCASE** letters in the syntax. They are not case sensitive.

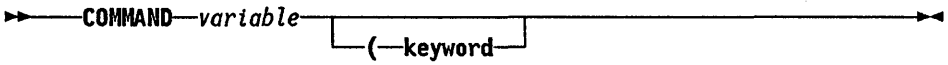
  Note: Network File System commands must be issued in lowercase at the client. Because of this restriction, they are shown in **bold lowercase**.

  Keywords are shown in **bold lowercase** letters in the syntax. They should be entered exactly as shown, but are not case sensitive.

  Variables appear in *italic lowercase* letters in the syntax. They represent user-supplied names or values.



- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.



A Note on Data Set Names

In this book, all data set names are assumed to have a high-level qualifier of TCPIP (this is the default condition). See your local System Administrator if your TCP/IP for MVS installation has implemented a different high-level qualifier. The high-level qualifier for TCP/IP for MVS's libraries may be customized at time of installation (refer to *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance*).

Recommended Reading

For a concise overview and introduction to internetworking with TCP/IP, you can refer to:

Internetworking With TCP/IP: Principles, Protocols, and Architecture. Douglas Comer. Prentice Hall. (New Jersey, 1988). You can obtain this book from IBM by ordering publication number SC09-1302

Introducing IBM's Transmission Control Protocol/Internet Protocol Products, GC09-1307.

Publications Referenced in This Book

At times, this book references other books to find more information. Following is a list of the books referenced:

IBM AIX Operating System Technical Reference: System Calls and Subroutines, SC23-2125

IBM AIX X-Windows Programmer's Reference, SC23-2118

IBM AIX X-Windows User's Guide, SC23-2017

IBM VM/SP System Facilities for Programming, SC24-5288

Networking on the Sun Workstation: Remote Procedure Call Programming Guide, SUN Microsystems.

Other Books That You May Reference

In addition to the books referenced in this book, you may want to consult other books in the TCP/IP library. Following is a list of other books in the library:

IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance, SC09-1256

IBM Transmission Control Protocol/Internet Protocol for MVS: User's Guide, SC09-1255

IBM Transmission Control Protocol/Internet Protocol for VM: Command Reference Manual, GC09-1204

IBM Transmission Control Protocol/Internet Protocol for VM: Installation and Maintenance Manual, GC09-1203

IBM Transmission Control Protocol/Internet Protocol for VM: Network File System and Remote Procedure Call Manual, SC09-1274

IBM Transmission Control Protocol/Internet Protocol for VM: Programmer's Manual, GC09-1206

IBM Transmission Control Protocol for the Personal System/2 Computer: Command Reference and Installation Manual, SC09-1270

IBM AIX PS/2 Transmission Control Protocol/Internet Protocol User's Guide, SC23-2047

IBM RT AIX Interface Program for Use with TCP/IP: Version 2.2.1, SC23-2005.

For information about the IBM 8232 LAN Channel Station, you may want to refer to the following books:

IBM 8232 LAN Channel Station: Installation and Testing, GA27-3796

IBM LAN Channel Station Operator Guide, GA27-3785

IBM LAN Channel Support Program, Version 1.0: User's Guide, SC30-3458.

Contents

Chapter 1. Computer Networks and TCP/IP Protocols 1-1
Network Basics 1-1
Gateways 1-2
Internetwork Communication 1-2
TCP/IP Protocols and Network Software 1-3
Internet Protocol 1-4
Address Resolution Protocol 1-4
Transmission Control Protocol 1-4
User Datagram Protocol 1-5
Other Network Protocols 1-5
Internet Addressing 1-5
Routing 1-6
Example of Using TCP/IP 1-7

Chapter 2. TCP/UDP/IP API (Pascal Language) 2-1
Software Requirements 2-2
Data Structures 2-2
Connection State 2-2
Connection Information Record 2-4
Notification Record 2-5
File Specification Record 2-12
Procedure Calls 2-12
Notifications 2-13
TCP/UDP Initialization Procedures 2-13
TCP/UDP Termination Procedure 2-13
Handling External Interrupts 2-13
TCP Communication Procedures 2-14
Ping Interface 2-14
Monitor Procedures 2-15
UDP Communication Procedures 2-15
Raw IP Interface 2-15
Timer Routines 2-16
Host Lookup Routines 2-16
Other Routines 2-16
Notifications 2-17
GetNextNote 2-17
Handle 2-18
Unhandle 2-18
TCP/UDP Initialization Procedures 2-19
TcpNameChange 2-19
BeginTcpIp 2-20
StartTcpNotice 2-20
TCP/UDP Termination Procedure 2-21
EndTcpIp 2-21
Handling External Interrupts 2-21
TcpExtRupt 2-22
TcpVmcfRupt 2-22
TCP Communication Procedures 2-23

TcpOpen and TcpWaitOpen 2-23
TcpFSend, TcpSend, and TcpWaitSend 2-25
TcpFReceive, TcpReceive, and TcpWaitReceive 2-28
TcpClose 2-31
TcpAbort 2-32
TcpStatus 2-33
Ping Interface 2-34
PingRequest 2-34
Monitor Procedures 2-35
MonCommand 2-35
MonQuery 2-36
UDP Communication Procedures 2-38
UdpOpen 2-38
UdpSend 2-39
UdpNReceive 2-40
UdpReceive 2-41
UdpClose 2-41
Raw IP Interface 2-42
RawIpOpen 2-42
RawIpReceive 2-43
RawIpSend 2-44
RawIpClose 2-45
Timer Routines 2-45
CreateTimer 2-46
ClearTimer 2-46
SetTimer 2-46
DestroyTimer 2-47
Host Lookup Routines 2-47
GetHostNumber 2-47
GetHostResol 2-48
GetHostString 2-48
GetIdentity 2-49
IsLocalAddress 2-50
IsLocalHost 2-50
Other Routines 2-51
GetSmsg 2-51
ReadXlateTable 2-51
SayCalRe 2-52
SayConSt 2-52
SayIntAd 2-53
SayIntNum 2-53
SayNotEn 2-54
SayPorTy 2-54
SayProTy 2-54
AddUserNote 2-55
Pascal Return Codes 2-55

Chapter 3. C Socket Application Program Interface 3-1
Programming With Sockets 3-1
A Typical TCP Socket Session 3-2
A Typical UDP Socket Session 3-3
Software Requirements 3-3

C Socket Quick Reference 3-3
The Socket Library 3-4
accept() 3-5
bind() 3-6
close() 3-8
connect() 3-8
gethostbyaddr() 3-9
gethostbyname() 3-10
gethostname() 3-11
getsockname() 3-11
listen() 3-12
read() and readv() 3-12
recv() and recvfrom() 3-13
select() 3-15
send() and sendto() 3-16
socket() 3-17
write() and writev() 3-18

Chapter 4. X-Windows Interface 4-1
Software Requirements 4-1
How the X-Windows Interface Works 4-1
X Defaults 4-3
EBCDIC-ASCII Translation 4-3
Creating an Application 4-3
Running an Application 4-4
X-Windows Quick Reference 4-4
Opening and Closing Display 4-5
Creating and Destroying Windows 4-5
Manipulating Windows 4-5
Changing Window Attributes 4-6
Obtaining Window Information 4-6
Properties and Atoms 4-7
Manipulating Window Properties 4-7
Setting Window Selections 4-7
Manipulating Colormaps 4-7
Manipulating Color Cells 4-8
Creating and Freeing Pixmaps 4-8
Manipulating Graphics Contexts 4-8
Clearing and Copying Areas 4-9
Drawing Lines 4-10
Filling Areas 4-10
Loading and Freeing Fonts 4-10
Querying Character String Sizes 4-11
Drawing Text 4-11
Transferring Images 4-12
Manipulating Cursors 4-12
Handling Window Manager Functions 4-12
Manipulating Keyboard Settings 4-13
Controlling the Screen Saver 4-14
Manipulating Hosts and Access Control 4-14
Handling Events 4-14
Enabling and Disabling Synchronization 4-15

Using Default Error Handling 4-15
Communicating with Window Managers 4-16
Keyboard Event Functions 4-17
Manipulating Regions 4-17
Using Cut and Paste Buffers 4-18
Querying Visual Types 4-18
Manipulating Images 4-18
Manipulating Bitmaps 4-19
Using the Resource Manager 4-19
Display Functions 4-20
Extension Routines 4-23
Associate Table Functions 4-24
X-Windows Toolkit 4-25
Purpose 4-25
Contents 4-25
Requirements 4-25
Defining Widgets 4-27

Chapter 5. Remote Procedure Calls 5-1
The RPC Interface 5-1
Software Requirements 5-1
Remote Procedure Call Quick Reference 5-1
Remote Procedure Call Library 5-5
auth_destroy() 5-5
authnone_create() 5-6
authunix_create() 5-6
authunix_create_default() 5-6
callrpc() 5-6
clnt_call() 5-7
clnt_destroy() 5-8
clnt_freeres() 5-8
clnt_geterr() 5-8
clnt_pcreateerror() 5-9
clnt_perrno() 5-9
clnt_perror() 5-9
clntraw_create() 5-10
clnttcp_create() 5-10
clntudp_create() 5-11
get_myaddress() 5-11
mvs_xdr_enum() 5-11
pmap_getmaps() 5-12
pmap_getport() 5-12
pmap_rmtcall() 5-13
pmap_set() 5-13
pmap_unset() 5-14
registerrpc() 5-14
rpc_createerr 5-15
svc_destroy() 5-15
svc_fds 5-15
svc_freeargs() 5-15

svc_getargs() 5-16
svc_getcaller() 5-16
svc_getreq() 5-16
svc_register() 5-17
svc_run() 5-17
svc_sendreply() 5-17
svc_unregister() 5-18
svcerr_auth() 5-18
svcerr_decode() 5-18
svcerr_noproc() 5-19
svcerr_noprog() 5-19
svcerr_progvers() 5-19
svcerr_systemerr() 5-19
svcerr_weakauth() 5-20
svcraw_create() 5-20
svctcp_create() 5-20
svcudp_create() 5-21
xdr_accepted_reply() 5-21
xdr_array() 5-21
xdr_authunix_parms() 5-22
xdr_bool() 5-22
xdr_bytes() 5-22
xdr_callhdr() 5-23
xdr_callmsg() 5-23
xdr_double() 5-23
xdr_enum() 5-24
xdr_float() 5-25
xdr_inline() 5-25
xdr_int() 5-26
xdr_long() 5-26
xdr_opaque() 5-26
xdr_opaque_auth() 5-27
xdr_pmap() 5-27
xdr_pmaplist() 5-27
xdr_reference() 5-28
xdr_rejected_reply() 5-28
xdr_replymsg() 5-28
xdr_short() 5-29
xdr_string() 5-29
xdr_u_int() 5-29
xdr_u_long() 5-30
xdr_u_short() 5-30
xdr_union() 5-30
xdr_void() 5-31
xdr_wrapstring() 5-31
xprt_register() 5-31
xprt_unregister() 5-31

Appendix A. VMCF Interface A-1

Sending and Receiving Special Messages A-1
Locating Program Call Numbers A-2
Program Call Sequences A-3
Parameters Passed to External Interrupt Routines A-5
Disabling Reception of Interrupts A-5
Manipulating the System Mask A-5
Updating Control Register 0 . . . . . A-6
Data Structures A-7
General Information A-10
Use of VMCF Parm List Fields A-10
Use of VMCF Interrupt Header Fields A-10
TCP/UDP/IP Initialization and Termination Procedures A-11
Begin TCP/IP Service A-11
Specifying the Notifications to Receive A-11
End TCP/IP Service A-12
Open TCP Connection A-12
Send TCP Data A-13
Receive TCP Data with the FRECEIVEtcp Function A-14
Receive TCP Data with the RECEIVEtcp Function A-15
Close a TCP Connection A-15
Abort a TCP Connection A-16
Obtain Current Status of TCP Connection A-16
Close a UDP Port A-16
Open a UDP Port A-17
Send UDP Data A-17
Receive UDP Data A-18
Determine Whether an Address is Local A-18
Instruct TCPIP to Obey a File of Commands A-18
Obtain Status Information from TCPIP A-19
Send an ICMP Echo Request A-20
Tell TCPIP That Your Program Will Use a Particular IP Protocol . . . A-20
Tell TCPIP That Your Program Will No Longer Use a Particular IP Protocol A-21
Send Raw IP Packets A-21
Receive Raw IP Packets of a Given Protocol A-22
Notifications A-22
BUFFERspaceAVAILABLE A-23
CONNECTIONstateCHANGED A-23
DATAdelivered A-24
PINGresponse A-24
RAWIPpacketsDELIVERED A-25
RAWIPspaceAVAILABLE A-25
RESOURCESavailable A-26
UDPdatagramDELIVERED A-26
UDPdatagramSPACEavailable A-27
UDPresourcesAVAILABLE A-27
URGENTpending A-28

Appendix B. Interface to SMTP Address Space B-1
Format of Batch SMTP Command Data Sets B-1
SMTP Responses B-1
SMTP Path-Address Handling B-2

SMTP Commands B-3
DATA B-3
HELO B-4
HELP B-4
MAIL FROM B-5
NOOP B-5
QUEU B-5
QUIT B-6
RCPT TO B-6
RSET B-6
TICK B-7
VERB B-7
VRFY B-7
Unimplemented Commands B-8
Batch SMTP Example B-8

Appendix C. Assembler Calls for the Pascal API C-1
RTcpExtRupt C-1
RTcpVmcfRupt C-1
AddUserNote C-2

Appendix D. Network File System Server Exit Routines D-1
Related Publications D-1
Login Exit D-1
Requirements D-2
Register Contents D-2
Contents of Parameter List D-2
Using the Parameter List D-4
System Initialization D-4
Start of New User Session D-5
User Logon Request D-5
New Password Supplied D-6
User Timed Out D-7
Logout Has Been Requested D-7
System Termination D-8
Sample Parameter List - Assembler Language DSECT D-8
Security Exit D-10
Requirements D-10
Contents of Parameter List D-11
Using the Parameter List D-12
Validate Allocate Request D-12
Validate Write Request D-13
Validate Read Request D-14
Return Security Permissions D-15
Sample Parameter List - Assembler Language DSECT D-15
Archive Exit D-16
Requirements D-17
Register Contents D-17
Contents of Parameter List D-17
Using the Parameter List D-19
System Initialization D-20
Information From Archive Requested D-20

Retrieve From Archive Requested D-22
Read Request D-23
Write Request D-24
Delete Request D-24
Create Request D-25
System Termination D-26
Sample Parameter List - Assembler Language DSECT D-27
Account Exit D-28
When It Is Called D-28
Requirements D-29
Register Contents D-29
Contents of Parameter List D-29
Using the Parameter List D-32
System Initialization D-32
Start of New User Session D-32
User Request Complete D-33
User Interval Expiration D-34
User Data Set Usage D-34
User Termination D-35
System Termination D-35
Sample Parameter List - Assemble Language DSECT D-36

Appendix E. Sample Programs E-1
A Sample Pascal Application E-1
A Sample C Socket Communications Server E-6
A Sample C Socket Communications Client E-10
A Sample X-Windows Application E-13

Appendix F. Related Protocol Specifications F-1

Glossary X-1

Index X-9

Figures

1-1. Examples of Bus, Ring, and Point-to-Point Network Topologies . . 1-1
1-2. A Typical Internetwork 1-2
1-3. Relationship of TCP/IP to Network Software 1-3
1-4. File Transfer Using TCP/IP 1-7
2-1. Pascal Declaration of Connection State Type 2-3
2-2. Pascal Declaration of Connection Information Record 2-4
2-3. Pascal Declaration of Socket Type 2-5
2-4. Notification Record 2-6
2-5. Pascal Declaration of File Specification Record 2-12
2-6. Monitor Query Record 2-37
4-1. MVS X-Windows Application to Server 4-2
A-1. Routine to Locate the VMCF CVT A-3
A-2. Standard Program Call Sequence A-4
A-3. VMCF Parameter List A-4
A-4. IUCV Parameter List A-4
A-5. IUCV Query Parameter List A-4
A-6. External Interrupt Routine Parameter List A-5
A-7. Read System Mask Parameter List A-5
A-8. Set System Mask Parameter List A-5
A-9. Store Then And System Mask Parameter List A-6
A-10. Store Then Or System Mask Parameter List A-6
A-11. Read Control Register 0 Parameter List A-6
A-12. Set Control Register 0 Parameter List A-6
A-13. VMCF Parameter List Fields A-7
A-14. Equates for the CALLCODE Field A-8
A-15. Equates for the CALLCODE Field A-8
A-16. Equates for Connection States A-9
A-17. Equates for Notification Mask in the HANDLEnotice Call A-9
A-18. Assembler Format of the Connection Information Record A-9
A-19. Miscellaneous Constants A-10
A-20. Assembler Format of the SpecOfFileType Record for MVS A-19
A-21. MonQueryRecordType A-19
A-22. BUFFERspaceAVAILABLE Notification A-23
A-23. CONNECTIONstateCHANGED Notification A-23
A-24. DATAdelivered Notification A-24
A-25. PINGresponse Notification A-24
A-26. RAWIPpacketsDELIVERED Notification A-25
A-27. RAWIPspaceAVAILABLE Notification A-25
A-28. RESOURCESavailable Notification A-26
A-29. UDPdatagramDELIVERED Notification A-26
A-30. UDPdatagramsPACEavailable Notification A-27
A-31. UDPresourcesAVAILABLE Notification A-27
A-32. URGENTpending Notification A-28
E-1. Example of a Pascal Application Program E-1
E-2. Example of a C Socket Communications Server E-6
E-3. Example of a C Socket Communications Client E-10
E-4. Example of an X-Windows Application E-13

Tables

1-1. Address Fields of Internet Addresses 1-6
2-1. TCP Connection States 2-3
2-2. Pascal Language Interface Summary - Notifications 2-13
2-3. Pascal Language Interface Summary - TCP/UDP Initialization Procedures 2-13
2-4. Pascal Language Interface Summary - TCP/UDP Termination Procedure 2-13
2-5. Pascal Language Interface Summary - Handling External Interrupts 2-13
2-6. Pascal Language Interface Summary - TCP Communication Procedures 2-14
2-7. Pascal Language Interface Summary - Ping Interface 2-14
2-8. Pascal Language Interface Summary - Monitor Procedures 2-15
2-9. Pascal Language Interface Summary - UDP Communication Procedures 2-15
2-10. Pascal Language Interface Summary - Raw IP Interface 2-15
2-11. Pascal Language Interface Summary - Timer Routines 2-16
2-12. Pascal Language Interface Summary - Host Lookup Routines . . . 2-16
2-13. Pascal Language Interface Summary - Other Routines 2-16
2-14. Pascal Language Return Codes 2-55
3-1. C Socket Quick Reference 3-3
4-1. Differences in Header File Names 4-4
4-2. Opening and Closing Display 4-5
4-3. Creating and Destroying Windows 4-5
4-4. Manipulating Windows 4-5
4-5. Changing Window Attributes 4-6
4-6. Obtaining Window Information 4-6
4-7. Properties and Atoms 4-7
4-8. Manipulating Window Properties 4-7
4-9. Setting Window Selections 4-7
4-10. Manipulating Colormaps 4-7
4-11. Manipulating Color Cells 4-8
4-12. Creating and Freeing Pixmaps 4-8
4-13. Manipulating Graphics Contexts 4-8
4-14. Clearing and Copying Areas 4-9
4-15. Drawing Lines 4-10
4-16. Filling Areas 4-10
4-17. Loading and Freeing Fonts 4-10
4-18. Querying Character String Sizes 4-11
4-19. Drawing Text 4-11
4-20. Transferring Images 4-12
4-21. Manipulating Cursors 4-12
4-22. Handling Window Manager Functions 4-12
4-23. Manipulating Keyboard Settings 4-13
4-24. Controlling the Screen Saver 4-14
4-25. Manipulating Hosts and Access Control 4-14
4-26. Handling Events 4-14
4-27. Enabling and Disabling Synchronization 4-15
4-28. Using Default Error Handling 4-15

4-29. Communicating with Window Managers 4-16
4-30. Keyboard Event Functions 4-17
4-31. Manipulating Regions 4-17
4-32. Using Cut and Paste Buffers 4-18
4-33. Querying Visual Types 4-18
4-34. Manipulating Images 4-18
4-35. Manipulating Bitmaps 4-19
4-36. Using the Resource Manager 4-19
4-37. Display Functions 4-20
4-38. Extension Routines 4-23
4-39. Associate Table Functions 4-24
4-40. Differences in Header File Names for Toolkits 4-26
5-1. Remote Procedure Call Quick Reference 5-2
A-1. VMCF CVT Program Call Number Field Names A-2
B-1. Default Values of SMTP Commands and Data B-2
D-1. Login Exit Routines D-3
D-2. System Initialization Codes and Fields D-4
D-3. Start of New User Session Codes and Fields D-5
D-4. User Logon Request Codes and Fields D-5
D-5. New Password Supplied Codes and Fields D-6
D-6. User Timed Out Codes and Fields D-7
D-7. Logoff Has Been Requested Codes and Fields D-7
D-8. System Termination Codes and Fields D-8
D-9. Security Exit Routines D-11
D-10. Validate Allocate Request Codes and Fields D-12
D-11. Validate Write Request Codes and Fields D-13
D-12. Validate Read Request Codes and Fields D-14
D-13. Return Security Permissions Codes and Fields D-15
D-14. Archive Exit Routines D-18
D-15. System Initialization Codes and Fields D-20
D-16. Information from Archives Requested Codes and Fields D-20
D-17. Retrieve From Archive Requested Codes and Fields D-22
D-18. Read Request Codes and Fields D-23
D-19. Write Request Codes and Fields D-24
D-20. Delete Request Codes and Fields D-24
D-21. Create Request Codes and Fields D-25
D-22. System Termination Codes and Fields D-26
D-23. Account Exit Routines D-30
D-24. System Initialization Codes and Fields D-32
D-25. Start of New User Session Codes and Fields D-32
D-26. User Request Complete Codes and Fields D-33
D-27. User Interval Expiration Codes and Fields D-34
D-28. User Data Set Storage Codes and Fields D-34
D-29. User Termination Codes and Fields D-35
D-30. System Termination Codes and Fields D-36

Chapter 1. Computer Networks and TCP/IP Protocols

This chapter provides background information about computer networks, explains the concept of internetwork communication using Transmission Control Protocol/Internet Protocol (TCP/IP), and introduces the IBM TCP/IP for MVS product.

Network Basics

A computer network is a collection of computer nodes physically connected by a suitable communications medium. A computer node can be a microcomputer, computer workstation, minicomputer, or mainframe. The arrangement and connection of network nodes is known as the network topology.

Figure 1-1 shows several common network topologies.

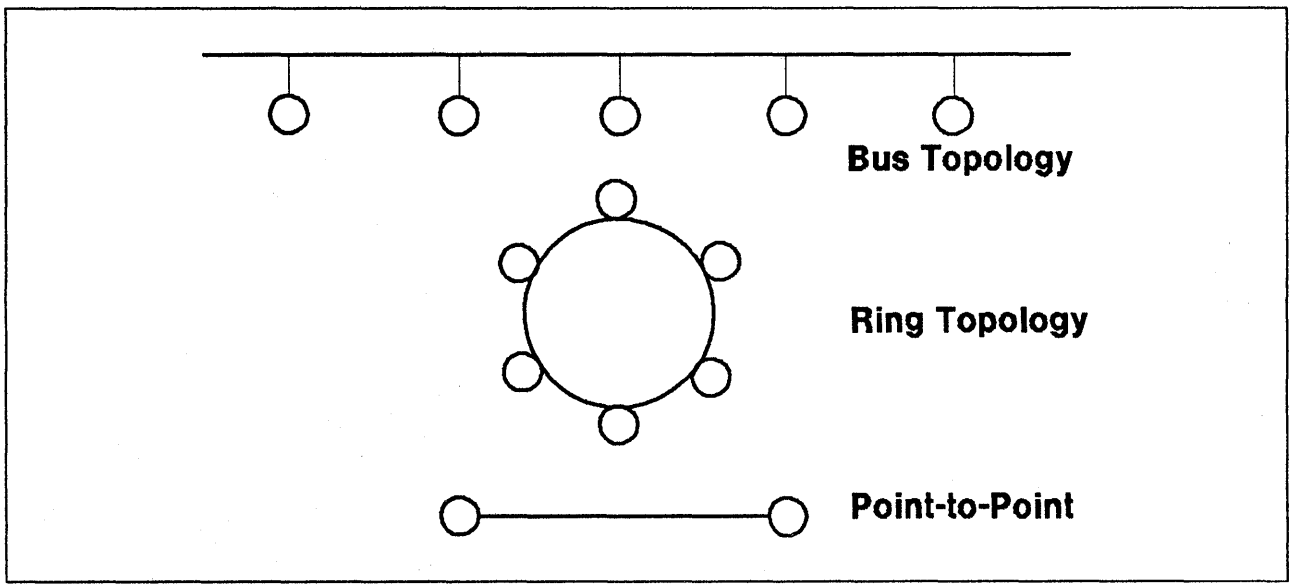


Figure 1-1. Examples of Bus, Ring, and Point-to-Point Network Topologies

The purpose of a computer network is to provide communication between nodes, resource sharing, and distributed computing. Communication applications include electronic mail, remote logon, and file transfer. Resource sharing refers to the access to limited resources, such as disk space and printers, by many computers on the network. Distributed computing refers to distribution of workload among hosts. A designated computer on the network (the server) makes a specialized service available to other computers on the network (the clients). Different computers provide different services for the benefit of the entire network.

A network where all nodes are treated the same, regardless of size, is called a peer-to-peer network.

Gateways

Networks are linked together through a common node called a gateway. The gateway performs all protocol conversion required for communication across networks. The network to which a node is physically connected is the local network; the network to which it is linked by a gateway is the foreign network. This local/foreign concept applies not only to networks, but also to hosts.

Internetwork Communication

An internetwork or *internet* is a collection of packet-switched networks interconnected by gateways to form a single, large virtual network.

Conceptually, an internet is equivalent to universal connectivity. It means all nodes on all interconnected networks can communicate as if they were all on the same physical network, regardless of their specific hardware or software architecture. This cooperation among otherwise incompatible networks and systems is known as *interoperability*.

Figure 1-2 shows the various ways networks can be interconnected in an internet.

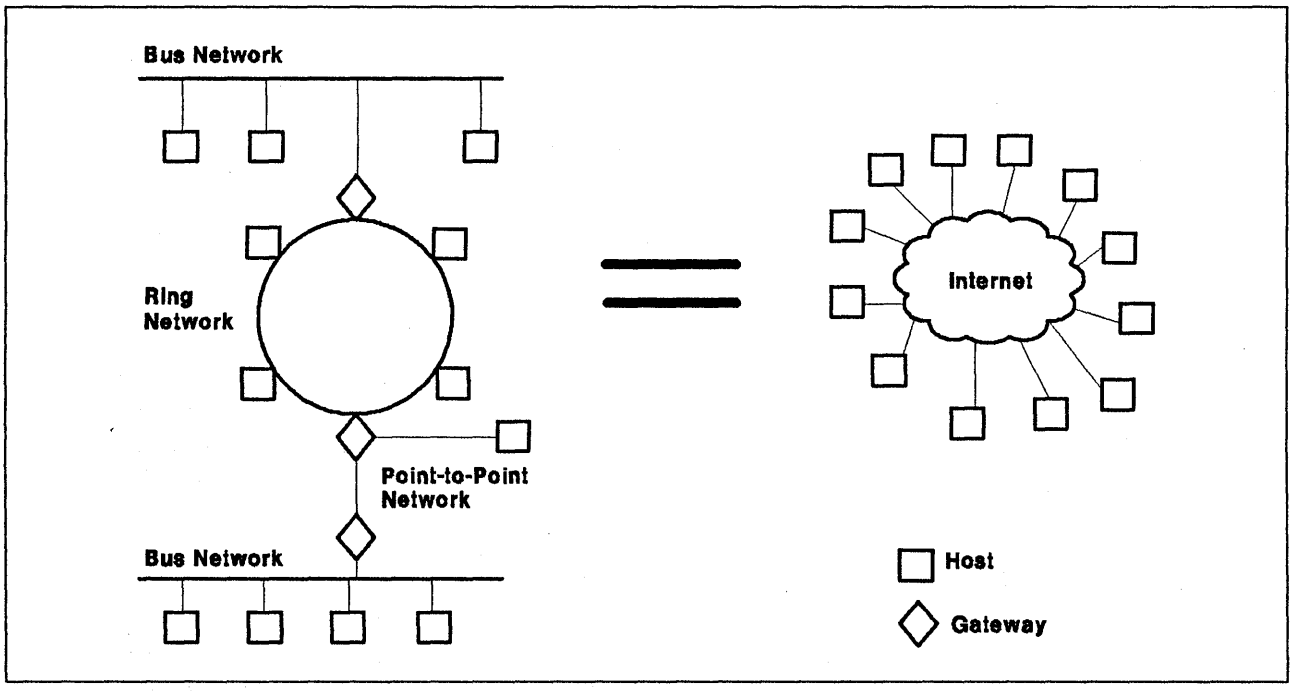


Figure 1-2. A Typical Internetwork

When capitalized, internet, that is, Internet, refers to a specific internetwork that includes ARPANET, MILNET, and NSFnet.

Each network connection of each node on an internetwork must be assigned a unique address according to conventions discussed later in this chapter. This is different from a hardware address, which is often preset by the manufacturer. Also, internet addresses follow a standard format, while different hardware types use different address lengths and formats.

TCP/IP Protocols and Network Software

Network protocols are formal descriptions of the sequence and content of data packets exchanged between network nodes. Because network protocols are implemented in the network software, these terms are often used interchangeably.

Internetwork communication is dependent on TCP/IP, a family of nonproprietary network level protocols, collectively referred to as a protocol suite. TCP/IP allows disparate packet-switched computer networks to function as a single coordinated entity. Originally developed to link military, government research and university networks, TCP/IP now has many commercial users and applications.

The TCP/IP protocol suite forms a layered structure of protocols (see Figure 1-3) ranging from low-level hardware-dependent software to high-level applications. Each TCP/IP layer provides services to the layer above it and uses the services provided by the layer below it. The lowest layer, which is next to the hardware, is not defined by TCP/IP. This layer consists of the hardware specific network protocols.

Figure 1-3 shows where TCP/IP protocols are positioned in relation to network software.

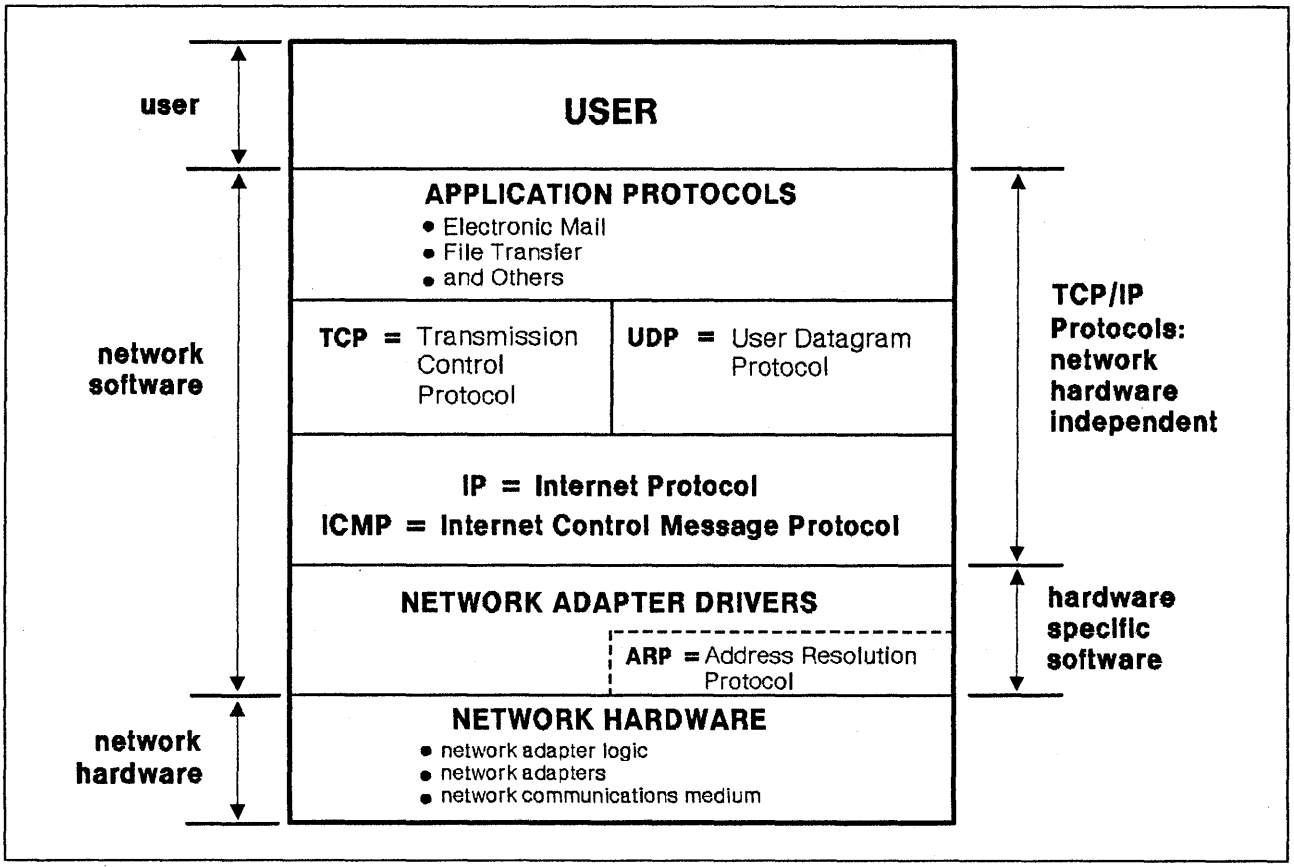


Figure 1-3. Relationship of TCP/IP to Network Software

Following are brief descriptions of the layers making up the TCP/IP protocol suite.

Internet Protocol

The Internet Protocol (IP) provides the basic transport mechanism for communication between hosts on the different networks that make up an internetwork. IP is responsible for making the underlying interconnected networks appear to the layers above as a single, large, virtual network. IP is thus responsible for implementing the internet concept, which it accomplishes by routing packets from a host on one network through a series of gateways to a host on another network. At the internet level, all communication is host-to-host, using fixed length addresses to identify source and destination hosts. The protocol layers above only need to know each host's internet address to make a connection.

In computer networks that use TCP/IP protocols, information is transmitted between nodes in the form of *packets* of data. Outgoing packets can be fragmented into multiple smaller packets. The packet of each fragment is automatically prefixed with an IP header. Incoming packets are reassembled, if necessary, and stripped of the header before being passed on to the next higher protocol layer, usually TCP or UDP. IP does not acknowledge receipt of a packet, nor is it responsible for retransmitting or providing flow and error control. Reliable delivery must be ensured by a higher-level protocol, such as TCP.

Integral to every IP implementation is the Internet Control Message Protocol (ICMP), used for reporting errors in datagram processing. Although ICMP is a basic part of IP, it treats IP as a higher-level protocol.

Address Resolution Protocol

The Address Resolution Protocol (ARP) dynamically maps internet addresses to hardware addresses on a local network.

Transmission Control Protocol

The Transmission Control Protocol (TCP) provides a reliable vehicle for delivering packets between hosts on an internet. TCP takes a stream of data, breaks it into datagrams, sends each one individually using IP, and reassembles the datagrams at the destination node. If any datagrams are lost or damaged during transmission, TCP detects this fact and resends the missing datagrams. The received data stream is a reliable copy of the transmitted data stream.

The interface to TCP is a set of library calls similar to the calls made by an application program to an operating system when manipulating files. TCP communicates asynchronously with applications in a general environment of interconnected networks, and assumes the presence of IP as the underlying protocol at the network level.

User Datagram Protocol

The User Datagram Protocol (UDP) allows application programs to send messages to other programs with a minimum of protocol conversion. Unlike TCP, UDP is a connectionless datagram protocol that requires minimal overhead, but does not guarantee delivery. UDP may be used instead of TCP when an application does not want to incur the overhead of TCP connection setup and breakdown, and is able to do its own acknowledgement and retransmission processing to ensure reliable data transfer. The interface to UDP is a set of library calls.

Other Network Protocols

The user interfaces directly with the Application Protocols layer. This layer consists of several independent protocols that implement the following applications:

File Transfer Protocol
The File Transfer Protocol (FTP) allows copies of files to be sent across an internet. Usually, a password is required to retrieve a file from a foreign host.

Simple Mail Transfer Protocol
The Simple Mail Transfer Protocol (SMTP) allows electronic mail to be exchanged among hosts across an internet.

Remote Terminal Protocol
The Telnet Protocol (Telnet) allows remote logon to hosts across an internet.

Internet Addressing

Each node on an internet must have a unique address called an *internet address.* This address is a 32-bit integer. Internet addresses are usually expressed in the form `www.xxx.yyy.zzz`, where each field is the decimal representation of one octet of the address. For example, the address whose hexadecimal representation is `X'82638001'` would be expressed as `130.99.128.1`. Addresses on the Internet[1] are administered by SRI International[2]. If you have your own internet, you are its administrator. It is strongly recommended that you have an Internet address.

A higher-level naming method called *domain naming* is used to eliminate the need for users to know numerical internet addresses. When a user specifies a domain name, it is resolved into an internet address by a domain name server.

Internet addresses have two parts: a network number and a host number on that network. The four-octet address is divided into network number and host number in any of three different ways, depending on the range of addresses into which the address falls. The first range of addresses is reserved for *Class A* networks, which are very large networks. The second range of addresses is reserved for *Class B* networks, which are medium-size networks. The third range of addresses is reserved for *Class*

---

[1] A specific internetwork that includes ARPANET, MILNET and NSFnet. These networks use the TCP/IP protocol suite.

[2] SRI International, 333 Ravenswood Avenue, Menlo Park, CA. 94025, 1-800-235-3155.

C networks, which are small networks. Some addresses are reserved for special purposes and for future classes of networks.

The division of addresses between network number and host number is as follows:

Table 1-1. Address Fields of Internet Addresses

Network Class	Network Number	Host Number
Class A	First octet	Last three octets
Class B	First two octets	Last two octets
Class C	First three octets	Last octet

A network can be divided into multiple smaller subnetworks called *subnets*. For example, a Class B network can be divided into multiple subnets, each having the same number of hosts as a Class C network. This extension to the IP addressing scheme allows a site to be seen from outside as having one network number. The Class B network could consist of multiple networks, each of smaller size.

To allow for subnet addressing, the host number portion of an internet address is divided into a subnet part and a host part. The network portion and subnet part are logically concatenated to form the subnetwork identifier. For example, if a Class B network had an address of `130.42`, it could be divided into 254 subnetworks with addresses ranging from `130.42.1` through `130.42.254` (0 and 255 are reserved).

On each network, a special host number is reserved for a broadcast address. This is the host number consisting of all 0 bits or all 1 bits. For example, the broadcast host number on a Class B network is 0 or 65 535. The broadcast host number on a Class C network, or on each subnet of a Class B network is 0 or 255.

Routing

Routing is the process of deciding where to send a packet based on its destination address. There are two kinds of routing involved in communications within an internet: direct and indirect.

Direct routing is used when the source and destination nodes are on the same network within the internet. The source node maps the destination internet address into a hardware address and sends packets to the destination node at this address. This mapping is normally performed through a translation table, but if no match can be found for a destination internet address, the Address Resolution Protocol (ARP) provides a default address.

Indirect routing is used when the source and destination nodes are on different networks within the internet. The source node sends packets to a gateway, bridge, or router on the same network using direct routing. From there, the packets are forwarded through intermediate gateways, bridges, or routers, as required, until they arrive at the destination network. Direct routing is then used to forward the packets to the destination host on that network. Each host, gateway, bridge, and router in the internet has a routing table that defines paths to other nodes in the internet.

Example of Using TCP/IP

To send a large file from a source node on one network to a destination node on another network, the source node waits for concurrence from the destination node. In this scenario, the source node is the client and the destination node is the server for the file transfer application (see Figure 1-4).

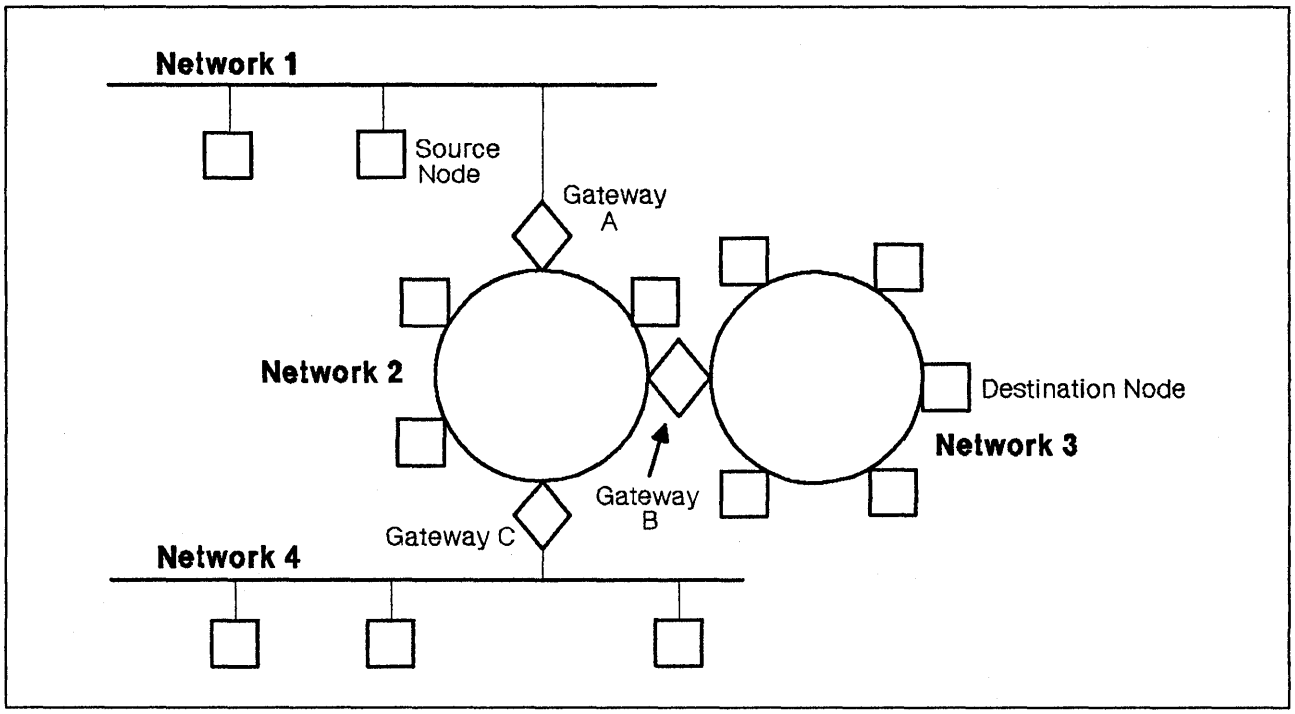


Figure 1-4. File Transfer Using TCP/IP

The file transfer application on the source node takes the file and uses the reliable data stream connection services of TCP to send the file to the file transfer application on the destination node. The source node's TCP layer breaks the file into packets and uses the packet routing services of IP to individually send each packet to TCP on the destination node. Packets do not necessarily arrive in sequence. It is up to the destination node's TCP layer to sort them out and restore the original file. If packets are lost in transmission, the source node resends the lost packets.

The source node's IP layer determines whether the destination node is on the local network or on a foreign network. If the destination node is on the local network, the source node's IP uses the services of the low-level network protocol to send the packet to the destination node's IP layer. If the destination node is on a foreign network, the source node's IP layer decides which gateway should route the datagram and uses the services of the low-level network protocol to send the packet to that gateway's IP layer. The gateway's IP layer in turn routes the packet through the foreign network until the destination node's IP layer is reached.

Chapter 2. TCP/UDP/IP API (Pascal Language)

This chapter describes the Pascal Language Application Program Interface (API) provided with the TCP/IP for MVS product. This interface allows programmers to write application programs that use the TCP, UDP, and IP layers of the TCP/IP protocol suite.

You should have experience in Pascal Language programming and be familiar with the principles of internetwork communication in order to use the Pascal Language API.

Your program uses procedure calls to initiate communication with the TCPIP address space. Most of these procedure calls return with a code that indicates success, or the type of failure incurred by the call. The TCPIP address space starts asynchronous communication by sending you notifications.

The general sequence of operations is as follows:

1. Start up TCP/UDP/IP service (**BeginTcpIp**, **StartTcpNotice**)
2. Specify the set of notifications that TCP/UDP/IP may send you (**Handle**)
3. Establish a connection (**TcpOpen**, **UdpOpen**, **RawIpOpen**, **TcpWaitOpen**)

   If using **TcpOpen**, you must wait for the appropriate notification that a connection has been established.
4. Transfer data buffer to or from the TCPIP address space (**TcpSend**, **TcpFSend**, **TcpWaitSend**, **TcpReceive**, **TcpFReceive**, **TcpWaitReceive**, **UdpSend**, **UdpNReceive**, **RawIpSend**, **RawIpReceive**)

   Note: **TcpWaitReceive** and **TcpWaitSend** are synchronous calls.
5. Check status returned from TCPIP in the form of notifications (**GetNextNote**)
6. Repeat the data transfer operations (steps 4 and 5) until the data is exhausted
7. Terminate the connection (**TcpClose**, **UdpClose**, **RawIpClose**)

   If using **TcpClose**, you must wait for the connection to terminate.
8. Terminate the communication service (**EndTcpIp**).

Control is returned to you, in most instances, after the initiation of your request. When appropriate, some procedures have alternative wait versions that return only after completion of the request. The bodies of the Pascal procedures are in the TCPIP.ATCPPSRC data set.

A sample program is supplied with the TCP/IP for MVS program. It is contained in the PSAMPLE member of the TCPIP.ATCPPSRC data set, and illustrates the use of the interface procedures. Refer to "A Sample Pascal Application" on page E-1 for a listing of the sample program.

Software Requirements

To develop programs in Pascal that interface directly to the TCP, UDP, and IP protocol boundaries, you require the following:

- IBM VS Pascal Compiler & Library (5668-767).

Data Structures

Programs containing Pascal Language API calls must include the appropriate data structures. The data structures are declared in the CMCOMM and CMCLIEN members of the TCPIP.COMMMAC data set. Your SYSLIB concatenation must include the TCPIP.COMMMAC library. To include these data sets in your program source, enter:

```
%include CMCOMM
%include CMCLIEN
```

Additional include statements are required in programs that use certain calls. The following list shows which members of the TCPIP.COMMMAC data set need to be included for the various calls.

- CMRESGLB for **GetHostResol**
- CMINTER for **GetHostNumber**, **GetHostString**, **IsLocalAddress**, and **IsLocalHost**
- CMMON for **MonCommand**, and **MonQuery**.

The load modules are in the TCPIP.COMMTXT data set. Include this data set in your SYSLIB concatenation when you are creating a load module to link an application program.

Connection State

ConnectionState is the current state of the connection. See Figure 2-1 on page 2-3 for the Pascal declaration of the ConnectionStateType data type. ConnectionStateType is used in StatusInfoType and NotificationInfoType. It defines the client program's view of the state of a TCP connection, in a form more readily usable than the formal TCP connection state defined by RFC 793. Refer to Table 2-1 on page 2-3 for the mapping between TCP states and ConnectionStateType.

```
ConnectionStateType =
(
    CONNECTIONclosing,
    LISTENING,
    NONEXISTENT,
    OPEN,
    RECEIVINGonly,
    SENDINGonly,
    TRYINGtoOPEN
);
```

Figure 2-1. Pascal Declaration of Connection State Type

CONNECTIONclosing
Indicates that no more data can be transmitted on this connection because it is going through the TCP connection closing sequence.

LISTENING
Indicates that you are waiting for a foreign site to open a connection.

NONEXISTENT
Indicates that a connection no longer exists.

OPEN
Indicates that data can go either way on the connection.

RECEIVINGonly
Indicates that data can be received, but cannot be sent on this connection, because the client has done a **TcpClose**.

SENDINGonly
Indicates that data can be sent out, but cannot be received on this connection, because the foreign application has done a **TcpClose** or equivalent.

TRYINGtoOPEN
Indicates that you are trying to contact a foreign site to establish a connection.

Table 2-1. TCP Connection States

TCP State	Pseudo-State
CLOSED	NONEXISTENT
LAST-ACK, CLOSING, TIME-WAIT	If there is incoming data that the client program has not received, then RECEIVINGonly, else CONNECTIONclosing.
CLOSE-WAIT	If there is incoming data that the client program has not received, then OPEN, else SENDINGonly.
ESTABLISHED	OPEN
FIN-WAIT-1, FIN-WAIT-2	RECEIVINGonly
LISTEN	LISTENING
SYN-SENT, SYN-RECEIVED	TRYINGtoOPEN

Connection Information Record

The connection information record is used as a parameter in several of the procedure calls. It enables you and the TCP/IP for MVS program to exchange information about the connection. The Pascal declaration is shown in Figure 2-2. See "TcpOpen and TcpWaitOpen" on page 2-23 and "TcpStatus" on page 2-33 for further details on the use of each field.

```
StatusInfoType =
    record
    Connection: ConnectionType;
    OpenAttemptTimeout: integer;
    Security: SecurityType;
    Compartment: CompartmentType;
    Precedence: PrecedenceType;
    BytesToRead: integer;
    UnackedBytes: integer;
    ConnectionState: ConnectionStateType;
    LocalSocket: SocketType;
    ForeignSocket: SocketType;
    end;
```

Figure 2-2. Pascal Declaration of Connection Information Record

Connection
Is a number identifying the connection that is being described. This connection number is different from the connection number displayed by the NETSTAT command. (Refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: User's Guide* for more information about the NETSTAT command.)

OpenAttemptTimeout
Is the number of seconds that TCP continues to attempt to open a connection. You specify this number. If the limit is exceeded, TCP stops trying to open the connection and shuts down any partially open connection.

Security, Compartment, Precedence
Are used only when working within a multilevel secure environment. These parameters are reserved for IBM use only.

BytesToRead
Is the number of data bytes received from the foreign host by TCP, but not yet delivered to the client. TCP maintains this value.

UnackedBytes
Is the number of bytes sent by your program but not yet sent to the foreign TCP, or the number of bytes sent to the foreign TCP, but not yet acknowledged.

LocalSocket
Is the local internet address and local port. Together, these form one end of a connection. The foreign socket forms the other end. See Figure 2-3 on page 2-5 for the Pascal declaration of the SocketType record.

```
InternetAddressType = UnsignedIntegerType;
PortType = UnsignedHalfWordType;
SocketType =
      record
      Address: InternetAddressType;
      Port: PortType;
      end;
```

Figure 2-3. Pascal Declaration of Socket Type

Address
Is the internet address.

Port
Is the port.

ForeignSocket
Is the foreign, or remote, internet address and its associated port. These form one end of a connection. The local socket forms the other end.

Notification Record

The notification record is used to provide event information. You receive this information by using the **GetNextNote** call (refer to "GetNextNote" on page 2-17 for more information). It is a variant record, the number of fields being dependent on the type of notification. Refer to Figure 2-4 on page 2-6 for the Pascal declaration of this record.

```
NotificationInfoType =
    record
    Connection: ConnectionType;
    Protocol: ProtocolType;
    case NotificationTag: NotificationEnumType of
        BUFFERspaceAVAILABLE:
            (
            AmountOfSpaceInBytes: integer
            );
        CONNECTIONstateCHANGED:
            (
            NewState: ConnectionStateType;
            Reason: CallReturnCodeType
            );
        DATAdelivered:
            (
            BytesDelivered: integer;
            LastUrgentByte: integer;
            PushFlag: Boolean
            );
        EXTERNALinterrupt:
            (
            RuptCode: integer
            );
        FRECEIVEerror:
            (
            ReceiveTurnCode: CallReturnCodeType;
            ReceiveRequestErr: Boolean;
            );
        FSENDresponse:
            (
            SendTurnCode: CallReturnCodeType;
            SendRequestErr: Boolean;
            );
        IOinterrupt:
            (
            DeviceAddress: integer;
            UnitStatus: UnsignedByteType;
            ChannelStatus: UnsignedByteType
            );
        IUCVinterrupt:
            (
            IUCVResponseBuf: IUCVBufferType
            );
        PINGresponse:
            (
            PingTurnCode: CallReturnCodeType;
            ElapsedTime: TimeStampType
            );
        RAWIPpacketsDELIVERED:
            (
            RawIpDataLength: integer;
            RawIpFullLength: integer;
            );
```

Figure 2-4 (Part 1 of 2). Notification Record

```
        RAWIPspaceAVAILABLE:
            (
            RawIpSpaceInBytes: integer;
            );
        RESOURCESavailable: ();
        SMSGreceived: ();
        TIMERexpired:
            (
            Datum: integer;
            AssociatedTimer: TimerPointerType
            );
        UDPdatagramDELIVERED:
            (
            DataLength: integer;
            ForeignSocket: SocketType;
            FullLength: integer
            );
        UDPdatagramSPACEavailable: ();
        UDPresourcesAVAILABLE: ();
        URGENTpending:
            (
            BytesToRead: integer;
            UrgentSpan: integer
            );
        USERdefinedNOTIFICATION:
            (
            UserData: UserNotificationDataType
            );
        USERdeliversLINE:
            (
            LengthOfUserData: integer;
            UnsolicitedRead: Boolean
            );
        USERwantsATTENTION:
            (
            WhichAttentionKey: char
            );
        end;
```

Figure 2-4 (Part 2 of 2). Notification Record

Connection
Is the client's connection number to which the notification applies. In the case of USERdefinedNOTIFICATION, this field is as supplied by the user in the **AddUserNote** call.

Protocol
In the case of USERdefinedNOTIFICATION, this field is as supplied by the user in the **AddUserNote** call. For all other notifications, this field is reserved.

NotificationTag
Is the type of notification being sent, and a set of fields dependent on the value of the tag. Possible tag values relevant to the TCP/UDP/IP interface and the corresponding fields are:

BUFFERspaceAVAILABLE

Notification given when space becomes available on a connection for which **TcpSend** (refer to "TcpFSend, TcpSend, and TcpWaitSend" on page 2-25) previously returned NObufferSPACE.

AmountOfSpaceInBytes:

The minimum number of bytes that the TCP/IP service has available for buffer space for this connection. The actual amount of buffer space might be more than this number.

CONNECTIONstateCHANGED

Indicates that a TCP connection has changed state.

NewState:

The new state for this connection.

Reason:

The reason for the state change. This field is meaningful only if the **NewState** field has a value of NONEXISTENT.

Usage Notes:

1. The sequence of state notifications for a connection is usually the following. For active open: OPEN, RECEIVINGonly or SENDINGonly, CONNECTIONclosing, NONEXISTENT. For passive open: TRYINGtoOPEN, OPEN, RECEIVINGonly or SENDINGonly, CONNECTIONclosing, NONEXISTENT.

   Your program should be prepared for any intermediate step or steps to be skipped.

2. The normal TCP connection closing sequence may lead to a connection staying in CONNECTIONclosing state for up to two minutes, corresponding to the TCP state TIME-WAIT.

3. Possible Reason codes giving the reason for a connection changing to NONEXISTENT are as follows.

   OK (means normal closing), UNREACHABLEnetwork, TIMEOUTopen OPENrejected, REMOTEreset, WRONGsecORprc, UNEXPECTEDsyn, FATALerror
   KILLEDbyCLIENT, TIMEOUTconnection, TCPipSHUTDOWN, DROPPEDbyOPERATOR.

DATAdelivered

Notification given when your buffer (named in an earlier **TcpReceive** or **TcpFReceive** request) contains data.

Usage Note: The data delivered should be treated as part of a byte-stream, not as a message. There is no guarantee that the data sent in one **TcpSend** (or equivalent) call on the foreign host is delivered in a single DATAdelivered notification, even if the PushFlag is set.

BytesDelivered:

Number of bytes of data delivered to you.

LastUrgentByte:

Number of bytes of urgent data remaining, including data just delivered.

PushFlag:
TRUE if the last byte of data was received with the push bit set.

EXTERNALinterrupt
Notification given when a simulated external interrupt occurs in your address space. The Connection and Protocol fields are not applicable.

RuptCode:
The interrupt type.

FRECEIVEerror
Notification given in place of DATAdelivered when a **TcpFReceive** that initially returned OK has terminated without delivering data.

ReceiveTurnCode:
The reason the **TcpFReceive** has failed or was canceled. If ReceiveRequestErr is set to FALSE, ReceiveTurnCode contains the same reason as the Reason field in the CONNECTIONstateCHANGED with NewState set to NONEXISTENT notification for this connection (see Usage Note 3 on page 2-8). ReceiveTurnCode could be OK, if the connection closed normally.

ReceiveRequestErr
If TRUE, the **TcpFReceive** was rejected during initial processing. If FALSE, the **TcpFReceive** was initially accepted, but was terminated due to connection closing.

Usage Note: Normally, you do not need to take any action upon receipt of this notification with ReceiveRequestErr set to FALSE, because your program receives a CONNECTIONstateCHANGED notification informing it that the connection has been terminated.

FSENDresponse
Notification given when a **TcpFSend** request is completed successfully or unsuccessfully.

SendTurnCode:
The status of the send operation.

SendRequestErr
If TRUE, the **TcpFSend** was rejected during initial processing or during retry after buffer space became available. If FALSE, the **TcpFSend** was canceled due to connection closing.

PINGresponse

PingTurnCode:
The status of the ping operation.

ElapsedTime:
The time elapsed between the sending of a request and the reception of a response. This time does not include the time spent in the simulated Virtual Machine Communication Facility (VMCF) communication between your program and the TCPIP address space. This field is valid only if **PingTurnCode** has a value of OK.

RAWIPpacketsDELIVERED

Notification given when your buffer (indicated in an earlier **RawIpReceive** request) contains a datagram. Only one datagram is delivered on each notification. Your buffer contains the entire IP header, plus as much of the datagram as fits in your buffer.

RawIpDataLength

The actual data length delivered to your buffer. If this is less than RawIpFullLength, the datagram was truncated.

RawIpFullLength

Length of the packet, from the TotalLength field of the IP header.

RAWIPspaceAVAILABLE

When space becomes available after a client does a **RawIpSend** and receives a NObufferSPACE return code, the client receives this notification to indicate that space is now available.

RawIpSpaceInBytes:

The amount of space available always equals the maximum size IP datagram.

RESOURCESavailable

Notice given when resources needed for a **TcpOpen** or **TcpWaitOpen** are available. This notification is sent only if a previous **TcpOpen** or **TcpWaitOpen** returned ZEROresources.

SMSGreceived

Notification given when one or more Special Messages (Smsgs) arrive. The **GetSmsg** call is used to retrieve queued Smsgs. Refer to "Sending and Receiving Special Messages" on page A-1 for information on the SMSG command.

TIMERexpired

Notification given when a timer set through **SetTimer** expires.

Datum:

The data specified when **SetTimer** was called.

AssociatedTimer:

The address of the timer that expired.

UDPdatagramDELIVERED

Notification given when your buffer, indicated in an earlier **UdpNReceive** or **UdpReceive** request, contains a datagram. Your buffer contains the datagram excluding the UDP header.

Usage Note: If **UdpReceive** was used, your buffer contains the entire datagram excluding the header, with the length indicated by **DataLength**. If **UdpNReceive** was used, and DataLength is less than FullLength, your buffer contains a truncated datagram. The reason is that the length of your buffer was too small to contain the entire datagram.

DataLength:

Length of the data delivered to your buffer.

ForeignSocket:
The source of the datagram.

FullLength:
The length of the entire datagram, excluding the UDP header. This field is set only if **UdpNReceive** was used.

UDPdatagramSPACEavailable
Notification given when buffer space becomes available for a datagram for which **UdpSend** previously returned NObufferSPACE due to insufficient resources.

UDPresourcesAVAILABLE
Notice given when resources needed for a **UdpOpen** are available. This notification is sent only if a previous **UdpOpen** returned UDPZeroRESOURCES.

URGENTpending
Notification given when there is urgent data pending on a TCP connection.

BytesToRead:
The number of incoming bytes not yet delivered to the client.

UrgentSpan:
Number of undelivered bytes to the last known urgent pointer. No urgent data is pending if this is negative.

USERdefinedNOTIFICATION
Notice generated from data passed to **AddUserNote** by your program.

UserData:
A 40-byte field supplied by your program through **AddUserNote**. The Connection and Protocol fields are also set from the values supplied to **AddUserNote**.

USERdeliversLINE
Reserved for IBM use only.

USERwantsATTENTION
Reserved for IBM use only.

File Specification Record

The file specification record is used to fully specify a data set. The Pascal declaration is shown in Figure 2-5.

```
SpecOfFileType =
      record
      Owner: DirectoryNameType;
      Case SpecOfSystemType of
      VM:
      (
         VirtualAddress:VirtualAddressType;
         NewVirtualAddress:VirtualAddressType;
         DiskPassword: DirectoryNameType;
         Filename: DirectoryNameType;
         Filetype: DirectoryNameType;
         Filemode: FilemodeType
      );
      MVS:
      (
         DatasetPassword: DirectoryNameType;
         FullDatasetName: DatasetNameType;
         MemberName: MemberNameType;
         DDName: DDNameType
      );
      end;
```

Figure 2-5. Pascal Declaration of File Specification Record

DatasetPassword
The password for a password-protected data set. If a password is not needed, it should be left blank.

FullDatasetName
The fully-qualified data set name to be processed.

MemberName
The member of the data set to be processed if the data set is a partitioned data set.

DDName
Reserved for IBM use only.

Procedure Calls

Your program uses procedure calls to initiate communication with the TCPIP address space. Most of these procedure calls return with a code, which indicates success or the type of failure incurred by the call. Refer to Table 2-14 on page 2-55 for an explanation of the return codes.

Before invoking any of the other interface procedures, use **BeginTcpIp** or **StartTcpNotice** to start up the TCP/UDP/IP service. Once the TCP/UDP/IP service has begun, use the **Handle** procedure to specify a set of notifications that the TCP/UDP/IP service may send you. To terminate the TCP/UDP/IP service, use the **EndTcpIp** procedure.

The following tables give short descriptions of the various Pascal Language API procedures and functions. The table also refers you to a page where additional information on the use of the procedure can be obtained.

Notifications

Table 2-2. Pascal Language Interface Summary - Notifications

Procedure Call	Description	Page
GetNextNote	Retrieves the next notification.	2-17
Handle	Specifies the types of notifications that your program will process.	2-18
Unhandle	Specifies notification types that your program will no longer process.	2-18

TCP/UDP Initialization Procedures

Table 2-3. Pascal Language Interface Summary - TCP/UDP Initialization Procedures

Procedure Call	Description	Page
TcpNameChange	Identifies the name of the address space running the TCP/IP for MVS program when the address space has a name other than TCPIP.	2-19
BeginTcpIp	Establishes communication with the TCP/IP services.	2-20
StartTcpNotice	Establishes communication with the TCP/IP services.	2-20

TCP/UDP Termination Procedure

Table 2-4. Pascal Language Interface Summary - TCP/UDP Termination Procedure

Procedure Call	Description	Page
EndTcpIp	Terminates communication with the TCP/IP services.	2-21

Handling External Interrupts

Table 2-5 (Page 1 of 2). Pascal Language Interface Summary - Handling External Interrupts

Procedure Call	Description	Page
TcpExtRupt	Notifies the TCP interface of the arrival of a simulated external interrupt.	2-22
RTcpExtRupt	A version of **TcpExtRupt**.	C-1

Table 2-5 (Page 2 of 2). Pascal Language Interface Summary - Handling External Interrupts		
Procedure Call	**Description**	**Page**
TcpVmcfRupt	Notifies the TCP interface of the arrival of a simulated external VMCF interrupt.	2-22
RTcpVmcfRupt	A version of **TcpVmcfRupt**.	C-1

TCP Communication Procedures

Table 2-6. Pascal Language Interface Summary - TCP Communication Procedures		
Procedure Call	**Description**	**Page**
TcpOpen	Initiates a TCP connection.	2-23
TcpWaitOpen	Initiates a TCP connection and waits for the establishment of the connection.	2-23
TcpFSend	Sends TCP data.	2-25
TcpSend	Sends TCP data.	2-25
TcpWaitSend	Sends TCP data and waits until TCPIP accepts it.	2-25
TcpFReceive	Establishes a buffer to receive TCP data.	2-28
TcpReceive	Establishes a buffer to receive TCP data.	2-28
TcpWaitReceive	Establishes a buffer to receive TCP data and waits for the reception of the data.	2-28
TcpClose	Begins the TCP one-way closing sequence.	2-31
TcpAbort	Shuts down a TCP connection immediately.	2-32
TcpStatus	Obtains the current status of a TCP connection.	2-33

Ping Interface

Table 2-7. Pascal Language Interface Summary - Ping Interface		
Procedure Call	**Description**	**Page**
PingRequest	Sends an Internet Control Message Protocol (ICMP) echo request.	2-34

Monitor Procedures

Table 2-8. Pascal Language Interface Summary - Monitor Procedures		
Procedure Call	**Description**	**Page**
MonCommand	Instructs TCP to read a specific data set and execute the commands that it contains.	2-35
MonQuery	Performs control functions and retrieves internal TCPIP control blocks.	2-36

UDP Communication Procedures

Table 2-9. Pascal Language Interface Summary - UDP Communication Procedures		
Procedure Call	**Description**	**Page**
UdpOpen	Requests communication with UDP on a specified socket.	2-38
UdpSend	Sends a UDP datagram to a specified foreign socket.	2-39
UdpNReceive	Notifies the TCPIP address space that you are willing to receive UDP datagram data.	2-40
UdpReceive	Notifies the TCPIP address space that you are willing to receive UDP datagram data.	2-41
UdpClose	Terminates use of a UDP socket.	2-41

Raw IP Interface

Table 2-10. Pascal Language Interface Summary - Raw IP Interface		
Procedure Call	**Description**	**Page**
RawIpOpen	Informs the TCPIP address space that the client wants to send and receive IP packets of a specified protocol.	2-42
RawIpReceive	Specifies a buffer to receive raw IP packets of a specified protocol.	2-43
RawIpSend	Sends raw IP packets of a specified protocol.	2-44
RawIpClose	Informs the TCPIP address space that the client no longer handles the protocol.	2-45

Timer Routines

Table 2-11. Pascal Language Interface Summary - Timer Routines		
Procedure Call	**Description**	**Page**
CreateTimer	Allocates a timer.	2-46
ClearTimer	Resets a timer.	2-46
SetTimer	Sets a timer to expire after a specified interval.	2-46
DestroyTimer	Deallocates a timer.	2-47

Host Lookup Routines

Table 2-12. Pascal Language Interface Summary - Host Lookup Routines		
Procedure Call	**Description**	**Page**
GetHostNumber	Converts a host name to an internet address using static tables.	2-47
GetHostResol	Converts a host name to an internet address using a domain name resolver.	2-47
GetHostString	Converts an internet address to a host name using static tables.	2-48
GetIdentity	Returns environment information.	2-49
IsLocalAddress	Determines if an internet address is local.	2-50
IsLocalHost	Determines if a host name is local, remote, loopback, or unknown.	2-50

Other Routines

Table 2-13 (Page 1 of 2). Pascal Language Interface Summary - Other Routines		
Procedure Call	**Description**	**Page**
GetSmsg	Retrieves one queued special message (Smsg).	2-51
ReadXlateTable	Reads a binary translation table data set.	2-51
SayCalRe	Converts a return code into a descriptive message.	2-52
SayConSt	Converts a connection state into a descriptive message.	2-52
SayIntAd	Converts an internet address into a name or dotted-decimal form.	2-53
SayIntNum	Converts an internet address into its dotted-decimal form.	2-53
SayNotEn	Converts a notification enumeration type into a descriptive message.	2-54

Table 2-13 (Page 2 of 2). Pascal Language Interface Summary - Other Routines		
Procedure Call	**Description**	**Page**
SayPorTy	Converts a port number into a descriptive message or into EBCDIC.	2-54
SayProTy	Converts the protocol type into a descriptive message or into EBCDIC.	2-54
AddUserNote	Adds a USERdefinedNOTIFICATION notification to the note queue.	2-55

Notifications

The TCPIP address space sends you notifications to inform you of asynchronous events. Also, some notifications are generated in your address space by the TCP interface. Notifications can be received only after **BeginTcp** or **StartTcpNotice**.

The notifications are received by the TCP interface and kept in a queue. Use **GetNextNote** to get the next notification. The notifications are in Pascal variant record form (refer to Figure 2-4 on page 2-6).

GetNextNote

Use this procedure to retrieve notifications from the queue. This procedure returns the next notification queued for you.

```
procedure GetNextNote
      (
   var    Note: NotificationInfoType;
          ShouldWait: Boolean;
   var    ReturnCode: integer
      );
      external;
```

Parameter	Description
Note:	Next notification is stored here when ReturnCode is OK.
ShouldWait:	Set ShouldWait to TRUE if you want **GetNextNote** to wait until a notification becomes available. Set ShouldWait to FALSE if you want **GetNextNote** to return immediately. When ShouldWait is set to FALSE, ReturnCode is set to NOoutstandingNOTIFICATIONS if no notification is currently queued.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
NOoutstandingNOTIFICATIONS
NOTyetBEGUN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Handle

Use the **Handle** procedure to specify that you want to receive notifications in the given set. You must always use it after calling the **BeginTcpIp** procedure and before accessing the TCP/IP services. This Pascal set may contain any of the NotificationEnumType values shown in Figure 2-4 on page 2-6.

```
procedure Handle
        (
            Notifications: NotificationSetType;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
Notifications:	The set of notification types to be handled.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
NOTyetBEGUN
TCPipSHUTDOWN
ABNORMALcondition
FATALerror.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Unhandle

Use this procedure when you no longer want to receive notifications in the given set.

If you request to unhandle the DATAdelivered notification, the **Unhandle** procedure returns with a code of INVALIDrequest.

```
procedure Unhandle
        (
            Notifications: NotificationSetType;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
Notifications:	The set of notifications that you no longer want to receive.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
INVALIDrequest
NOTyetBEGUN
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

TCP/UDP Initialization Procedures

The following procedures affect all present and future connections. Use these procedures to initialize the TCP/IP environment for your program.

TcpNameChange

Use this procedure if the address space running the TCP/IP for MVS program is not named TCPIP and is not the same as specified in the TcpipUserid statement of the TCPIP.TCPIP.DATA data set (refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* book).

If required, this procedure must be called before the **BeginTcpIp** or the **StartTcpNotice** procedure.

```
procedure TcpNameChange
        (
            NewNameOfTcp: DirectoryNameType
        );
        external;
```

Parameter	Description
NewNameOfTcp:	Specifies the name of the address space running TCP/IP.

BeginTcpIp

Use **BeginTcpIp** to inform the TCPIP address space that you want to start using its services. If your program handles simulated external interrupts itself, use **StartTcpNotice** instead of **BeginTcpIp**. Refer to Appendix A, "VMCF Interface" on page A-1 for information on simulated external interrupt support.

```
procedure BeginTcpIp
        (
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
NOtcpIPservice
TCPipSHUTDOWN
VIRTUALmemoryTOOsmall.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

StartTcpNotice

Use this procedure instead of **BeginTcpIp** when you want to handle simulated external interrupts yourself. You establish your own external interrupt handler.

If your program does not use simulated VMCF (refer to Appendix A, "VMCF Interface" on page A-1 for information on the simulated Virtual Machine Communication Facility interface), set the ClientDoesVmcf parameter to FALSE. Later, when your program receives a simulated external interrupt that it does not handle, including a VMCF interrupt, inform the TCP interface by calling **TcpExtRupt**. The TCP interface then processes the interrupt.

If your program uses simulated VMCF itself, set the ClientDoesVmcf parameter to TRUE. Your program must use the VMCF AUTHORIZE function to establish a VMCF interrupt buffer. Later, when your program receives a VMCF interrupt that it does not handle, inform the TCP interface by calling **TcpVmcfRupt** with the address of your VMCF interrupt buffer. When your program receives a non-VMCF simulated external interrupt that it does not handle, call **TcpExtRupt** as explained above.

```
procedure StartTcpNotice
        (
            ClientDoesVmcf: Boolean;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
ClientDoesVmcf:	Set to FALSE if your program does not use simulated VMCF. Otherwise set to TRUE.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
ALREADYclosing
NOTCPIPservice
VIRTUALmemoryTOOsmall
FATALerror.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

TCP/UDP Termination Procedure

EndTcpIp

Use **EndTcpIp** when you have finished with the TCP/IP services. This procedure releases ports and protocols in use that are not permanently reserved. It causes TCP to clean up any data structures it has associated with you.

```
procedure EndTcpIp;
        external;
```

Handling External Interrupts

The following procedures allow you to pass simulated external interrupts to the TCP interface. You must call the **StartTcpNotice** initialization routine before you can begin using the external interrupt calls.

TcpExtRupt

Use this procedure when:

1. You initiated the TCP/IP service by calling **StartTcpNotice** with ClientDoesVmcf set to TRUE, and your external interrupt handler receives a non-VMCF interrupt not handled by your program. See "TcpVmcfRupt" on page 2-22 for the handling of VMCF interrupts.
2. You initiated the TCP/IP service by calling **StartTcpNotice** with ClientDoesVmcf set to FALSE, and your external interrupt handler receives any interrupt not handled by your program.

RTcpExtRupt is a version of **TcpExtRupt**. Refer to Appendix C, "Assembler Calls for the Pascal API" on page C-1 for more information.

```
procedure TcpExtRupt
        (
    const  RuptCode: integer
        );
        external;
```

Parameter	Description
RuptCode:	The external interrupt code you received.

TcpVmcfRupt

Use this procedure when you initiated the TCP/IP service by calling **StartTcpNotice** with ClientDoesVmcf set to TRUE, and your external interrupt handler receives a VMCF interrupt not handled by your program.

RTcpVmcfRupt is a version of **TcpVmcfRupt** that can be called directly from an assembler interrupt handler. Refer to Appendix C, "Assembler Calls for the Pascal API" on page C-1 for more information.

```
procedure TcpVmcfRupt
        (
            VmcfHeaderAddress: integer
        );
        external;
```

Parameter	Description
VmcfHeaderAddress:	Is the address of your VMCF interrupt buffer as specified in the VMCF AUTHORIZE function that your program issued at initialization.

TCP Communication Procedures

The following procedures apply to a particular client connection. Use these procedures to establish a connection and to communicate. You must call either the **BeginTcpIp** or the **StartTcpNotice** initialization routine before you can begin using TCP communication procedures.

TcpOpen and TcpWaitOpen

Use **TcpOpen** or **TcpWaitOpen** to initiate a TCP connection. **TcpOpen** returns immediately, and connection establishment proceeds asynchronously with your program's other operations. The connection is fully established when your program receives a CONNECTIONstateCHANGED notification with NewState set to OPEN. **TcpWaitOpen** does not return until the connection is established, or until an error occurs.

There are two types of **TcpOpen** calls: passive open, and active open. A passive open call sets the connection state to LISTENING. An active open call sets the connection state to TRYINGtoOPEN.

If a **TcpOpen** or **TcpWaitOpen** call returns ZEROresources, and your application handles RESOURCESavailable notifications, you receive a RESOURCESavailable notification when sufficient resources are available to process an open call. The first open your program issues after a RESOURCESavailable notification is guaranteed not to get the ZEROresources return code.

```
procedure TcpOpen
      (
   var    ConnectionInfo: StatusInfoType;
   var    ReturnCode: integer
      );
      external;
```

```
procedure TcpWaitOpen
      (
   var    ConnectionInfo: StatusInfoType;
   var    ReturnCode: integer
      );
      external;
```

Parameter	Description
ConnectionInfo:	Is a connection information record as indicated below.
Connection:	Set this field to UNSPECIFIEDconnection. When the call returns, the field contains the number of the new connection if ReturnCode is OK.

ConnectionState: For active open, set this field to TRYINGtoOPEN. For passive open, set this field to LISTENING.

OpenAttemptTimeout:
Set this field to specify how long, in seconds, TCP is to continue attempting to open the connection. If the connection is not fully established during that time, TCP reports the error to you. If you used **TcpOpen**, you receive a notification. The type of notification that you receive is CONNECTIONstateCHANGED. It has a new state of NONEXISTENT and a reason of TIMEOUTopen. If you used **TcpWaitOpen** it returns with ReturnCode set to TIMEOUTopen.

Security: This field is reserved. Set it to DEFAULTsecurity.

Compartment: This field is reserved. Set it to DEFAULTcompartment.

Precedence: This field is reserved. Set it to DEFAULTprecedence.

LocalSocket: Active Open: In general, you can use an address of UNSPECIFIEDaddress (TCPIP uses the home address corresponding to the network interface used to route to the foreign address) and a port of UNSPECIFIEDport (TCPIP assigns a port number, in the range of 1 000 to 65 534). You can specify the address and/or the port if particular values are required by your application. The address must be a valid home address for your node, and the port must be available (not reserved, and not in use by another application).

Passive Open: You generally specify a predetermined port number, which is known by another program, that will do an active open to connect to your program. Alternatively, you can use UNSPECIFIEDport to let TCPIP assign a port number, obtain the port number through **TcpStatus**, and transmit it to the other program through an existing TCP connection or manually. You generally specify an address of UNSPECIFIEDaddress, so that the active open to your port succeeds no matter what home addresses it was sent to.

ForeignSocket: Active Open: The address and port must both be specified, because TCPIP cannot actively initiate a connection without knowing the destination address and port.

Passive Open: If your program is offering a service to anyone who wants it, specify an address of UNSPECIFIEDaddress and a port of UNSPECIFIEDport. You can specify a particular address and port if you want to accept an active open only from a certain foreign application.

ReturnCode: Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
CONNECTIONalreadyEXISTS
DROPPEDbyOPERATOR (**TcpWaitOpen** only)
LOCALportNOTavailable
NOsuchCONNECTION
NOTyetBEGUN
OPENrejected (**TcpWaitOpen** only)
PARAMlocalADDRESS
PARAMstate
PARAMtimeout
PARAMunspecADDRESS
PARAMunspecPORT
REMOTEreset (**TcpWaitOpen** only)
SOFTWAREerror
TCPipSHUTDOWN
TIMEOUTconnection (**TcpWaitOpen** only)
TIMEOUTopen (**TcpWaitOpen** only)
TOOmanyOPENS
UNEXPECTEDsyn (**TcpWaitOpen** only)
UNREACHABLEnetwork (**TcpWaitOpen** only)
WRONGsecORprc (**TcpWaitOpen** only)
ZEROresources.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

TcpFSend, TcpSend, and TcpWaitSend

TcpFSend and **TcpSend** are the asynchronous ways of sending data on a TCP connection. Both procedures return to your program immediately (do not wait under any circumstance).

TcpWaitSend is a simple synchronous method of sending data on a TCP connection. It does not return immediately if the TCPIP address space has insufficient space to accept the data being sent.

TcpFSend and **TcpSend** differ in the way that they handle VMCF when the TCPIP address space has insufficient buffer space to accept the data being sent. Both start by issuing a VMCF SEND function to transfer your data. Normally, the TCPIP address space issues a VMCF RECEIVE, thus completing the VMCF transaction.

In the case of insufficient buffer space, TCPIP responds to **TcpSend** with a VMCF REJECT, completing the VMCF transaction (unsuccessfully). When space becomes available, another complete VMCF transaction is performed to send a BUFFERspaceAVAILABLE notification.

In the case of a **TcpFSend** with insufficient buffer space, TCPIP does not respond to the VMCF SEND until buffer space becomes available, at which time the transaction is completed with a VMCF RECEIVE.

TcpSend returns to your program after the VMCF response from TCPIP is received. In contrast, because the VMCF response from **TcpFSend** may be delayed, **TcpFSend** returns before the VMCF response is received. An OK return code from **TcpFSend** indicates only the successful initiation of the VMCF transaction.

The advantage of **TcpFSend** is that the VMCF transactions necessary to send data are reduced in the case where a program can send data faster than the TCP connection can carry it. Its disadvantages are that it is limited to 50 outstanding VMCF sends and therefore 50 **TcpFSends**, and is slightly more complicated to use because you have to wait for an FSENDresponse notification (generated internally by the TCP interface) between successive **TcpFSends**.

The advantage of **TcpSend** is that it does not involve an outstanding VMCF transaction. Thus, there is no imposed VMCF-related limit. Also, **TcpSend** is simpler to use because you can issue successive **TcpSend**s without waiting for a notification. The disadvantage of **TcpSend** is that it is less efficient than **TcpFSend** if your program can send data faster than the TCP connection can carry it.

Your program can issue successive **TcpWaitSend** calls. Buffer shortage conditions are handled transparently. Any errors that occur are likely to be nonrecoverable errors, or are caused by a connection that has terminated.

If you receive any of the codes listed for Reason in the CONNECTIONstateCHANGED notification, except for OK, the connection was terminated for the indicated reason. Your program should not issue a **TcpClose**, but the connection is not completely terminated until your program receives a NONEXISTENT notification for the connection.

```
procedure TcpFSend
        (
            Connection: ConnectionType;
            Buffer: Address31Type;
            BufferLength: integer;
            PushFlag: Boolean;
            UrgentFlag: Boolean;
    var     ReturnCode: integer
        );
        external;
```

```
procedure TcpSend
        (
            Connection: ConnectionType;
            Buffer: Address31Type;
            BufferLength: integer;
            PushFlag: Boolean;
            UrgentFlag: Boolean;
    var     ReturnCode: integer
        );
        external;
```

```
procedure TcpWaitSend
        (
            Connection: ConnectionType;
            Buffer: Address31Type;
            BufferLength: integer;
            PushFlag: Boolean;
            UrgentFlag: Boolean;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
Connection:	Is the connection number, as returned by **TcpOpen** or **TcpWaitOpen** in the Connection field of the StatusInfoType record.
Buffer:	Is the address of the buffer containing the data to send.
BufferLength:	Is the length of data in the buffer. Maximum is 8 192.
PushFlag:	Is set to force the data, and previously queued data, to be sent immediately to the foreign application.
UrgentFlag:	Is set to mark the data as 'urgent'. The semantics of urgent data is dependent on your application. **Note:** Use urgent data with caution. If the foreign application follows the Telnet-style use of urgent data, it may flush all urgent data until a special character sequence is encountered.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
BADlengthARGUMENT
CANNOTsendDATA

FATALerror
FSENDstillPENDING
NObufferSPACE (**TcpSend** only)
NOsuchCONNECTION
NOTyetBEGUN
NOTyetOPEN
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Usage Notes:

1. A successful **TcpFSend**, **TcpSend**, and **TcpWaitSend** means that TCP has received the data to be sent and stored it in its internal buffers. TCP then puts the data in packets and transmits it when the conditions permit.
2. Data sent in a **TcpFSend**, **TcpSend**, or **TcpWaitSend** request may be split up into numerous packets by TCP, or the data may wait in TCP's buffer space and share a packet with other **TcpFSend**, **TcpSend**, or **TcpWaitSend**, requests.
3. The PushFlag is used to give the user the ability to affect when TCP sends the data.

   Setting the PushFlag to FALSE allows TCP to buffer the data and wait until it has enough data to transmit so as to utilize the transmission line more efficiently. There can be some delay before the foreign host receives the data.

   Setting the PushFlag to TRUE instructs TCP to packetize and transmit any buffered data from previous Send requests along with the data in the current **TcpFSend**, **TcpSend**, or **TcpWaitSend** request without delay or consideration of transmission line efficiency. A successful send does not imply that the foreign application has actually received the data, only that the data will be sent as soon as possible.
4. **TcpWaitSend** is intended for programs that manage a single TCP connection. It is not suitable for use by multiconnection servers.

TcpFReceive, TcpReceive, and TcpWaitReceive

TcpFReceive and **TcpReceive** are the asynchronous ways of specifying a buffer to receive data for a given connection. Both procedures return to your program immediately. A return code of OK means that the request has been accepted. When received data has been placed in your buffer, your program receives a DATAdelivered notification. If your program uses **TcpFReceive**, it may receive an FRECEIVEerror notification instead of DATAdelivered, indicating that the receive request was rejected, or that it was initially accepted but was later canceled due to connection closing.

TcpWaitReceive is the synchronous interface for receiving data from a TCP connection. **TcpWaitReceive** does not return to your program until data has been received into your buffer, or until an error occurs. It is therefore not necessary that **TcpWaitReceive** receive a notification when data is delivered. The BytesRead parameter is set to the number of bytes received by the data delivery, but if the number is less than zero, the parameter indicates an error.

TcpReceive uses a complete VMCF transaction (SEND by your address space followed by REJECT by TCP/IP address space) to tell the TCPIP address space that your program is ready to receive, and another complete VMCF transaction (SEND by TCPIP address space followed by RECEIVE by your address space) to deliver the received data. In contrast, the entire **TcpFReceive** cycle is completed in one VMCF transaction. The TCP interface does a VMCF SEND/RECEIVE to inform TCPIP that your program is ready to receive. This transaction remains uncompleted until data is ready to be placed in your buffer. At that time the TCPIP address space does a VMCF REPLY, completing the transaction.

The advantage of **TcpFReceive** is that it requires less VMCF transactions to receive data, thus increasing efficiency. The disadvantage is that each outstanding **TcpFReceive** means an outstanding VMCF transaction. You are limited to 50 outstanding VMCF transactions (per address space), thus 50 outstanding **TcpFReceives**.

The advantage of **TcpReceive** is that you are not subject to the limit of 50 outstanding receives (per address space). The disadvantage is that there are twice as many VMCF transactions involved in receiving data, thus more overhead.

The only programming difference between **TcpFReceive** and **TcpReceive** is the generation of FRECEIVEerror notifications for **TcpFReceive**.

```
procedure TcpFReceive
        (
            Connection: ConnectionType;
            Buffer: Address31Type;
            BytesToRead: integer;
    var     ReturnCode: integer
        );
        external;
```

```
procedure TcpReceive
        (
            Connection: ConnectionType;
            Buffer: Address31Type;
            BytesToRead: integer;
    var     ReturnCode: integer
        );
        external;
```

```
procedure TcpWaitReceive
        (
            Connection: ConnectionType;
            Buffer: Address31Type;
            BytesToRead: integer;
    var     BytesRead: integer
        );
        external;
```

Parameter	Description
Connection:	Is the connection number, as returned by **TcpOpen** or **TcpWaitOpen** in the Connection field of the StatusInfoType record.
Buffer:	Is the address of the buffer to contain the received data.
BytesToRead:	Is the size of the buffer. TCPIP usually buffers the incoming data until this many bytes are received. Data is delivered sooner if the sender specified the PushFlag, or if the sender does a **TcpClose** or equivalent. The largest usable buffer is 8 192 bytes. Specifying BytesToRead of more than 8 192 bytes may not cause an error return, but only 8 192 bytes of the buffer will be used. **Usage Note:** The order of **TcpFReceive** or **TcpReceive** calls on multiple connections, and the order of DATAdelivered notifications among the connections, are not necessarily related.
BytesRead:	Will be set when **TcpWaitReceive** returns. If it is greater than zero, it indicates the number of bytes received into your buffer. If it is less than or equal to zero, it indicates an error. **Possible BytesRead Error values:** OK+ ABNORMALcondition FATALerror TIMEOUTopen+ UNREACHABLEnetwork+ BADlengthARGUMENT NOsuchCONNECTION NOTyetBEGUN NOTyetOPEN OPENrejected+ RECEIVEstillPENDING REMOTEreset+ UNEXPECTEDsyn+ WRONGsecORprc+ DROPPEDbyOPERATOR+ FATALerror+ KILLEDbyCLIENT+

TCPipSHUTDOWN+
TIMEOUTconnection+
REMOTEclose.

Usage Notes (TcpWaitReceive):

1. For BytesRead OK, the function was initiated, but the connection is no longer receiving for an unspecified reason. Your program does not have to issue **TcpClose**, but the connection is not completely terminated until a NONEXISTENT notification is received for the connection.
2. For BytesRead REMOTEclose, the foreign host has closed the connection. Your program should respond with **TcpClose**.
3. If you receive any of the codes marked with +, the function was initiated but the connection has now been terminated (see Usage Note 3 on page 2-8). Your program should not issue **TcpClose**, but the connection is not completely terminated until NONEXISTENT notification is received for the connection.
4. **TcpWaitReceive** is intended to be used by programs that manage a single TCP connection. It is not suitable for use by multiconnection servers.
5. A return code of TCPipSHUTDOWN may be returned either because the connection initiation has failed, or because the connection has been terminated due to shutdown. In either case, your program should not issue any more TCP/IP calls.

ReturnCode: Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
BADlengthARGUMENT
FATALerror
NOsuchCONNECTION
NOTyetBEGUN
NOTyetOPEN
RECEIVEstillPENDING
REMOTEclose
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

TcpClose

Use this procedure to begin the TCP one-way closing sequence. During this closing sequence, you, the local client, cannot send any more data. Data may be delivered to you until the foreign application also closes. **TcpClose** also causes all data sent on that connection by your application, and buffered by TCPIP, to be sent to the foreign application immediately.

```
procedure TcpClose
        (
            Connection: ConnectionType;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
Connection:	Is the connection number, as returned by **TcpOpen** or **TcpWaitOpen** in the Connection field of the StatusInfoType record.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
ALREADYclosing
NOsuchCONNECTION
NOTyetBEGUN
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Usage Notes:

1. If you receive the notification CONNECTIONstateCHANGED with a NewState of SENDINGonly, the remote application has done **TcpClose** (or equivalent function) and is receiving only. Respond with **TcpClose** when you have finished sending data on the connection.
2. The connection is fully closed when you receive the notification CONNECTIONstateCHANGED, with a NewState field set to NONEXISTENT.

TcpAbort

Use this procedure to shut down a specific connection immediately. Data sent by your application on the aborted connection may be lost. TCP sends a reset packet to notify the foreign host that you have aborted the connection, but there is no guarantee that the reset will be received by the foreign host.

```
procedure TcpAbort
        (
            Connection: ConnectionType;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
Connection:	Is the connection number, as returned by **TcpOpen** or **TcpWaitOpen** in the Connection field of the StatusInfoType record.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
NOsuchCONNECTION
NOTyetBEGUN
TCPipSHUTDOWN.

The connection is fully terminated when you receive the notification CONNECTIONstateCHANGED with the NewState field set to NONEXISTENT.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

TcpStatus

Use **TcpStatus** to obtain the current status of a TCP connection. Your program sets the Connection field of the ConnectionInfo record to the number of the connection whose status you want.

```
procedure TcpStatus
      (
   var    ConnectionInfo: StatusInfoType;
   var    ReturnCode: integer
      );
      external;
```

Parameter	Description
ConnectionInfo:	If ReturnCode is OK, the following fields are returned.
OpenAttemptTimeout:	If the connection is in the process of being opened (including a passive open), this field is set to the number of seconds remaining before the open is terminated if it has not completed. Otherwise, it is set to WAITforever.
BytesToRead:	Is the number of bytes of incoming data queued for your program (waiting for **TcpReceive**, **TcpFReceive**, or **TcpWaitReceive**).
UnackedBytes:	Is the number of bytes sent by your program but not yet sent to the foreign TCP, or the number of bytes sent to the foreign TCP, but not yet acknowledged.
ConnectionState:	Is the current connection state.

LocalSocket:	Is the local socket, consisting of a local address and a local port.
ForeignSocket:	Is the foreign socket, consisting of a foreign address and a foreign port.

ReturnCode: Indicates the success or failure of the call.

Possible ReturnCode values:

OK
ABNORMALcondition
NOsuchCONNECTION
NOTyetBEGUN
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Usage Note: Your program cannot monitor connection state changes exclusively through polling with **TcpStatus**. It must receive CONNECTIONstateCHANGED notifications through **GetNextNote**, for the TCP interface to work properly.

Ping Interface

The Ping Interface lets a client send an ICMP echo request to a foreign host. You must call either the **BeginTcpIp** or the **StartTcpNotice** initialization routine before you can begin using the Ping Interface.

PingRequest

Use this procedure to send an ICMP echo request to a foreign host. When a response is received or the timeout limit is reached, you receive a PingResponse notification.

The **PingRequest** procedure is used by the PING user command. Refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: User's Guide* for more information on the PING command.

```
procedure PingRequest
        (
            ForeignAddress: InternetAddressType;
            Length: integer;
            Timeout: integer;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
ForeignAddress:	Address of foreign host to be 'pinged'.

Length: The length of the ping packet, excluding the IP header. The range of values for this field are 8 to 512 bytes.

Timeout: How long to wait for a response, in seconds.

ReturnCode: Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
BADlengthARGUMENT
CONNECTIONalreadyEXISTS
NObufferSPACE
NOTyetBEGUN.

Usage Note: CONNECTIONalreadyEXISTS, in this context, means a ping request is already outstanding.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Monitor Procedures

Two monitor procedures, **MonCommand** and **MonQuery**, provide a mechanism for querying and controlling the TCPIP address space.

MonCommand and **MonQuery** are described in the CMMON member of the TCPIP.COMMMAC data set. Any program using these procedures must include CMMON after the include statements for CMCOMM and CMCLIEN.

MonCommand

Use the **MonCommand** procedure to instruct TCPIP to read a specific data set and execute the commands found there. This procedure updates TCP/IP internal tables and parameters while the TCPIP address space is running. For example, the type and destination of run-time tracing can be modified dynamically using **MonCommand**. This procedure is used by the OBEYFILE command (refer to *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* for more information about the OBEYFILE command). You must be in the TCPIP obey list to use the **MonCommand** procedure.

```
procedure MonCommand
        (
    const  FileSpec: SpecOfFileType;
    var    ReturnCode: integer
        );
        external;
```

Parameter	Description
FileSpec:	Specifies fully a data set in a manner that allows access to that data set. The TCPIP address space must be authorized to access the data set. The SpecOfFileType record is shown on page 2-12.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
ERRORinPROFILE
HASnoPASSWORD
INCORRECTpassword
INVALIDuserID
INVALIDvirtualADDRESS
MINIDISKinUSE
MINIDISKnotAVAILABLE
NOTyetBEGUN
PROFILEnotFOUND
SOFTWAREerror
TCPipSHUTDOWN
UNAUTHORIZEDuser
UNIMPLEMENTEDrequest.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

MonQuery

The **MonQuery** procedure is used to obtain status information, or to request TCPIP to perform certain actions. This procedure is used by the NETSTAT command. For more information about the NETSTAT command, refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: User's Guide*.

```
procedure MonQuery
        (
            QueryRecord: MonQueryRecordType;
            Buffer: integer;
            BufSize: integer;
    var     ReturnCode: integer;
    var     Length: integer
        );
        external;
```

Parameter	Description
Buffer:	Is the address of the buffer to receive data.
BufSize:	Is the size of the buffer.

ReturnCode: Indicates success or failure of call.

Length: Is the length of the data returned in the buffer.

QueryRecord: Your program sets up a QueryRecord to specify the type of status information to be retrieved. The MonQueryRecordType is shown in Figure 2-6.

```
MonQueryRecordType =
      record
      case QueryType: MonQueryType of
      QUERYhome, QUERYgateways, QUERYcontrolBLOCKS,
      QUERYstartTIME, QUERYtelnetSTATUS,
      QUERYdevicesANDlinks,
      QUERYhomeONLY: ();
      QUERYudpPORTowner:
      (
        QueryPort: PortType
      );
      COMMANDcpCMD:
      (
        CpCmd: WordType
      );
      COMMANDdropCONNECTION:
      (
        Connection: ConnectionType
      );
      end; { MonQueryRecordType }
```

Figure 2-6. Monitor Query Record

The only QueryType values available for customer use are:

QUERYhomeONLY: Is used to obtain a list of the home internet addresses recognized by your TCPIP. Your program sets the Buffer to the address of a variable of type HomeOnlyListType, and the BufSize to its length. When **MonQuery** returns, Length is set to the length of the Buffer that was used, if ReturnCode is OK. Divide the Length by sizeof(InternetAddressType) to get the number of the home addresses that are returned.

COMMANDdropCONNECTION:
Is used to instruct the TCPIP address space to drop a TCP connection. The connection is reset, and the client process owning the connection is sent a NONEXISTENT notification with the Reason field set to DROPPEDbyOPERATOR. Your program sets the Connection field to the number of the connection to be dropped. The connection number is the number displayed by the NETSTAT CONN or the NETSTAT TELNET command, and is not the same number used to refer to the connection by the client program that owns the connection. Refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: User's Guide* for

information on the NETSTAT command. The address space running your program that uses COMMANDdropCONNECTION must be in the TCPIP obey list.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
NOTyetBEGUN
TCPipSHUTDOWN
UNAUTHORIZEDuser
UNIMPLEMENTEDrequest.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

UDP Communication Procedures

This section describes the programming interface for the User Datagram Protocol (UDP) provided in the TCP/IP for MVS product.

UdpOpen

This procedure requests acceptance of UDP datagrams on the specified socket and allows datagrams to be sent from the specified socket. When the socket port is unspecified, UDP selects a port and returns it to the socket port field. When the socket address is unspecified, UDP uses the default local address. If specified, the address must be a valid home address for your node.

Usage Note: When the local address is specified, only the UDP datagrams addressed to it are delivered.

If the ReturnCode indicates the open was successful, use the returned ConnIndex value on any further actions pertaining to this UDP socket.

```
procedure UdpOpen
      (
var  LocalSocket: SocketType;
var  ConnIndex: ConnectionIndexType;
var  ReturnCode: CallReturnCodeType
      );
      external;
```

Parameter	Description
LocalSocket:	Is the local socket (address and port pair).
ConnIndex:	Is the ConnIndex value returned from **UdpOpen**.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
LOCALportNOTavailable
NOTyetBEGUN
SOFTWAREerror
TCPipSHUTDOWN
UDPlocalADDRESS
UDPzeroRESOURCES.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Usage Note: If a **UdpOpen** call returns UDPzeroRESOURCES, and your application handles UDPresourcesAVAILABLE notifications, you receive a UDPresourcesAVAILABLE notification when sufficient resources are available to process a **UdpOpen** call. The first **UdpOpen** your program issues after a UDPresourcesAVAILABLE notification is guaranteed not to get the UDPzeroRESOURCES return code.

UdpSend

This procedure sends a UDP datagram to the specified foreign socket. The source socket is the local socket selected in the **UdpOpen** that returned the ConnIndex value that was used. The buffer does not include the UDP header. This header is supplied by TCPIP.

When there is no buffer space to process the data, an error is returned. In this case, wait for a subsequent UDPdatagramSPACEavailable notification.

```
procedure UdpSend
        (
            ConnIndex: ConnectionIndexType;
            ForeignSocket: SocketType;
            BufferAddress: integer;
            Length: integer;
    var     ReturnCode: CallReturnCodeType
        );
        external;
```

Parameter	Description
ConnIndex:	Is the ConnIndex value returned from **UdpOpen**.
ForeignSocket:	Is the foreign socket (address and port) to whom the datagram is to be sent.
BufferAddress:	Is the address of your buffer containing the UDP datagram to be sent, excluding UDP header.
Length:	Is the length of the datagram to be sent, excluding UDP header. Maximum is 8 492 bytes.

ReturnCode: Indicates success or failure of call.

Possible ReturnCode values:

OK
BADlengthARGUMENT
NObufferSPACE
NOsuchCONNECTION
NOTyetBEGUN
SOFTWAREerror
TCPipSHUTDOWN
UDPunspecADDRESS
UDPunspecPORT.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

UdpNReceive

This procedure notifies the TCPIP address space that you are willing to receive UDP datagram data. This call returns immediately. The data buffer is not valid until you receive a UDPdatagramDELIVERED notification.

```
procedure UdpNReceive
        (
            ConnIndex: ConnectionIndexType;
            BufferAddress: integer;
            BufferLength: integer;
    var     ReturnCode: CallReturnCodeType
        );
        external;
```

Parameter	Description
ConnIndex:	Is the ConnIndex value returned from **UdpOpen**.
BufferAddress:	Is the address of your buffer that will be filled with a UDP datagram.
BufferLength:	Is the length of your buffer. If you specify a length larger than 8 492 bytes, only the first 8 492 bytes are used.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
NOsuchCONNECTION
NOTyetBEGUN
RECEIVEstillPENDING
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

UdpReceive

This procedure notifies the TCPIP address space that you are willing to receive UDP datagram data.

UdpReceive is for compatibility with old programs only. New programs should use the **UdpNReceive** procedure, which allows you to specify the size of your buffer.

If you use **UdpReceive**, TCPIP may put a datagram of up to 2 012 bytes in your buffer. If a larger datagram is sent to your port when **UdpReceive** is pending, the datagram is discarded without notification.

Note: No data is transferred from the TCPIP address space in this call. It only tells TCPIP that you are waiting for a datagram. Data has been transferred when a UDPdatagramDELIVERED notification is received.

```
procedure UdpReceive
        (
            ConnIndex: ConnectionIndexType;
            DatagramAddress: integer;
    var     ReturnCode: CallReturnCodeType
        );
        external;
```

Parameter	Description
ConnIndex:	Is the ConnIndex value returned from **UdpOpen**.
DatagramAddress:	Is the address of your buffer that will be filled in with a UDP datagram.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
NOsuchCONNECTION
NOTyetBEGUN
SOFTWAREerror
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

UdpClose

This procedure closes the UDP socket specified in the ConnIndex field. All incoming datagrams on this connection are discarded.

```
procedure UdpClose
      (
          ConnIndex: ConnectionIndexType;
   var    ReturnCode: CallReturnCodeType
      );
      external;
```

ConnIndex: Is the ConnIndex value returned from **UdpOpen.**

ReturnCode: Indicates success or failure of call.

Possible ReturnCode values:

OK
ABNORMALcondition
FATALerror
NOsuchCONNECTION
NOTyetBEGUN
SOFTWAREerror
TCPipSHUTDOWN.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Raw IP Interface

The Raw IP interface lets a client program send and receive arbitrary IP packets on any IP protocol except TCP and UDP. Only one client can use any given protocol at one time. Only clients in the obey list can use the Raw IP interface. For further information on the obey list, refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* book.

RawIpOpen

This command tells the TCPIP address space that the client wants to send and receive packets of the specified protocol.

Protocols 6 and 17 must not be used. They specify the TCP (6) and UDP (17) protocols. When you specify 6, 17 or a protocol that has been opened by another address space, you receive the LOCALportNOTavailable return code.

```
procedure RawIpOpen
      (
          ProtocolNo: integer;
   var    ReturnCode: integer
      );
      external;
```

Parameter	Description
ProtocolNo:	Is the number of the IP protocol.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
LOCALportNOTavailable
NOTyetBEGUN
SOFTWAREerror
TCPipsHUTDOWN
UNAUTHORIZEDuser.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Usage Note: You can open the ICMP protocol, but your program receives only those ICMP packets that are not interpreted by the TCPIP address space.

RawIpReceive

Use this command to specify a buffer to receive Raw IP packets of the specified protocol. You get the notification RAWIPpacketsDELIVERED when a packet is put in the buffer.

```
procedure RawIpReceive
        (
            ProtocolNo: integer;
            Buffer: Address31Type;
            BufferLength: integer;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
ProtocolNo:	Is the number of the IP protocol.
Buffer:	Address of your buffer.
BufferLength:	Length of your buffer. If you specify a length greater than 8 492 bytes, only the first 8 492 bytes are used.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
NOsuchCONNECTION
NOTyetBEGUN
SOFTWAREerror
TCPipsHUTDOWN
UNAUTHORIZEDuser.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

RawIpSend

This command sends IP packets of the given protocol number. The entire packet, including the IP header, must be in the buffer. The TCPIP address space uses the total length field of the IP header to determine where each packet ends. Subsequent packets begin at the next double-word (8-byte) boundary within the buffer.

The packets in your buffer are transmitted as is with the following exceptions.

- They may be fragmented. The fragment offset and flag fields in the header are filled in.
- The version field in the header is filled in.
- The checksum field in the header is filled in.
- The source address field in the header is filled in.

You get the return code NOsuchCONNECTION if the client is not handling the protocol, or if a packet in the buffer has another protocol. The return code BADlengthARGUMENT is received when:

- The DataLength is less than 40 bytes or more than 8KB.
- NumPackets is 0.
- A packet is greater than 2 048 bytes.
- All packets do not fit into DataLength.

A ReturnCode value of NObufferSPACE indicates that the data is rejected because TCPIP is out of buffers. When buffer space is available, the notification RAWIPspaceAVAILABLE is sent to the client.

```
procedure RawIpSend
        (
            ProtocolNo: integer;
            Buffer: Address31Type;
            DataLength: integer;
            NumPackets: integers;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
ProtocolNo:	Is the number of the IP protocol.
Buffer:	Address of your buffer containing packets to send.
DataLength:	Total length of data in your buffer.
NumPackets:	Number of packets in your buffer.
ReturnCode:	Indicates success or failure of call.

Usage Note: If your buffer contains multiple packets to send, some of the packets may have been sent even if ReturnCode is not OK.

Possible ReturnCode values:

OK
BADlengthARGUMENT
NObufferSPACE
NOsuchCONNECTION
NOTyetBEGUN
SOFTWAREerror
TCPipSHUTDOWN
UNAUTHORIZEDuser.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

RawIpClose

This command tells the TCPIP address space that the client does not handle the protocol any longer. Any queued incoming packets are discarded.

When the client is not handling the protocol, a return code of NOsuchCONNECTION is received.

```
procedure RawIpClose
        (
            ProtocolNo: integer;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
ProtocolNo:	Is the number of the IP protocol.
ReturnCode:	Indicates success or failure of call.

Possible ReturnCode values:

OK
NOsuchCONNECTION
NOTyetBEGUN
SOFTWAREerror
TCPipSHUTDOWN
UNAUTHORIZEDuser.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

Timer Routines

The timer routines are used with the TCP/UDP/IP interface. You must call either the **BeginTcpIp** or the **StartTcpNotice** initialization routine before you can begin using the timer routines.

CreateTimer

This procedure allocates a timer. The timer is not set in any way. Refer to the **SetTimer** procedure to activate the timer.

```
procedure CreateTimer
        (
    var     T: TimerPointerType
        );
        external;
```

Parameter	Description
T:	Is set to a timer pointer that can be used in subsequent **SetTimer**, **ClearTimer**, and **DestroyTimer** calls.

ClearTimer

This procedure resets the timer to prevent it from timing out.

```
procedure ClearTimer
        (
            T: TimerPointerType
        );
        external;
```

Parameter	Description
T:	A timer pointer, as returned by a previous **CreateTimer** call.

SetTimer

This procedure sets a timer to expire after a specified time interval. Specify the amount of time in seconds. When it times out, you receive the TIMERexpired notification, which contains the data and the timer pointer.

Usage Note: This procedure resets any previous time interval set on this timer.

```
procedure SetTimer
        (
            T: TimerPointerType;
            AmountOfTime: integer;
            Data: integer
        );
        external;
```

Parameter	Description
T:	A timer pointer, as returned by a previous **CreateTimer** call.

AmountOfTime:	The time interval in seconds.
Data:	An integer value to be returned with the TIMERexpired notification.

DestroyTimer

This procedure deallocates or "frees" a timer that you created.

```
procedure DestroyTimer
      (
   var    T: TimerPointerType
      );
      external;
```

Parameter	Description
T:	A timer pointer, as returned by a previous **CreateTimer** call.

Host Lookup Routines

The host lookup routines (with the exception of **GetHostResol**) are declared in the CMINTER member of the TCPIP.COMMMAC data set. The host lookup routine **GetHostResol** is declared in the CMRESGLB member of the TCPIP.COMMMAC data set. Any program using these procedures must include CMINTER or CMRESGLB after the include statements for CMCOMM and CMCLIEN.

GetHostNumber

The main purpose of the **GetHostNumber** procedure is to resolve a host name into an internet address.

GetHostNumber uses a table lookup to convert the name of a host to an internet address, and returns this address to the HostNumber field. When the name is a dotted-decimal number, **GetHostNumber** returns the integer represented by that dotted-decimal. The dotted-decimal representation of a 32-bit number has one decimal integer for each of the 4 bytes, separated by dots. For example, `14.0.0.7` for `X'0E000007'`. Refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* book for information on how to create host lookup tables.

The HostNumber field is set to NOhost if the host is not found.

```
procedure GetHostNumber
      (
   const  Name: string;
   var    HostNumber: InternetAddressType
      );
      external;
```

Parameter	Description
Name:	The name or dotted-decimal number to be converted.
HostNumber:	Set to the converted address, or NOhost if conversion fails.

GetHostResol

The main purpose of the **GetHostResol** procedure is to resolve a host name into an internet address by using a name server.

GetHostResol passes the query to the remote name server through the resolver. The name server converts the name of a host to an internet address, and returns this address in the HostNumber field. If the name server does not respond or does not find the name, the host name is converted to a host number by table lookup. When the name is a dotted-decimal number, the integer represented by that dotted-decimal is returned. The dotted-decimal representation of a 32-bit number has one decimal integer for each of the 4 bytes, separated by dots. For example, `14.0.0.7` for X'0E000007'.

The HostNumber field is set to NOhost if the host is not found.

```
procedure GetHostResol
        (
    const  Name: string;
    var    HostNumber: InternetAddressType
        );
        external;
```

Parameter	Description
Name:	The name or dotted-decimal number to be converted.
HostNumber:	Set to the converted address, or NOhost if conversion fails.

GetHostString

The **GetHostString** procedure call uses a table lookup to convert an internet address to a host name, and returns this string in the Name field. The first host name found in the lookup is returned. If no host name is found, a gateway or network name is returned. If no gateway or network name is found, a null string is returned.

```
procedure GetHostString
        (
            Address: InternetAddressType;
    var     Name: SiteNameType
        );
        external;
```

Parameter	Description
Address:	The address to be converted.
Name:	Set to the corresponding host, gateway, or network name, or to null string if no match found.

GetIdentity

This procedure returns the following information:

- The user ID of the TSO user or the job name of a batch job that has invoked it
- The host machine name
- The network domain name
- The user ID of the TCPIP address space.

The host machine name and domain name are extracted from the HostName and DomainOrigin statements, respectively, in the *userid*.TCPIP.DATA data set. If the *userid*.TCPIP.DATA data set does not exist, the TCPIP.TCPIP.DATA data set is used. If a HostName statement is not specified, then the default host machine name is the name specified by the TCP/IP for MVS installer during installation (refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance*). The TCPIP address space user ID is extracted from the TcpipUserid statement in the *userid*.TCPIP.DATA data set; if the statement is not specified, the default is TCPIP.

```
procedure GetIdentity
      (
   var    UserId:  DirectoryNameType;
   var    HostName, DomainName: String;
   var    TcpIpServiceName: DirectoryNameType;
   var    Result: integer
      );
      external;
```

Parameter	Description
UserId:	The user ID of the TSO user or the job name of a batch job that has invoked **GetIdentity**.
HostName:	The host machine name.
DomainName:	The network domain name.
TcpIpServiceName:	The user ID of the TCPIP address space.
Result:	Indicates success or failure of the call.

IsLocalAddress

This procedure queries the TCPIP address space to determine whether the HostAddress is one of the addresses recognized for this host. If the address is local, it returns OK. If the address is not local, it returns NONlocalADDRESS.

```
procedure IsLocalAddress
        (
            HostAddress: InternetAddressType;
    var     ReturnCode: integer
        );
        external;
```

Parameter	Description
HostAddress:	The host address to be tested.
ReturnCode:	Indicates whether host address is local, or may indicate an error.

Possible ReturnCode values:

OK
NONlocalADDRESS
TCPipSHUTDOWN
ABNORMALcondition
FATALerror.

For a description of Pascal ReturnCodes, see Table 2-14 on page 2-55.

IsLocalHost

This procedure returns the correct host class for Name, which may be a host name or a dotted-decimal address.

The host classes are:

HOSTlocal	Is an internet address for the local host.
HOSTloopback	Is one of the dummy internet addresses used to designate various levels of loopback testing.
HOSTremote	Is a known host name for some remote host.
HOSTunknown	Is an unknown host name (or other error).

```
procedure IsLocalHost
        (
    const   Name: string;
    var     Class: HostClassType
        );
        external;
```

Parameter	Description
Name:	Is the host name.
Class:	Is the host class.

Other Routines

GetSmsg

Your program should call this procedure after receiving an SMSGreceived notification. Each call to **GetSmsg** retrieves one queued Smsg. Your program should exhaust all queued Smsgs, by calling **GetSmsg** repeatedly until the Success field returns with a value of FALSE. After a value of FALSE is returned, do not call GetSmsg again until you receive another SMSGreceived notification.

Refer to "Sending and Receiving Special Messages" on page A-1 for information on the SMSG command and how to enable the reception of Smsgs.

```
procedure GetSMsg
      (
   var    Smsg: SmsgType;
   var    Success: Boolean;
      );
      external;
```

Parameter	Description
Smsg:	Will be set to the returned Smsg if Success is set to TRUE.
Success:	TRUE if Smsg returned, otherwise FALSE.

ReadXlateTable

This routine reads the binary translation table data set specified by Tablename, and fills in the AtoETable and EtoATable translation tables.

```
procedure ReadXlateTable
      (
   var    TableName: DirectoryNameType;
   var    AtoETable: AtoEType;
   var    EtoATable: EtoAType;
   var    TranslateTableSpec: SpecOfFileType;
   var    ReturnCode: integer
      );
      external;
```

Parameter	Description
TableName:	The name of the translate table. ReadXlateTable tries to read *userid*.TableName.TCPXLBIN. If that file exists but it has a bad format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If *userid*.TableName.TCPXLBIN does not exist, ReadXlateTable tries to read TCPIP.TableName.TCPXLBIN. ReturnCode reflects the status of reading that file.
AtoETable:	Will be filled in with ASCII-to-EBCDIC table if return code is OK.
EtoATable:	Will be filled in with EBCDIC-to-ASCII table if return code is OK.
TranslateTableSpec:	If ReturnCode is OK, TranslateTableSpec contains the complete specification of the file that ReadXlateTable used. If ReturnCode is not OK, TranslateTableSpec contains the complete specification of the last file that ReadXlateTable tried to use.
ReturnCode:	Indicates success or failure of a call.

Possible ReturnCode values:

OK
ERROROpeningORreadingFILE
FILEformatINVALID.

SayCalRe

This function returns a printable string describing the return code passed in CallReturn.

```
function SayCalRe
        )
            CallReturn: integer
        ):
        WordType;
        external;
```

Parameter	Description
CallReturn:	The return code to be described.

SayConSt

This function returns a printable string describing the connection state passed in State. For example, if **SayConSt** is invoked with the type identifier RECEIVINGonly, it returns the message "Receiving only".

```
function SayConSt
        (
            State: ConnectionStateType
        ):
        Wordtype;
        external;
```

Parameter	Description
State:	The connection state to be described.

SayIntAd

This function converts the internet address specified by InternetAddress to a printable string. The address is looked up in TCPIP.HOSTS.ADDRINFO, and the name is returned if found. If it is not found, the dotted-decimal format of the address is returned.

```
function SayIntAd
        (
            InternetAddress: InternetAddressType
        ):
        WordType;
        external;
```

Parameter	Description
Internet Address:	The internet address to be converted.

SayIntNum

This function converts the internet address specified by InternetAddress to a printable string, in dotted-decimal form.

```
function SayIntNum
        (
            InternetAddress: InternetAddressType
        ):
        Wordtype;
        external;
```

Parameter	Description
Internet Address:	The internet address to be converted.

SayNotEn

This function returns a printable string describing the notification enumeration type passed in Notification. For example, if **SayNotEn** is invoked with the type identifier EXTERNALinterrupt, it returns the message "Other external Interrupt received".

```
function SayNotEn
        (
            Notification: NotificationEnumType
        );
        Wordtype;
        external;
```

Parameter	Description
Notification:	The notification enumeration type to be described.

SayPorTy

This function returns a printable string describing the port number passed in Port, if it is a well-known port number such as the Telnet port. Otherwise, the EBCDIC representation of the number is returned.

```
function SayPorTy
        (
            Port: PortType
        ):
        WordType;
        external;
```

Parameter	Description
Port:	The port number to be described.

SayProTy

This function converts the protocol type specified by Protocol to a printable string if it is a well-known protocol number such as 6 (TCP). Otherwise, the EBCDIC representation of the number is returned.

```
function SayProTy
        (
            Protocol: ProtocolType
        ):
        WordType;
        external;
```

Parameter	Description
Protocol:	The number of the protocol to be described.

AddUserNote

This procedure can be called to add a USERdefinedNOTIFICATION notification to the note queue and wake up **GetNextNote** if it is waiting for a notification. See Appendix C, "Assembler Calls for the Pascal API" on page C-1 for more information.

Pascal Return Codes

When using Pascal procedure calls, check to determine whether or not the call has been completed successfully. Use the **SayCalRe** function (see "SayCalRe" on page 2-52) to convert the ReturnCode parameter to a printable form.

The **SayCalRe** function converts a return code value into a descriptive message. For example, if **SayCalRe** is invoked with the integer constant BADlengthARGUMENT, it returns the message "Invalid buffer length specified". Refer to Table 2-14 for a description of Pascal return codes and their equivalent message text from **SayCalRe**.

Most return codes are self-explanatory in the context where they occur. The return codes you see as a result of issuing a TCP/UDP/IP request are in the range -128 to 0. Refer to the Explanatory Notes at the end of Table 2-14 for more information.

Table 2-14 (Page 1 of 3). Pascal Language Return Codes

Return Code	Numeric Value	Message Text
OK	0	OK.
ABNORMALcondition[1]	-1	Abnormal condition during inter-address communication. (VMCF Rc = *nn* User = *xxxxxxxx*)
ALREADYclosing	-2	Connection already closing.
BADlengthARGUMENT	-3	Invalid length specified.
CANNOTsendDATA[2]	-4	Cannot send data.
CLIENTrestart	-5	Client reinitialized TCP/IP service.
CONNECTIONalreadyEXISTS	-6	Connection already exists.
DESTINATIONunreachable	-7	Destination address is unreachable.
ERRORinPROFILE	-8	Error in profile data set. Details are in PROFILE.TCPERROR.
FATALerror[3]	-9	Fatal inter-address communications error. (VMCF Rc = *nn* User = *xxxxxxxx*)
HASnoPASSWORD	-10	No password in RACF directory.
INCORRECTpassword	-11	TCPIP not authorized to access data set.

Table 2-14 (Page 2 of 3). Pascal Language Return Codes		
Return Code	**Numeric Value**	**Message Text**
INVALIDrequest	-12	Invalid request.
INVALIDuserID	-13	Invalid user ID.
INVALIDvirtualADDRESS	-14	Invalid virtual address.
KILLEDbyCLIENT	-15	You aborted the connection.
LOCALportNOTavailable	-16	The requested local port is not available.
MINIDISKinUSE	-17	Data set is in use by someone else and cannot be accessed.
MINIDISKnotAVAILABLE	-18	Data set not available.
NObufferSPACE[4]	-19	No more space for data currently available.
NOmoreINCOMINGdata	-20	The foreign host has closed this connection.
NONlocalADDRESS	-21	The internet address is not local to this host.
NOoutstandingNOTIFICATIONS	-22	No outstanding notifications.
NOsuchCONNECTION	-23	No such connection.
NOtcpIPservice	-24	No TCP/IP service available.
NOTyetBEGUN	-25	Not yet begun TCP/IP service.
NOTyetOPEN	-26	The connection is not yet open.
OPENrejected	-27	Foreign host rejected the open attempt.
PARAMlocalADDRESS	-28	TcpOpen error: invalid local address.
PARAMstate	-29	TcpOpen error: invalid initial state.
PARAMtimeout	-30	Invalid timeout parameter.
PARAMunspecADDRESS	-31	TcpOpen error: unspecified foreign address in active open.
PARAMunspecPORT	-32	TcpOpen error: unspecified foreign port in active open.
PROFILEnotFOUND	-33	TCPIP cannot read profile data set.
RECEIVEstillPENDING	-34	Receive still pending on this connection.
REMOTEclose	-35	Foreign host unexpectedly closed the connection.
REMOTEreset	-36	Foreign host aborted the connection.
SOFTWAREerror	-37	Software error in TCP/IP!
TCPipSHUTDOWN	-38	TCP/IP service is being shut down.
TIMEOUTconnection	-39	Foreign host is no longer responding.
TIMEOUTopen	-40	Foreign host did not respond within OPEN timeout.
TOOmanyOPENS	-41	Too many open connections already exist.

Table 2-14 (Page 3 of 3). Pascal Language Return Codes

Return Code	Numeric Value	Message Text
UNAUTHORIZEDuser	-43	You are not authorized to issue this command.
UNEXPECTEDsyn	-44	Foreign host violated the connection protocol.
UNIMPLEMENTEDrequest	-45	Unimplemented TCP/IP request.
UNKNOWNhost	-46	Destination host is not known.
UNREACHABLEnetwork	-47	Destination network is unreachable.
UNSPECIFIEDconnection	-48	Unspecified connection.
VIRTUALmemoryTOOsmall	-49	Client address space has too little storage.
WRONGsecORprc	-50	Foreign host disagreed on security or precedence.
YOURend	-55	Client has ended TCP/IP service.
ZEROresources	-56	TCP cannot handle any more connections now.
UDPlocalADDRESS	-57	Invalid local address for UDP.
UDPunspecADDRESS	-59	Address unspecified when specification necessary.
UDPunspecPORT	-60	Port unspecified when specification necessary.
UDPzeroRESOURCES	-61	UDP cannot handle any more traffic.
FSENDstillPENDING	-62	FSend still pending on this connection.
DROPPEDbyOPERATOR	-79	Connection dropped by operator.
ERRORopeningORreadingFILE	-80	Error opening or reading data set.
FILEformatINVALID	-81	Data set format invalid.

Explanatory Notes:

1. ABNORMALcondition — The actual VMCF return code is available in the external integer variable LastVmcfCode, and is included in the output of **SayCalRe** if called immediately after the error is detected.
2. CANNOTsendDATA — Cannot send data on this connection because the connection state is invalid for sending data.
3. FATALerror — The actual VMCF return code is available in the external integer variable LastVmcfCode, and is included in the output of **SayCalRe** if called immediately after the error is detected.
4. NObufferSPACE — Applies to this connection only. There may still be space available for other connections.

Chapter 3. C Socket Application Program Interface

This chapter describes the C socket API provided with the TCP/IP for MVS product. Use the socket routines when you want your C Language programs to communicate across networks with other programs. You can, for example, make use of socket routines when you write a client program that must communicate with a server program running on another computer.

Knowledge of the C Language programming is required to use the sockets. For further information on C sockets, refer to *IBM AIX Operating System Technical Reference: System Calls and Subroutines* (SC23-2125).

TCP/IP for MVS supports a subset of the socket library described in SC23-2125 (Order Number SX23-0711).

Programming With Sockets

A socket is a communication endpoint between two processes, in effect, a virtual port. Two sockets, one associated with a process on one computer and one associated with another process on another computer, cooperate to allow the two tasks to communicate. Socket applications can also be processes running in different address spaces on the same local host.

The TCP/IP for MVS program supports two types of socket: stream sockets and datagram sockets.

Stream sockets are connected by a set pathway and transmit data reliably in both directions between two processes. Because the pair of sockets are reliable, you use stream sockets most frequently for long communications of multiple packets. In the TCP/IP for MVS program, the TCP and IP protocols handle data delivery for stream sockets.

Datagram sockets, on the other hand, are connectionless and have no fixed pathway. Consequently, datagram sockets transmit data unreliably in both directions. Delivery of datagrams is not guaranteed, and data may be lost or duplicated in either direction. Datagram sockets are generally used for brief communications or client broadcasting where only one packet travels in each direction. The UDP and IP protocols handle data delivery for datagram sockets.

Your choice of the type of socket to use may be limited if you write an application program that follows a well-known application protocol. Well-known application protocols are documented in the TCP/IP RFC specifications. You must use the type of socket called for in the corresponding RFC. If you are writing an application program to be used with an existing client or server, you must use the same type of socket used by the corresponding program.

A socket is characterized by a local address or name. These addresses or names are fixed structures comprised of an address family and information specific to it. TCP/IP for MVS only supports the DARPA Internet address domain or address family.

Socket names, defined in the socket.h header file, hence, consist of a port number and an internet address.

Note: Some of the port numbers, called well-known port numbers, are reserved for servers that offer well-known services. For further information, refer to TCP/IP RFC 1010.

A Typical TCP Socket Session

The use of TCP sockets is divided into passive (server) and active (client) processes. While some commands are necessary to both types, some are role-specific.

Once a connection is made, it exists until its socket is closed. During the connection, data is either delivered, or an error code is returned by the TCPIP address space.

The general sequence of socket calls needed to be able to send and receive data follows:

1. Create a stream socket **s** with the **socket()** call.
2. Bind the socket **s** to a local address with the **bind()** call.
3. Server: Alert the TCPIP machine of your readiness to communicate with the **listen()** call.
4. Server: Accept a connection on the socket **s** using a temporary socket, say **ns**, with the **accept()** call.

   The socket **ns** is now a dedicated connection between the local and foreign hosts. The socket **s** remains ready to accept other connections.

   Client: Connect the socket **s** to a foreign host with the **connect()** call.

   The socket **s** is now a dedicated connection between the local and foreign hosts. Only one connection per local address is allowed.
5. Server: Read and write data on the socket **ns** until all desired data has been exchanged.

   Close the socket **ns** using the **close()** call.

   Client: Read and write data on the socket **s** until all desired data has been exchanged.
6. Server: Accept another connection from socket **s**, or close the original socket using the **close()** call (ending the TCPIP session).

   Client: Close the socket **s** using the **close()** call (ending the TCPIP session).

Refer to pages E-6 and E-10 in Appendix E, Sample Programs for a sample C Socket communications server and client.

A Typical UDP Socket Session

UDP socket processes are not clearly divided into server and client roles. Instead, the distinction is between connected and unconnected sockets. While an unconnected socket can be used to communicate with any host, a connected socket is a dedicated pathway able to send data to, and receive data from, one host only.

Both connected and unconnected sockets send their data over the network without verification. Once a packet has been placed on the network by the TCPIP address space, its existence, and hence delivery, is not guaranteed.

The general sequence of socket calls needed to be able to send and receive data follows:

1. Create a datagram socket **s** with the **socket()** call.
2. Bind the socket **s** to a local address with the **bind()** call.
3. Unconnected Socket: Send and receive data on socket **s** using the **sendto()** and **recvfrom()** calls until all desired exchanges have taken place.

   Connected Socket: Connect the socket **s** using the **connect()** call.

   Send and receive data on socket **s** using the **recv()** and **send()** calls until all desired exchanges have taken place.
4. Close the socket **s** using the **close()** call (ending the TCPIP session).

Software Requirements

To develop programs with the C Language that interface to TCP and UDP you require the following:

- IBM C for System/370, Compiler Licensed Program (5688-040) and IBM C for System/370, Library Licensed Program (5688-039) for compilation of the programs.
- IBM VS Pascal Compiler & Library (5668-767) **or** IBM VS Pascal Library (5668-717) for execution of the programs.

C Socket Quick Reference

The table below summarizes each socket call supported by the TCP/IP for MVS program, and points to the page where you can find detailed information.

Table 3-1 (Page 1 of 2). C Socket Quick Reference

Socket() Call	Description	Page
accept()	Accepts a connection request from a foreign host.	3-5
bind()	Assigns a local address to the socket.	3-6
close()	Closes the socket associated with the socket descriptor *s*.	3-8
connect()	Requests a connection to a foreign host.	3-8
gethostbyaddr()	Returns information about a host specified by an address.	3-9

Table 3-1 (Page 2 of 2). C Socket Quick Reference		
Socket() Call	**Description**	**Page**
gethostbyname()	Returns information about a host specified by a name.	3-10
gethostname()	Returns the standard name of the current host.	3-11
getsockname()	Obtains local socket name.	3-11
listen()	Indicates that a stream socket is ready for a connection request from a foreign host.	3-12
read()	Reads a set number of bytes into a buffer.	3-12
readv()	Obtains data from a socket and reads this data into specified buffers.	3-12
recv()	Receives messages from a connected socket.	3-13
recvfrom()	Receives messages from a datagram socket, regardless of its connection status.	3-13
select()	Detects whether read is possible on a group of sockets.	3-15
send()	Transmits messages to a connected socket.	3-16
sendto()	Transmits messages to a datagram socket, regardless of its connection status.	3-16
socket()	Requests that a socket be created.	3-17
write()	Writes a set number of bytes from a buffer to a socket.	3-18
writev()	Writes data in the buffers specified by an array of iovec structures.	3-18

The Socket Library

The C socket library uses data structures that are declared in the TCPIP.COMMMAC data set. The C socket routines are in the TCPIP.COMMTXT data set and the headers are in the TCPIP.COMMMAC data set.

Socket programs need the include files listed below.

- types.h
- ctype.h
- tcperrno.h
- in.h
- ioctl.h
- manifest.h
- netdb.h
- socket.h
- uio.h

To access the socket libraries, you must include the statements below at the beginning of each program:

```
#define MVS
#include <manifest.h>
```

A sample linkedit for a file MYFILE in 31-bit mode is:

```
INCLUDE OBJ(MYFILE)
INCLUDE SYSLIB(AMPZMVSB)
MODE AMODE(31),RMODE(ANY)
ENTRY CEESTART
NAME MYFILE(R)
```

A sample linkedit for a file MYFILE in 24-bit mode is:

```
INCLUDE OBJ(MYFILE)
INCLUDE SYSLIB(AMPZRP01)
MODE AMODE(24),RMODE(24)
ENTRY CEESTART
NAME MYFILE(R)
```

A description of each socket library call and its relevant parameters follows.

accept()

The **accept()** call waits for a connection request on the unconnected stream socket with descriptor *s*. If there is a connection request queued, or as soon as one arrives, **accept()** accepts the connection as follows. A new socket, whose descriptor is the return value from **accept()**, is created. This socket is bound to the same local address as the original socket *s* (refer to "bind()" on page 3-6). The TCPIP address space connects this new socket to the foreign host that requested the connection. The original socket remains free to receive requests for more connections, up to the *backlog* parameter specified in the **listen()** call (refer to "listen()" on page 3-12).

If no pending connections are present on the queue, the **accept()** call blocks the caller until a connection is available. The accepted socket is used for subsequent calls to **read()**, **write()** and **select()**.

Use **accept()** when you are writing a server, and you want to wait for and accept connection requests from clients.

```
#include <types.h>
#include <socket.h>
#include <in.h>

int accept(s, name, namelen)
int s;
struct sockaddr_in *name;
int *namelen;
```

Parameter	Description
s	The integer-valued socket descriptor.
name	The internet address of the connecting socket that is filled by **accept()** before it returns. The exact format of *name* is determined by the AF_INET domain in which communication occurs.
namelen	The size of *name* in bytes.

Return Values

The **accept()** call returns a nonnegative integer as the socket descriptor if it successfully created and connected to the socket address where the connection request originated. If the **accept()** call fails, a value of -1 is returned. If **accept()** does not receive any connection requests, it does not return.

bind()

The **bind()** call assigns a unique local address to the socket with descriptor *s*. The address assigned to the socket depends on the values assigned to the fields of the *name* parameter. Each *name* is a combination of a port number and a machine address. Each socket must use a different combination.

Since TCP/IP for MVS supports internet protocols exclusively, the *name* is always a sockaddr_in structure. This structure is defined in in.h, and has the following three fields:

```
short    sin_family;
ushort   sin_port;
struct   in_addr sin_addr;
```

The in_addr structure is also defined in in.h, and has one field:

```
u_long   s_addr;
```

You make assignments to the fields of *name* based on the role played by the particular application. In the case of servers that wait for connections before entering into communication with other hosts, a predetermined port number must be used. Certain "well-known" port numbers are reserved for specific applications; these are listed in RFC 1010. Other port numbers may be assigned as desired, but you must ensure that clients wishing to access the server know with which port to connect (refer to "connect()" on page 3-8).

You can have the TCPIP address space assign a free port to a socket by setting the sin_port field equal to zero (0). This can be done when a client function uses **bind()**

to assign itself a particular machine address, but where the specific port used is unimportant.

Note: The TCPIP address space expects all values assigned to *name* to be in network byte order, rather than host byte order. The C Language macros ntohs(), ntohl(), htonl(), and htons() are used to translate values from network-to-host-short, network-to-host-long, host-to-network-long, and host-to-network-short, respectively. (MVS host order and internet network order are identical. These C Language macros can be used for portability purposes.)

The sin_addr field is used to specify the machine address with which the socket will be associated. If you do not know the address of the machine, or if the machine has more than one address, **gethostbyname()** can be used to obtain a list of the local host's internet addresses. Any one of these addresses can be assigned to the s_addr field of sin_addr. Refer to "gethostname()" on page 3-11 for more information.

The **bind()** call does not have to request a specific address. Certain applications (for example, a gateway server offering a function to all of its networks, clients using **bind()** to request only a specific port, any application on a single address machine) may desire or require that the address be left unspecified. In these and similar cases, the INADDR_ANY C Language macro (defined in the in.h include file) can be used instead of a specific address. This is most important in the case of a server offering a service to multiple networks. By leaving the address unspecified, the server is free to accept all connection requests made for its port, regardless of which of the gateway's addresses was used to deliver the request. If a specific address is used, the server is only able to accept requests made to that address.

The sin_family field is always assigned the AF_INET C Language macro (defined in the in.h include file), specifying that communications take place in the AF_INET domain.

Note: After a successful call to **listen()** or **connect()**, **getsockname()** can be used to determine the actual address and port assigned to the socket. Refer to "getsockname()" on page 3-11 for details.

See "A Sample C Socket Communications Server" on page E-6 and "A Sample C Socket Communications Client" on page E-10 for examples of these assignments.

```
#include <types.h>
#include <socket.h>
#include <in.h>

int bind(s, name, namelen)
int s;
struct sockaddr_in *name;
int namelen;
```

Parameter	Description
s	The integer-valued socket descriptor returned by the associated **socket()** call.

name	Points to a sockaddr_in structure containing the requested address and port number. On return, modified to reflect assigned address and port.
namelen	The size of *name* in bytes.

Return Values

The **bind()** call returns a value of 0 if it is successful. The binding of a stream socket is not complete until a successful call to **listen()** or **connect()** is made. Applications using stream sockets should check the return values of **listen()** and **connect()** before using any function that requires a bound stream socket. (Refer to "getsockname()" on page 3-11.)

close()

The **close()** call shuts down the socket associated with the socket descriptor *s*, and frees resources allocated to the socket. If *s* refers to an open TCP connection, the connection is closed.

```
int close(s)
int s;
```

Parameter	Description
s	The descriptor of the socket to discard.

Return Values

The **close()** call returns a value of 0 if successful. If the **close()** call fails, a value of -1 is returned.

connect()

The **connect()** call is used by the client side of socket-based applications to establish a connection with a server. The address and port of the server are specified in *name* (see "bind()" on page 3-6 for a description of the fields of *name*). The server must have successfully called **bind()** and **listen()** before a connection can be made. Otherwise, the connection request is refused.

The **connect()** call is occasionally used with datagram sockets. It permanently specifies the peer socket to which datagrams are sent. If you permanently connect the two sockets with this call, you can use the **send()** call to transmit data rather than the **sendto()** call.

```
#include <types.h>
#include <socket.h>
#include <in.h>

int connect(s, name, namelen)
int s;
struct sockaddr_in *name;
int namelen;
```

Parameter	Description
s	The integer-valued socket descriptor returned on the creation of the socket by the **socket()** call.
name	The pointer to a socket address structure containing the address of the foreign socket to which a connection will be attempted.
namelen	The size of *name* in bytes.

Return Values

The **connect()** call returns a value of 0 if successful. **connect()** performs two tasks when called for a stream socket: it completes the binding necessary for a stream socket, and it attempts to create a connection with a foreign socket. If either of these steps fail, **connect()** returns -1. If **connect()** is used with a datagram socket, a return value of -1 indicates locally detected errors.

gethostbyaddr()

The **gethostbyaddr()** call returns a pointer to a hostent structure for the host name specified on the call. This information is obtained from a name server.

The hostent structure is defined in the netdb.h header file, and contains the following elements:

Element	Description
h_name	Official name of the host.
h_aliases	An array, terminated with a NULL pointer, of alternative names for the host.
h_addrtype	The type of address being returned. The call always sets this value to AF_INET.
h_length	The length of the address in bytes.
h_addr_list	An array, terminated with a NULL pointer, of pointers to the network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list, provided for backward compatibility.

```
#include <netdb.h>

struct hostent *gethostbyaddr(addr, addrlen, domain)
char *name;
int addrlen, domain;
```

Parameter	Description
addr	The pointer to a structure containing the address of the socket. (An unsigned long for AF_INET.)
addrlen	The size of *addr* in bytes.
domain	The address domain supported (AF_INET).

Return Values

The return value points to static data that is overwritten by subsequent calls. The **gethostbyaddr()** call returns a pointer to a hostent structure on success.

A NULL pointer is returned if an error occurs or if the host name is unknown.

gethostbyname()

The **gethostbyname()** call returns a pointer to a hostent structure for the host name specified on the call. This information is obtained from a name server.

The hostent structure is defined in the netdb.h header file, and contains the following elements:

Element	Description
h_name	Official name of the host.
h_aliases	An array, terminated with a NULL pointer, of alternative names for the host.
h_addrtype	The type of address being returned. The call always sets this value to AF_INET.
h_length	The length of the address in bytes.
h_addr_list	An array, terminated with a NULL pointer, of pointers to the network addresses for the host. Host addresses are returned in network byte order.
h_addr	The first address in h_addr_list, provided for backward compatibility.

```
#include <netdb.h>

struct hostent *gethostbyname(name)
char *name;
```

Parameter	Description
name	The pointer to a socket address structure containing the address of the socket.

Return Values

The return value points to static data that is overwritten by subsequent calls. The **gethostbyname()** call returns a pointer to a hostent structure on success.

A NULL pointer is returned if an error occurs or if the host name is unknown.

gethostname()

The **gethostname()** call returns the standard host name of the current host. The host name is obtained from the TCPIP.TCPIP.DATA data set.

```
int gethostname(name, namelen)
char *name;
int namelen;
```

Parameter	Description
name	The pointer to the name of the current host.
namelen	The size of *name* in bytes.

Return Values

Upon successful completion of the **gethostname()** call, a value of 0 is returned. If the **gethostname()** call fails, a value of -1 is returned.

getsockname()

The **getsockname()** call stores the current name for the socket specified by the *s* parameter into the structure pointed to by the *name* parameter.

```
#include <in.h>

int getsockname(s, name, namelen)
int s;
struct sockaddr_in *name;
int *namelen;
```

Parameter	Description
s	The integer-valued socket descriptor.
name	The pointer to a socket address structure containing the address of the socket.
namelen	The size of *name* in bytes.

Return Values

Upon successful completion of the **getsockname()** call, a value of 0 is returned. If the **getsockname()** call fails, a value of -1 is returned. Stream sockets are not assigned a name until after a successful call to either **listen()** or **connect()** (see "bind()" on page 3-6 for details).

listen()

The **listen()** call creates a queue of length *backlog* for connection requests to the stream socket with descriptor *s*. **listen()** indicates a willingness to accept client connections. If it is not called, no connections are accepted.

```
int listen (s, backlog)
int s, backlog;
```

Parameter	Description
s	The integer-valued socket descriptor.
backlog	Defines the maximum length for the queue of pending connections. The only *backlog* value supported in this implementation is 1.

Return Values

The **listen()** call returns a value of 0 if it is successful. **listen()** performs two tasks: it completes the binding necessary for a steam socket, and it creates a connection-request queue of length *backlog*. If either of these tasks fail, **listen()** returns a -1.

read() and readv()

The **read()** call reads the number of bytes set by the *nbyte* parameter from the open stream socket denoted by the socket descriptor *s*, and places those bytes into the buffer pointed to by the *buf* parameter.

```
int read(s, buf, nbyte)
int s;
char *buf;
unsigned int nbyte;
```

Parameter	Description
s	The integer-valued socket descriptor.
buf	The pointer to the buffer storing the incoming data.
nbyte	The set number of bytes read into the buffer.

The **readv()** call applies to stream sockets and performs the same function as **read()**. It reads data into the buffers specified by the array of iovec structures pointed to by the *iov* parameter.

The iovec structure is defined in the uio.h header file, and contains the following elements:

Element	Description
`caddr_t`	iov_base
`int`	iov_len

Each iovec entry specifies the base address and length of an area in memory where data should be placed. The **readv()** call completely fills out an area before moving to the next.

```
#include <types.h>
#include <uio.h>

int readv(s, iov, iovcnt)
int s;
struct iovec *iov;
int iovcnt;
```

Parameter	Description
s	A socket descriptor as defined previously.
iov	The array of iovec structures pointed to.
iovcnt	The number of iovec structures pointed to by the *iov* parameter.

Return Values

Upon successful completion, the **read()** and **readv()** calls return the number of bytes actually read and placed in the buffer; this number can be less than the value of the *nbyte* parameter. A value of 0 is returned when the connection is closed. If **read()** or **readv()** fails, a value of -1 is returned.

recv() and recvfrom()

The **recv()** and **recvfrom()** calls receive data on a socket with descriptor *s* and store it in a buffer. The **recv()** call applies to connected sockets only; the **recvfrom()** call applies to any datagram socket, whether connected or not.

These calls return the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If no

datagram packets are available at the socket with descriptor *s*, the receive calls wait for a message to arrive, and block the caller.

```
#include <types.h>
#include <socket.h>

int recv(s, buf, nbyte, flags)
int s;
char *buf;
int nbyte, flags;
```

Parameter	Description
s	The integer-valued socket descriptor.
buf	The pointer to the buffer that receives the data.
nbyte	The length of the buffer pointed to by the *buf* parameter.
flags	Provided only for compatibility with other socket implementations. Should be zero.

Other parameters apply to the **recvfrom()** call.

```
#include <types.h>
#include <socket.h>
#include <in.h>

int recvfrom(s, buf, len,
flags, name, namelen)
int s;
char *buf;
int len, flags;
struct sockaddr_in *name;
int *namelen;
```

Parameter	Description
s	The integer-valued socket descriptor.
buf	The pointer to the buffer that receives the data.
len	The length of the buffer pointed to by the *buf* parameter.
flags	Provided only for compatibility with other socket implementations. Should be zero.
name	A pointer to a socket address structure from which data is being received. If *name* is a nonzero value, the source address will be returned.
namelen	The size of *name* in bytes.

Return Values

Upon successful completion, the length of the message/datagram in bytes is returned. If the **recv()** or **recvfrom()** calls fail, a value of -1 is returned. Return values of -1 indicate some locally-detected errors.

select()

The **select()** call monitors for activity on a set of different sockets to see if any of them are ready for reading (receiving), or until a timeout period expires. TCP/IP for MVS does not support selecting for writing or exceptional conditions.

The address of the bitmask is passed in *readfds*. The socket descriptors from 0 through *num_fds*-1 are examined. On return, **select()** replaces the mask of the socket descriptors examined with the mask of the sockets that are ready for reading. The total number of ready socket descriptors is returned in *nfound*.

The set of descriptors available for reading are stored as bit fields in unsigned long integers.

```
#include <types.h>

nfound=select(num_fds, readfds, 0, 0, timeout);
int num_fds, nfound;
unsigned long *readfds;
struct timeval *timeout;
```

Parameter	Description
num_fds	Number of file descriptors represented in the mask (≤32).
readfds	Points to the bit mask representing the sockets to be checked. Each bit in the read mask corresponds to a socket descriptor. To set the bit for an arbitrary socket descriptor, **s**: `readfds |= (1 << s);`

timeout	The parameter *timeout* is a pointer to a timeval structure containing the timeout value. The timeval structure, included by types.h, has two fields:
	`long tv_sec;`
	`long tv_usec;`
	Only the first field is relevant in this implementation (the tv_usec field can be set, but is not used). If the **select()** function should wait indefinitely for data to arrive on any of the *num_fds* sockets, timeval should be a pointer to NULL. Otherwise, tv_sec should be set to the desired timeout value, in seconds. If **select()** should poll the sockets and return immediately, set tv_sec to 0.

Note: The zeroes (third and fourth parameters) are required.

send() and sendto()

The **send()** and **sendto()** calls send packets on the socket with descriptor *s*. The **send()** call applies to all connected sockets, and the **sendto()** call applies to any datagram socket, whether connected or not.

When used with stream sockets, **send()** blocks until the packet is delivered, or until an error condition is encountered. No indication of failure to deliver is implied in the return value of either call when used with datagram sockets.

```
#include <types.h>
#include <socket.h>

int send(s, buf, nbyte, flags)
int s;
char *buf;
int nbyte, flags;
```

Parameter	Description
s	The integer-valued socket descriptor.
buf	The pointer to the buffer containing the message to transmit.
nbyte	The length of the message pointed to by the *buf* parameter.
flags	Provided only for compatibility with other socket implementations. Should be zero.

Other parameters pertain to the **sendto()** call as follows. Give the address of the target socket by *name*, with *namelen* specifying its size. Specify the length of the message with *len*. If the message is too long to pass through the underlying protocol, an error is returned.

```
#include <types.h>
#include <socket.h>
#include <in.h>

int sendto(s, msg, len,
flags, name, namelen)
int s;
char *msg;
int len, flags;
struct sockaddr_in *name;
int namelen;
```

Parameter	Description
s	The integer-valued socket descriptor.
msg	The pointer to the buffer containing the message to transmit.
len	The length of the message in the buffer pointed to by the *msg* parameter.

flags	Provided only for compatibility with other socket implementations. Should be zero.
name	A pointer to a socket address structure to which data is being sent.
namelen	The size of *name* in bytes.

Return Values

Upon successful completion, the number of characters sent is returned. If the **send()** or **sendto()** call fails, a value of -1 is returned. Return values of -1 indicate some locally-detected errors.

socket()

The **socket()** call creates an end point for communication and returns a positive integer value socket descriptor *s*. The parameters specify the type of socket protocol supported for the socket, and address domain.

TCP/IP for MVS program supports the following socket definitions.

- The types of socket supported are stream sockets and datagram sockets. These are specified as SOCK_STREAM and SOCK_DGRAM, respectively. Refer to "Programming With Sockets" on page 3-1 for further information about these socket types.
- The *protocol* parameter specifies a particular protocol to be used with the socket. When you specify *protocol* as 0, the **socket()** call defaults to the correct protocol for the type of returned socket requested. The protocol supported is TCP for stream sockets and UDP for datagram sockets.
- The address domain supported is the DARPA Internet address, specified as AF_INET.

```
#include <types.h>
#include <socket.h>

int socket (af, type, protocol)
int af, type, protocol;
```

Parameter	Description
af	The address domain supported (AF_INET).
type	The type of socket created, either SOCK_STREAM or SOCK_DGRAM.
protocol	The protocol used with the socket as an integer. Possible values are 0, IPPROTO_UDP or IPPROTO_TCP.

Return Values

If successful, **socket()** returns a valid socket descriptor. If an error has occurred, **socket()** returns a value of -1.

write() and writev()

The **write()** call writes the number of bytes set by the *nbyte* parameter to the open stream socket denoted by the socket descriptor *s* from a buffer specified by the *buf* parameter.

The *s* parameter is a socket descriptor obtained from a **socket()** or **accept()** call.

```
int write(s, buf, nbyte)
int s;
char *buf;
unsigned int nbyte;
```

Parameter	Description
s	The integer-valued socket descriptor.
buf	The pointer to the buffer storing the outgoing data.
nbyte	The set number of bytes written into the buffer.

The **writev()** call applies to stream sockets and functions like **write()**. It writes data to the socket from the buffers specified by the array of iovec structures pointed to by the *iov* parameter.

The iovec structure is defined in the uio.h header file, and contains the following elements:

Element	Description
`caddr_t`	iov_base
`int`	iov_len

Each iovec entry specifies the base address and length of an area in memory from which data is transferred. The **writev()** call completely transfers an area before moving to the next.

```
#include <types.h>
#include <uio.h>

int writev(s, iov, iovcnt)
int s;
struct iovec *iov;
int iovcnt;
```

Parameter	Description
s	The integer-valued socket descriptor.
iov	The array of iovec structures pointed to.
iovcnt	The number of iovec structures pointed to by the *iov* parameter.

Return Values

Upon successful completion, the **write()** and **writev()** calls return the number of bytes actually written. If **write()** or **writev()** fails, a value of -1 is returned.

Chapter 4. X-Windows Interface

This chapter contains information of benefit to someone writing application programs using the X Window System[1] protocol developed by the Massachusetts Institute of Technology. The X-Windows API allows users to interface MVS to any X-Windows server on a TCP/IP-based network to provide the MVS client with real-time interactive 2-dimensional bitmap and vector graphics. For more detailed information on the X-Windows API, see the *IBM AIX X-Windows Programmer's Reference*, SC23-2118 or the X-Windows programmer's book you are using.

Software Requirements

Application programs using the X-Windows API are written in C and require the following:

- IBM C for System/370, Compiler Licensed Program (5688-040)
- IBM C for System/370, Library Licensed Program (5688-039)
- IBM VS Pascal Compiler & Library (5668-767) **or** IBM VS Pascal Library (5668-717).

How the X-Windows Interface Works

The TCP/IP for MVS X-Windows API provides programmers with a set of X-Windows calls. X-Windows is a network-transparent windowing protocol that operates under the AIX™ Operating System, or on any server with bitmapped display terminals running the X11 standard. This interface assumes that a reliable bidirectional byte-stream is available for communication between the application and the server in the TCP/IP for MVS programming environment.

In an X-Windows environment, the X Server distributes user input to and accepts output requests from various client programs located either on the same system or elsewhere in a network. The client code uses sockets to communicate with the server.

Figure 4-1 on page 4-2 is a high-level abstraction of the different parts of the system. The application programmer need only be concerned with the client API to write his or her code.

---

[1] Trademark of Massachusetts Institute of Technology.

AIX is a trademark of the International Business Machines Corporation.

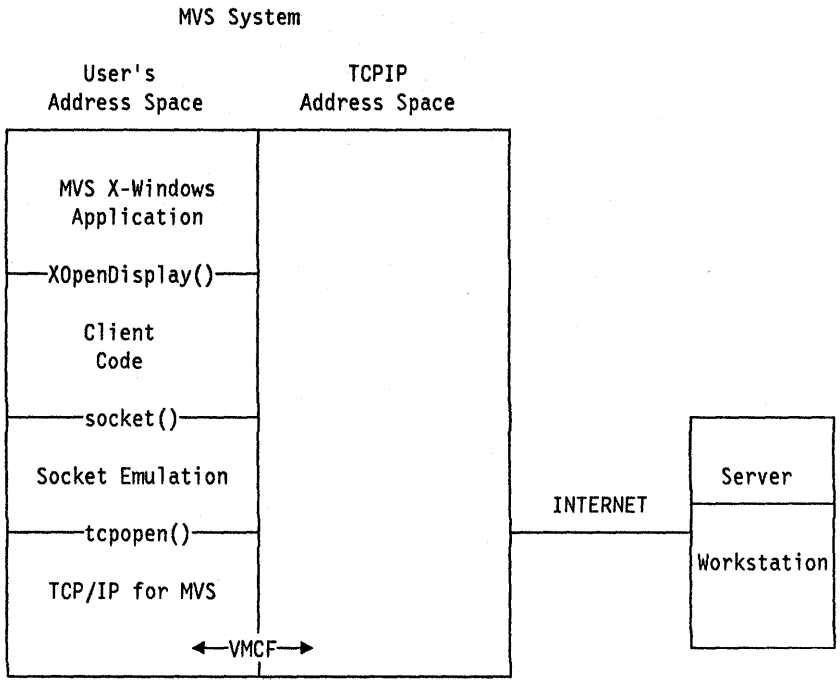


Figure 4-1. MVS X-Windows Application to Server. The communication path from the MVS X-Windows Application to the server involves the client code and TCP/IP for MVS.

The application program you create is the client part of a client-server relationship; you write the program, and the X Server provides it independence from the hardware. There is one X Server for each virtual terminal that runs X-Windows. In this chapter, the term display refers to a logical virtual terminal with its associated keyboard, locator, and server unless it is explicitly stated otherwise.

Each client can interact with many X Servers, and each X Server can interact with many clients.

Your application can call the **XOpenDisplay**() routine to start communications with your workstation. The client code creates the appropriate X Protocol and opens a socket to the server running on your workstation. This socket is emulated, using the appropriate TCP/IP for MVS program calls, to establish a connection to your server over which a two-way communication can take place.

The X Protocol generated by the client code uses an ASCII representation for character strings. A transformation from EBCDIC to ASCII (or from ASCII to EBCDIC) is performed automatically for you when you use the client API, because MVS uses the EBCDIC representation for character strings.

X Defaults

X-Windows allows you to create a data set containing defaults that an application can consult to alter its functionality. Typically, this data set contains hints about the window size, placement, coloring, font usage, and other functional details of the application.

On an AIX system, these hints are usually found in the user's home directory in a file called .Xdefaults. Under the X-Windows interface used with the TCP/IP for MVS program, this information is found in the user's data set called *userid*.X.DEFAULTS. Each line of this data set represents information for an application. A typical data set might look like this:

```
mvsload*Geometry: =400x100+0-0
mvsload*Background: blue
mvsload*Foreground: white
mvsload*BorderWidth: 3
mvsload*Font: Rom10.500
```

An application can use this data set to automatically modify the characteristics of the windows displayed by a particular application. For example, the `mvsload` program above might look in this data set for automatic window sizing and placement, rather than prompting you for the information. It allows the application user to automatically tailor some of the characteristics of the application at the time the application is run, rather than having to pass parameters on the command line. For more information about the format of this data set, consult the *IBM AIX X-Windows User's Guide* (SC23-2017), *IBM AIX X-Windows Programmer's Reference* (SC23-2118), or the X-Windows documentation for your workstation.

EBCDIC-ASCII Translation

Because MVS uses the EBCDIC representation for character strings, and the Socket API protocol uses the ASCII representation, a translation is needed. The translation table used by the X-Windows client code can be changed by the user by creating a *userid*.STANDARD.TCPXLBIN data set. The data set called TCPIP.STANDARD.TCPXLBIN is used if you do not have your own translation table. If this data set is not available to the client, a built-in table is used instead and a warning message is issued by the client.

Refer to *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* for information on creating your own translation table.

Creating an Application

To create an application that uses the X-Windows interface, study the application program interface for the X-Windows protocol running on your workstation. The sample programs provided illustrate the simple creation of windows and some of the graphics capabilities available to you. You must write your application in the C language, and you should be familiar with creating, compiling, and generating applications with this language in the MVS environment.

To generate your application, you must include the libraries TCPIP.X11LIB, TCPIP.OLDXLIB and TCPIP.COMMTXT in your SYSLIB declaration so the linkage editor can search these libraries for the client interface code.

Due to the characteristics of file naming conventions in MVS, some header files have different names than they do on the original MIT distribution of the X Window System. In porting applications to MVS, includes of these header files must be changed for the C compiler to work. Table 4-1 on page 4-4 describes the name changes.

Table 4-1. Differences in Header File Names

MIT Distribution Name	X-Windows API Name
errno.h	Xerrno.h
copyright.h	cpyright.h
cursorfont.h	cursfont.h
Xatomtype.h	Xatomtyp.h
Xprotostr.h	Xprotost.h
Xresource.h	Xresourc.h

Running an Application

Many applications are not coded with the specific name of a server with which they are to communicate. This information is provided to the application in the *userid*.XWINDOWS.DISPLAY data set. A line of this data set may look like this:

```
ROYAL.CSC.IBM.COM:0.0
```

The application can then communicate with the server associated with display 0 on the machine "royal.csc.ibm.com." You can dynamically change the server you want to communicate with. You could also pass this information to the application in a number of other ways. However, if the application uses the **XOpenDisplay**() call and passes a NULL pointer, the value entered in *userid*.XWINDOWS.DISPLAY is used as the address of the server. Consult your local operations group about the names used for hosts on your Local Area Network if this is not clear.

Alternatively, you can specify an internet address of your server in *userid*.XWINDOWS.DISPLAY. like this:

```
129.42.3.100:0.0
```

This permits you to use an application while you are waiting for your system administration group to update the TCP/IP name tables to include your workstation.

Refer to page E-13 in Appendix E, Sample Programs for a sample X-Windows application.

X-Windows Quick Reference

The following is a Quick Reference list of the subroutines supported by the TCP/IP for MVS product. These subroutines are tabled alphabetically and grouped according to the type of function provided.

The AIX extensions described in the *IBM AIX X-Windows Programmer's Reference* (SC23-2118) are not supported by the X-Windows API provided by the TCP/IP for MVS library routines.

For a comprehensive description of X-Windows subroutines and the X-Windows Toolkit, including the configuration and operation of the X-Windows programming interface, consult the *IBM AIX X-Windows Programmer's Reference* (SC23-2118) and "X-Windows Toolkit" on page 4-25, respectively.

Opening and Closing Display

Table 4-2. Opening and Closing Display

Subroutine	Description
XCloseDisplay()	Closes a display.
XFree()	Frees in-memory data created by Xlib function.
XNoOp()	Executes a NoOperation protocol request.
XOpenDisplay()	Opens a display.

Creating and Destroying Windows

Table 4-3. Creating and Destroying Windows

Subroutine	Description
XConfigureWindow()	Configures the specified window.
XCreateSimpleWindow()	Creates unmapped InputOutput subwindow.
XCreateWindow()	Creates unmapped subwindow.
XDestroySubwindows()	Destroys all subwindows of specified window.
XDestroyWindow()	Unmaps and destroys window and all subwindows.

Manipulating Windows

Table 4-4 (Page 1 of 2). Manipulating Windows

Subroutine	Description
XCirculateSubwindows()	Circulates a subwindow up or down.
XCirculateSubwindowsUp()	Raises the lowest mapped child of window.
XCirculateSubwindowsDown()	Lowers the highest mapped child of window.
XLowerWindow()	Lowers the specified window.
XMapRaised()	Maps and raises the specified window.
XMapSubwindows()	Maps all subwindows of the specified window.

Table 4-4 (Page 2 of 2). Manipulating Windows	
Subroutine	**Description**
XMapWindow()	Maps the specified window.
XMoveResizeWindow()	Changes the specified window's size and location.
XMoveWindow()	Moves the specified window.
XRaiseWindow()	Raises the specified window.
XResizeWindow()	Changes the specified window's size.
XRestackWindows()	Restacks a set of windows from top to bottom.
XSetWindowBorderWidth()	Changes the border width of the window.
XUnmapSubwindows()	Unmaps all subwindows of the specified window.
XUnmapWindow()	Unmaps the specified window.

Changing Window Attributes

Table 4-5. Changing Window Attributes	
Subroutine	**Description**
XChangeWindowAttributes()	Changes one or more window attributes.
XSetWindowBackground()	Sets the window's background to specified pixel.
XSetWindowBackgroundPixmap()	Sets the window's background to specified pixmap.
XSetWindowBorder()	Changes the window's border to specified pixel.
XSetWindowBorderPixmap()	Changes window's border tile.
XTranslateCoordinates()	Transforms coordinates between windows.

Obtaining Window Information

Table 4-6. Obtaining Window Information	
Subroutine	**Description**
XGetGeometry()	Gets current geometry of specified drawable.
XGetWindowAttributes()	Gets current attributes for specified window.
XQueryPointer()	Gets pointer coordinates and root window.
XQueryTree()	Obtains the IDs of the children and parent windows.

Properties and Atoms

Table 4-7. Properties and Atoms	
Subroutine	**Description**
XGetAtomName()	Gets a name for the specified atom ID.
XInternAtom()	Gets an atom for the specified name.

Manipulating Window Properties

Table 4-8. Manipulating Window Properties	
Subroutine	**Description**
XChangeProperty()	Changes the property for specified window.
XDeleteProperty()	Deletes a property for the specified window.
XGetWindowProperty()	Gets atom type and property format for window.
XListProperties()	Gets the specified window's property list.
XRotateWindowProperties()	Rotates properties in property array.

Setting Window Selections

Table 4-9. Setting Window Selections	
Subroutine	**Description**
XConvertSelection()	Converts a selection.
XGetSelectionOwner()	Gets the selection owner.
XSetSelectionOwner()	Sets the selection owner.

Manipulating Colormaps

Table 4-10. Manipulating Colormaps	
Subroutine	**Description**
XCopyColormapAndFree()	Creates a new colormap from specified colormap.
XCreateColormap()	Creates a colormap.
XFreeColormap()	Frees the specified colormap.
XQueryColor()	Queries the RGB value for a specified pixel.
XQueryColors()	Queries the RGB values for array of pixels.
XSetWindowColormap()	Sets the colormap of the specified window.

Manipulating Color Cells

Table 4-11. Manipulating Color Cells	
Subroutine	**Description**
XAllocColor()	Allocates a read-only color cell.
XAllocColorCells()	Allocates read/write color cells.
XAllocColorPlanes()	Allocates read/write color resources.
XAllocNamedColor()	Allocates a read-only color cell by name.
XFreeColors()	Frees colormap cells.
XLookupColor()	Looks up colorname.
XStoreColor()	Stores an RGB value into a single colormap cell.
XStoreColors()	Stores RGB values into colormap cells.
XStoreNamedColor()	Sets a pixel color to the named color.

Creating and Freeing Pixmaps

Table 4-12. Creating and Freeing Pixmaps	
Subroutine	**Description**
XCreatePixmap()	Creates a pixmap of a specified size.
XFreePixmap()	Frees all storage associated with specified pixmap.

Manipulating Graphics Contexts

Table 4-13 (Page 1 of 2). Manipulating Graphics Contexts	
Subroutine	**Description**
XChangeGC()	Changes the components in the specified Graphics Context (GC).
XCopyGC()	Copies components from a source GC to a destination GC.
XCreateGC()	Creates a new GC.
XFreeGC()	Frees the specified GC.
XGContextFromGC()	Obtains the GContext resource ID for GC.
XQueryBestTile()	Gets best fill tile shape.
XQueryBestSize()	Gets best size of tile, stipple, or cursor.
XQueryBestStipple()	Gets best stipple shape.
XSetArcMode()	Sets the arc mode of the specified GC.

Table 4-13 (Page 2 of 2). Manipulating Graphics Contexts	
Subroutine	**Description**
XSetBackground()	Sets the background of the specified GC.
XSetClipmask()	Sets the clip_mask of specified GC to specified pixmap.
XSetClipOrigin()	Sets the clip origin of the specified GC.
XSetClipRectangles()	Sets clip_mask of GC to list of rectangles.
XSetDashes()	Sets the dashed line style components of specified GC.
XSetFillRule()	Sets the fill rule of the specified GC.
XSetFillStyle()	Sets the fill style of the specified GC.
XSetFont()	Sets the current font of the specified GC.
XSetForeground()	Sets the foreground of the specified GC.
XSetFunction()	Sets display function in specified GC.
XSetGraphicsExposures()	Sets graphics-exposure flag of specified GC.
XSetLineAttributes()	Sets the line-drawing components of the GC.
XSetPlaneMask()	Sets the plane mask of the specified GC.
XSetState()	Sets foreground, background, plane mask, and function in GC.
XSetStipple()	Sets the stipple of the specified GC.
XSetSubwindowMode()	Sets subwindow mode of the specified GC.
XSetTile()	Sets the fill tile of the specified GC.
XSetTSOrigin()	Sets the tile or stipple origin of specified GC.

Clearing and Copying Areas

Table 4-14. Clearing and Copying Areas	
Subroutine	**Description**
XClearArea()	Clears a rectangular area of window.
XClearWindow()	Clears the entire window.
XCopyArea()	Copies drawable area between drawables of the same root and the same depth.
XCopyPlane()	Copies single bit-plane of drawable.

Drawing Lines

Table 4-15. Drawing Lines	
Subroutine	**Description**
XDraw()	Draws an arbitrary polygon or curve that is defined by the specified list of Vertexes as specified in *vlist*.
XDrawArc()	Draws single arc in drawable.
XDrawArcs()	Draws multiple arcs in specified drawable.
XDrawFilled()	Draws arbitrary polygons or curves and then fills them in.
XDrawLine()	Draws a single line between two points in drawable.
XDrawLines()	Draws multiple lines in the specified drawable.
XDrawPoint()	Draws a single point in specified drawable.
XDrawPoints()	Draws multiple points in specified drawable.
XDrawRectangle()	Draws outline of single rectangle in drawable.
XDrawRectangles()	Draws outline of multiple rectangles in drawable.
XDrawSegments()	Draws multiple line segments in specified drawable.

Filling Areas

Table 4-16. Filling Areas	
Subroutine	**Description**
XFillArc()	Fills single arc in drawable.
XFillArcs()	Fills multiple arcs in drawable.
XFillPolygon()	Fills a polygon area in drawable.
XFillRectangle()	Fills single rectangular area in drawable.
XFillRectangles()	Fills multiple rectangular areas in drawable.

Loading and Freeing Fonts

Table 4-17 (Page 1 of 2). Loading and Freeing Fonts	
Subroutine	**Description**
XFreeFont()	Unloads font and free storage used by font.
XFreeFontInfo()	Frees the font information array.
XFreeFontNames()	Frees a font name array.
XFreeFontPath()	Frees data returned by XGetFontPath.

Table 4-17 (Page 2 of 2). Loading and Freeing Fonts	
Subroutine	**Description**
XGetFontPath()	Gets the current font search path.
XGetFontProperty()	Gets the specified font property.
XListFontsWithInfo()	Gets names and information about loaded fonts.
XLoadFont()	Loads a font.
XLoadQueryFont()	Loads and queries font in one operation.
XListFonts()	Gets a list of available font names.
XQueryFont()	Gets information about a loaded font.
XSetFontPath()	Sets the font search path.
XUnloadFont()	Unloads the specified font.

Querying Character String Sizes

Table 4-18. Querying Character String Sizes	
Subroutine	**Description**
XQueryTextExtents()	Gets a 1-byte character string bounding box from server.
XQueryTextExtents16()	Gets a 2-byte character string bounding box from server.
XTextExtents()	Gets a bounding box of a 1-byte character string.
XTextExtents16()	Gets a bounding box of a 2-byte character string.
XTextWidth()	Gets the width of an 8-bit character string.
XTextWidth16()	Gets the width of a 2-byte character string.

Drawing Text

Table 4-19. Drawing Text	
Subroutine	**Description**
XDrawImageString()	Draws 8-bit image text in specified drawable.
XDrawImageString16()	Draws 2-byte image text in specified drawable.
XDrawString()	Draws 8-bit text in specified drawable.
XDrawString16()	Draws 2-byte text in specified drawable.
XDrawText()	Draws 8-bit complex text in specified drawable.
XDrawText16()	Draws 2-byte complex text in specified drawable.

Transferring Images

Table 4-20. Transferring Images	
Subroutine	**Description**
XGetImage()	Gets image from rectangle in drawable.
XGetSubImage()	Copies rectangle on display to image.
XPutImage()	Puts image from memory into rectangle in drawable.

Manipulating Cursors

Table 4-21. Manipulating Cursors	
Subroutine	**Description**
XCreateFontCursor()	Creates a cursor from a standard font.
XCreateGlyphCursor()	Creates a cursor from font glyphs.
XDefineCursor()	Defines a cursor for a window.
XFreeCursor()	Frees a cursor.
XQueryBestCursor()	Gets useful cursor sizes.
XRecolorCursor()	Changes the color of a cursor.
XUndefineCursor()	Undefines a cursor for a window.

Handling Window Manager Functions

Table 4-22 (Page 1 of 2). Handling Window Manager Functions	
Subroutine	**Description**
XAddToSaveSet()	Adds a window to the client's save-set.
XAllowEvents()	Allows events to be processed after a device is frozen.
XChangeActivePointerGrab()	Changes the active pointer grab.
XChangePointerControl()	Changes the interactive feel of pointer device.
XChangeSaveSet()	Adds or removes a window from the client's save-set.
XGetInputFocus()	Gets the current input focus.
XGetPointerControl()	Gets the current pointer parameters.
XGrabButton()	Grabs a mouse button.
XGrabKey()	Grabs a single key of the keyboard.
XGrabKeyboard()	Grabs the keyboard.
XGrabPointer()	Grabs the pointer.

Table 4-22 (Page 2 of 2). Handling Window Manager Functions	
Subroutine	**Description**
XGrabServer()	Grabs the server.
XInstallColormap()	Installs a colormap.
XKillClient()	Removes a client.
XListInstalledColormaps()	Gets a list of currently installed colormaps.
XRemoveFromSaveSet()	Removes a window from the client's save-set.
XReparentWindow()	Changes the parent of a window.
XSetCloseDownMode()	Changes the close down mode.
XSetInputFocus()	Sets the input focus.
XUngrabButton()	Ungrabs a mouse button.
XUngrabKey()	Ungrabs a key.
XUngrabKeyboard()	Ungrabs the keyboard.
XUngrabPointer()	Ungrabs the pointer.
XUngrabServer()	Ungrabs the server.
XUninstallColormap()	Uninstalls a colormap.
XWarpPointer()	Moves the pointer to arbitrary point on the screen.

Manipulating Keyboard Settings

Table 4-23 (Page 1 of 2). Manipulating Keyboard Settings	
Subroutine	**Description**
XAutoRepeatOff()	Turns off keyboard auto-repeat.
XAutoRepeatOn()	Turns on keyboard auto-repeat.
XBell()	Sets the volume of the bell.
XChangeKeyboardControl()	Changes keyboard settings.
XChangeKeyboardMapping()	Changes the mapping of symbols to keycodes.
XDeleteModifiermapEntry()	Deletes an entry from XModifierKeymap structure.
XFreeModifiermap()	Frees XModifierKeymap structure.
XGetKeyboardControl()	Gets the current keyboard settings.
XGetKeyboardMapping()	Gets the mapping of symbols to keycodes.
XGetModiferMapping()	Gets keycodes to be modifiers.
XGetPointerMapping()	Gets the mapping of buttons on the pointer.
XInsertModifiermapEntry()	Adds an entry to XModifierKeymap structure.
XNewModifiermap()	Creates the XModifierKeymap structure.

Table 4-23 (Page 2 of 2). Manipulating Keyboard Settings	
Subroutine	**Description**
XQueryKeymap()	Gets the state of the keyboard keys.
XSetPointerMapping()	Sets the mapping of buttons on the pointer.
XSetModifierMapping()	Sets keycodes to be modifiers.

Controlling the Screen Saver

Table 4-24. Controlling the Screen Saver	
Subroutine	**Description**
XActivateScreenSaver()	Activates the screen saver.
XForceScreenSaver()	Turns the screen saver on or off.
XGetScreenSaver()	Gets the current screen saver settings.
XResetScreenSaver()	Resets the screen saver.
XSetScreenSaver()	Sets the screen saver.

Manipulating Hosts and Access Control

Table 4-25. Manipulating Hosts and Access Control	
Subroutine	**Description**
XDisableAccessControl()	Disables access control.
XEnableAccessControl()	Enables access control.
XListHosts()	Gets the list of hosts.
XSetAccessControl()	Changes access control.

Handling Events

Table 4-26 (Page 1 of 2). Handling Events	
Subroutine	**Description**
XCheckIfEvent()	Checks event queue for specified event without blocking.
XCheckMaskEvent()	Removes the next event that matches a specified mask without blocking.
XCheckTypedEvent()	Gets the next event that matches event type.
XCheckTypedWindowEvent()	Gets the next event for specified window.

Table 4-26 (Page 2 of 2). Handling Events	
Subroutine	**Description**
XCheckWindowEvent()	Removes next event that matches the specified window and mask without blocking.
XEventsQueued()	Checks the number of events in the event queue.
XFlush()	Flushes the output buffer.
XGetMotionEvents()	Gets the motion history for specified window.
XIfEvent()	Checks event queue for specified event and removes it.
XMaskEvent()	Removes the next event that matches a specified mask.
XNextEvent()	Gets the next event and removes it from the queue.
XPeekEvent()	Peeks at the event queue.
XPeekIfEvent()	Checks event queue for specified event.
XPending()	Returns the number of events that are pending.
XPutBackEvent()	Pushes event back to top of event queue.
XSelectInput()	Selects events to be reported to the client.
XSendEvent()	Sends an event to a specified window.
XSync()	Flushes the output buffer and waits until all requests are completed.
XWindowEvent()	Removes next event that matches the specified window and mask.

Enabling and Disabling Synchronization

Table 4-27. Enabling and Disabling Synchronization	
Subroutine	**Description**
XSetAfterFunction()	Sets the previous after function.
XSynchronize()	Enables or disables synchronization.

Using Default Error Handling

Table 4-28 (Page 1 of 2). Using Default Error Handling	
Subroutine	**Description**
XDisplayName()	Gets name of display currently being used.
XGetErrorText()	Gets error text for specified error code.
XGetErrorDatabaseText()	Gets error text from the error database.

Table 4-28 (Page 2 of 2). Using Default Error Handling	
Subroutine	**Description**
XSetErrorHandler()	Sets error handler.
XSetIOErrorHandler()	Sets error handler for unrecoverable I/O errors.

Communicating with Window Managers

Table 4-29. Communicating with Window Managers	
Subroutine	**Description**
XGetClassHint()	Gets the class of a window.
XFetchName()	Gets the name of a window.
XGetIconName()	Gets the name of an icon window.
XGetIconSizes()	Gets the values of icon size atom.
XGetNormalHints()	Gets size hints for window in normal state.
XGetSizeHints()	Gets the values of type WM_SIZE_HINTS properties.
XGetStandardColormap()	Gets colormap associated with specified atom.
XGetTransientForHint()	Gets WM_TRANSIENT_FOR property for window.
XGetWMHints()	Gets the value of the window manager's hints atom.
XGetZoomHints()	Gets values of the zoom hints atom.
XSetCommand()	Sets the value of the command atom.
XSetClassHint()	Sets the class of a window.
XSetIconName()	Assigns a name to an icon window.
XSetIconSizes()	Sets the values of icon size atom.
XSetNormalHints()	Sets size hints for window in normal state.
XSetSizeHints()	Sets the values of type WM_SIZE_HINTS properties.
XSetStandardColormap()	Sets colormap associated with specified atom.
XSetStandardProperties()	Specifies a minimum set of properties.
XSetTransientForHint()	Sets WM_TRANSIENT_FOR property for window.
XSetWMHints()	Sets the value of the window manager's hints atom.
XSetZoomHints()	Sets values of the zoom hints atom.
XStoreName()	Assigns a name to a window.

Keyboard Event Functions

Table 4-30. Keyboard Event Functions	
Subroutine	**Description**
XKeycodeToKeysym()	Converts keycode to a keysym value.
XKeysymToKeycode()	Converts keysym value to keycode.
XKeysymToString()	Converts keysym value to keysym name.
XLookupKeysym()	Translates keyboard event into keysym value.
XLookupMapping()	Gets mapping of keyboard event from keymap file.
XLookupString()	Translates keyboard event into character string.
XRebindCode()	Changes the keyboard mapping in keymap file.
XRebindKeysym()	Maps character string to specified keysym and modifiers.
XRefreshKeyboardMapping()	Refreshes stored modifier and keymap information.
XStringToKeysym()	Converts keysym name to keysym value.
XUseKeymap()	Changes keymap files.
XGeometry()	Parses window geometry given padding and font values.
XGetDefault()	Gets default window options.
XParseColor()	Obtains RGB values from color name.
XParseGeometry()	Parses standard window geometry options.

Manipulating Regions

Table 4-31 (Page 1 of 2). Manipulating Regions	
Subroutine	**Description**
XClipBox()	Generates the smallest enclosing rectangle in region.
XCreateRegion()	Creates a new empty region.
XEmptyRegion()	Determines whether a specified region is empty.
XEqualRegion()	Determines whether two regions are the same.
XIntersectRegion()	Computes intersection of two regions.
XDestroyRegion()	Frees storage associated with specified region.
XOffsetRegion()	Moves specified region by specified amount.
XPointInRegion()	Determines if a point lies in specified region.
XPolygonRegion()	Generates a region from points.
XRectInRegion()	Determines if a rectangle lies in specified region.
XSetRegion()	Sets the GC to the specified region.

Table 4-31 (Page 2 of 2). Manipulating Regions	
Subroutine	**Description**
XShrinkRegion()	Reduces specified region by specified amount.
XSubtractRegion()	Subtracts two regions.
XUnionRegion()	Computes union of two regions.
XUnionRectWithRegion()	Creates a union of source region and rectangle.
XXorRegion()	Gets the difference between union and intersection of regions.

Using Cut and Paste Buffers

Table 4-32. Using Cut and Paste Buffers	
Subroutine	**Description**
XFetchBuffer()	Gets data from specified cut buffer.
XFetchBytes()	Gets data from first cut buffer.
XRotateBuffers()	Rotates the cut buffers.
XStoreBuffer()	Stores data in specified cut buffer.
XStoreBytes()	Stores data in first cut buffer.

Querying Visual Types

Table 4-33. Querying Visual Types	
Subroutine	**Description**
XGetVisualInfo()	Gets a list of visual information structures.
XMatchVisualInfo()	Gets visual information matching screen depth and class.

Manipulating Images

Table 4-34 (Page 1 of 2). Manipulating Images	
Subroutine	**Description**
XAddPixel()	Increases each pixel in pixmap by constant value.
XCreateImage()	Allocates memory for XImage structure.
XDestroyImage()	Frees memory for XImage structure.
XGetPixel()	Gets a pixel value in an image.

Table 4-34 (Page 2 of 2). Manipulating Images	
Subroutine	**Description**
XPutPixel()	Sets a pixel value in an image.
XSubImage()	Creates an image that is a subsection of a specified image.

Manipulating Bitmaps

Table 4-35. Manipulating Bitmaps	
Subroutine	**Description**
XCreateBitmapFromData()	Includes a bitmap in C program.
XCreatePixmapFromBitmapData()	Creates pixmap using bitmap data.
XDeleteContext()	Deletes data associated with window and context type.
XFindContext()	Gets data associated with window and context type.
XReadBitmapFile()	Reads in a bitmap from a file.
XSaveContext()	Stores data associated with window and context type.
XUniqueContext()	Allocates a new context.
XWriteBitmapFile()	Writes out a bitmap to a file.

Using the Resource Manager

Table 4-36 (Page 1 of 2). Using the Resource Manager	
Subroutine	**Description**
Xpermalloc()	Allocates memory that is never freed.
XrmGetFileDatabase()	Creates a database from specified file.
XrmGetResource()	Retrieves a resource from a database.
XrmGetStringDatabase()	Creates a database from specified string.
XrmInitialize()	Initializes the resource manager.
XrmMergeDatabases()	Merges two databases.
XrmParseCommand()	Stores command options in a database.
XrmPutFileDatabase()	Copies database into specified file.
XrmPutLineResource()	Stores a single resource entry in a database.
XrmPutResource()	Stores resource in a database.
XrmPutStringResource()	Stores string resource in a database.
XrmQGetResource()	Retrieves a quark from a database.

Table 4-36 (Page 2 of 2). Using the Resource Manager	
Subroutine	**Description**
XrmQGetSearchList()	Gets a resource search list of database levels.
XrmQGetSearchResource()	Gets a quark search list of database levels.
XrmQPutResource()	Stores binding and quarks in a database.
XrmQPutStringResource()	Stores string binding and quarks in a database.
XrmQuarkToString()	Converts a quark to a character string.
XrmStringToQuark()	Converts character string to a quark.
XrmStringToQuarkList()	Converts character strings to quark list.
XrmStringToBindingQuarkList()	Converts strings to bindings and quarks.
XrmUniqueQuark()	Allocates a new quark.

Display Functions

Table 4-37 (Page 1 of 4). Display Functions	
Subroutine	**Description**
AllPlanes() XAllPlanes()	Returns all bits suitable for use in plane argument.
BitMapBitOrder() XBitMapOrder()	Returns either the most or least significant bit in each bitmap unit.
BitMapPad() XBitMapPad()	Returns the multiple of bits padding each scanline.
BitMapUnit() XBitMapUnit()	Returns the size of a bitmap's unit in bits.
BlackPixel() XBlackPixel()	Returns black pixel value of screen specified.
BlackPixelOfScreen() XBlackPixelOfScreen()	Returns black pixel value of screen specified.
CellsOfScreen() XCellsOfScreen()	Returns number of colormap cells.
ConnectionNumber() XConnectionNumber()	Returns file descriptor of connection.
CreatePixmapCursor() XCreatePixmapCursor()	Creates a pixmap of a specified size.
CreateWindow() XCreateWindow()	Creates an unmapped subwindow for a specified parent window.
DefaultColormap() XDefaultColormap()	Returns default colormap ID for allocation on screen specified.

Table 4-37 (Page 2 of 4). Display Functions	
Subroutine	**Description**
DefaultColormapOfScreen() XDefaultColormapOfScreen	Returns default colormap ID of screen specified.
DefaultDepth() XDefaultDepth()	Returns depth of default root window.
DefaultDepthOfScreen() XDefaultDepthOfScreen()	Returns default depth of screen specified.
DefaultGC() XDefaultGC()	Returns default GC of default root window.
DefaultGCOfScreen() XDefaultGCOfScreen()	Returns default GC of screen specified.
DefaultScreen() XDefaultScreen()	Obtains the default screen referenced in the XOpenDisplay routine.
DefaultScreenofDisplay() XDefaultScreenofDisplay()	Returns the default screen of the display specified.
DefaultRootWindow() XDefaultRootWindow()	Obtains the root window for the default screen specified.
DefaultVisual() XDefaultVisual()	Returns default visual type of the screen specified.
DefaultVisualOfScreen() XDefaultVisualOfScreen()	Returns default visual type of the screen specified.
DisplayCells() XDisplayCells()	Displays number of entries in the default colormap.
DisplayHeight() XDisplayHeight()	Displays height of screen in pixels.
DisplayHeightMM() XDisplayHeightMM()	Displays height of screen in millimeters.
DisplayOfScreen() XDisplayOfScreen()	Displays the type of screen specified.
DisplayPlanes() XDisplayPlanes()	Displays the depth (number of planes) of the root window of the screen specified.
DisplayString() XDisplayString()	Displays the string passed to XOpenDisplay when current display was opened.
DisplayWidth() XDisplayWidth()	Displays width of specified screen in pixels.
DisplayWidthMM() XDisplayWidthMM()	Displays width of specified screen in millimeters.
DoesBackingStore() XDoesBackingStore()	Indicates whether specified screen supports backing stores.

Table 4-37 (Page 3 of 4). Display Functions	
Subroutine	**Description**
DoesSaveUnders() XDoesSaveUnders()	Indicates whether specified screen supports save unders.
EventMaskOfScreen() XEventMaskOfScreen()	Returns initial root event mask for specified screen.
HeightMMOfScreen() XHeightMMOfScreen()	Returns height of specified screen in millimeters.
HeightOfScreen() XHeightOfScreen()	Returns height of specified screen in pixels.
ImageByteOrder() XImageByteOrder()	Specifies required byte order for each scanline unit of an image.
IsCursorKey()	Returns TRUE if keysym is on cursor key.
IsFunctionKey()	Returns TRUE if keysym is on function keys.
IsKeypadKey()	Returns TRUE if keysym is on keypad.
IsMiscFunctionKey()	Returns TRUE if keysym is on miscellaneous function keys.
IsModifierKey()	Returns TRUE if keysym is on modifier keys.
IsPFKey()	Returns TRUE if keysym is on PF keys.
LastKnownRequestProcessed() XLastKnownRequestProcessed()	Extracts full serial number of last known request processed by X Server.
MaxCmapsOfScreen() XMaxCmapsOfScreen()	Returns maximum number of colormaps supported by specified screen.
MinCmapsOfScreen() XMinCmapsOfScreen()	Returns minimum number of colormaps supported by specified screen.
NextRequest() XNextRequest()	Extracts full serial number to be used for next request to be processed by X Server.
PlanesOfScreen() XPlanesOfScreen()	Returns depth (number of planes) in specified screen.
ProtocolRevision() XProtocolRevision()	Returns minor protocol revision number (zero) of X Server associated with display.
ProtocolVersion() XProtocolVersion()	Returns major version number (11) of protocol associated with display.
QLength() XQLength()	Returns length of event queue for display.
RootWindow() XRootWindow()	Returns root window of current screen.
RootWindowOfScreen() XRootWindowOfScreen()	Returns root window of specified screen.
ScreenCount() XScreenCount()	Returns number of screens available.

Table 4-37 (Page 4 of 4). Display Functions	
Subroutine	**Description**
ScreenOfDisplay() XScreenOfDisplay()	Returns pointer to screen of display specified.
ServerVendor() XServerVendor()	Returns pointer to a null-determined string that identifies owner of X Server implementation.
VendorRelease() XVendorRelease()	Returns number related to vendor's release of the X Server.
WhitePixel() XWhitePixel()	Returns white pixel value for current screen.
WhitePixelOfScreen() XWhitePixelOfScreen()	Returns white pixel value of specified screen.
WidthMMOfScreen() XWidthMMOfScreen()	Returns width of specified screen in millimeters.
WidthOfScreen() XWidthOfScreen()	Returns width of specified screen in pixels.

Extension Routines

X-Windows Extension Routines allow you to create extensions of the core Xlib functions with the same performance characteristics. The basic protocol requests for X-Windows extensions are:

- **XQueryExtension**
- **XListExtensions**
- **XFreeExtensionList**

Table 4-38 (Page 1 of 2). Extension Routines	
Subroutine	**Description**
XAllocID()	Returns a resource ID that can be used when creating new resources.
XESetCloseDisplay()	Defines a procedure to call when XCloseDisplay is called.
XESetCopyGC()	Defines a procedure to call when a GC is copied.
XESetCreateFont()	Defines a procedure to call when XLoadQueryFont is called.
XESetCreateGC()	Defines a procedure to call when an new GC is created.
XESetError()	Suppresses the call to an external error handling routine and defines an alternative routine for error handling.
XESetErrorString()	Defines a procedure to call when an I/O error is detected.
XESetEventToWire()	Defines a procedure to call when an event must be converted from the host to wire format.

Table 4-38 (Page 2 of 2). Extension Routines	
Subroutine	**Description**
XESetFreeFont()	Defines a procedure to call when XFreeFont is called.
XESetFreeGC()	Defines a procedure to call when a GC is freed.
XESetWireToEvent()	Defines a procedure to call when an event is converted from the wire to host format.
XFreeExtensionList()	Frees memory allocated by XListExtensions.
XListExtensions()	Returns list of all extensions supported by the server.
XQueryExtension()	Indicates whether named extension is present.

Associate Table Functions

When it is necessary to associate arbitrary information with resource IDs, the **XAssocTable** allows you to associate your own data structures with X resources such as bitmaps, pixmaps, fonts, and windows.

An **XAssocTable** can be used to *type* X resources. For example, to create three or four types of windows with different properties, each window ID is associated with a pointer to a user-defined window property data structure. (A generic type, called XID, is defined in Xlib.h.)

Observe the following guidelines when using an XAssocTable:

- Ensure the correct display is active before initiating an XASSOCTABLE function, because all XIDs are relative to a specified display.
- Restrict the size of the table (number of buckets in the hashing system) to a power of 2, and assign no more than 8 XIDs per bucket to maximize the efficiency of the table.

There is no restriction on the number of XIDs per table or display, or the number of displays per table.

Table 4-39. Associate Table Functions	
Subroutine	**Description**
XCreateAssocTable ()	Returns pointer to newly created associate table.
XDeleteAssoc()	Deletes entry from specified associate table.
XDestroyAssocTable()	Frees memory allocated to specified associate table.
XLookUpAssoc()	Obtains data from specified associate table.
XMakeAssoc()	Creates entry in specified associate table.

X-Windows Toolkit

The Toolkit is a library package layered on top of X-Windows that allows you to simplify the design of applications by providing an underlying set of common user interface functions. Included are mechanisms for defining and expanding intercomponent and intracomponent interaction independently, masking implementation details from both the application and component implementer.

The X-Windows Toolkit is policy-free, making the definition, implementation, and enforcement of policy a function of the application environment, as defined by you.

For additional information on the X-Windows Toolkit, see the *IBM AIX X-Windows Programmer's Reference*, SC23-2118.

Purpose

The X-Windows Toolkit functions manage the following:

- Toolkit initialization
- Widgets and Widget Geometry
- Memory
- Window, file, and timer events
- Input focus
- Selections
- Resources and resource conversion
- Translation of events
- Graphics contexts
- Pixmaps
- Errors and warnings.

Contents

Each X-Windows Toolkit consists of:

- A set of programming mechanisms, called Intrinsics, used to build widgets.
- An architectural model to help programmers design new widgets, with enough flexibility to accommodate different application interface layers.
- A consistent interface, in the form of a coordinated set of widgets and composition policies, some of which are application domain specific while others are common across several application domains.

Requirements

The following libraries must be included in the SYSLIB declaration:

- TCPIP.XTLIB (Xt Intrinsics)
- TCPIP.XAWLIB (Athena Widget Set)
- TCPIP.XR11LIB (Hewlett-Packard Widget Set)
- TCPIP.X11LIB
- TCPIP.OLDXLIB

Applications that use the X-Windows Toolkit must include the following header files:

- <Xlib.h>
- <Intrinsi.h>
- <StringDf.h>

and possibly:

- <Atoms.h>
- <Shell.h>

The applications should also include the additional headers for each widget class to be used, such as <Label.h> or <Scroll.h>. The object library file of the Intrinsic is named Xtlib.

Again, many of the names of header files needed by these toolkits and widget sets have been shortened because of the file name conventions in MVS. The name changes in general shorten the MVS file name to eight characters or less. Table 4-40 describes the name changes.

Table 4-40. Differences in Header File Names for Toolkits

MIT **Distribution Name**	**X-Windows Toolkit Name**
Cardinals.h	Cardinal.h
Composite.h	Composit.h
Constraint.h	Constrai.h
Cascade.h[1]	HPCascad.h
CascadeP.h[1]	HPCascaP.h
Form.h[1]	HPForm.h
Intrinsic.h	Intrinsi.h
ScrollBar.h	ScrollBa.h
StringDefs.h	StringDf.h
TEDisplayP.h	TEDisplP.h
TESourceP.h	TESourcP.h
TMprivate.h	TMprivat.h
WorkSpace.h	WorkSpac.h

Note:

1. Cascade.h, CascadeP.h, and Form.h are the Hewlett-Packard widget include files renamed to avoid conflict with files of the same names belonging to the Athena widgets. The Athena files have kept the same names as on the MIT distribution.

When using the Athena Widget Set, and your application uses text widgets, you must explicitly INCLUDE SYSLIB(TEXT) when you load your application code, as not all entry points are defined as external references in TCPIP.XAWLIB.

Likewise, the Hewlett-Packard Widget Set has two routines which you may have to explicitly include. These are PRIMITIV and TEXTEDIT.

Defining Widgets

The fundamental data type of the X-Windows Toolkit is the widget. A widget is allocated dynamically and contains state information. Every widget belongs to one widget class that is allocated statically and initialized. The widget class contains the operations allowed on widgets of that class.

For the list of X-Windows Toolkit routines, refer to *IBM AIX X-Windows Programmer's Reference* (SC23-2118), or the X-Windows programmer's manual you are using.

Chapter 5. Remote Procedure Calls

The Remote Procedure Call (RPC) protocol permits remote execution of subroutines across a TCP/IP network. RPC, together with the External Data Representation (XDR) protocol, defines a standard for representing data that is independent of internal protocols or formatting. Remote Procedure Calls can communicate between processes on the same or different hosts.

This chapter describes the high-level RPC routines implemented in the TCP/IP for MVS program. Only the RPC programming interface to the C Language, and communication between processes on different hosts is discussed here.

For more detailed information on RPC and XDR, refer to the SUN Microsystems documentation, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide*, and protocol specifications in the following RFCs:

- *Remote Procedure Call Protocol Specification* (RFC 1057)
- *XDR: External Data Representation Standard* (RFC 1014).

The RPC Interface

The RPC interface enables programmers to write distributed applications using high-level RPCs rather than lower level calls based on sockets.

In the RPC process, the client is communicating with a server. The client invokes a procedure to send a call message to the server. When the message arrives, the server calls a dispatch routine, and performs the requested service. The server sends back a reply message, after which the original procedure call returns to the client program with a value derived from the reply message.

To use the RPC interface, you must be familiar with programming in the C Language, and possess a working knowledge of networking concepts.

Software Requirements

Programs that use Remote Procedure Calls require the following:

- IBM C for System/370, Compiler Licensed Program (5688-040)
- IBM C for System/370, Library Licensed Program (5688-039)
- IBM VS Pascal Compiler & Library (5668-767) **or** IBM VS Pascal Library (5668-717).

Remote Procedure Call Quick Reference

Table 5-1 on page 5-2 lists the RPCs supported by the TCP/IP for MVS program.

---

[1] Trademark of American Telephone and Telegraph Company.

Table 5-1 (Page 1 of 3). Remote Procedure Call Quick Reference		
Remote Procedure Call	**Description**	**Page**
auth_destroy()	Destroys authentication information.	5-5
authnone_create()	Creates and returns a null RPC authentication handle.	5-6
authunix_create()	Creates and returns a UNIX[1]-based authentication handle.	5-6
authunix_create_default()	Calls authunix_create() with default parameters.	5-6
callrpc()	Calls remote procedures.	5-6
clnt_call()	Calls remote procedures.	5-7
clnt_destroy()	Destroys client's RPC handle.	5-8
clnt_freeres()	De-allocates resources assigned for decoding RPC.	5-8
clnt_geterr()	Copies error structure from client's handle to local address.	5-8
clnt_pcreateerror()	Indicates why a client handle cannot be created.	5-9
clnt_perrno()	Writes error message indicating why RPC failed.	5-9
clnt_perror()	Writes error message indicating why RPC failed.	5-9
clntraw_create()	Creates client transport handle for use in a single task.	5-10
clnttcp_create()	Creates an RPC client for the remote program using TCP transport.	5-10
clntudp_create()	Creates an RPC client for the remote program using UDP transport.	5-11
get_myaddress()	Returns local host's internet address.	5-11
mvs_xdr_enum()	Translates C-enumerated numbers to their external representations.	5-11
pmap_getmaps()	Returns list of current program to port mappings on specified remote host.	5-12
pmap_getport()	Returns port number associated with remote program.	5-12
pmap_rmtcall()	Instructs remote host to make RPC call on the client's behalf.	5-13
pmap_set()	Sets mapping of server program to port on local machine.	5-13
pmap_unset()	Resets mappings on the local machine.	5-14
registerrpc()	Registers procedure with local RPC portmapper.	5-14
rpc_createerr()	Global variable set when any RPC client creation routine fails.	5-15
svc_destroy()	Destroys RPC service transport handle.	5-15
svc_fds()	Specifies read descriptor bit mask on the service transport machine.	5-15
svc_freeargs()	Frees storage allocated for arguments.	5-15
svc_getargs()	Decodes arguments from an RPC service transport handle.	5-16
svc_getcaller()	Obtains the network address of the client associated with the service transport handle.	5-16

Table 5-1 (Page 2 of 3). Remote Procedure Call Quick Reference		
Remote Procedure Call	**Description**	**Page**
svc_getreq()	Implements asynchronous event processing, and returns control to program after all sockets have been serviced.	5-16
svc_register()	Registers procedures on portmapper.	5-17
svc_run()	Accepts RPC requests, and calls appropriate service.	5-17
svc_sendreply()	Sends results of RPC to caller.	5-17
svc_unregister()	Removes local mapping.	5-18
svcerr_auth()	Returns error reply when service cannot execute RPC due to authentication errors.	5-18
svcerr_decode()	Returns error reply when service cannot decode its parameters.	5-18
svcerr_noproc()	Returns error reply when service cannot call procedure requested.	5-19
svcerr_noprog()	Returns error reply when service cannot call program requested.	5-19
svcerr_progvers()	Returns error reply when service cannot call version of program requested.	5-19
svcerr_systemerr()	Returns error reply when service detects system error not handled.	5-19
svcerr_weakauth()	Returns error reply when service cannot execute RPC because of weak authentication parameters.	5-20
svcraw_create()	Creates service transport handle to simulate RPC programs in a single task.	5-20
svctcp_create()	Creates TCP based service transport.	5-20
svcudp_create()	Creates UDP based service transport.	5-21
xdr_accepted_reply()	Translates RPC reply messages.	5-21
xdr_array()	Translates array to its external representation.	5-21
xdr_authunix_parms()	Translates UNIX-based authentication information.	5-22
xdr_bool()	Translates booleans to their external representations.	5-22
xdr_bytes()	Translates counted byte strings.	5-22
xdr_callhdr()	Translates RPC call message header.	5-23
xdr_callmsg()	Translates RPC call messages.	5-23
xdr_double()	Translates C double-precision numbers to their external representations.	5-23
xdr_enum()	Translates C-enumerated numbers to their external representations.	5-24
xdr_float()	Translates C floating-point numbers to their external representations.	5-25

Table 5-1 (Page 3 of 3). Remote Procedure Call Quick Reference		
Remote Procedure Call	**Description**	**Page**
xdr_inline()	Invokes inline routine, and returns pointer to continuous piece of XDR buffer.	5-25
xdr_int()	Translates C integers to their external representations.	5-26
xdr_long()	Translates C long integers to their external representations.	5-26
xdr_opaque()	Translates fixed-size opaque data to its external representation.	5-26
xdr_opaque_auth()	Translates RPC authentication data.	5-27
xdr_pmap()	Translates port map elements.	5-27
xdr_pmaplist()	Translates list of port mappings.	5-27
xdr_reference()	Provides pointer "chasing" within structures.	5-28
xdr_rejected_reply()	Translates rejected RPC reply messages.	5-28
xdr_replymsg()	Translates RPC reply messages.	5-28
xdr_short()	Translates between C short integers and their external representations.	5-29
xdr_string()	Translates between C strings and their external representations.	5-29
xdr_u_int()	Translates between C unsigned integers and their external representations.	5-29
xdr_u_long()	Translates between C unsigned long integers and their external representations.	5-30
xdr_u_short()	Translates between C unsigned short integers and their external representations.	5-30
xdr_union()	Translates between a discriminated C union and its external representations.	5-30
xdr_void()	Returns a value of one.	5-31
xdr_wrapstring()	Translates strings to their external representation.	5-31
xprt_register()	Registers service transport handles with the RPC service package, and modifies the service transport.	5-31
xprt_unregister()	Unregisters RPC service transport handle before it is destroyed.	5-31

Remote Procedure Call Library

In addition to the C and Pascal Libraries, include TCPIP.COMMTXT in SYSLIB for linking. When compiling, include TCPIP.COMMMAC and the standard C SYSLIB allocation.

The RPC routines are in the TCPIP.COMMTXT data set and the headers are in the TCPIP.COMMMAC data set.

The following statements are required at the beginning of each program that uses RPC calls.

```
#define MVS
#include "rpc.h"
```

A sample linkedit for a file MYFILE in 31-bit mode is:

```
INCLUDE OBJ(MYFILE)
INCLUDE SYSLIB(AMPZMVSB)
MODE AMODE(31),RMODE(ANY)
ENTRY CEESTART
NAME MYFILE(R)
```

A sample linkedit for a file MYFILE in 24-bit mode is:

```
INCLUDE OBJ(MYFILE)
INCLUDE SYSLIB(AMPZRP01)
MODE AMODE(24),RMODE(24)
ENTRY CEESTART
NAME MYFILE(R)
```

A description of each RPC routine and its relevant parameters follows.

auth_destroy()

This procedure destroys (discards, frees storage for reuse) the authentication information for *auth*. Once this procedure is called, *auth* points to NULL.

```
void
auth_destroy(auth)
        AUTH *auth;
```

Parameter	Usage
auth	A pointer to authentication information.

authnone_create()

This procedure creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

```
AUTH *
authnone_create()
```

authunix_create()

This procedure creates and returns an authentication handle that contains UNIX-based authentication information.

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
        char *host;
int uid, gid, len, *aup_gids;
```

Parameter	Usage
host	A pointer to to the symbolic name of the host where the desired server is located.
uid	The user's user ID.
gid	The user's group ID.
len	The length of the information pointed to by *aup_gids*.
aup_gids	A pointer to a counted array of groups to which the user belongs.

authunix_create_default()

This procedure calls **authunix_create()** with default parameters.

```
AUTH *
authunix_create_default()
```

callrpc()

This procedure calls the remote procedure described by *prognum, versnum*, and *procnum* running on the *host* system. It can encode and decode the parameters for transfer, and returns a 0 if it was successful, and an *enum clnt_stat* cast to an integer if it was not. The results of the remote procedure call are returned to *out*.

```
enum clnt_stat
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
        char *host, *in, *out;
u_long prognum, versnum, procnum;
xdrproc_t inproc, outproc;
```

Parameter	Usage
host	A pointer to the symbolic name of the host where the desired server is located.
prognum	Used to identify the remote procedure's program number.
versnum	Used to identify the remote procedure's version number.
procnum	Used to identify the remote procedure's procedure number.
inproc	The XDR procedure used to encode the remote procedure's arguments.
in	A pointer to the remote procedure's arguments.
outproc	The XDR procedure used to decode the remote procedure's results.
out	A pointer to the remote procedure's results.

Notes:

1. **clnt_perrno()** can be used to translate the return code into messages.
2. **callrpc** cannot call the procedure **xdr_enum**. Refer to "xdr_enum()" on page 5-24 for more information.
3. This procedure uses UDP as its transport layer. Refer to "clntudp_create()" on page 5-11 for more information.

clnt_call()

This procedure calls the remote procedure (*procnum*) associated with the client handle (*clnt*).

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
        CLIENT *clnt;
u_long procnum;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval tout;
```

Parameter	Usage
clnt	Points to a client handle that can be obtained using **clntudp_create()**, **clnttcp_create()**, or **clntraw_create()**.
procnum	Used to identify the remote procedure number.
inproc	The XDR procedure used to encode *procnum*'s arguments.
in	Points to the remote procedure's arguments.
outproc	The XDR procedure used to decode the remote procedure's results.
out	Points to the remote procedure's results.
tout	The time allowed for the server to respond in units of 0.1 seconds.

clnt_destroy()

This procedure destroys a client RPC transport handle. This procedure involves the deallocation of private data resources, including *clnt*. Once this procedure is used, *clnt* points to NULL. Open sockets associated with *clnt* must be closed.

```
void
clnt_destroy(clnt)
        CLIENT *clnt;
```

Parameter	Usage
clnt	Points to a client handle that can be created using **clntudp_create()**, **clnttcp_create()**, or **clntraw_create()**.

clnt_freeres()

This procedure deallocates any resources that were assigned by the system to decode the results of an RPC call. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
clnt_freeres(clnt, outproc, out)
        CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

Parameter	Usage
clnt	Points to a client handle, that can be obtained using **clntudp_create()**, **clnttcp_create()**, or **clntraw_create()**.
outproc	The XDR procedure used to decode the remote procedure's results.
out	Points to the remote procedure's results.

clnt_geterr()

This procedure copies the error structure from the client handle to the structure at address *errp*.

```
void
clnt_geterr(clnt, errp)
        CLIENT *clnt;
struct rpc_err *errp;
```

Parameter	Usage
clnt	Points to a client handle, that can be obtained using **clntudp_create()**, **clnttcp_create()**, or **clntraw_create()**.
errp	Points to the address into which the error structure is copied.

clnt_pcreateerror()

This procedure writes a message to the standard error device indicating why a client handle cannot be created. This procedure is used after the **clntraw_create()**, **clnttcp_create()**, or **clntudp_create()** calls fail as indicated by a returned value of NULL.

```
void
clnt_pcreateerror(s)
        char *s;
```

Parameter	Usage
s	Points to a string that is to be printed in front of the message. The string is followed by a colon.

clnt_perrno()

This procedure writes a message to the standard error device corresponding to the condition indicated by *stat*. This procedure should be used after **callrpc()** if there is an error.

```
void
clnt_perrno(stat)
        enum clnt_stat stat;
```

Parameter	Usage
stat	The client status.

clnt_perror()

This procedure writes a message to the standard error device indicating why an RPC call failed. This procedure should be used after **clnt_call()** if there is an error.

```
void
clnt_perror(clnt, s)
        CLIENT *clnt;
char *s;
```

Parameter	Usage
clnt	Points to a client handle, that can be obtained using **clntudp_create()**, **clnttcp_create()**, or **clntraw_create()**.
s	Points to a string that is to be printed in front of the message. The string is followed by a colon.

clntraw_create()

This procedure creates a dummy client for the remote double (*prognum, versnum*). Because messages are passed using a buffer within the local process's address space, the server should also use the same address space. This allows the simulation of RPC programs within one address space. Refer to "svcraw_create()" on page 5-20 for more information.

This procedure returns NULL if it fails.

```
CLIENT *
clntraw_create(prognum, versnum)
        u_long prognum, versnum;
```

Parameter	Usage
prognum	The remote program number.
versnum	The version number of the remote program.

clnttcp_create()

This procedure creates an RPC client transport handle for the remote program specified by (*prognum, versnum*). The client uses TCP as the transport layer.

This procedure returns NULL if it fails.

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
        struct sockaddr_in *addr;
u_long prognum, versnum;
int *sockp;
u_int sendsz, recvsz;
```

Parameter	Usage
addr	Points to the internet address of the remote program. If *addr* points to a port number of zero (0), *addr* is set to the port the remote program is receiving on. The remote PORTMAP service is used for this.
prognum	The remote program number.
versnum	The version number of the remote program.
sockp	Points to the socket (virtual port). If *sockp* is RPC_ANYSOCK, then this routine opens a new socket and sets *sockp*.
sendsz	The size of the send buffer. Use zero to get the default.
recvsz	The size of the receive buffer. Use zero to get the default.

clntudp_create()

This procedure creates a client transport handle for the remote program (*prognum*) with version (*versnum*). UDP is used as the transport layer.

Note: This procedure should not be used with procedures that use large arguments or return large results. UDP RPC messages can only contain 8K bytes of encoded data.

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
        struct sockaddr_in *addr;
u_long prognum, versnum;
struct timeval wait;
int *sockp;
```

Parameter	Usage
addr	Points to the internet address of the remote program. If *addr* points to a port number of zero (0), *addr* is set to the port the remote program is receiving on. The remote PORTMAP service is used for this.
prognum	The remote program number.
versnum	The version number of the remote program.
wait	UDP resends the call request at intervals of *wait* time, until either a response is received or the call times out. The time out length is set using the **clnt_call()** procedure.
sockp	Points to the socket (virtual port). If *sockp* is RPC_ANYSOCK, this routine opens a new socket and sets *sockp*.

get_myaddress()

This procedure puts the local host's internet address into *addr*.

The port number (*addr->sin_port*) is set to **htons(PMAPPORT)**, which is 111.

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Parameter	Usage
addr	Points to the location where the local internet address is placed.

mvs_xdr_enum()

This XDR procedure translates between C-enumerated groups and their external representation. Refer to the **Usage Note:** with "xdr_enum()" on page 5-24 for more information.

```
bool_t
mvs_xdr_enum(xdrs, ep, size)
        XDR *xdrs;
enum_t *ep;
int size;
```

Parameter	Usage
xdrs	Points to an XDR stream.
ep	Points to the enumerated type variable.
size	The size of the enumerated type variable.

pmap_getmaps()

This procedure returns a list of current program to port mappings on the remote host specified by *addr*.

```
struct pmaplist *
pmap_getmaps(addr)
        struct sockaddr_in *addr;
```

Parameter	Usage
addr	Points to the remote host's internet address.

pmap_getport()

This procedure returns the port number associated with the remote program (*prognum*), with version (*versnum*), and transport protocol (*protocol*). A return value of zero indicates that the mapping does not exist or that the remote PORTMAP could not be contacted. If the remote PORTMAP server could not be contacted, the global variable *rpc_createerr* contains the RPC status.

```
u_short
pmap_getport(addr, prognum, versnum, protocol)
        struct sockaddr_in *addr;
u_long prognum, versnum, protocol;
```

Parameter	Usage
addr	Points to the remote host's internet address.
prognum	The program number to be mapped.
versnum	The version number of the program to be mapped.
protocol	The transport protocol used by the program.

pmap_rmtcall()

This procedure instructs PORTMAP on the remote host to make an RPC call to a procedure on that host, on your behalf. This procedure should only be used for PING type functions.

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc,
  out, tout, portp)
        struct sockaddr_in *addr;
u_long prognum, versnum, procnum, *portp;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval tout;
```

Parameter	Usage
addr	Points to the remote host's internet address.
prognum	The remote program number.
versnum	The version number of the remote program.
procnum	Used to identify the procedure to be called.
inproc	The XDR procedure used to encode the remote procedure's arguments.
in	Points to the remote procedure's arguments.
outproc	The XDR procedure used to decode the remote procedure's results.
out	Points to the remote procedure's results.
tout	The time out period for the remote request.
portp	If the call from the remote PORTMAP service is successful, *portp* contains the port number of the triple (*prognum, versnum, procnum*).

pmap_set()

This is part of the interface to the PORTMAP service.

This procedure sets the mapping of the program (specified by *prognum, versnum,* and *protocol*) to *port* on the local machine. This procedure is automatically called by the **svc_register()** procedure. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
pmap_set(prognum, versnum, protocol, port)
        u_long prognum, versnum, protocol;
u_short port;
```

Parameter	Usage
prognum	The local program number.
versnum	The version number of the local program.
protocol	The transport protocol used by the local program.
port	The port to which the local program is mapped.

pmap_unset()

This is part of the interface to the PORTMAP service.

This procedure removes the mappings associated with *prognum* and *versnum* on the local machine. All ports for each transport protocol currently mapping the *prognum* and *versnum* are removed from the PORTMAP service. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
pmap_unset(prognum, versnum)
        u_long prognum, versnum;
```

Parameter	Usage
prognum	The local program number.
versnum	The version number of the local program.

registerrpc()

This procedure registers a procedure (*prognum, versnum, procnum*) with the local PORTMAP server, and creates a control structure to remember the server procedure and its XDR routine. The control structure is used by **svc_run()**. When a request arrives for the program (*prognum, versnum, procnum*), the procedure *procname* is called. Procedures registered using **registerrpc()** are accessed using the UDP transport layer. This routine returns a 0 if it succeeds, and a -1 if it does not.

Note: **xdr_enum()** cannot be used as an argument to **registerrpc**. Refer to "xdr_enum()" on page 5-24 for more information.

```
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
        u_long prognum, versnum, procnum;
char *(*procname) ();
xdrproc_t inproc, outproc;
```

Parameter	Usage
prognum	The program number to register.
versnum	The version number to register.
procnum	The procedure number to register.
procname	The procedure that is called when the registered program is requested. *procname* must accept a pointer to its arguments, and return a static pointer to its results.
inproc	The XDR routine used to decode the arguments.
outproc	The XDR routine that encodes the results.

rpc_createerr

A global variable that is set when any RPC client creation routine fails. Use **clnt_pcreateerror()** to print the message.

```
struct  rpc_createerr rpc_createerr;
```

svc_destroy()

This procedure destroys the RPC service transport handle *xprt*, which becomes NULL after this routine is called.

```
void
svc_destroy(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svc_fds

This is a global variable that specifies the read descriptor bit mask on the service machine. This is only of interest if the service programmer decides to write an asynchronous event processing routine; otherwise **svc_run()** should be used. Writing asynchronous routines in the MVS environment is not simple because there is no direct relationship between the descriptors used by the socket routines and the Event Control Blocks commonly used by MVS programs for coordinating concurrent activities.

Warning: Do not modify this variable.

```
int    svc_fds;
```

svc_freeargs()

This procedure frees storage allocated to decode the arguments received by **svc_getargs()**. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
svc_freeargs(xprt, inproc, in)
        SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameter	Usage
xprt	Points to the service transport handle.
inproc	The XDR routine used to decode the arguments.
in	Points to the input arguments.

svc_getargs()

This procedure uses the XDR routine *inproc* to decode the arguments of an RPC request associated with the RPC service transport handle *xprt*. The results are placed at address *in*. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
svc_getargs(xprt, inproc, in)
        SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameter	Usage
xprt	Points to the service transport handle.
inproc	The XDR routine used to decode the arguments.
in	Points to the decoded arguments.

svc_getcaller()

This macro obtains the socket address of the client associated with the service transport handle *xprt*.

```
struct sockaddr_in *
svc_getcaller(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svc_getreq()

This procedure is used instead of **svc_run()** to implement asynchronous event processing. The routine returns control to the program once all sockets associated with the read descriptor bit mask have been serviced.

```
void
svc_getreq(rdfds)
        int rdfds;
```

Parameter	Usage
rdfds	The read descriptor bit mask.

svc_register()

This procedure associates the program described by (*prognum, versnum*) with the service dispatch routine *dispatch*. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
        SVCXPRT *xprt;
u_long prognum, versnum protocol;
void (*dispatch) ();
```

Parameter	Usage
xprt	Points to the service transport handle.
prognum	The program number to be mapped.
versnum	The version number of the program to be mapped.
dispatch	The dispatch routine associated with *prognum* and *versnum*.
	The structure of the dispatch routine is as follows:

```
dispatch(request, xprt)
        struct svc_req *request;
SVCXPRT *xprt;
```

protocol	The protocol used. This value is generally one of: • 0 (zero) • IPPROTO_UDP • IPPROTO_TCP When a value of 0 is used, the service is not registered with PORTMAP.

Note: When using a toy RPC service transport created with **svcraw_create**, a call to **xprt_register** must be made immediately after a call to **svc_register**.

svc_run()

This procedure does not return control. It accepts RPC requests, and calls the appropriate service using **svc_getreq()**.

```
svc_run()
```

svc_sendreply()

This procedure is called by the service dispatch routine to send the results of the call to the caller. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
svc_sendreply(xprt, outproc, out)
        SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Parameter	Usage
xprt	Points to the caller's transport handle.
outproc	The XDR procedure used to encode the results.
out	Points to the results.

svc_unregister()

This procedure removes all local mappings of (*prognum, versnum*) to dispatch routines and (*prognum, versnum,* *) to port numbers.

```
void
svc_unregister(prognum, versnum)
        u_long prognum, versnum;
```

Parameter	Usage
prognum	The program number that is removed.
versnum	The version number of the program that is removed.

svcerr_auth()

This procedure is called by a service dispatch routine that refuses to execute an RPC request because of authentication errors.

```
void
svcerr_auth(xprt, why)
        SVCXPRT *xprt;
enum auth_stat why;
```

Parameter	Usage
xprt	Points to the service transport handle.
why	The reason the call was refused.

svcerr_decode()

This procedure is called by a service dispatch routine that cannot decode its parameters.

```
void
svcerr_decode(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svcerr_noproc()

This procedure is called by a service dispatch routine that does not implement the procedure requested.

```
void
svcerr_noproc(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svcerr_noprog()

This procedure is used when the desired program is not registered.

```
void
svcerr_noprog(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svcerr_progvers()

This procedure is used when the desired version of a program is not registered.

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svcerr_systemerr()

This procedure is called by a service dispatch routine when it detects a system error that is not handled by the protocol.

```
void
svcerr_systemerr(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

svcerr_weakauth()

This procedure is called by a service dispatch routine that cannot execute an RPC call because of correct but weak authentication parameters.

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

Note: This is the equivalent of: `svcerr_auth(xprt, AUTH_TOOWEAK)` .

svcraw_create()

This procedure creates a local RPC service transport used for timings, to which it returns a pointer. Messages are passed using a buffer within the address space of the local process; hence, the client process must also use the same address space. This allows the simulation of RPC programs within one computer. Refer to "clntraw_create()" on page 5-10 for more information.

This routine returns NULL if it fails.

```
SVCXPRT *
svcraw_create()
```

svctcp_create()

This procedure creates a TCP-based service transport to which it returns a pointer.

This routine returns NULL if it fails.

```
SVCXPRT *
svctcp_create(sockp, send_buf_size, recv_buf_size)
        int sockp;
u_int send_buf_size, recv_buf_size;
```

Parameter	Usage
sockp	Points to the socket (virtual port). If *sockp* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.
send_buf_size	The size of the send buffer. Specify zero (0) to choose the default.

recv_buf_size The size of the receive buffer. Specify zero (0) to choose the default.

svcudp_create()

This procedure creates a UDP-based service transport to which it returns a pointer.

This routine returns NULL if it fails.

```
SVCXPRT *
svcudp_create(sockp)
        int sockp;
```

Parameter	Usage
sockp	Points to the socket associated with the service transport handle. If *sockp* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.

Warning: UDP can only transmit 8K bytes of data per message.

xdr_accepted_reply()

This XDR procedure is used to translate RPC reply messages.

```
bool_t
xdr_accepted_reply(xdrs, ar)
        XDR *xdrs;
struct accepted_reply *ar;
```

Parameter	Usage
xdrs	Points to an XDR stream.
ar	Points to the reply to be represented.

xdr_array()

This filter primitive translates between an array and its external representation. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
char **arrp;
u_int *sizep, maxsize, elsize;
xdrproc_t elproc;
```

Parameter	Usage
xdrs	Points to an XDR stream.
arrp	The address of the pointer to the array.

sizep	Points to the element count of the array.
maxsize	The maximum number of elements accepted.
elsize	The size of each of the array's elements (found using **sizeof()**).
elproc	The XDR routine that translates an individual array element.

xdr_authunix_parms()

This XDR procedure translates UNIX-based authentication information.

```
bool_t
xdr_authunix_parms(xdrs, aupp)
        XDR *xdrs;
struct authunix_parms *aupp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
aupp	Points to the authentication information.

xdr_bool()

This primitive translates between booleans and their external representation. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_bool(xdrs, bp)
        XDR *xdrs;
bool_t *bp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
bp	Points to the Boolean.

xdr_bytes()

This XDR procedure translates between counted byte strings and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
char **sp;
u_int *sizep, maxsize;
```

Parameter	Usage
xdrs	Points to an XDR stream.
sp	Points to the byte string.
sizep	Points to the byte string size.
maxsize	The maximum size of the byte string.

xdr_callhdr()

This XDR procedure translates an RPC message header.

```
void
xdr_callhdr(xdrs, chdr)
        XDR *xdrs;
struct rpc_msg *chdr;
```

Parameter	Usage
xdrs	Points to an XDR stream.
chdr	Points to the call header.

xdr_callmsg()

This XDR procedure is used to translate RPC call messages (header and authentication; not argument data).

```
void
xdr_callmsg(xdrs, cmsg)
        XDR *xdrs;
struct rpc_msg *cmsg;
```

Parameter	Usage
xdrs	Points to an XDR stream.
cmsg	Points to the call message.

xdr_double()

This XDR procedure translates between C double-precision numbers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_double(xdrs, dp)
        XDR *xdrs;
double *dp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
dp	Points to a double-precision number.

xdr_enum()

This XDR procedure translates between C-enumerated groups and their external representation. This routine returns a 1 if it succeeds, and a 0 if it does not.

Usage Note: The **xdr_enum()** procedure is a macro that calls the **mvs_xdr_enum()** procedure. The **xdr_enum()** macro cannot be used as a parameter in another procedure such as **callrpc()** and **registerrpc()**.

```
bool_t
xdr_enum(xdrs, ep)
        XDR *xdrs;
enum_t *ep;
```

Parameter	Usage
xdrs	Points to an XDR stream.
ep	Points to the enumerated number.

When calling the procedures **callrpc** and **registerrpc**, a stub procedure must be created for both server and client before the procedure of the application program using **xdr_enum**. This procedure should look like the following:

```
void
static xdr_enum_t(xdrs, ep)
        XDR *xdrs;
enum_t *ep;
{
        xdr_enum(xdrs, ep)
}
```

The **xdr_enum_t** procedure is used as the *inproc* and *outproc* in both the client and server RPCs.

A sample RPC client would contain the following line:

```
                        :

error = callrpc(argv[1],ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,
                &innumber,xdr_enum_t,&outnumber);

                        :
```

A sample RPC server would contain the following line:

```
                    ⋮

registerrpc(ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,
            xdr_enum_t);

                    ⋮
```

xdr_float()

This XDR procedure translates between C floating-point numbers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_float(xdrs, fp)
        XDR *xdrs;
float *fp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
fp	Points to the floating-point number.

xdr_inline()

This XDR procedure returns a pointer to a continuous piece of the XDR stream's buffer. The value is `long *` rather than `char *` because the external data representation of any object is always an integer multiple of 32 bits.

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
int len;
```

Parameter	Usage
xdrs	Points to an XDR stream.
len	The byte length of the desired buffer.

Note: **xdr_inline()** may return NULL if there is not sufficient space in the stream buffer to satisfy the request.

xdr_int()

This XDR procedure translates between C integers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_int(xdrs, ip)
        XDR *xdrs;
int *ip;
```

Parameter	Usage
xdrs	Points to an XDR stream.
ip	Points to the integer.

xdr_long()

This XDR procedure translates between C long integers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_long(xdrs, lp)
        XDR *xdrs;
long *lp;
```

Parameter	Usage
xdrs	Point to an XDR stream.
lp	Points to the long integer.

xdr_opaque()

This XDR procedure translates between fixed size opaque data and its external representation. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
char *cp;
u_int cnt;
```

Parameter	Usage
xdrs	Points to an XDR stream.
cp	Points to the opaque object.
cnt	The size of the opaque object.

xdr_opaque_auth()

This XDR procedure is used to translate RPC message authentications.

```
bool_t
xdr_opaque_auth(xdrs, ap)
        XDR *xdrs;
struct opaque_auth *ap;
```

Parameter	Usage
xdrs	Points to an XDR stream.
ap	Points to the opaque authentication information.

xdr_pmap()

This XDR procedure translates an RPC procedure identification such as is used in calls to PORTMAP.

```
bool_t
xdr_pmap(xdrs, regs)
        XDR *xdrs;
struct pmap *regs;
```

Parameter	Usage
xdrs	Points to an XDR stream.
regs	Points to the PORTMAP parameters.

xdr_pmaplist()

This XDR procedure translates a variable number of RPC procedure identifications such as PORTMAP creates.

```
bool_t
xdr_pmaplist(xdrs, rp)
        XDR *xdrs;
struct pmaplist **rp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
rp	Points to the PORTMAP data array.

xdr_reference()

This XDR procedure provides pointer "chasing" within structures. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Parameter	Usage
xdrs	Points to an XDR stream.
pp	Points to a pointer.
size	The size of (**sizeof()**) to which the structure ***pp* points.
proc	The XDR procedure that translates an individual element of the type addressed by the pointer.

xdr_rejected_reply()

This XDR procedure is used to translate RPC reply messages.

```
bool_t
xdr_rejected_reply(xdrs, rr)
        XDR *xdrs;
struct rejected_reply *rr;
```

Parameter	Usage
xdrs	Points to an XDR stream.
rr	Points to the rejected reply.

xdr_replymsg()

This XDR procedure is used to translate RPC reply messages.

```
bool_t
xdr_replymsg(xdrs, rmsg)
        XDR *xdrs;
struct rpc_msg *rmsg;
```

Parameter	Usage
xdrs	Points to an XDR stream.
rmsg	Points to the reply message.

xdr_short()

This XDR procedure translates between C short integers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_short(xdrs, sp)
        XDR *xdrs;
short *sp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
sp	Points to the short integer.

xdr_string()

This XDR procedure translates between C strings and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
char **sp;
u_int maxsize;
```

Parameter	Usage
xdrs	Points to an XDR stream.
sp	Points to a pointer to the string.
maxsize	The maximum size of the string.

Usage Note: This procedure uses the *userid*.STANDARD.TCPXLBIN translation table. If this table is not found, TCPIP.STANDARD.TCPXLBIN is used.

xdr_u_int()

This XDR procedure translates between C unsigned integers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_u_int(xdrs, up)
        XDR *xdrs;
unsigned *up;
```

Parameter	Usage
xdrs	Points to an XDR stream.
up	Points to the unsigned integer.

xdr_u_long()

This XDR procedure translates between C unsigned long integers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
unsigned long *ulp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
ulp	Points to the unsigned long integer.

xdr_u_short()

This XDR procedure translates between C unsigned short integers and their external representations. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_u_short(xdrs, usp)
        XDR *xdrs;
unsigned short *usp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
usp	Points to the unsigned short integer.

xdr_union()

This XDR procedure translates between a discriminated C union and its external representation. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Parameter	Usage
xdrs	Points to an XDR stream.
dscmp	Points to the union's discriminant.
unp	Points to the union.
choices	Points to an array detailing which XDR procedure to use on each arm of the union.
dfault	The default XDR procedure to use.

xdr_void()

This XDR procedure always returns a value of 1.

```
bool_t
xdr_void()
```

xdr_wrapstring()

This XDR procedure is the equivalent of calling **xdr_string()** with a maximum size of MAXUNSIGNED. It is useful because many RPC procedures implicitly invoke two-parameter XDR routines, and **xdr_string()** is a three-parameter routine. This routine returns a 1 if it succeeds, and a 0 if it does not.

```
bool_t
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
char **sp;
```

Parameter	Usage
xdrs	Points to an XDR stream.
sp	Points to a pointer to the string.

Usage Note: This procedure uses the *userid*.STANDARD.TCPXLBIN translation table. If this table is not found, TCPIP.STANDARD.TCPXLBIN is used.

xprt_register()

This procedure registers service transport handles with the RPC service package. This routine also modifies the global variable **svc_fds**.

```
void
xprt_register(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

xprt_unregister()

This procedure unregisters an RPC service transport handle. A transport handle should be unregistered with the RPC service package before it is destroyed. This routine also modifies the global variable **svc_fds**.

```
void
xprt_unregister(xprt)
        SVCXPRT *xprt;
```

Parameter	Usage
xprt	Points to the service transport handle.

Appendix A. VMCF Interface

You can communicate directly with the TCPIP address space using the simulated Virtual Machine Communication Facility (VMCF) calls, instead of the Pascal API or C Sockets. You can use the simulated VMCF calls when:

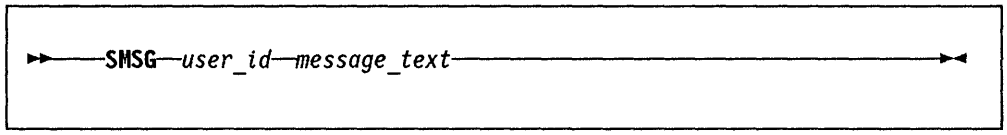
1. You want to write your program in assembler.
2. You add TCP/IP communication to an existing complex program, and it may be difficult or impossible for your program to monitor TCP/IP events through the **GetNextNote** interface.

If your program drives the simulated VMCF interface directly, do not link any of the TCP interface library modules with your program. Consequently, you will not be able to use any of the auxiliary routines such as the Say functions and timer routines. (You must use MVS timer support, or support provided by your existing program.)

For further information on the VMCF interface, refer to *IBM VM/SP System Facilities for Programming*, SC24-5288.

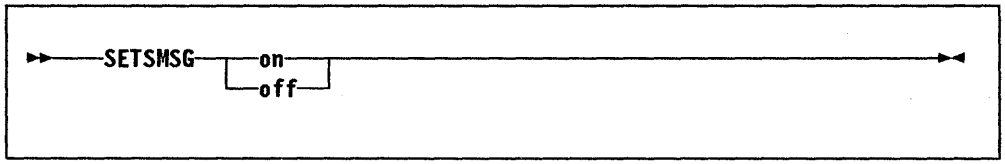
Sending and Receiving Special Messages

The SMSG command is used to send a special message (Smsg) to a user program, a started task, or a batch job. The format of the SMSG command is as follows:



Parameter	Description
user_id	Is the name of the user program, the started task, or the batch job to receive the Smsg.
message_text	Is text of the Smsg to be sent.

When TCP/IP establishes the session with VMCF, it does not enable Smsgs. If your application program accepts and processes Smsgs, you must enable the reception of Smsgs before using the services provided by the application program. By default, reception of Smsgs is disabled. The SETSMSG command is used to enable and disable the reception of Smsgs. The format of the SETSMSG command is as follows:



Parameter	Description
on	Enables the reception of Smsgs.
off	Disables the reception of Smsgs. This is the default.

Locating Program Call Numbers

Before calling any VMCF routines, locate the correct program call (PC) number to invoke the desired function. The PC numbers are contained in VMCF CVT (MVPXVMCV, a member of TCPIP.COMMMAC). This is pointed to by the SSCTSUSE field of the VMCF subsystem entry. You can locate the correct SSCT by following the chain beginning with the JESSSCT field. See Table A-1 for a list of labels and associated tasks.

Table A-1. VMCF CVT Program Call Number Field Names

Function Performed	Field Name
VMCF	PCVMCF
Set System Mask	PCSSM
Read System Mask	PCREADSM
Set Control Register 0	PCSETCR0
Read Control Register 0	PCREDCR0
Store then "or" System Mask	PCSTOSM
Store then "and" System Mask	PCSTNSM

Figure A-1 on page A-3 is a sample routine to locate the VMCF vector table.

```
                        :

         USING PSA,0
         L     R1,FLCCVT              Get address of CVT
         DROP  R0                     Done with PSA
         USING CVT,R1
         L     R1,CVTJESCT            Get address of JES Control Table
         USING JESCT,R1
         ICM   R1,B'1111',JESSSCT     Get address of subsystem
*                                       communications table
         BZ    ERROR1                 VMCF not loaded
         USING SSCT,R1
LOOP     CLC   SSCTSNAM,=C'VMCF'      Correct entry?
         BE    FOUND                  Yes, get CVT address
         ICM   R1,B'1111',SSCTSCTA    Get pointer to next entry
         BNZ   LOOP                   Check next entry
ERROR1   DS    0H
*        VMCF entry could not be found
*        Appropriate error processing goes here
FOUND    L     R1,SSCTSUSE            Get address of VMCF CVT
         USING MVPXVMCV,R1
*        At this point, the VMCF CVT address is in register 1
*        and it may be saved for use by later parts of the program.

                        :
```

Figure A-1. Routine to Locate the VMCF CVT

Program Call Sequences

The VMCF and IUCV facilities provided with TCP/IP for MVS are similar to the VMCF and IUCV facilities of VM. Each uses identical parameter formats; the VMCBLOK for VMCF and the IPARML for IUCV. They differ in the manner in which they are invoked.

On MVS, VMCF and IUCV are invoked via a Program Call (PC) instruction. The correct PC number for each of the following functions is determined from the VMCF CVT. The standard sequence for issuing a program call instruction on MVS is shown in Figure A-2 on page A-4. On return from the PC instruction, register 13 is intact. Many other registers are not. Make sure the save area is based by a register whose contents are restored, or subsequent execution will be unpredictable.

```
        LA    R1,PARMS              Pointer to VMCF parameter list
        ESAR  R14                   Save secondary address
        STM   R14,R12,12(R13)       Save callers registers
        L     R2,PCNUMBER           Get PC Number of routine to call
        PC    0(R2)                 Call routine
        L     R14,12(,R13)          Get saved secondary address
        SSAR  R14                   Restore secondary address
        LM    R2,R12,28(R13)        Restore remainder of registers
```

Figure A-2. Standard Program Call Sequence

When calling VMCF, the parameter list contains one or three parameters. The first parameter is a VMCBLOK, containing the actual request. The second and third parameters are only used for an authorize request. They contain the address of the routine to receive control on a simulated external interrupt and a token to be passed to that routine. The calling sequence is shown in Figure A-3.

```
R1   ->   A(.....)  ->  VMCBLOK
          A(.....)  ->  A(.....)    ->   Interrupt_routine
          A(.....)  ->  Token
```

Figure A-3. VMCF Parameter List

When calling IUCV, the parameter list may contain two, three, or four parameters. The first parameter is the *Iucv_function_code*. This function code is the value that would be in register 0 were an IUCV request made on VM. The second parameter is the *IPARML* for the request. For a declare buffer request, parameter three is the address of the interrupt routine to get control when a simulated external interrupt occurs, and parameter four is a token to be passed to that routine. This is shown in Figure A-4.

```
R1   ->   A(.....)  ->  Iucv_Function_Code
          A(.....)  ->  IPARML
          A(.....)  ->  A(.....)    ->   Interrupt_routine
          A(.....)  ->  Token
```

Figure A-4. IUCV Parameter List

The one exception is for an IUCV query request. For that request, the second parameter is a word that will get the contents of register 0 and the third parameter is a word that will get the contents of register 1 as defined by the IUCV query function. Figure A-5 shows this parameter list.

```
R1   ->   A(.....)  ->  Iucv_Function_Code
          A(.....)  ->  Query_register0
          A(.....)  ->  Query_register1
```

Figure A-5. IUCV Query Parameter List

Parameters Passed to External Interrupt Routines

When the external interrupt routine receives control, it is called with two parameters. The first parameter is the half word interrupt code of the interrupt. The second parameter is the token that was passed at VMCF authorize or IUCV declare buffer time. The calling sequence is the same, so it is possible to use the same routine to process both VMCF and IUCV interrupts. The parameter list is shown in Figure A-6.

```
R1   ->   A(.....)  ->  Interrupt_Code
          A(.....)  ->  Token
```

Figure A-6. External Interrupt Routine Parameter List

The remainder of the data associated with the interrupt is in the buffer that was specified on the VMCF authorize or the IUCV declare buffer. The interrupt routine is executed from an Interruption Request Block (IRB) running in the same task as the one that issued the VMCF authorize or IUCV declare buffer.

Disabling Reception of Interrupts

To disable the reception of VMCF or IUCV interrupts, a pseudo-system mask and pseudo-control register 0 are supported, along with routines for fetching and setting them.

Manipulating the System Mask

The system mask may be manipulated with either **set-system-mask**, **read-system-mask**, **store-then-and-system-mask**, and **store-then-or-system-mask**. **Set-system-mask** and **read-system-mask** each take a one byte parameter. **Set-system-mask** takes the new system mask contents. **Read-system-mask** returns the present system mask contents. **Store-then-and-system-mask** and **store-then-or-system-mask** take two one byte parameters. The first is a byte to contain the old system mask. The second is a byte to AND or OR with the old system mask. These parameter lists are shown in the following figures.

```
R1   ->   A(.....)  ->  Old_System_Mask
```

Figure A-7. Read System Mask Parameter List

```
R1   ->   A(.....)  ->  New_System_Mask
```

Figure A-8. Set System Mask Parameter List

```
R1   ->  A(.....)  ->  Old_System_Mask
         A(.....)  ->  And_Mask
```

Figure A-9. Store Then And System Mask Parameter List

```
R1   ->  A(.....)  ->  Old_System_Mask
         A(.....)  ->  Or_Mask
```

Figure A-10. Store Then Or System Mask Parameter List

Updating Control Register 0

Two routines are provided to update control register 0. These are read control register 0 and set control register 0. These are similar to read system mask and set system mask, and each takes a single fullword parameter of control register 0. The following figures show this parameter list.

```
R1   ->  A(.....)  ->  Old_Control_Register0
```

Figure A-11. Read Control Register 0 Parameter List

```
R1   ->  A(.....)  ->  New_Control_Register0
```

Figure A-12. Set Control Register 0 Parameter List

Data Structures

Your program uses the standard 40-byte VMCF parameter list (parm list) to submit VMCF requests to the TCPIP address space. VMCF interrupts result in the similar 40-byte VMCF interrupt header being stored in your address space. In this appendix, fields in the parameter list and interrupt header are referred to using the field names in the following pseudo-DSECT:

```
V1       DS    X
V2       DS    X
FUNC     DS    H
MSGID    DS    F
JOBNAME  DS    CL8
VADA     DS    A
LENA     DS    F
VADB     DS    A
LENB     DS    F
* User-doubleword field is divided into the following fields:
ANINTEGR DS    F
CONN     DS    H
CALLCODE DS    X
RETCODE  DS    X
```

Figure A-13. VMCF Parameter List Fields

The VMCF functions used to communicate with the TCP/IP for MVS program are:

SEND
SEND/RECV
RECEIVE
REPLY
REJECT

In addition, your program must use the AUTHORIZE and UNAUTHORIZE functions to initialize and halt the use of VMCF.

Figure A-14 on page A-8 shows the equates for the CALLCODE field, used when you are initiating a function from your program.

```
ABORTtcp              EQU   100
BEGINtcpIPservice     EQU   101
CLOSEtcp              EQU   102
CLOSEudp              EQU   103
ENDtcpIPservice       EQU   104
HANDLEnotice          EQU   105
IShostLOCAL           EQU   106
MONITORcommand        EQU   107
MONITORquery          EQU   108
OPENtcp               EQU   110
OPENudp               EQU   111
RECEIVEtcp            EQU   113
NRECEIVEudp           EQU   115
SENDtcp               EQU   118
SENDudp               EQU   119
STATUStcp             EQU   120
FRECEIVEtcp           EQU   121
FSENDtcp              EQU   122
CLOSErawIP            EQU   123
OPENrawIP             EQU   124
RECEIVErawIP          EQU   125
SENDrawIP             EQU   126
PINGreq               EQU   127
```

Figure A-14. Equates for the CALLCODE Field. These equates are used when you are initiating a function from your program.

Figure A-15 shows the equates for the CALLCODE field, used when TCPIP sends a notification to your program.

```
BUFFERspaceAVAILABLE      EQU  10
CONNECTIONstateCHANGED    EQU  11
DATAdelivered             EQU  12
URGENTpending             EQU  15
UDPdatagramDELIVERED      EQU  16
UDPdatagramSPACEavailable EQU  17
RAWIPpacketsDELIVERED     EQU  24
RAWIPspaceAVAILABLE       EQU  25
RESOURCESavailable        EQU  28
UDPresourcesAVAILABLE     EQU  29
PINGresponse              EQU  30
```

Figure A-15. Equates for the CALLCODE Field. These equates are used when TCPIP sends a notification to your program.

Figure A-16 on page A-9 shows the equates used for the connection states.

```
CONNECTIONclosing EQU   0
LISTENING         EQU   1
NONEXISTENT       EQU   2
OPEN              EQU   3
RECEIVINGonly     EQU   4
SENDINGonly       EQU   5
TRYINGtoOPEN      EQU   6
```

Figure A-16. Equates for Connection States

Figure A-17 shows the equates used for notification mask in the HANDLEnotice call.

```
MaskBUFFERspaceAVAILABLE       EQU    X'00000001'
MaskCONNECTIONstateCHANGED     EQU    X'00000002'
MaskDATAdelivered              EQU    X'00000004'
MaskURGENTpending              EQU    X'00000020'
MaskUDPdatagramDELIVERED       EQU    X'00000040'
MaskUDPdatagramSPACEavailable  EQU    X'00000080'
MaskRAWIPpacketsDELIVERED      EQU    X'00004000'
MaskRAWIPspaceAVAILABLE        EQU    X'00008000'
MaskRESOURCESavailable         EQU    X'00040000'
MaskUDPresourcesAVAILABLE      EQU    X'00080000'
MaskPINGresponse               EQU    X'00100000'
```

Figure A-17. Equates for Notification Mask in the HANDLEnotice Call

Figure A-18 shows the assembler format of the Connection Information Record.

```
Connection             DS  H
OpenAttemptTimeout     DS  F
Security               DS  H
Compartment            DS  H
Precedence             DS  X
BytesToRead            DS  F
UnackedBytes           DS  F
ConnectionState        DS  X
LocalSocket.Address    DS  F
LocalSocket.Port       DS  H
ForeignSocket.Address  DS  F
ForeignSocket.Port     DS  H
```

Figure A-18. Assembler Format of the Connection Information Record

Figure A-19 on page A-10 shows Miscellaneous constants.

```
UNSPECIFIEDconnection   -48
DEFAULTsecurity           0
DEFAULTcompartment        0
DEFAULTprecedence         0
UNSPECIFIEDaddress        0
UNSPECIFIEDport           X'FFFF'
ANintegerFLAGrequestERR   X'80000000'
```

Figure A-19. Miscellaneous Constants

General Information

Except when noted, TCPIP responds immediately to your VMCF requests, and you should wait for the response before issuing another TCPIP request.

Use of VMCF Parm List Fields

V1: You can set the VMCPAUTS flag in V1 for the AUTHORIZE function to disallow VMCF communication with any address space other than TCPIP. If you do not set the VMCPAUTS flag in V1, you must check the JOBNAME field when processing each interrupt, to ensure that interrupts from other address spaces are not misinterpreted as coming from TCPIP. V1 must be zero for all functions other than AUTHORIZE.

V2: Must be 0.

MSGID: You must use a unique number for each outstanding transaction. A simple method is to number your transactions consecutively.

JOBNAME: You must set this field to the user ID of the TCPIP address space.

Use of the remaining fields is described individually for each TCP/IP function.

Use of VMCF Interrupt Header Fields

V1: If the interrupt is in response to a transaction initiated by your address space, the VMCMRESP flag is set. If the TCPIP address space responds using the REJECT function, the VMCMRJCT flag is also set. This flag by itself does not usually indicate that the transaction was unsuccessful. Your program should check the RETCODE field, as described for each function.

CALLCODE: If the interrupt is in response to a transaction initiated by your address space (VMCMRESP flag set in V1), CALLCODE is the same as set by your program when it initiated the transaction.

RETCODE: Return codes reported in this field are taken from the same set used by Pascal programs (see "Pascal Return Codes" on page 2-55). Further information is given in the description of each function.

TCP/UDP/IP Initialization and Termination Procedures

Begin TCP/IP Service

Your program must perform this function after doing a VMCF AUTHORIZE, but before performing any other TCP/IP function. It informs TCPIP that your address space will use TCPIP services. An "End TCP/IP Service" function is logically performed first, in the case where your address space already has TCPIP resources allocated.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CALLCODE: BEGINtcpIPservice
```

The TCPIP address space responds using the VMCF REJECT function. The VMCF interrupt header, stored in your address space by the response interrupt, contains a return code in the RETCODE field. The return code can be any of those listed for the **BeginTcpIp** Pascal procedure (page 2-20).

Specifying the Notifications to Receive

Your program performs this function to specify the notification types to be received from TCPIP. The VADB field in the VMCF parm list contains a notification mask, with 1 bit set for each notification you want to handle. The bit to be set for each notification type is shown in Figure A-17 on page A-9.

Each HANDLEnotice call must specify all the notification types to be handled. Notification types specified in previous HANDLEnotice calls are not remembered.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     Note mask
LENB:     0
CALLCODE: HANDLEnotice
```

The TCPIP address space responds using the VMCF REJECT function. The VMCF interrupt header, stored in your address space by the response interrupt, contains a return code in the RETCODE field. The return code can be any of those listed for the **Handle** Pascal procedure (page 2-18).

End TCP/IP Service

Your program should perform this function when it has finished using TCPIP services. All existing TCP connections are reset (aborted), all existing UDP port opens are canceled, and all IP protocols are released.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CALLCODE: ENDtcpIPservice
```

The TCPIP address space responds using the VMCF REJECT function. The VMCF interrupt header, stored in your address space by the response interrupt, indicates a return code of OK in the RETCODE field.

Open TCP Connection

Use this function to initiate a TCP connection. Note that your program sends a Connection Information Record to TCPIP. Figure A-18 on page A-9 gives the assembler format of the record. Figure A-19 on page A-10 gives the equates for the assorted constants used to set up the record. See the section on the Pascal **TcpOpen** procedure (page 2-23) for further information on the usage of the fields of the Connection Information Record.

```
FUNC:     SEND/RECV
VADA:     Address of Connection Information Record initialized by
          your program
LENA:     Length of Connection Information Record
VADB:     Address of Connection Information Record to be filled in
          with TCPIP reply
LENB:     Length of Connection Information Record
CONN:     UNSPECIFIEDconnection
CALLCODE: OPENtcp
```

If the open attempt cannot be initiated, the TCPIP address space responds using the VMCF REJECT function. The VMCF interrupt header, which is stored in your address space by the response interrupt, contains a return code in the RETCODE field. The return code can be any of those listed for the **TcpOpen** Pascal procedure.

If the open attempt is not immediately rejected, the TCPIP address space will use the VMCF RECEIVE function to receive the Connection Information Record describing the connection to be opened. If the connection then cannot be initiated, TCPIP responds using the VMCF REJECT function. The RETCODE field in the VMCF interrupt header will be set as described in the previous paragraph.

If the open was successfully initiated, the TCPIP address space responds using the VMCF REPLY function to send back the updated Connection Information Record. The Connection field contains the connection number of the new connection. In

the VMCF interrupt header, stored in your address space by the response interrupt, the RETCODE field will indicate OK, and the CONN field also contains the connection number of the new connection. The connection is not yet open; your program receives notification(s) during the opening sequence. See the section on the Pascal NotificationInfoType ("Notification Record" on page 2-5) for more information about this notification. Also refer to "Notifications" on page A-22.

Send TCP Data

Use either SENDtcp or FSENDtcp to send data on a TCP connection. Refer to the **TcpFSend** and **TcpSend** Pascal procedures (page 2-25) for the advantages and disadvantages of using each function, and for general information about sending TCP data.

```
FUNC:     SEND
VADA:     Address of data
LENA:     Length of data
VADB:     1 if push desired, else 0
LENB:     1 if urgent data, else 0
CONN:     Connection number from open
CALLCODE: SENDtcp or FSENDtcp
```

If TCPIP can successfully queue the data for sending, it responds with the VMCF RECEIVE function. The VMCF interrupt header, stored in your address space by the response interrupt, indicates a RETCODE of OK

If TCPIP cannot queue the data for sending, it responds with the VMCF REJECT function. In the VMCF interrupt header, stored in your address space by the response interrupt, the RETCODE field indicates the type of error. The return code can be any of those listed for the **TcpSend** Pascal procedure.

TcpFSend is the same as SENDtcp. If TCPIP cannot accept the data due to a buffer shortage, it does not respond immediately. Instead, it waits until space is available, and then uses VMCF RECEIVE to receive the data. While it is waiting, if the connection is reset, it responds with VMCF REJECT, with a return code as described above. In summary, TCPIP may not respond immediately to FSENDtcp, and its response after waiting may indicate either success or failure. If TCPIP responds with REJECT, your program can check the ANintegerFLAGrequestERR bit in ANINTEGR to determine if the request was rejected during initial or retry processing (bit on) or because of connection closing (bit off).

Your program need not wait for a response from FSENDtcp. It can issue functions involving other connections, before receiving a response from FSENDtcp.

There is a limit of 50 outstanding VMCF transactions per address space; your program can therefore have FSENDtcp functions outstanding on only 50 connections at a time. If your application needs more outstanding sends, use the SENDtcp function.

Receive TCP Data with the FRECEIVEtcp Function

Use the FRECEIVEtcp function to tell TCPIP that you are ready to receive data on a specified TCP connection. Note that TCPIP does not respond until data is received or the connection is closed. Consequently, each outstanding FRECEIVEtcp function results in an outstanding VMCF transaction. There is a limit of 50 outstanding VMCF transactions per address space; you can therefore have FRECEIVEtcp functions outstanding on only 50 connections at one time. If your application needs more outstanding receives, use the RECEIVEtcp function.

Your program need not wait for a response from FRECEIVEtcp. It can issue functions involving other connections, before receiving a response from FRECEIVEtcp.

Refer to the **TcpFReceive** Pascal procedure (page 2-28) for general information about receiving TCP data.

```
FUNC:     SEND/RECV
VADA:     0
LENA:     1
VADB:     Address of buffer to receive data
LENB:     Length of buffer to receive data
CONN:     Connection number from open
CALLCODE: FRECEIVEtcp
```

If TCPIP accepts the request, your program will receive no response until TCPIP is ready to deliver data to your buffer, or until the request is canceled, because the connection has closed without delivering data.

When TCPIP is ready to deliver data for this connection, it issues a VMCF REPLY function. Significant fields in the VMCF interrupt header, stored in your address space by the response interrupt, are:

LENB: The residual count. Subtract this from the size of your buffer (LENB value in parameter list) to determine the number of bytes actually delivered.

ANINTEGR: The high-order byte is nonzero if data was pushed; otherwise, it is zero. The low-order three bytes are interpreted as a 24-bit integer, indicating the offset of the byte following the last byte of urgent data, measured from the first byte of data delivered to your buffer. If it is zero or a negative number, then there is no urgent data pending.

CONN: The connection number.

RETCODE: OK

If TCPIP responds with the VMCF REJECT function (VMCFRJCT flag set in the VMCF interrupt header), either:

1. It did not accept the request, in which case the ANintegerFLAGrequestERR bit in ANINTEGR is on.

2. It accepted the request initially, but the connection closed before data was delivered. ANintegerFLAGrequestERR bit in ANINTEGR is off. In this case, the RETCODE field indicates one of the reason codes listed for CONNECTIONstateCHANGED with the NewState field set to NONEXISTENT. See Usage Note 3 on page 2-8 for further information. Note that your program does not have to take any special action in this case, because it receives one or more CONNECTIONstateCHANGED notifications indicating that the connection is closing.

Receive TCP Data with the RECEIVEtcp Function

Use the RECEIVEtcp function to tell TCPIP that you are ready to receive data on a specified TCP connection. Unlike FRECEIVEtcp, TCPIP responds immediately to RECEIVEtcp. You can have more than 50 receives pending using RECEIVEtcp without exceeding the limit of 50 outstanding VMCF transactions.

Refer to the **TcpReceive** Pascal procedure for (page 2-28) general information about receiving TCP data.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     Length of buffer to receive data
CONN:     Connection number from open
CALLCODE: RECEIVEtcp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer, stored in your address space, indicates whether the request was successful. If yes (RETCODE of OK), you will later receive a DATAdelivered notification delivering data to your buffer. If not, then the return code is one of those listed for the **TcpReceive** Pascal procedure.

Close a TCP Connection

Use the CLOSEtcp function to initiate the closing of a TCP connection. Refer to the **TcpClose** Pascal procedure (page 2-31) for general information about the close function.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CONN:     Connection number from open
CALLCODE: CLOSEtcp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer, stored in your address space, contains the return code. It is one of those listed for the **TcpClose** Pascal procedure.

Abort a TCP Connection

Use the ABORTtcp function to abort a TCP connection. Refer to the **TcpAbort** Pascal procedure (page 2-32) for general information about the abort function.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CONN:     Connection number from open
CALLCODE: ABORTtcp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer, stored in your address space, contains the return code. It is one of those listed for the **TcpAbort** Pascal procedure.

Obtain Current Status of TCP Connection

Use the STATUStcp function to obtain a Connection Information Record giving the current status of a TCP connection. Refer to Figure A-18 on page A-9 for the assembler format of the Connection Information Record. Refer to the **TcpStatus** Pascal procedure (page 2-33) for general information about the status function.

```
FUNC:     SEND/RECV
VADA:     0
LENA:     1
VADB:     Address of Connection Information Record to fill in
LENB:     Length of Connection Information Record to fill in
CONN:     Connection number from open
CALLCODE: STATUStcp
```

TCPIP responds with the VMCF REPLY function, filling in the record whose address you supplied in LENB. The record is valid only if the return code, which is in the RETCODE field of the VMCF interrupt header, returns OK. Otherwise, the return code is one of those listed for the **TcpStatus** Pascal procedure.

Close a UDP Port

Use the CLOSEudp function to close a UDP port. Refer to the **UdpClose** Pascal procedure (page 2-41) for general information about the close UDP function.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CONN:     Connection number
CALLCODE: CLOSEudp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field in the VMCF interrupt header can be any of the return codes listed for the **UdpClose** Pascal procedure. If the return code is OK, and your program specified UNSPECIFIEDport as the port number, the actual port number assigned is encoded in the CONN field of the interrupt header. Add 32 768 to the CONN field, using unsigned arithmetic, to compute the port number.

Open a UDP Port

Use the OPENudp function to open a UDP port. Refer to the **UdpOpen** Pascal procedure (page 2-38) for general information about the UDP close function.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     Local port number or UNSPECIFIEDport
LENB:     Local address
CONN:     Connection number: An arbitrary number, which your program
          will use in subsequent actions involving this port.
CALLCODE: OPENudp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field in the VMCF interrupt header may be any of the return codes listed for the **UdpOpen** Pascal procedure.

Send UDP Data

Use the SENDudp function to send a UDP datagram. Refer to the **UdpSend** Pascal procedure (page 2-39) for general information about the UDP send function.

```
FUNC:     SEND
VADA:     Address of datagram data
LENA:     Length of datagram data (up to 8492 bytes)
VADB:     Destination port number
LENB:     Destination address
CONN:     Connection number
CALLCODE: SENDudp
```

If TCPIP can send the datagram, it responds with the VMCF RECEIVE function. The RETCODE field in the VMCF interrupt header, stored in your address space by the response interrupt, contains OK. If TCPIP cannot send the datagram, it responds with the VMCF REJECT function. The RETCODE field contains one of the return codes listed for the **UdpSend** Pascal procedure.

Receive UDP Data

Use the NRECEIVEudp function to tell TCPIP that your program is ready to receive a UDP datagram on a particular port. TCPIP responds immediately to inform you whether it accepted the request. If it has, your program receives a UDPdatagramDELIVERED notification when a datagram arrives. Refer to the **UdpNReceive** Pascal procedure (page 2-40) for further information about receiving UDP datagrams.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     0
LENB:     Size of your buffer to receive datagram
CONN:     Connection number
CALLCODE: NRECEIVEudp
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt header, which is stored in your address space, contains one of the return codes listed for the **UdpNReceive** Pascal procedure.

Determine Whether an Address is Local

Use the IShostLOCAL function to determine whether a given internet address is one of your host's local addresses. Refer to the **IsLocalAddress** Pascal procedure (page 2-50) for general information about this procedure.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     Internet address to be tested
LENB:     0
CONN:     UNSPECIFIEDconnection
CALLCODE: IShostLOCAL
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt header contains the return code, as described in the **IsLocalAddress** Pascal procedure section.

Instruct TCPIP to Obey a File of Commands

Use the MONITORcommand function to instruct TCPIP to obey a file of commands. Refer to the **MonCommand** Pascal procedure (page 2-35) for more information. Refer to the *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* book for more information on the OBEYFILE command, which uses the **MonCommand** procedure.

```
Owner            DS   CL8
DatasetPassword  DS   CL8
FullDatasetName  DS   CL44
MemberName       DS   CL8
DDName           DS   CL8
```

Figure A-20. Assembler Format of the SpecOfFileType Record for MVS

```
FUNC:     SEND/RECV
VADA:     Address of SpecOfFile record
LENA:     Length of SpecOfFile record
VADB:     0
LENB:     0
CONN:     UNSPECIFIEDconnection
CALLCODE: MONITORcommand
```

If TCPIP cannot process the request, it responds immediately with the VMCF REJECT function. Otherwise, it uses the VMCF RECEIVE function to receive the SpecOfFile record provided by your program. It then attempts to process the file, and uses the VMCF REJECT function to report the return code. In either case, the return code is one of those specified for the **MonCommand** Pascal procedure.

Obtain Status Information from TCPIP

Use the MONITORquery function to obtain status information from the TCPIP address space or to request that it perform certain functions. Refer to the **MonQuery** Pascal procedure (page 2-36) for general information. Assembler formats of constants and records used with MONITORquery are:

```
COMMANDcpCMD           EQU   6
COMMANDdropCONNECTION  EQU   8
QUERYhomeONLY          EQU   9
```

```
QueryType    DS    X
* For QueryType = QUERYhomeONLY: No other fields
* For QueryType = COMMANDcpCMD:
CpCmd        DS    H      Length of command
             DS    100C   Command
* For QueryType = COMMANDdropCONNECTION:
             ORG   CpCmd
Connection   DS    H
```

Figure A-21. MonQueryRecordType

The Pascal type HomeOnlyListType is an array of 64 InternetAddressType elements. InternetAddressType is a fullword.

```
FUNC:     SEND/RECV
VADA:     Address of MonQueryRecord describing request
LENA:     Length of MonQueryRecord
VADB:     Address of return buffer
LENB:     Length of return buffer
CONN:     UNSPECIFIEDconnection
CALLCODE: MONITORquery
```

If TCPIP cannot process the request, it responds immediately with the VMCF REJECT function. Otherwise, it uses the VMCF RECEIVE function to receive the MonQueryRecord describing your request, followed by either a VMCF REPLY to send the response to your return buffer (not applicable to COMMANDdropCONNECTION), or a VMCF REJECT to send a return code but no return data. Refer to the **MonQuery** Pascal procedure for information on the return codes and the data returned (if any).

Send an ICMP Echo Request

Use the PINGreq function to send an ICMP echo request (Ping request) to a specified host and wait a specified time for a response. Refer to the **PingRequest** Pascal procedure (page 2-34) for general information.

```
FUNC:     SEND
VADA:     0
LENA:     1
VADB:     Internet address of foreign host
LENB:     Length of ping packet
ANINTEGR: Timeout
CALLCODE: PINGreq
```

TCPIP uses the VMCF REJECT function to respond to the request. The RETCODE field of the VMCF interrupt header, which is stored in your address space by the response interrupt, contains one of the return codes listed for the **PingRequest** Pascal procedure. If the return code is OK, your program receives a PINGresponse notification later.

Tell TCPIP That Your Program Will Use a Particular IP Protocol

Use the OPENrawip function to tell TCPIP that your program is ready to send and receive packets of a specified IP protocol. Refer to the **RawIpOpen** Pascal procedure (page 2-42) for general information.

```
FUNC:     SEND/RECV
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CONN:     Protocol number
CALLCODE: OPENrawip
```

TCPIP uses the VMCF REJECT function to respond to the request. The RETCODE field of the VMCF interrupt header, which is stored in your address space by the response interrupt, contains one of the return codes listed for the **RawIpOpen** Pascal procedure.

Tell TCPIP That Your Program Will No Longer Use a Particular IP Protocol

Use the CLOSErawip function to tell TCPIP that your program is ready to cease sending and receiving packets of a specified IP protocol. Refer to the **RawIpClose** Pascal procedure (page 2-45) for information.

```
FUNC:     SEND/RECV
VADA:     0
LENA:     1
VADB:     0
LENB:     0
CONN:     Protocol number
CALLCODE: CLOSErawip
```

TCPIP uses the VMCF REJECT function to respond to the request. The RETCODE field of the VMCF interrupt header, stored in your address space by the response interrupt, contains one of the return codes listed for the **RawIpClose** Pascal procedure.

Send Raw IP Packets

Use the SENDrawip function to send raw IP packets of a given protocol. Refer to the **RawIpSend** Pascal procedure (page 2-44) for general information.

```
FUNC:     SEND/RECV
VADA:     Address of buffer containing packets to send
LENA:     Length of buffer
VADB:     0
LENB:     0
CONN:     (Number of packets shifted left 8 bits) + protocol number
CALLCODE: SENDrawip
```

If TCPIP immediately determines that the request cannot be fulfilled, it responds with the VMCF REJECT function. Otherwise, it uses the VMCF RECEIVE function to receive your data, followed by VMCF REJECT. The RETCODE field of the VMCF

interrupt header, which is stored in your address space by the response interrupt, contains one of the return codes listed for the **RawIpSend** Pascal procedure.

Receive Raw IP Packets of a Given Protocol

Use the RECEIVErawip function to tell TCPIP that your program is ready to receive raw IP packets of a given protocol. Your program receives a RAWIPpacketsDELIVERED notification when a packet arrives. Refer to the **RawIpReceive** Pascal procedure (page 2-43) and the section on the Pascal NotificationInfoType (page 2-5) for general information.

```
FUNC:     SEND/RECV
VADA:     0
LENA:     1
VADB:     0
LENB:     Length of your buffer
CONN:     Protocol number
CALLCODE: RECEIVErawip
```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt header, stored in your address space by the response interrupt, contains one of the return codes listed for the **RawIpReceive Pascal** procedure.

Notifications

Described below are the VMCF interrupt headers that are stored in your address space for the various types of notifications. Also indicated is the action that your program should take.

Refer to the Pascal NotificationInfoType record ("Notification Record" on page 2-5) for general information on the various notification types.

The VMCF transaction for a notification must be completed before TCPIP sends your program another notification. Your program must therefore be careful to take the VMCF actions specified below, or TCPIP cannot communicate further with your program.

BUFFERspaceAVAILABLE

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
VADB:     Space available to send on this connection, in bytes.
          Currently always 8192
CONN:     Connection number
CALLCODE: BUFFERspaceAVAILABLE
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-22. BUFFERspaceAVAILABLE Notification

CONNECTIONstateCHANGED

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
VADB:     New connection state (see Figure A-16)
LENB:     Reason for state change, if new state is NONEXISTENT
CONN:     Connection number
CALLCODE: CONNECTIONstateCHANGED
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-23. CONNECTIONstateCHANGED Notification

DATAdelivered

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
LENA:     Length of data being delivered
VADB:     Non-zero if data was pushed, else zero.
LENB:     The offset of the byte following the last byte of urgent
          data, measured from the first byte of data delivered to your
          buffer.  If zero or negative then there is no urgent data
          pending.
CONN:     Connection number
CALLCODE: DATAdelivered
RETCODE:  OK

Your program should issue the VMCF RECEIVE function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     RECEIVE
VADA:     Address of your buffer to receive data.  Buffer should be
          at least as long as indicated by LENA.  LENA will be no
          larger than the buffer length you specified using the
          RECEIVEtcp function.
```

Figure A-24. DATAdelivered Notification

PINGresponse

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
VADB:     High order word of elapsed time, in TOD clock format
          Valid only if ANINTEGR is zero
LENB:     Low order word of elapsed time, in TOD clock format
          Valid only if ANINTEGR is zero
ANINTEGR: Return code from ping operation
CALLCODE: PINGresponse
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-25. PINGresponse Notification

RAWIPpacketsDELIVERED

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
ANINTEGR: Total length of datagram being delivered (including IP header)
LENA:     Length of data (including IP header) that TCPIP will
          deliver to you.
CONN:     Low-order byte is protocol number, 3 high order bytes
          is number of packets, currently always 1.
CALLCODE: RAWIPpacketsDELIVERED
RETCODE:  OK

Your program should issue the VMCF RECEIVE function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     RECEIVE
VADA:     Address of your buffer to receive data.  Buffer should be
          at least as long as indicated by LENA.
```

Figure A-26. RAWIPpacketsDELIVERED Notification

RAWIPspaceAVAILABLE

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
LENB:     Space available.  Always equals maximum IP datagram size.
CONN:     Protocol number
CALLCODE: RAWIPspaceAVAILABLE
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-27. RAWIPspaceAVAILABLE Notification

RESOURCESavailable

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
CALLCODE: RESOURCESavailable
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-28. RESOURCESavailable Notification

UDPdatagramDELIVERED

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
LENA:     Length of data being delivered.
VADB:     Source port
LENB:     Source address
ANINTEGR: Length of entire datagram excluding UDP header.  If larger
          than LENA then the
          datagram was too large to fit into the buffer size specified
          in NRECEIVEudp call, and has been truncated.
CONN:     Connection number
CALLCODE: UDPdatagramDELIVERED
RETCODE:  OK

Your program should issue the VMCF RECEIVE function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     RECEIVE
VADA:     Address of your buffer to receive data.  Buffer should be
          at least as long as indicated by LENA.
```

Figure A-29. UDPdatagramDELIVERED Notification

UDPdatagramSPACEavailable

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
CONN:     Connection number
CALLCODE: UDPdatagramSPACEavailable
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-30. UDPdatagramSPACEavailable Notification

UDPresourcesAVAILABLE

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
CALLCODE: UDPresourcesAVAILABLE
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-31. UDPresourcesAVAILABLE Notification

URGENTpending

```
FUNC:     SEND
JOBNAME:  Name of the TCPIP address space
VADB:     Number of bytes of queued incoming data not yet received
          by your program.
LENB:     Subtract 1 from LENB to get the offset of the byte following
          the last byte of urgent data, measured from the first byte not
          yet received by your program.  If this quantity is zero or
          negative then there is no urgent data pending.
CONN:     Connection number
CALLCODE: URGENTpending
RETCODE:  OK

Your program should issue the VMCF REJECT function, with VMCF parm
list copied from the interrupt header, with the following fields changed:

V1:       0
V2:       0
FUNC:     REJECT
```

Figure A-32. URGENTpending Notification

Appendix B. Interface to SMTP Address Space

This appendix is intended for the reader interested in details of the interfaces to the SMTP address space. It may also be of interest to implementers of electronic mail preparation programs that communicate with the IBM TCP/IP for MVS implementation of the Simple Mail Transfer Protocol (SMTP).

There are two interfaces to the SMTP address space:

1. Interface from the TCP/IP network. SMTP commands and replies can be sent and received interactively over a TCP network connection. Mail from TCP network sites destined for local MVS users (or users on a Network Job Entry (NJE) network attached to the local MVS system) arrives over this interface. All commands and data received and transmitted through this interface use ASCII characters.
2. Interface from the Job Entry System (JES) spool, including any connected NJE nodes. SMTP commands can be written into a SYSOUT data set, with an external writer name of the SMTP address space. SMTP processes each of the commands in the data set in sequence, exactly as if they had been transmitted over a TCP connection. This is how mail is sent from local MVS users to recipients on the TCP network. Batch SMTP data sets must contain commands and data in EBCDIC characters.

Format of Batch SMTP Command Data Sets

Data sets containing SMTP commands can be written to the JES spool as SYSOUT data sets. These SYSOUT data sets contain either punch or netdata records. Data sets originate from users on the same system as the SMTP address space or from users on any system connected to the host system through an NJE network.

Files containing SMTP commands can have either fixed-length or variable-length records. All trailing blanks on each record are truncated. Records longer than 8 192 characters are also truncated.

The SMTPNOTE command packaged with the TCP/IP for MVS program converts mail destined for TCP network recipients into batch SMTP format, and spools the batch SMTP file to the SMTP address space.

SMTP Responses

SMTP commands arrive over a TCP connection or over a batch SMTP connection. With either connection, a response to each command is generated. All responses are prefixed with a three-digit number. A simple interpretation of the response is possible by inspecting the first digit of the response code:

First Digit	Description
1	Not used for SMTP.
2	Positive response; command accepted.
3	Positive response; now send the data associated with the command.
4	Temporary negative response. Try again later.
5	Permanent negative response. The command has been rejected.

If SMTP commands arrive over a TCP connection, all responses (positive or negative) are returned over that TCP connection. If SMTP commands arrive over a batch SMTP connection, all responses are written to a batch SMTP response file.

If verbose mode is enabled for a batch SMTP connection, SMTP returns the batch SMTP response file to the origination point of the spool file. The origination point is determined from information in the NETDATA header if the file arrives in netdata format, or the MAIL FROM command if the file arrives in punch format. If the batch SMTP connection is not in verbose mode, the batch SMTP response file is not returned to the point of origin.

If an error occurs during the processing of commands over a batch SMTP connection, such as reception of a negative response (with first digit of 4 or 5), an error report is mailed back to the sender of the mail. The sender is determined from the last MAIL FROM command received that was both syntactically and semantically valid. If the sender cannot be determined from a MAIL FROM command, the sender is assumed to be the origination point of the batch SMTP command file. The error report mailed to the sender includes the batch SMTP response file and the text of the undeliverable mail.

All SMTP commands and data that arrive over TCP/IP or batch SMTP connections are subject to the following restrictions. The default values are as follows:

Table B-1. Default Values of SMTP Commands and Data	
Maximum length of command line	512 characters
Maximum length of data line	1 024 characters
Maximum length of path	256 characters
Maximum length of domain name	256 characters
Maximum length of user name	256 characters

SMTP Path-Address Handling

The syntax of a *path_address* used in both MAIL FROM and RCPT TO commands is defined in RFC 821 (refer to "MAIL FROM" on page B-5 and "RCPT TO" on page B-6). Path addresses are simplified and rewritten according to the following rules:

1. If the local part of a mailbox name includes a percent sign (%) and the domain of the mailbox is the host system, SMTP rewrites the address by treating that

portion of the local part to the right of the % as the real destination host. For example, the path address

```
John%yourhost@ourhost.our.edu
```

is rewritten by SMTP running at ourhost.our.edu as:

```
John@yourhost
```

2. Path addresses with source routes are accepted and rewritten to remove the domain name of the host system. For example, the path address

```
@ourhost.our.edu,@next.host.edu:John@yourhost
```

is rewritten by SMTP running at ourhost.our.edu as:

```
@next.host.edu:John@yourhost
```

SMTP Commands

This section describes the SMTP commands that are recognized by the TCP/IP for MVS implementation of SMTP.

DATA

The DATA command is used after a HELO command, a MAIL FROM command, and at least one RCPT TO command have been accepted. Following receipt of a reply message (354), the sender must transmit the body of the mail.

The body of the mail is terminated by a record that contains a single period (.). When receiving mail over a TCP connection, this ASCII period should be followed by an ASCII CRLF character. If any record in the body of the mail begins with a period, the sending SMTP program must convert the period into a pair of periods (..). When the receiving SMTP encounters a record that begins with two periods in the body of the mail, it discards the leading period. This convention permits the body of mail to contain records that would otherwise be interpreted as signaling the end of the body of mail. These rules must be followed over both TCP and batch SMTP connections.

The SMTPNOTE command included in the TCP/IP for MVS program performs this period doubling on all mail spooled to SMTP. If the body of the mail in a batch SMTP command is not explicitly terminated by a record with a single period, SMTP adds one.

After a period has been received, the SMTP connection is reset to the initial state (that is, before any sender or recipients have been specified). Additional MAIL FROM, RCPT TO, DATA, and other commands can now be sent. If there is no more mail to be delivered, terminate the connection with the QUIT command. If a QUIT command is not found at the end of a batch SMTP command file, it is implied.

If SMTP runs out of local mail storage space, it sends a reply with a 451 code. If the length of the body of the mail exceeds the **MaxMailBytes** parameter (defined in the SMTP configuration data sets to be 512KB), SMTP sends a reply with a 552 code. For more information on **MaxMailBytes**, refer to *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance.*

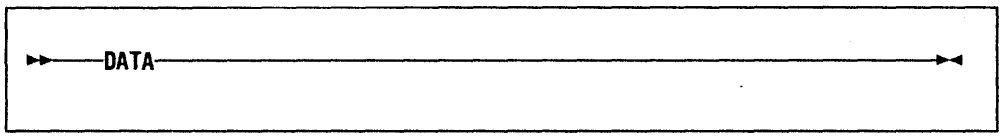
When mail arrives over a batch SMTP connection from an NJE network host, the RFC822 header fields are modified to reflect that mail has been transferred to the TCP network recipients through a gateway. The mail header records: *From*, *Sender*, *ReSent-From*, *ReSent-Sender*, *Reply-To*, *ReSent-Reply-To*, and *Return-Path* may contain NJE network addresses. For the TCP network recipient to be able to respond to this mail, the NJE network addresses must have the domain name of the gateway added to the path.

For example, mail in batch SMTP format spooled to the SMTP address space running at OURHOST.OUR.EDU, with SMTP runtime options GATEWAY and NJEDOMAIN BITNET, and containing the mail header record:

```
Sender: JOHN@YOURHOST
```

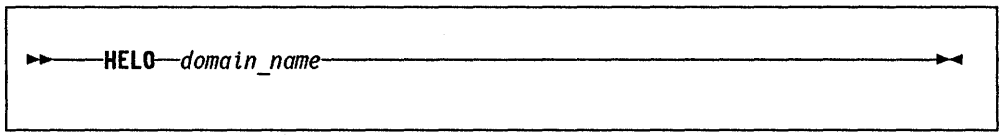
is rewritten as

```
From: JOHN%YOURHOST.BITNET@ourhost.our.edu
```



HELO

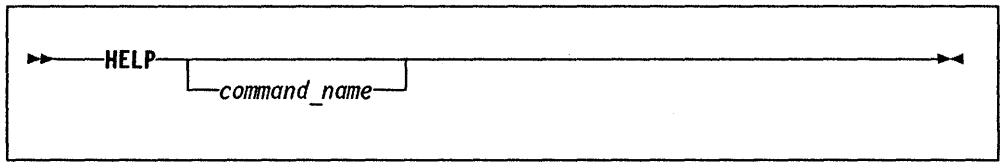
The HELO command must be sent once before a MAIL FROM command to identify the *domain_name* of the sending host to SMTP.



domain_name
Is the domain name of the sending host.

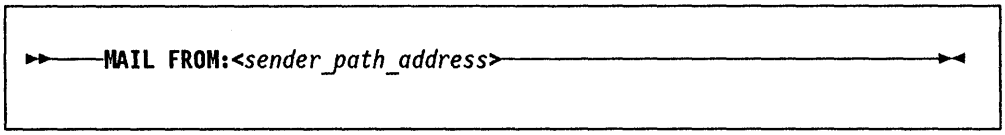
HELP

The HELP command returns a multiline reply with some very general information about the SMTP commands. You can issue the HELP command with the parameter *command_name* to request information about a specific SMTP commands.



MAIL FROM

The MAIL FROM command must be used once after a HELO command to specify the sender of the mail.

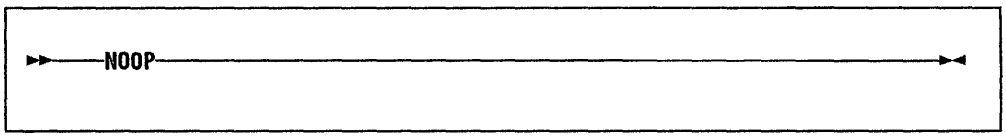


sender_path_address
Is the full path address of the sender of the mail.

For information on how to specify path address, refer to the "SMTP Path-Address Handling" on page B-2.

NOOP

This command returns an OK return code when SMTP is responding.



QUEU

This command returns a multiline reply with detailed information about mail queued for delivery. Information is returned about mail in the spool queue, mail currently active, mail currently queued, mail in the retry queue, and mail in the undeliverable queue. This command can be issued from a batch SMTP session, in which case it must be preceded by a VERB ON command. If the batch SMTP file is in punch format, the QUEU command must be preceded by appropriate HELO and MAIL FROM commands.

Spool Queue Contains mail that is destined for recipients on the local MVS system, or for recipients on an NJE system attached to the local MVS system. This queue is generally empty because SMTP can deliver this mail quickly by spooling it to the local recipient or NJE.

Active Indicates that if SMTP is currently transmitting to a TCP network destination, all the mail queued for that destination is shown to be Active.

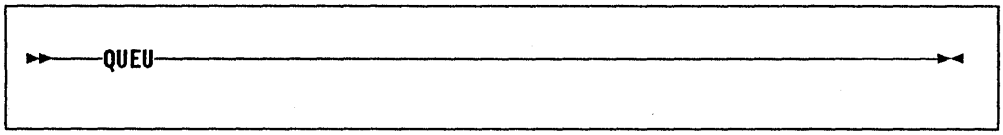
Queued All mail that arrived over a batch SMTP connection (and mail from TCP connections that is to be forwarded to another TCP network destination through source routing) is placed on the queued list. As soon as SMTP receives resources from the TCPIP address space, mail that is queued is "promoted" to be Active.

Retry Queue Contains mail placed here after SMTP has tried to transmit mail to each of the TCP network hosts, but was unable to either open a connection or complete delivery over the connection. After a

number of minutes specified by RETRYINT, mail is promoted from the Retry Queue to the Queued state. For more information about the RETRYINT variable, refer to *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance.*

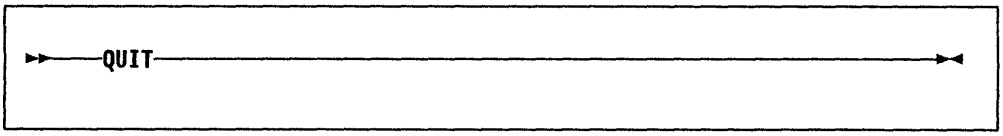
Undeliverable Queue

Contains mail placed here if SMTP cannot deliver mail to a local MVS recipient (or a recipient on the NJE network attached to the local MVS system) because spool space on the local MVS system is full. After spool space has become available and SMTP has been restarted, delivery is attempted again.



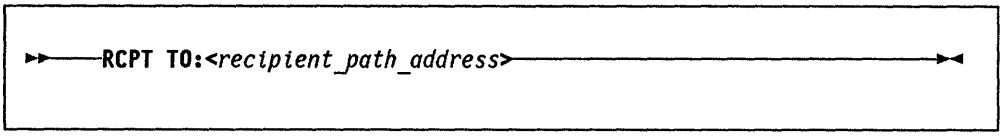
QUIT

This command terminates an SMTP session.



RCPT TO

You can use the RCPT TO command any number of times to specify the recipients of the mail. You must use the RCPT TO command after a MAIL FROM command. If the recipient's host is unknown to the local host system, the RCPT TO command receives a negative reply. If a name server is used for domain name resolution, MX records are used to resolve the recipient's internet address.

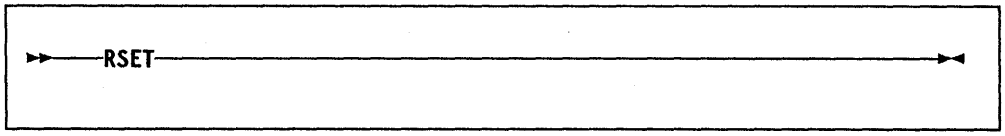


recipient_path_address

Is the full path address of a recipient of the mail.

RSET

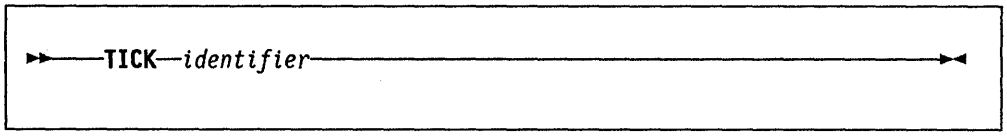
This command resets the SMTP connection to the initial state. The sender and recipient buffers are erased, and the connection is ready to begin a new mail transaction.



TICK

Use this command in conjunction with the VERB ON command to insert an identifier into the batch SMTP response file. This command may be useful for some mail systems that keep track of batch SMTP response files.

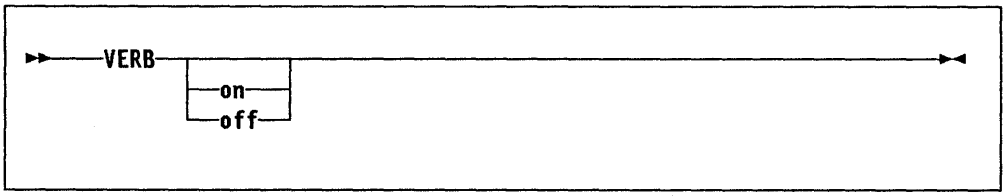
The TICK command has no effect when it is issued over a TCP connection to SMTP.



VERB

Use this command to enable or disable verbose mode. When verbose mode is enabled, batch SMTP commands and associated replies are recorded in the batch SMTP response file. If verbose mode is disabled (the default), only the replies (not the commands) are recorded in the batch SMTP response file. When verbose mode is enabled, the batch SMTP response file is sent back to the origination point of the batch SMTP command file.

When the VERB command is issued over a TCP connection to SMTP, there is no effect.



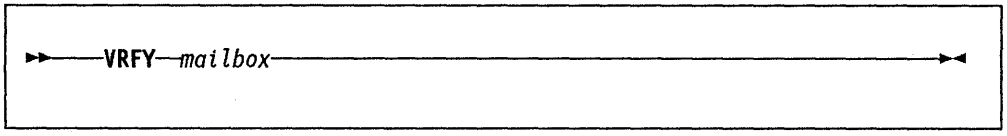
VRFY

Use this command to verify the existence of a given mailbox on the local host. Specify the user ID of a user on the local host to see if that user ID exists.

SMTP commands and responses issued to TCP host abc.com:

```
VRFY sleepy
    250 sleepy@abc.com
```

The EXPN command is implemented as VRFY.



mailbox

Is the user ID of the mailbox.

Unimplemented Commands

The SEND, SOML, SAML, and TURN SMTP commands are not implemented.

Batch SMTP Example

Listed below is an example of sending mail from an NJE network host to two TCP network recipients. The NJE network is BITNET, and the NJE host is named YOURHOST. This batch SMTP file is spooled to SMTP at OURHOST which is running with the run-time arguments GATEWAY and NJEDOMAIN BITNET. OURHOST is a BITNET host and is also connected through a TCP network to the hosts *rsch.our.edu* and *ai.our.edu.*

```
HELO YOURHOST
MAIL FROM:<SNEEZY@YOURHOST>
RCPT TO:<msgs@rsch.our.edu>
RCPT TO:<sneezy@ai.our.edu>
DATA
Date: Fri, 28 Jul 89 04:23:35 CST
From: Sneezy Dwarf<SNEEZY@YOURHOST>
To:   <msgs@rsch.your.edu>
Cc:   <sneezy@ai.your.edu>
Subject: It's a Boy!

Snow White-Dwarf and I are pleased to announce the arrival of:

          Sneezy Edward Dwarf Jr.

born on Tuesday, July 25, 1989 at 10:47 pm.  He weighs 6lbs, 15.5
ounces and measures 18.5 inches from toe to honker.
Mother and Father are doing fine.

QUIT
```

SMTP rewrites the *From:* line to reflect that the mail has been transferred from an NJE network (in this case, BITNET) to a TCP network. The TCP network recipients receive:

```
From: Sam Sneezy <SNEEZY%YOURHOST.BITNET@ourhost.our.edu>
```

Appendix C. Assembler Calls for the Pascal API

The content of this appendix is Internal Product Information and must not be used as programming interface information.

RTcpExtRupt

This procedure is a version of **TcpExtRupt** Pascal procedure and is callable directly from an assembler interrupt handler. The following is a sample calling sequence.

```
          LA    R13,PASCSAVE
          LA    R1,EXTPARM
          L     R15,=V(RTCPEXTR)
          BALR  R14,R15
                 .
                 .
RUPTCODE  DS    H       Store interrupt code here before calling XTCPEXTR
PASCSAVE  DS    18F     Register save area
ENV       DC    F'0'    Zero initially.  It will be filled in with
                        an environment address.  Pass it unchanged
                        in subsequent calls to RTCPEXTR.
EXTPARM   DC    A(ENV)
          DC    A(RUPTCODE)
```

RTcpVmcfRupt

This procedure is a version of **TcpVmcfRupt** Pascal procedure and is callable directly from an assembler interrupt handler. The following is a sample calling sequence.

```
          LA    R13,PASCSAVE
          LA    R1,VMCFPARM
          L     R15,=V(RTCPVMCF)
          BALR  R14,R15
                 .
                 .
PASCSAVE  DS    18F     Register save area
ENV       DC    F'0'    Zero initially.  It will be filled in with
                        an environment address.  Pass it unchanged
                        in subsequent calls to RTCPVMCF.
VMCFPARM  DC    A(ENV)
          DC    A(VMCFBUF)    Address of your VMCF interrupt buffer.
```

AddUserNote

This procedure can be called from assembler code to add a USERdefinedNOTIFICATION notification to the note queue, and wake up **GetNextNote** if it is waiting for a notification. The following is a sample calling sequence.

```
            LA    R13,PASCSAVE
            LA    R1,PASCPARM
            L     R15,=V(ADDUSERN)
            BALR  R14,R15
                   .
                   .
PASCSAVE    DS    18F      Register save area
ENV         DC    F'0'     Zero initially.  It will be filled in with
                           an environment address.  Pass it unchanged
                           in subsequent calls to ADDUSERN.
DATA1       DS    H        Data for Connection field of notification.
DATA2       DS    C        Data for Protocol field of notification.
DATA3       DS    XL40     Data for UserData field of notification.
RC          DS    F        AddUserNote stores return code here.

PASCPARM    DC    A(ENV)
            DC    A(DATA1)
            DC    A(DATA2)
            DC    A(DATA3)
            DC    A(RC)
```

Possible ReturnCode values:

OK
NObufferSPACE.

Appendix D. Network File System Server Exit Routines

The content of this appendix is Product Sensitive and is intended to provide a customized interface to remote procedure calls, and must not be used for any other purpose.

This appendix contains information about exits provided by the Network File System server code, which can be used by your site's technical support personnel to create routines to do the processing your installation requires. The exits provided are:

- Login
- Security
- Archive
- Account.

The site routines should be executed as part of the Network File System server.

Related Publications

The *IBM Transmission Control Protocol/Internet Protocol for MVS: User's Guide* contains information relevant to the user in the client environment.

The *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* book contains information about installing the server, messages, warnings and information that it could generate, operator commands and configuration.

Login Exit

The login exit routine is called when a client tries to use either the **mvslogin** or **mvslogout** commands. The exit routine can use Security Authorization Facility (SAF) or a customized authorization facility.

This exit will have a parameter list passed from the server and is called in one of three ways:

1. **For a login:** User verification, password checking, and new password processing. The exit routine returns a value to the server.
2. **For a logout:** Can do cleanup or other required processing.
3. **When a time-out occurs:** The time-out interval can be extended or the server can be told to proceed with a forced logout.

Requirements

Check and be sure you observe and follow each of the following requirements. The login exit routine must:

- Be link-edited with the name NFSXU1.
- Be reentrant.
- Reside in an APF-authorized library because when it receives control it is part of an APF-authorized task.
- Obtain a Global Storage Block (see Request code 4 description in Table D-1 on page D-3). The address of this block is returned to the Network File System in the parameter list, and is passed back to the exit in each subsequent call. This block contains user exit data that will be needed as long as the Network File System server is active.

  Note: Access to the Global Storage Block (GSB) must be controlled by the user-written exits to ensure that updates to common data occur from a single task at a time. This block is shared with the security exit.

- Obtain a User Storage Block (see Request code 8 description in Table D-1 on page D-3). The address of this block is returned to the Network File System in the parameter list, and is passed back to the exit on each subsequent call related to this machine-user combination. This block will contain a save area and user data needed for this session.

  Note: Access to the User Storage Block (USB) is controlled by the server. The Network File System will use single thread access to this exit routine for each machine-user. This block is shared with the security and archive exits.

Register Contents

Standard operating system (O/S) register conventions are used. On entry to the login routine, the registers contain:

Register 1 Address of a fullword that contains the address of the parameter list for this exit routine.

Register 13 Address of the caller's save area.

Register 14 Caller's return address.

Register 15 Address of the entry point for this exit routine.

Note: Address parameters will have a null value (0) if the related data does not exist.

Contents of Parameter List

The parameter list is made up of 18 contiguous fullwords. Table D-1 on page D-3 provides the contents of each word and information about each system request code. Detailed information about what this exit routine should do for each event follows the table.

Table D-1 (Page 1 of 2). Login Exit Routines		
Fullword Number	**Field**	**Description (Contents)**
1	Request code	System request code set by the Network File System server before call to this exit, for the following conditions: **4** System initialization **8** Start of new user Session **12** User login request **16** New password supplied **20** User timed out **24** Logout has been requested **28** System termination
2	Return code	Codes generated and returned by this routine: **0** Login successful. **4** Invalid user ID. Message returned. **8** Invalid password. Message returned. **12** Password expired. Message returned. **16** Password required. Message returned. **20** User ID required. Message returned. **24** Timeout interval extended; value supplied. **28** Logout forced because of server timeout. **32** Request invalid. **36** User not authorized for this service.
3	Address of client system name	Address of character string terminated by single byte containing X'00'.
4	Client IP address	Number
5	Client user ID number	Number
6	Client group ID number	Number
7	Address of MVS user ID	Address of character string terminated by single byte containing X'00', conforming to MVS standards.
8	Address of MVS group name	Address of character string terminated by single byte containing X'00', conforming to security system standards.
9	Address of MVS user password	Address of character string terminated by single byte containing X'00', conforming to MVS and security system standards.
10	Address of new user password	Address of character string terminated by single byte containing X'00', conforming to MVS and security system standards.
11	Session timeout value	Number, specified in seconds.
12	Address of USB	Size and content installation-dependent, generated at start of a user session.

Table D-1 (Page 2 of 2). Login Exit Routines		
Fullword Number	**Field**	**Description (Contents)**
13	Address of GSB	Size and content installation-dependent, generated at system initialization.
14	Address of message supplied by this exit routine	Address of character string terminated by single byte containing X'00'.
15, 16	Reserved for future definition	
17	Authentication type	Valid types are: **0** None **1** UNIX-type credentials **2** Short hand UNIX credentials **3** DES credentials
18	User time delta	Number, specified in seconds. Amount of time user system is West of Greenwich.

Using the Parameter List

This section describes how the data in the parameter list is used by the Network File System server and by the login exit routine. A request code is set by the Network File System server before each call to this exit routine. Each topic below describes an event, for which some fields are set on entry. This exit routine sets a return code for each event.

System Initialization

This routine is used once at system initialization, and must acquire and initialize the GSB. The codes and fields used are shown in Table D-2.

Table D-2. System Initialization Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**4**
On exit	2	Return code	**0** Initialization successful
On exit	13	Global Storage Block	Address

Start of New User Session

This routine is used when a new client machine-user combination is recognized by the Network File System server. A USB is acquired and analyzed at this time if access is permitted. If access is not permitted, an **mvslogin** command must be issued by the Network File System client. The codes and fields used are shown in Table D-3.

Table D-3. Start of New User Session Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**8**
On entry	3	Client name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number
On entry	6	Client Group ID number	Number
On exit	2	Return code	**0** Login successful
On exit	12	User Storage Block	Address

User Logon Request

This routine is called when **mvslogin** is used. The installation security system should be called to determine if the user is properly authorized. The codes and fields used are shown in Table D-4.

Table D-4 (Page 1 of 2). User Logon Request Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**12**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS Group Name	Character string
On entry	9	MVS User Password	Character string
On entry	11	Session Timeout Value	Number of seconds
On entry	12	User Storage Block	Address
On entry	13	Global Storage Block	Address
On entry	17	Authentication Type	Number

Table D-4 (Page 2 of 2). User Logon Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	18	User Time Delta	Number of seconds
On exit	2	Return code	**0** Login successful **4** Invalid user ID **8** Invalid password **12** Password expired **16** Password required **20** User ID required

New Password Supplied

This routine is used when the user enters a new password. Calls can be made to the installation security system. The codes and fields used are shown in Table D-5.

Table D-5. New Password Supplied Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**16**
On entry	3	Client name	Character string
On entry	4	Client IP address	Number
On entry	5	client ID number	Number
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS Group Name	Character string
On entry	9	MVS User Password	Character string
On entry	10	New User Password	Character string
On entry	11	Session Timeout Value	Number of seconds
On entry	12	User Storage Block	Address
On entry	13	Global Storage Block	Address
On entry	17	Authentication Type	Number
On entry	18	User Time Delta	Number of seconds
On exit	2	Return code	**0** Login successful **4** Invalid user ID **8** Invalid password **12** Password expired **16** Password required **20** User ID required

User Timed Out

This routine is used at the expiration of the timeout interval when no new user input has been received. A new timeout interval can be provided or a logoff can be forced for the user. The codes and fields used are shown in Table D-6.

Table D-6. User Timed Out Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**20**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS Group Name	Character string
On entry	9	MVS User Password	Character string
On entry	11	Session Timeout Value	Number of seconds
On entry	12	User Storage Block	Address
On entry	13	Global Storage Block	Address
On entry	17	Authentication Type	Number
On exit	18	User Time Delta	Number of seconds
On exit	2	Return code	**24** Extend timeout interval. Value supplied. **28** Force timeout logout.

Logout Has Been Requested

This routine is used at logout to return storage obtained at the start of the session and to perform any related termination processing. The codes and fields used are shown in Table D-7.

Table D-7 (Page 1 of 2). Logoff Has Been Requested Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**24**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number

Table D-7 (Page 2 of 2). Logoff Has Been Requested Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS Group Name	Character string
On entry	9	MVS User Password	Character string
On entry	11	Session Timeout Value	Number of seconds
On entry	12	User Storage Block	Address
On entry	13	Global Storage Block	Address
On entry	17	Authentication Type	Number
On entry	18	User Time Delta	Number of seconds
On exit	2	Return code	**0** Logout successful

System Termination

This routine is used at Network File System server termination to terminate processing and to release the storage used for the GSB. The codes and fields used are shown in Table D-8.

Table D-8. System Termination Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**28**
On exit	2	Return code	**0** Exit termination successful

Sample Parameter List - Assembler Language DSECT

This listing is found in TCPIP.COMMMAC(LOGIN). If this listing differs from that shipped with the source code, the latter is more accurate.

```
------------------------------------------------
*
*       Copyright (C) 1987, 1988, Sun Microsystems, Inc.
*                          and Electronic Data Systems Corp.
*       Licensed Materials - Property of IBM
*       5798-061 (C) Copyright IBM Corporation
*       All rights reserved.
*       U.S. Government Users Restricted Rights -
*       Use, duplication or disclosure restricted
*       by GSA ADP Schedule Contract with IBM Corporation
*
```

```
*
* NFS login exit data structure (LEDS)
*
*        All strings are terminated with x'00'
*        NULL (x'00') pointers are used for unavailable data
*
LEDS     DSECT
LEDSRQ   DS     F                       REQUEST FROM CALLER
LEDSIN   EQU    4                       SYSTEM INITIALIZATION
LEDSSS   EQU    8                       START OF USER SESSION
LEDSLO   EQU    12                      LOGON REQUEST
LEDSPS   EQU    16                      NEW PASSWORD SUPPLIED
LEDSTM   EQU    20                      TIMEOUT OCCURRED
LEDSLF   EQU    24                      LOGOFF REQUEST/FORCED
LEDSXX   EQU    28                      SYSTEM TERMINATION
MAXREQ   EQU    28                      MAXIMUM REQUEST VALUE
*
LEDSRC   DS     F                       EXIT STATUS
LEDSOK   EQU    0                       LOGON SUCCESSFUL
LEDSIU   EQU    4                       INVALID USERID, MSG RETURNED
LEDSIP   EQU    8                       INVALID PASSWORD, MSG RETURNED
LEDSPX   EQU    12                      PASSWORD EXPIRED, MSG RETURNED
LEDSPR   EQU    16                      PASSWORD REQUIRED, MSG RETURNED
LEDSUR   EQU    20                      USERID REQUIRED, MSG RETURNED
LEDSXT   EQU    24                      EXTEND TIMEOUT, USE NEW INTERV
LEDSFT   EQU    28                      FORCE TIMEOUT LOGOFF
LEDSIR   EQU    32                      INVALID REQUEST BYTE
LEDSUN   EQU    36                      USER NOT AUTHORIZED FOR SERV
*
LEDSM    DS     A                       ADDRESS OF AIX MACHINE NAME
LEDSIA   DS     F                       INTERNET PROTOCOL (IP) ADDRESS
LEDSU    DS     F                       AIX USER ID
LEDSG    DS     F                       AIX GROUP
LEDSMU   DS     A                       ADDRESS OF MVS USER-ID
LEDSMG   DS     A                       ADDRESS OF MVS GROUP NAME
LEDSMP   DS     A                       ADDRESS OF MVS USER PASSWORD
LEDSNP   DS     A                       ADDRESS OF NEW USER PASSWORD
LEDSTO   DS     F                       SESSION TIMEOUT VALUE IN SECOND
LEDSXS   DS     A                       ADDRESS OF EXIT CONTROLLED STRG
LEDSXG   DS     A                       ADDRESS OF EXIT GLOBAL STORAGE
*                                       SET ON INITIAL CALL ONLY
LEDSXD   DS     A                       ADDRESS OF EXIT SUPPLIED MSG
         DS     F                       RESERVED
         DS     F                       RESERVED
LEDSAU   DS     F                       AUTHENTICITY TYPE TO USE
LEDSUT   DS     F                       USER TIME ADJ(MIN WEST OF GMT)
*
LEDSLEN  EQU    *-LEDS                  LENGTH OF EXIT DATA STRUCTURE
*
         MEND
```

Security Exit

The security exit routine verifies that a user is authorized to access a specific MVS data set or data set member in the access mode requested.

The three access modes and what they permit are as follows:

1. **Allocate:** Read, write, create, delete or rename the data set.
2. **Write:** Read or write the data set.
3. **Read:** Read the data set.

This exit will have a parameter list passed from the server and is called:

1. **For Data Set Access:** When a logged in user tries to access, read or write an MVS data set or data set member.
2. **For Data Set Allocation:** When a logged in user tries to allocate (create), rename, or delete a specific data set.
3. **For Getting Access Mode or Permissions:** The Network File System server needs the access mode or the permissions that a user has for a specific data set.

A return code is set by the exit routine indicating whether or not the request is allowed. This exit is not called at startup or shutdown.

Requirements

Each of the following requirements must be followed. The security exit routine must:

- Be link-edited with the name NFSXU2.
- Be reentrant.
- Reside in an APF-authorized library because when it receives control it is part of an APF-authorized task.

  You do not have to allocate any control blocks, because the Network File System server makes its control blocks available to this exit routine.

Notes:

1. Access to the USB is controlled by the server. The Network File System will single thread access to the security exit for one machine-user. The USB is shared with the login and archive exits.
2. Access to the GSB must be controlled by these exit routines to be sure that updates to common data occur from a single task at a time. The GSB is shared with the login exit.

Register Contents

Standard O/S register conventions are used. On entry to this routine the registers contain:

Register 1 Address of a fullword that contains the address of the parameter list for this exit routine.

Register 13 Address of the caller's save area.

Register 14 Caller's return address.

Register 15 Address of the entry point for this exit routine.

Contents of Parameter List

The parameter list is 17 contiguous fullwords. Table D-9 provides the contents of each word, as well as information about each system request code. Detailed information about what this exit routine should do for each event follows the table.

Table D-9 (Page 1 of 2). Security Exit Routines

Fullword Number	Field	Description (Contents)
1	Request code	System request code set by the Network File System server before call to this exit, for the following conditions: **4** Validate allocate request **8** Validate write request **12** Validate read request **16** Return security permissions
2	Return code	Codes generated and returned by this routine: **0** Access allowed **4** Access denied **8** Permissions returned **12** Invalid request
3	Address of client system name	Address of character string terminated by single byte containing X'00'.
4	Client IP address	Number
5	Client user ID number	Number
6	Client group ID number	Number
7	Address of MVS user ID	Address of character string terminated by single byte containing X'00'. This conforms to MVS standards.
8	Address of MVS data set name	Address of character string terminated by single byte containing X'00'. This conforms to MVS standards.
9	Address of MVS data set member name	Address of character string terminated by single byte containing X'00'. This conforms to MVS standards.

Table D-9 (Page 2 of 2). Security Exit Routines		
Fullword Number	**Field**	**Description (Contents)**
10	Address of data set volume name	Address of character string terminated by single byte containing X'00'. Contains the volume serial number where the requested data set resides.
11	Address of catalog volume name	Address of character string terminated by single byte containing X'00'. Contains the volume serial number for the catalog that contains the entry for the requested data set.
12	Address of catalog data set name	Address of character string terminated by single byte containing X'00'. Contains MVS data set name for the catalog that contains the entry for the requested data set.
13	Address of USB	Size and content installation dependent. Generated at the start of a user session.
14	Address of GSB	Size and content installation dependent. Generated at system initialization.
15, 16	Reserved for future definition	
17	Permissions returned	Valid types are: **0** None **1** Allocate allowed **2** Write allowed **4** Read allowed
Note: Address parameters will have null value (0) if the related data does not exist.		

Using the Parameter List

This section describes how the data in the parameter list is used by the Network File System server and by the security exit routine. A request code is set by the Network File System server before each call to this exit routine.

Each topic below describes an event, for which some fields are set on entry. This exit routine sets a return code for each event.

Validate Allocate Request

This routine is used when a user tries to allocate, rename, or delete a specific MVS data set or data set member. The codes and fields used are shown in Table D-10.

Table D-10 (Page 1 of 2). Validate Allocate Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**4**
On entry	3	Client machine name	Character string

Table D-10 (Page 2 of 2). Validate Allocate Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	4	Client IP address	Number
On entry	5	Client ID number	Number
On entry	6	Client ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	Catalog volume name	Character string
On entry	12	Catalog data set name	Character string
On entry	13	User Storage Block	Address
On entry	14	Global Storage Block	Address
On exit	2	Return code	**0** Access allowed **4** Access denied

Validate Write Request

This routine is called when a user tries to write to a specific MVS data set or data set member. The codes and fields used are shown in Table D-11.

Table D-11 (Page 1 of 2). Validate Write Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**8**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string

Table D-11 (Page 2 of 2). Validate Write Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	11	Catalog volume name	Character string
On entry	12	Catalog data set name	Character string
On entry	13	User Storage Block	Address
On entry	14	Global Storage Block	Address
On exit	2	Return code	**0** Access allowed **4** Access denied

Validate Read Request

This routine is used when a user tries to read from a specific MVS data set or data set member. The codes and fields used are shown in Table D-12.

Table D-12. Validate Read Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**12**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	Catalog volume name	Character string
On entry	12	Catalog data set name	Character string
On entry	13	User Storage Block	Address
On entry	14	Global Storage Block	Address
On exit	2	Return code	**0** Access allowed **4** Access denied

Return Security Permissions

This routine is used when the access mode or permissions that a user has for a specific data set are requested. The codes and fields used are shown in Table D-13.

Table D-13. Return Security Permissions Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**16**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client User ID number	Number
On entry	6	Client Group ID number	Number
On entry	7	MVS User ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	Catalog volume name	Character string
On entry	12	Catalog data set name	Character string
On entry	13	User Storage Block	Address
On entry	14	Global Storage Block	Address
On exit	2	Return code	**8** Permissions returned
On exit	17	Permissions returned	

Sample Parameter List - Assembler Language DSECT

This listing is found in TCPIP.COMMMAC(SECURE). If this listing differs from that shipped with the source code, the latter is more accurate.

```
------------------------------------------------
*
*       Copyright (C) 1987, 1988, Sun Microsystems, Inc.
*                         and Electronic Data Systems Corp.
*       Licensed Materials - Property of IBM
*       5798-061 (C) Copyright IBM Corporation
*       All rights reserved.
*       U.S. Government Users Restricted Rights -
*       Use, duplication or disclosure restricted
*       by GSA ADP Schedule Contract with IBM Corporation
*
*  MVS/NFS  security exit file access data structure (FEDS)
*
```

```
*        All strings are terminated with x'00'
*        NULL (x'00') pointers are used for unavailable data
*
FEDS     DSECT
FEDSRQ   DS      F       REQUEST
FEDSAR   EQU     4       VALIDATE ALLOCATE REQUEST
FEDSWR   EQU     8       VALIDATE WRITE REQUEST
FEDSRR   EQU     12      VALIDATE READ REQUEST
FEDSSP   EQU     16      RETURN SECURITY PERMISSIONS
FEDMAXRQ EQU     16      MAX VALUE REQUEST FIELD CAN CONTAIN
*
FEDSRC   DS      F       EXIT STATUS
FEDSAA   EQU     0       ACCESS ALLOWED
FEDSAD   EQU     4       ACCESS DENIED
FEDSPR   EQU     8       PERMISSIONS RETURNED
FEDSIR   EQU     12      INVALID REQUEST
*
FEDSM    DS      A       ADDRESS OF CLIENT MACHINE NAME
FEDSIA   DS      F       INTERNET PROTOCOL (IP) ADDRESS
FEDSU    DS      F       CLIENT USER ID NUMBERS
FEDSG    DS      F       CLIENT GROUP ID NUMBERS
FEDSMU   DS      A       ADDRESS OF MVS USER-ID
FEDSDN   DS      A       ADDRESS OF MVS DATA SET NAME
FEDSMN   DS      A       ADDRESS OF MEMBER NAME IF PDS
*                        IF MEMBER NAME, ACTION IS ON MEMBER
FEDSDV   DS      A       ADDRESS OF DATASET VOLUME
FEDSVN   DS      A       ADDRESS OF CATALOG VOLUME NAME
FEDSCN   DS      A       ADDRESS OF CATALOG DATASET NAME
FEDSXS   DS      A       ADDRESS OF EXIT CONTROLLED STORAGE
*                        (FROM LOGON EXIT)
FEDSXG   DS      A       ADDRESS OF EXIT GLOBAL STORAGE
*                                SET ON INITIAL CALL ONLY
         DS      F       RESERVED
         DS      F       RESERVED
FEDSRPM  DS      F       PERMISSIONS RETURNED IF REQUESTED
FEDSAP   EQU     1       ALLOCATE PERMISSION ALLOWED
FEDSWP   EQU     2       WRITE PERMISSION ALLOWED
FEDSRP   EQU     4       READ PERMISSION ALLOWED
FEDSLEN  EQU     FEDS-*  LENGTH OF EXIT DATA STRUCTURE
*
         MEND
```

Archive Exit

The archive exit routine is used when a Network File System client tries to read, write, create, delete, or rename an MVS data set that cannot be located. First, the installation exit is called, then an archive management package such as HSM may find out if the data set is in the installation's archives. After the search, data in an archived data set can be returned to the server, or a request for the data set to be restored can be issued. The restoration will be done depending upon the server's request and the capabilities of the archive package. This exit is also called during system startup and shutdown.

Requirements

Each of the following requirements must be followed. The archive exit routine must:

- Be link-edited with the name NFSXU3.
- Be reentrant.
- Reside in an APF-authorized library because when it receives control it is part of an APF-authorized task.
- Obtain a GSB (see Request code 4 description in Table D-14 on page D-18). The address of this block is returned to the Network File System in the parameter list and is passed back to the exit in each subsequent call. This block contains user exit data that will be needed as long as the Network File System server is active.

  Note: Access to the GSB must be controlled by the user written exits to ensure that updates to common data occur from a single task at a time.

  The USB that was created by the login exit will be used by this routine.

  Note: Access to the USB is controlled by the server. The Network File System uses single thread access to this exit routine for each machine-user. The USB is shared with the security exit.

Register Contents

Standard O/S register conventions are used. On entry to this routine the registers contain:

Register 1	Address of a fullword that contains the address of the parameter list for this exit routine.
Register 13	Address of the caller's save area.
Register 14	Caller's return address.
Register 15	Address of the entry point for this exit routine.

Contents of Parameter List

The parameter list is 23 contiguous fullwords. Table D-14 on page D-18 provides the contents of each word and information about each system request code. Detailed information about what this exit routine should do for each event follows the table.

Table D-14 (Page 1 of 2). Archive Exit Routines

Fullword Number	Field	Description (Contents)
1	Request code	System request code set by the Network File System server before call to this exit, for the following conditions: **4** System initialization **8** Information from archive requested **12** Retrieve from archive requested **16** Read request **20** Write request **24** Delete request **28** Create request **32** System termination
2	Return code	Codes generated and returned by this routine: **0** Data set restored, re-issue lookup **4** Data set does not exist **8** Access privilege refused **12** Device not available **16** Data set deleted from archives **20** Information returned **24** Invalid request
3	Address of client system name	Address of character string terminated by single byte containing X'00'.
4	Client IP address	Number
5	Client user ID number	Number
6	Client group ID number	Number
7	Address of MVS user ID	Address of character string terminated by single byte containing X'00'. This conforms to MVS standards.
8	Address of MVS data set name	Address of character string terminated by single byte containing X'00'. This conforms to MVS standards.
9	Address of MVS data set member name	Address of character string terminated by single byte containing X'00'. This conforms to MVS and security system standards.
10	Address of MVS data set volume name	Address of character string terminated by single byte containing X'00'.
11	Address of User Storage Block	Size and content installation dependent. Acquired at the start of a user session.
12	Address of Global Storage Block	Size and content installation dependent. Acquired at system initialization.

Table D-14 (Page 2 of 2). Archive Exit Routines		
Fullword Number	**Field**	**Description (Contents)**
13	Address of message supplied by this exit routine	Address of character string terminated by single byte containing X'00'.
14, 15	Reserved for future definition	
16	Data set organization	First byte of DSORG field in Format 1 DSCB for requested data set. **0** Unknown **1** Sequential data set **2** Partitioned sequential data set **3** Direct access data set **4** Indexed sequential data set **5** VSAM **6** VSAM entry sequenced data set **7** VSAM relative record data set **8** VSAM key sequenced data set
17	File size	Size of requested data set in bytes.
18	Creation date (Packed value)	Julian date when requested data set was created. Format: X'00YYDDDS'; YY is the year, DDD is the day, S is the sign.
19	Archive date (Packed value)	Julian date when requested data set was archived. Format: X'00YYDDDS'; YY is the year, DDD is the day, S is the sign.
20	Archive time (Packed value) unsigned	Time requested data set was archived. Format: X'00HHMMSS'; HH is the hour, MM is the minute, SS is the second.
21	Data set record format	Contains the value of DCB RECFM.
22	Data set record length	Record length information maintained in the archives.
23	Data set block size	Block size information maintained in the archives.
Note: Address parameters will have a null value (0) if the related data does not exist.		

Using the Parameter List

This section describes how the data in the parameter list is used by the Network File System server and by the archive exit routine. A request code is set by the Network File System server before each call to this exit routine.

Each topic below describes an event, for which some fields are set on entry. This exit routine sets a return code for each event.

System Initialization

This routine is used at system initialization to get the storage and initialize the GSB. The codes and fields used are shown in Table D-15.

Table D-15. System Initialization Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**4**
On exit	2	Return code	**0** Exit initialization successful.
On exit	12	Global Storage Block	Initialized

Information From Archive Requested

This routine is called when the Network File System server cannot locate the requested data set. The codes and fields used are shown in Table D-16.

Table D-16 (Page 1 of 3). Information from Archives Requested Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**8**
On entry	3	Client machine name	Name
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	User Storage Block	Address
On entry	12	Global Storage Block	Address

Table D-16 (Page 2 of 3). Information from Archives Requested Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On exit	2	Return code	**0** Data set restored, re-issue lookup. **4** Data set does not exist. **8** Access privilege refused. **12** Device not available. **20** Information returned.
On exit	16	Data set organization	First byte of DSORG field in Format 1 DSCB for requested data set. **0** Unknown. **1** Sequential data set. **2** Partitioned sequential data set. **3** Direct access data set. **4** Indexed sequential data set. **5** VSAM **6** VSAM entry sequenced data set. **7** VSAM relative record data set. **8** VSAM key sequenced data set.
On exit	17	File size	Size of requested data set in bytes.
On exit	18	Creation date	Date requested data set was created. Format: X'00YYDDDS'; YY is the year, DDD is the day, s is the sign.
On exit	21	Record format	Contains values of DCB RECFM
On exit	22	Record length	Information maintained in archives
On exit	23	Block size	Information maintained in archives

Table D-16 (Page 3 of 3). Information from Archives Requested Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On exit	19	Archive date	Packed value. Date requested data set was archived. Format: X'00YYDDDS'; YY is the year, DDD is the day, S is the sign.
On exit	20	Archive time	Packed unsigned value. Time requested data set was archived. Format: X'00HHMMSS'; HH is the hour, MM is the minute, SS is the second.

Retrieve From Archive Requested

This routine is called when the Network File System server determines that a data set must be restored from the archives. The codes and fields used are shown in Table D-17.

Table D-17 (Page 1 of 2). Retrieve From Archive Requested Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**12**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	User Storage Block	Address
On entry	12	Global Storage Block	Address

Table D-17 (Page 2 of 2). Retrieve From Archive Requested Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On exit	2	Return code	**0** Data set restored, re-issue lookup. **4** Data set does not exist. **8** Access privilege refused. **12** Device not available.

Read Request

This routine is called when the user tries to read a data set that cannot be located. This is a special case of Retrieve and is included in case unique processing is needed for a data set that will be read only. The codes and fields used are shown in Table D-18.

Table D-18. Read Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**16**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	User Storage Block	Address
On entry	12	Global Storage Block	Address
On exit	2	Return code	**0** Data set restored, re-issue lookup. **4** Data set does not exist. **8** Access privilege refused. **12** Device not available.

Write Request

This routine is called when the user tries to write to a data set that cannot be located. This is a special case of Retrieve and is included in case unique processing is needed for a data set that will be written to. The codes and fields used are shown in Table D-19.

Table D-19. Write Request Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**20**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	User Storage Block	Address
On entry	12	Global Storage Block	Address
On exit	2	Return code	**0** Data set restored, re-issue lookup. **4** Data set does not exist. **8** Access privilege refused. **12** Device not available.

Delete Request

This routine is called when the user tries to delete a data set that cannot be located. This routine is included in case unique processing is needed for a data set that will be deleted. The codes and fields used are shown in Table D-20.

Table D-20 (Page 1 of 2). Delete Request Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**24**
On entry	3	Client machine name	Character string

Table D-20 (Page 2 of 2). Delete Request Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	User Storage Block	Address
On entry	12	Global Storage Block	Address
On exit	2	Return code	**0** Data set restored, re-issue lookup. **4** Data set does not exist. **8** Access privilege refused. **12** Device not available. **16** Data set deleted from archives. The Network File System server expects that the exit routine has removed all references to the data set from the system.

Create Request

This routine is called when the user tries to create a data set that cannot be located by the archive routine. The codes and fields used are shown in Table D-21.

Table D-21 (Page 1 of 2). Create Request Codes and Fields			
When	**Fullword Number**	**Fields**	**Contents**
On entry	1	Request code	**28**
On entry	3	Client machine name	Character string

Table D-21 (Page 2 of 2). Create Request Codes and Fields			
When	**Fullword Number**	**Fields**	**Contents**
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	8	MVS data set name	Character string
On entry	9	MVS data set member name	Character string
On entry	10	MVS data set volume name	Character string
On entry	11	User Storage Block	Address
On entry	12	Global Storage Block	Address
On exit	2	Return code	**0** Data set restored, re-issue lookup. **4** Data set does not exist. **8** Access privilege refused. **12** Device not available. **16** Data set deleted from archives. The Network File System server expects that the exit routine has removed all references to the data set from the system.

System Termination

This routine is used to terminate processing and to release the storage used for the GSB. The codes and fields used are shown in Table D-22.

Table D-22 (Page 1 of 2). System Termination Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**32**
On entry	12	Global Storage Block	Address

Table D-22 (Page 2 of 2). System Termination Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On exit	2	Return code	**0** Exit termination is successful

Sample Parameter List - Assembler Language DSECT

This listing is found in TCPIP.COMMMAC(ARCHIVE). If this listing differs from that shipped with the source code, the latter is more accurate.

```
-----------------------------------------------
*
*       Copyright (C) 1987, 1988, Sun Microsystems, Inc.
*                          and Electronic Data Systems Corp.
*       Licensed Materials - Property of IBM
*       5798-061 (C) Copyright IBM Corporation
*       All rights reserved.
*       U.S. Government Users Restricted Rights -
*       Use, duplication or disclosure restricted
*       by GSA ADP Schedule Contract with IBM Corporation
*
* MVS/NFS archive exit data structure (ARCDS)
*
*       All strings are terminated with x'00'
*       NULL (x'00') pointers are used for unavailable data
*
ARCDS    DSECT
ARCDSRQ  DS     F                       REQUEST FROM CALLER
ARCDSIN  EQU    4                       SYSTEM INITIALIZATION
ARCDSDI  EQU    8                       D.S. INFORMATION REQUEST
ARCDSRA  EQU    12                      RETRIEVE FROM ARCHIVE
ARCDSRR  EQU    16                      READ REQUEST
ARCDSWR  EQU    20                      WRITE REQUEST
ARCDSDR  EQU    24                      DELETE REQUEST
ARCDSCR  EQU    28                      CREATE REQUEST
ARCDSTM  EQU    32                      SYSTEM TERMINATION
MAXREQ   EQU    32                      MAXIMUM REQUEST VALUE
*
ARCDSRC  DS     F                       EXIT STATUS
ARCDSRS  EQU    0                       ARCHIVE RESTORE SUCCESSFUL
ARCDSNO  EQU    4                       ENOENT - NO SUCH DATA SET
ARCDSAC  EQU    8                       EACCES - ACCESS NOT ALLOWED
ARCDSNX  EQU    12                      ENXIO _ DEVICE NOT AVAILABLE
ARCDSDL  EQU    16                      D.S. DELETED FROM ARCHIVES
ARCDSRI  EQU    20                      D.S. INFORMATION RETURNED
ARCDSIR  EQU    24                      INVALID REQUEST BYTE
*
ARCDSM   DS     A                       ADDRESS OF CLIENT MACHINE NAME
ARCDSIA  DS     F                       INTERNET PROTOCOL (IP) ADDRESS
ARCDSU   DS     F                       CLIENT USER ID
ARCDSG   DS     F                       CLIENT GROUP
ARCDSMU  DS     A                       ADDRESS OF MVS USER-ID
ARCDSDN  DS     A                       ADDRESS OF MVS DATA SET NAME
```

```
ARCDSMN  DS     A                       ADDRESS OF MEMBER NAME
ARCDSDV  DS     A                       ADDRESS OF VOLUME SERIAL
ARCDSXS  DS     A                       ADDRESS OF EXIT CONTROLLED STRG
ARCDSXG  DS     A                       ADDRESS OF EXIT GLOBAL STORAGE
*                                       SET ON INITIAL CALL ONLY
ARCDSXD  DS     A                       ADDRESS OF EXIT SUPPLIED MSG
         DS     F                       RESERVED
         DS     F                       RESERVED
*                              FOLLOWING VALUES ARE RETURNED BY EXIT
ARCDSDO  DS     F                       DATA SET TYPE
ARCDSSZ  DS     F                       FILE SIZE IN BYTES
ARCDSCD  DS     F                       CREATE DATE (00yyddds)
ARCDSAD  DS     F                       ARCHIVE DATE (00yyddds)
ARCDSAT  DS     F                       ARCHIVE TIME (00hhmmss)
ARCDSRF  DS     F                       DATA SET RECORD FORMAT
ARCDSRL  DS     F                       DATA SET RECORD LENGTH
ARCDSRS  DS     F                       DATA SET RECORD SIZE
ARCDSLEN EQU    *-ARCDS                 LENGTH OF EXIT DATA STRUCTURE
*
         MEND
```

Account Exit

The account exit routine is used when accounting information has been accumulated at various times during execution. Data is collected about resource utilization for each request or transaction as well as on activity to any data set. The data is identified with a specific user and can be summarized or written to a permanent data set.

When It Is Called

The account exit routine is called in the following cases:

- For startup and shutdown of the system, or general housekeeping (such as storage acquisition or release, opening and closing a data set).
- For a new machine and user starting, or when the user issues the **mvslogin** command. A USB can be created at this time.
- For completion of the processing of each request or transaction. Data such as elapsed time, active time, disk I/O counts, and TCP/IP I/O counts can be found and saved either in the USB or buffered into memory. I/O requests cannot be made.
- When use of data has been completed by a specific user. Data such as I/O counts and bytes transferred is available.
- When the time interval (15 minutes) is over for each user.
- When a user is logging off. The storage for the USB can be released.

Requirements

Each of the following requirements must be followed. The account exit routine must:

- Be link-edited with the name NFSXU4.
- Be reentrant.
- Reside in an APF-authorized library because when it receives control it is part of an APF-authorized task.
- Obtain a GSB (see Request code 4 description in Table D-23 on page D-30). The address of this block is returned to the Network File System in the parameter list and is passed back to the exit on each subsequent call. This block contains user exit data that will be needed while the Network File System server is active.

  Note: Access to the GSB must be controlled by the user-written exits to ensure that updates to common data occur from a single task at a time.
- Obtain a USB (see Request code 8 description in Table D-23 on page D-30). The address of this block is returned to the Network File System in the parameter list and is passed back to the exit on each subsequent call related to this machine-user combination. This block will contain a save area and a storage area for accumulating accounting information.

  Note: Access to the USB is controlled by the server. The Network File System server will use single thread access to this exit routine for each machine-user.

Register Contents

Standard O/S register conventions are used. On entry to this routine the registers contain:

Register 1	Address of a fullword that contains the address of the parameter list for this exit routine.
Register 13	Address of the caller's save area.
Register 14	Caller's return address.
Register 15	Address of the entry point for this exit routine.

Contents of Parameter List

The parameter list is 32 contiguous fullwords. Table D-23 on page D-30 provides the contents of each word and information about each system request code. Detailed information about what this exit routine should do for each event follows the table.

Table D-23 (Page 1 of 2). Account Exit Routines

Fullword Number	Field	Description (Contents)
1	Request code	System request code set by the Network File System server before call to this exit, for the following conditions: **4** System initialization **8** User initialization **12** User request complete **16** User interval expiration **20** User data set usage **24** User termination **28** System termination
2	Return code	Code generated and returned by this routine: **0** Processing successful **4** Processing error **8** Invalid request
3	Address of client system name	Address of character string terminated by single byte containing X'00'.
4	Client IP address	Number
5	Client user ID number	Number
6	Client group ID number	Number
7	Address of MVS user ID	Address of character string terminated by single byte containing X'00'. Conforms to MVS standards.
8	Address of User Storage Block	Size and content installation-dependent. Acquired at start of a user session.
9	Address of Global Storage Block	Size and content installation-dependent. Acquired at system initialization.
10	Address of error message	Address of character string terminated by single byte containing X'00'.
11, 12	Reserved for future use	
13, 14	Request start time	64 bits. TOD clock value at start of processing. Server sets the value when it receives the request.
15, 16	Request end time	64 bits. TOD clock value at the end of processing. Server sets the value immediately before returning the results of the request to the user.
17	Request program number	Program number of the server that processed the request.

Table D-23 (Page 2 of 2). Account Exit Routines

Fullword Number	Field	Description (Contents)
18	Request version number	Version number of the server that processed the request.
19	Request procedure number	Procedure number that was requested to be executed.
20	Request active time	Number of milliseconds request was actively worked on by the Network File System server tasks.
21	Request bytes read from TCP/IP	Number of bytes received by the Network File System server from TCP/IP for this request.
22	Request bytes written from TCP/IP	Number of bytes written by the Network File System server to TCP/IP for this request.
23	Request disk read count	Number of read operations to all data sets by the Network File System server needed to complete this request.
24	Request disk bytes read	Number of bytes read from disk by the Network File System server during all the read operations.
25	Request disk write count	Number of write operations to all data sets by the Network File System server needed to complete this request.
26	Request disk bytes written	Number of bytes written by the Network File System server during all write operations.
27	Address of data set name	Address of character string terminated by single byte containing X'00'. It is the address of the data set name used, in conformance to MVS standards.
28	Address of member name	Address of character string terminated by single byte containing X'00'. It is the address of the data set member used, in conformance to MVS standards.
29	Data set read count	Number of reads by user during this use of the data set.
30	Data set bytes read	Number of bytes transferred by this user for this use of the data set.
31	Data set write count	Number of write operations done by this user for this use of the data set.
32	Data set bytes written	Number of bytes written by this user for this use of the data set.

Note: Address parameters will have null value if the related data does not exist.

Using the Parameter List

This section describes how the data in the parameter list is used by the Network File System server and by the account exit routine. A request code is set by the Network File System server before each call to this exit routine.

Each topic below describes an event, for which some fields are set on entry. This exit routine sets a return code for each event.

System Initialization

This routine is used at system initialization and must acquire and initialize the GSB. The codes and fields used are shown in Table D-24.

Table D-24. System Initialization Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**4**
On exit	2	Return code. Global Storage Block created.	**0** Exit initialization successful.
On exit	9	Global Storage Block	Address

Start of New User Session

This routine is used at the first activity for a client session which can be a new machine-user or when a client issues an **mvslogin** command. This routine must also acquire and initialize the USB. The codes and fields used are shown in Table D-25.

Table D-25. Start of New User Session Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**8**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID	Character string
On entry	9	Global Storage Block	Address
On exit	2	Return code	**0** User initialization successful
On exit	8	User Storage Block	Address

User Request Complete

This routine is called when there is a request or transaction from a specific user that has been completed. The data can be collected, but a write cannot be done at this time. The codes and fields used are shown in Table D-26.

Table D-26. User Request Complete Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**12**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID name	Character string
On entry	8	User Storage Block	Address
On entry	9	Global Storage Block	Address
On entry	13, 14	Request start time	64 bits clock value
On entry	15, 16	Request end time	64 bits clock value
On entry	17	Request program number	Number
On entry	19	Request procedure number	Number
On entry	20	Request active time	Milliseconds
On entry	21	Request bytes read from TCP/IP	Bytes
On entry	22	Request bytes written from TCP/IP	Bytes
On entry	23	Request disk read count	Number
On entry	24	Request disk bytes read	Bytes
On entry	25	Request disk write count	Number
On entry	26	Request disk bytes written	Bytes
On exit	2	Return code	**0** Processing successful **4** Processing error
On exit	10	Message supplied by Exit routine	Site defined message for accounting purposes

User Interval Expiration

This routine is used every 15 minutes to collect information. Also used immediately before the user logs off. The codes and fields used are shown in Table D-27.

Table D-27. User Interval Expiration Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**16**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID name	Character string
On entry	8	User storage block	Address
On entry	9	Global storage block	Address
On exit	2	Return code	**0** Processing successful **4** Processing error
On exit	10	Message supplied	

User Data Set Usage

This routine is called when the data set has timed out (see the Processing Attributes Table in the *IBM Transmission Control Protocol/Internet Protocol for MVS: Installation and Maintenance* book, Configuring the Network File System Server chapter, for an explanation of timeouts). Data about the user's activity while using the data set is made available. The codes and fields used are shown in Table D-28.

Table D-28 (Page 1 of 2). User Data Set Storage Codes and Fields

When	Fullword Number	Field	Contents
On entry	1	Request code	**20**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID name	Character string
On entry	8	User Storage Block	Address
On entry	9	Global Storage Block	Address
On entry	27	Data set name	Address of

Table D-28 (Page 2 of 2). User Data Set Storage Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	28	Member name	Address of
On entry	29	Data set read count	Number
On entry	30	Data set bytes read	Bytes
On entry	31	Data set write count	Number
On entry	32	Data set bytes written	Bytes
On exit	2	Return code	**0** Processing successful **4** Processing error
On exit	10	Message supplied	

User Termination

This routine is used at logout to return storage obtained at the start of the session and performs any related termination processing. The codes and fields used are shown in Table D-29.

Table D-29. User Termination Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**24**
On entry	3	Client machine name	Character string
On entry	4	Client IP address	Number
On entry	5	Client user ID number	Number
On entry	6	Client group ID number	Number
On entry	7	MVS user ID name	Character string
On entry	8	User Storage Block	Address
On entry	9	Global Storage Block	Address
On exit	2	Return code	**0** User termination successful

System Termination

This routine is used at Network File System server termination to terminate processing and to release the storage used for the GSB. The codes and fields used are shown in Table D-30 on page D-36.

Table D-30. System Termination Codes and Fields			
When	**Fullword Number**	**Field**	**Contents**
On entry	1	Request code	**28**
On entry	9	Global Storage Block	Address
On exit	2	Return code	**0** Network File System server termination successful.

Sample Parameter List - Assemble Language DSECT

This listing is found in TCPIP.COMMMAC(ACCOUNT). If this listing differs from that shipped with the source code, the latter is more accurate.

```
------------------------------------------------
*
*       Copyright (C) 1987, 1988, Sun Microsystems, Inc.
*                          and Electronic Data Systems Corp.
*       Licensed Materials - Property of IBM
*       5798-061 (C) Copyright IBM Corporation
*       All rights reserved.
*       U.S. Government Users Restricted Rights -
*       Use, duplication or disclosure restricted
*       by GSA ADP Schedule Contract with IBM Corporation
*
*
* NFS Account exit data structure (ACCDS)
*
*       All strings are terminated with x'00'
*       NULL (x'00') pointers are used for unavailable data
*
ACTSDSCT DSECT
ACTSRQ   DS     F                       REQUEST FROM CALLER
ACTSSINT EQU    4                       - SYSTEM INITIALIZATION
ACTSUSIN EQU    8                       - USER INITIALIZATION
ACTSUSRC EQU    12                      - USER REQUEST COMPLETE
ACTSUSIX EQU    16                      - USER INTERVAL EXPIRATION
ACTSUSDS EQU    20                      - USER DATA SET USAGE
ACTSUSTM EQU    24                      - USER TERMINATION
ACTSSTRM EQU    28                      - SYSTEM TERMINATION
MAXREQ   EQU    28                      - MAXIMUM REQUEST VALUE
*
ACTSRC   DS     F                       EXIT RETURN CODE
ACTSOK   EQU    0                       - PROCESSING SUCCESSFUL
ACTSER   EQU    4                       - PROCESSING ERROR
ACTSIR   EQU    8                       - INVALID REQUEST
*
ACTSUMNA DS     A                       UNIX MACHINE NAME ADDRESS
ACTSIPA  DS     F                       INTERNET PROTOCOL (IP) ADDRESS
ACTSUUID DS     F                       UNIX USER ID
ACTSUGID DS     F                       UNIX GROUP ID
ACTSMUIA DS     A                       MVS USER-ID ADDRESS
```

```
ACTSUSBA DS     A                       USER STORAGE BLOCK ADDRESS
ACTSGSBA DS     A                       GLOBAL STORAGE BLOCK ADDRESS
ACTSESMA DS     A                       EXIT SUPPLIED MESSAGE ADDRESS
         DS     F                       - RESERVED
         DS     F                       - RESERVED
*
*        FOLLOWING ARE VALID WITH ACTSRQ = ACTSUSRC
*
ACTSRBEG DS     D                       REQUEST START TIME
ACTSREND DS     D                       REQUEST END TIME
ACTSRPGM DS     F                       REQUEST PROGRAM NUMBER
ACTSRVER DS     F                       REQUEST VERSION NUMBER
ACTSRPRC DS     F                       REQUEST PROCEDURE NUMBER
ACTSRACT DS     F                       REQUEST ACTIVE TIME
ACTSRCBR DS     F                       REQUEST CTC BYTES READ
ACTSRCBW DS     F                       REQUEST CTC BYTES WRITTEN
ACTSRDRC DS     F                       REQUEST DISK READ COUNT
ACTSRDBR DS     F                       REQUEST DISK BYTES READ
ACTSRDWC DS     F                       REQUEST DISK WRITE COUNT
ACTSRDBW DS     F                       REQUEST DISK BYTES WRITTEN
*
*        FOLLOWING ARE VALID WITH ACTRSQ = ACTSUSDS
*
ACTSDSNA DS     A                       DATA SET NAME ADDRESS
ACTSMBNA DS     A                       MEMBER NAME ADDRESS
ACTSDSRC DS     F                       DATA SET READ COUNT
ACTSDSBR DS     F                       DATA SET BYTES READ
ACTSDSWC DS     F                       DATA SET WRITE COUNT
ACTSDSBW DS     F                       DATA SET BYTES WRITTEN
*
ACTS     EQU    ACTSDSCT,*-ACTSDSCT
         MEND
```

Appendix E. Sample Programs

This appendix provides you with sample application programs to illustrate the Pascal, C Socket and X-Windows APIs.

A Sample Pascal Application

```
{*************************************************************************}
{*                                                                       *}
{* Memory-to-memory Data Transfer Rate Measurement                       *}
{*                                                                       *}
{* Pseudocode:  Establish access to TCP/IP Services                      *}
{*              Prompt user for operation parameters                     *}
{*              Open a connection (Sender:passive, Receiver:active)      *}
{*              If Sender:                                               *}
{*                Send 5M of data using TcpFSend                         *}
{*                Use GetNextNote to know when Send is complete          *}
{*                Print transfer rate after every 1M of data             *}
{*              else Receiver:                                           *}
{*                Receive 5M of data using TcpFReceive                   *}
{*                Use GetNextNote to know when data is delivered         *}
{*                Print transfer rate after every 1M of data             *}
{*              Close connection                                         *}
{*              Use GetNextNote to wait until connection is closed       *}
{*                                                                       *}
{*************************************************************************}
program SAMPLE;

%include CMALLCL
%include CMINTER
%include CMRESGLB

const
   BUFFERlength = 8192;             { same as MAXdataBUFFERsize      }
   PORTnumber   = 999;              { constant on both sides         }
   CLOCKunitsPERthousandth = '3E8000'x;

static
   Buffer         : packed array (.1..BUFFERlength.) of char;
   BufferAddress  : Address31Type;
   ConnectionInfo : StatusInfoType;
   Count          : integer;
   DataRate       : real;
   Difference     : TimeStampType;
   HostAddress    : InternetAddressType;
   IbmSeconds     : integer;
   Ignored        : integer;
   Line           : string(80);
   Note           : NotificationInfoType;
   PushFlag       : boolean;        { for TcpFSend                   }
```

Figure E-1 (Part 1 of 5). Example of a Pascal Application Program

```
   RealRate       : real;
   ReturnCode     : integer;
   SendFlag       : boolean;        { are we sending or receiving    }
   StartingTime   : TimeStampType;
   Thousandths    : integer;
   TotalBytes     : integer;
   UrgentFlag     : boolean;        { for TcpFSend                   }

   {*********************************************************************}
   {* Print message, release resources and reset environment           *}
   {*********************************************************************}
   procedure Restore ( const Message: string;
                       const ReturnCode: integer );
   begin
      Write(Message);
      if ReturnCode <> OK then
         Write(SayCalRe(ReturnCode));
      Writeln('');

      EndTcpIp;
      Close (Input);
      Close (Output);
   end;


begin
   TermOut (Output);
   TermIn (Input);

   { Establish access to TCP/IP services }
   BeginTcpIp (ReturnCode);
   if ReturnCode <> OK then begin
      Writeln('BeginTcpip: ',SayCalRe(ReturnCode));
      return;
   end;

   { Inform TCPIP which notifications will be handled by the program }
   Handle ((.DATAdelivered, BUFFERspaceAVAILABLE,
            CONNECTIONstateCHANGED, FRECEIVEerror,
            FSendResponse.), ReturnCode);
   if ReturnCode <> OK then begin
      Restore ('Handle: ', ReturnCode);
      return;
   end;

   { Prompt user for operation parameters                              }
   Writeln('Transfer mode: (Send or Receive)');
   ReadLn (Line);
   if (Substr(Line,1,1) = 's') or (Substr(Line,1,1) = 'S') then
      SendFlag := TRUE
   else
      SendFlag := FALSE;
```

Figure E-1 (Part 2 of 5). Example of a Pascal Application Program

```
Writeln('Host Name or Internet Address :');
ReadLn (Line);
GetHostResol (Line, HostAddress);
if HostAddress = NOhost then begin
   Restore ('GetHostResol failed. ', OK);
   return;
end;

{ Open a TCP connection: active for Send and passive for Receive }
{   - Connection value will be returned by TcpIp                 }
{   - initialize IBM reserved fields:  Security, Compartment     }
{     and Precedence                                             }
{ for Active open  - set Connection State to TRYINGtoOPEN        }
{                  - must initialize foreign socket              }
{ for Passive open - set ConnectionState to LISTENING            }
{                  - may leave foreign socket uninitialized to   }
{                    accept any open attempt                     }
with ConnectionInfo do begin
   Connection         := UNSPECIFIEDconnection;
   OpenAttemptTimeout := WAITforever;
   Security           := DEFAULTsecurity;
   Compartment        := DEFAULTcompartment;
   Precedence         := DEFAULTprecedence;
   if SendFlag then begin
      ConnectionState       := TRYINGtoOPEN;
      LocalSocket.Address   := UNSPECIFIEDaddress;
      LocalSocket.Port      := UNSPECIFIEDport;
      ForeignSocket.Address := HostAddress;
      ForeignSocket.Port    := PORTnumber;
   end
   else begin
      ConnectionState       := LISTENING;
      LocalSocket.Address   := HostAddress;
      LocalSocket.Port      := PORTnumber;
      ForeignSocket.Address := UNSPECIFIEDaddress;
      ForeignSocket.Port    := UNSPECIFIEDport;
   end;
end;
TcpWaitOpen (ConnectionInfo, ReturnCode);
if ReturnCode <> OK then begin
   Restore ('TcpWaitOpen: ', ReturnCode);
   return;
end;

{ Initialization }
BufferAddress := Addr(Buffer(.1.));
StartingTime  := ClockTime;
TotalBytes    := 0;
Count         := 0;
PushFlag      := false;     { let TcpIp buffer data for efficiency }
UrgentFlag    := false;              { none of out data is Urgent }
```

Figure E-1 (Part 3 of 5). Example of a Pascal Application Program

```
{ Issue first TcpFSend or TcpFReceive }
if SendFlag then
   TcpFSend (ConnectionInfo.Connection, BufferAddress,
               BUFFERlength, PushFlag, UrgentFlag, ReturnCode)
else
   TcpFReceive (ConnectionInfo.Connection, BufferAddress,
                  BUFFERlength, ReturnCode);

if ReturnCode <> OK then begin
   Writeln('TcpSend/Receive: ',SayCalRe(ReturnCode));
   return;
end;


{ Repeat until 5M bytes of data have been transferred }
while (Count < 5) do begin
   { Wait until previous transfer operation is completed }
   GetNextNote(Note, True, ReturnCode);
   if ReturnCode <> OK then begin
      restore('GetNextNote :',ReturnCode);
      return;
   end;

   { Proceed with transfer according to the Notification received   }
   { Notifications Expected :                                       }
   {  DATAdelivered - TcpFReceive function call is now complete     }
   {                - receive buffer contains data                  }
   {  FSENDresponse - TcpFSend function call is now complete        }
   {                - send buffer is now available for use          }
   {  FRECEIVEerror - if there was an error on TcpFReceive function }
   case Note.NotificationTag of
      DATAdelivered:
         begin
            TotalBytes := TotalBytes + Note.BytesDelivered;
            {issue next TcpFReceive                                }
            TcpFReceive (ConnectionInfo.Connection, BufferAddress,
                  BUFFERlength, ReturnCode);
            if ReturnCode <> OK then begin
               Restore('TcpFReceive: ',Note.SendTurnCode);
               return;
            end;
         end;
      FSENDresponse:
         begin
            if Note.SendTurnCode <> OK then begin
               Restore('FSENDresponse: ',Note.SendTurnCode);
               return;
            end
            else begin
               {issue next TcpFSend                          }
               TotalBytes := TotalBytes + BUFFERlength;
               TcpFSend (ConnectionInfo.Connection, BufferAddress,
                  BUFFERlength, PushFlag, UrgentFlag, ReturnCode);
               if ReturnCode <> OK then begin
```

Figure E-1 (Part 4 of 5). Example of a Pascal Application Program

```
                  Restore('TcpFSend: ',Note.SendTurnCode);
                  return;
               end;
            end;
         end;
      FRECEIVEerror:
         begin
            Restore('FRECEIVEerror: ', Note.ReceiveTurnCode);
            return;
         end;
      OTHERWISE
         begin
            Restore('UnExpected Notification ',OK);
            return;
         end;
   end; { Case on Note.NotificationTag }


   { is it time to print transfer rate? }
   if TotalBytes < 1048576 then
      continue;

   { Print transfer rate after every 1M bytes of data transferred }
   DoubleSubtract (ClockTime, StartingTime, Difference);
   DoubleDivide (Difference, CLOCKunitsPERthousandth, Thousandths,
                 Ignored);
   RealRate := (TotalBytes/Thousandths) * 1000.0;
   Writeln('Transfer Rate ', RealRate:1:0,' Bytes/sec.');

   StartingTime := ClockTime;
   TotalBytes   := 0;
   Count        := Count + 1;
end; {Loop while Count < 5 }

{ Close TCP connection and wait till partner also drops connection }
TcpClose (ConnectionInfo.Connection, ReturnCode);
if ReturnCode <> OK then begin
   Restore ('TcpClose: ', ReturnCode);
   return;
end;

{ when partner also drops connection, program will receive         }
{  CONNECTIONstateCHANGED notification with NewState = NONEXISTENT }
repeat
   GetNextNote (Note, True, ReturnCode);
   if ReturnCode <> OK then begin
      Restore ('GetNextNote: ', ReturnCode);
      return;
   end;
until (Note.NotificationTag = CONNECTIONstateCHANGED) &
      (Note.NewState = NONEXISTENT);

Restore ('Program terminated successfully. ', OK);
end.
```

Figure E-1 (Part 5 of 5). Example of a Pascal Application Program

A Sample C Socket Communications Server

```
/*********************************************************************
/
/    A sample C Socket communications server.
/
/    This process uses the TCP (SOCK_STREAM) communications protocol
/    to establish a reliable, connected pathway to the host running
/    the client process.  The socket is full-duplex.*
/
/    The function calls of chief importance are:
/
/    socket()     Creates the data-socket.
/    bind()       Associates the socket with a local address
/    listen()     Signals the TCP/IP space that the process is ready
/                 to accept foreign connection attempts.
/    accept()     Accepts a foreign connection, if there is data to read.
/                 Blocks otherwise, until data arrives.
/    read()       Reads data from the socket.
/    write()      Writes data to the socket.
/    close()      Closes the socket, terminating the connection.
/
/
/    NOTE:  This example performs no I/O, nor does it perform complete
/           error handling.  The error handling routine included,
/           errhand(), performs no useful function, just like the server
/           itself.
/
*********************************************************************/

/*********************/
/*   include files   */
/*********************/

#define MVS
#include <manifest.h>
#include <types.h>
#include <socket.h>
#include <tcperrno.h>
#include <netdb.h>
#include <in.h>
#include <uio.h>
#include <ioctl.h>
#include <ctype.h>
```

Figure E-2 (Part 1 of 4). Example of a C Socket Communications Server

```
/*********************/
/*   error handler   */
/*********************/

void errhand()
{
  exit(-1);              /* Error occurred. Quit. */
}

/*********************/
/*   server routine  */
/*********************/

main()
{
                         /* Client and server must choose port number */
  int port = 1965;       /* before beginning communications.          */
                         /* Certain ports are reserved - see RFC 1010 */

  int nbytes;                     /* Size of message buffer            */
  char buf[] = "the message";     /* The message buffer                */
  struct sockaddr_in name;        /* Complete address info             */
  int namelen;                    /* Size of name structure            */
  int s;                          /* The socket descriptor             */
  int ns;                         /* Accept's descriptor               */
  int backlog;                    /* Queue length                      */

/*********************/
/* Create the socket */
/*********************/

  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0)
  {                               /* If the socket function returns    */
    errhand();                    /* a negative value, the call to     */
  }                               /* socket failed.                    */

/*********************/
/* Set the fields of  */
/* the name structure */
/*********************/

  name.sin_family = AF_INET;      /* The communication domain          */
  name.sin_port   = htons(port); /* The port number to be used        */
  name.sin_addr.s_addr = INADDR_ANY
                                  /* Default local address             */
```

Figure E-2 (Part 2 of 4). Example of a C Socket Communications Server

```
/*******************/
/* Bind the socket */
/*******************/

  namelen = sizeof(name);
  if (bind(s, &name, namelen) < 0)
  {                                /* If bind() returns a negative      */
    errhand();                     /* value, the call failed.           */
  }

/*********************/
/* Inform the TCPIP */
/* address space of */
/* your readiness   */
/*********************/

  backlog = 1;
  if (listen(s, backlog) != 0)
  {                                /* If listen() returns               */
    errhand();                     /* a non-zero value,                 */
  }                                /* it failed.                        */

/*********************************/
/* Accept any queued connections */
/*********************************/

  ns = accept(s, &name, &namelen);
  if (ns == -1)
  {                                /* If accept() returns               */
    errhand();                     /* negative one, it failed           */
  }

/*****************************/
/* Read the client's message */
/*****************************/

  if (read(s, buf, nbytes) == -1)
  {                                /* If read() returns                 */
    errhand();                     /* negative one, it failed.          */
  }

/******************/
/* Send a message */
/******************/

  nbytes = sizeof(*buf);
  if (write(s, buf, nbytes) < 0)
  {                                /* If write() returns                */
    errhand();                     /* negative one, it failed.          */
  }
```

Figure E-2 (Part 3 of 4). Example of a C Socket Communications Server

```
/*******************/
/* Close the socket */
/* associated with  */
/* the accept call  */
/*******************/

  close(ns);

/*****************************/
/* Close the original socket */
/*****************************/

  close(s);

/**************/
/* That's it! */
/**************/
}
```

Figure E-2 (Part 4 of 4). Example of a C Socket Communications Server

A Sample C Socket Communications Client

```
/*********************************************************************
/
/    A sample C Socket communications client.
/
/    This process uses the TCP (SOCK_STREAM) communications protocol
/    to establish a reliable, connected pathway to the host running
/    the server process.  The socket is full-duplex.
/
/    The function calls of chief importance are:
/
/    socket()     Creates the data-socket.
/    connect()    Establishes a connection with the server's host.
/                 Connect will not return until a connection has been
/                 made, or an error condition discovered.
/    write()      Writes data to the socket.
/    read()       Reads data from the socket.
/    close()      Closes the socket, terminating the connection.
/
/
/    NOTE:  This example performs no I/O, nor does it perform complete
/           error handling.  The error handling routine included,
/           errhand(), performs no useful function, just like the client
/           itself.
/
*********************************************************************/

/*********************/
/*   include files   */
/*********************/

#define MVS
#include <manifest.h>
#include <types.h>
#include <socket.h>
#include <tcperrno.h>
#include <netdb.h>
#include <in.h>
#include <uio.h>
#include <ioctl.h>
#include <ctype.h>

/*********************/
/*   error handler   */
/*********************/

void errhand()
{
  exit(-1);              /* Error occurred. Quit. */
}
```

Figure E-3 (Part 1 of 3). Example of a C Socket Communications Client

```
/*********************/
/*   client routine  */
/*********************/

main()
{
                          /* Client and server must choose port number */
  int port = 1965;        /* before beginning communications.          */
                          /* Certain ports are reserved - see RFC 1010 */

  int nbytes;                      /* Size of message buffer           */
  char buf[] = "the message";      /* The message buffer               */
  char *host = "ForeignHost";      /* The server's local host          */
  char hostname[80];               /* The client's host's name         */
  struct hostent *hp;              /* Client's host's address info     */
  unsigned long hostaddr;          /* Client's actual address          */
  structure sockaddr_in myname;    /* Client's host's address and port */
  struct hostent *hostnm;          /* Resolved hostname info           */
  struct sockaddr_in name;         /* Complete address info            */
  int namelen;                     /* Size of name structure           */
  int s;                           /* The socket descriptor            */

  hostnm = gethostbyname(host);    /* Resolve the server's name        */
  if (hostnm == 0)
  {                                /* If a NULL pointer is returned,   */
    errhand();                     /* there was an error in the call   */
  }                                /* to gethostbyname()               */

/************************/
/* Set the fields of    */
/* the name structures */
/************************/

  name.sin_family = AF_INET;       /* The communication domain         */
  name.sin_port   = htons(port); /* The port number to be used       */
  name.sin_addr.s_addr = *((unsigned long *) hostnm->h_addr);
                                   /* Internet address field of hostnm */
  gethostname(hostname, sizeof(hostname); /* What is my host's name?  */
  hp = gethostbyname(hostname);    /* Determine host's address info    */
  hostaddr = hp->h_addr_list[0]; /* Use first internet address       */

  myname.sin_family = AF_INET      /* The communication domain         */
  myname.sin_port = 0;             /* Take any available port          */
  myname.sin_addr.s_addr = hostaddr;
                                   /* Client will receive on hostaddr  */

/*********************/
/* Create the socket */
/*********************/

  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s < 0)
  {                                /* If the socket function returns   */
    errhand();                     /* a negative value, the call to    */
  }                                /* socket failed.                   */
```

Figure E-3 (Part 2 of 3). Example of a C Socket Communications Client

```
/*******************/
/* Bind the socket */
/*******************/

  namelen = sizeof(myname);
  if (bind(s, &myname, namelen) < 0)
  {                              /* If bind() returns a negative      */
    errhand();                   /* value, the call failed.           */
  }

/***********************/
/* Connect the socket */
/***********************/

  namelen = sizeof(name);
  if (connect(s, &name, namelen) < 0)
  {                              /* If connect() returns a negative   */
    errhand();                   /* value, the call failed.           */
  }

/*******************/
/* Send a message */
/*******************/

  nbytes = sizeof(*buf);
  if (write(s, buf, nbytes) < 0)
  {                              /* If write() returns                */
    errhand();                   /* a negative value,                 */
  }                              /* it failed.                        */

/****************************/
/* Read the server's reply */
/****************************/

  if (read(s, buf, nbytes) < 0)
  {                              /* If read() returns                 */
    errhand();                   /* a negative value,                 */
  }                              /* it failed.                        */

/*********************/
/* Close the socket */
/*********************/

  close(s);

/**************/
/* That's it! */
/**************/
}
```

Figure E-3 (Part 3 of 3). Example of a C Socket Communications Client

A Sample X-Windows Application

Figure E-4 gives a simple example of an X-Windows application which opens the display and creates a window, waits 60 seconds, then destroys the window before ending.

```
/*
** This is a basic X-Window program, written using the X-Windows
** Application Program Interface (API).
*/
#include    <Xlib.h>
#include    <stdefs.h>
#include    <Xutil.h>

main(argc, argv)
int argc;
char    **argv;
{
    Display *dp;
    Window  w;
    char *cp;

/*
**  X will look up the value of the DISPLAY global variable
**  in the CENV group when passed a NULL pointer in XOpenDisplay.
*/
    dp = XOpenDisplay(NULL);
/*
**  Create a 200x200 window at xy(40, 40) with a black border.
*/
    w = XCreateSimpleWindow(dp, RootWindow(dp, 0), 40, 40, 200,
           200, 2, BlackPixel(dp, 0),
           WhitePixel(dp, 0));
/*
**  Map the window to the display...
**  This will cause the window to become visible on the screen.
*/
    XMapWindow(dp, w);
/*
**  Force X to write buffered requests.
*/
    XFlush(dp);
/*
**  Sleep for a minute.
*/
    sleep (60);
/*
**  Destroy the window and end the connection to the X Server.
*/
    XDestroyWindow(dp, w);
    XCloseDisplay(dp);
}
```

Figure E-4. Example of an X-Windows Application

Appendix F. Related Protocol Specifications

The following publications are used as the protocol specifications:

User Datagram Protocol, RFC 768, J. Postel

Trivial File Transfer Protocol, RFC 783, K.R. Sollins

Internet Protocol, RFC 791, J. Postel

Internet Control Message Protocol, RFC 792, J. Postel

Transmission Control Protocol, RFC 793, J. Postel

Simple Mail Transfer Protocol, RFC 821, J. Postel

Standard for the Format of ARPA Internet Text Messages, RFC 822, David H. Crocker

The DARPA Internet Gateway, RFC 823, R. Hinden, A. Sheltzer

An Ethernet Address Resolution Protocol, RFC 826, D. Plummer

Telnet Protocol Specification, RFC 854, J. Postel, J. Reynolds

Telnet Binary Transmission, RFC 856, J. Postel, J. Reynolds

Telnet Echo Option, RFC 857, J. Postel, J. Reynolds

A Standard for the Transmission of IP Datagrams over Public Data Networks, RFC 877, J.T. Korb

Telnet End Of Record Option, RFC 885, J. Postel

Telnet Terminal Type Option, RFC 930, M. Solomon, E. Wimmers

Internet Standard Subnetting Procedure, RFC 950, J. Mogul, J. Postel

DOD Internet Host Table Specification, RFC 952, K. Harrenstien, M. Stahl, E. Feinler

File Transfer Protocol, RFC 959, J. Postel

Mail Routing And The Domain System, RFC 974, C. Partridge

Assigned Numbers, RFC 1010, J. Reynolds, J. Postel

Official ARPA Internet Protocols, RFC 1011, J. Reynolds, J. Postel

X Window System Protocol, Version 11, RFC 1013, R. Scheifler

XDR: External Data Representation Standard, RFC 1014, SUN Microsystems Incorporated

Domain Names - Concepts and Facilities, RFC 1034, P. Mockapetris

Domain Names - Implementation and Specification, RFC 1035, P. Mockapetris

Internet Protocol on Network Systems HYPERchannel Protocol Specification, RFC 1044, K. Hardwick, J. Leckashaman

Remote Procedure Call Protocol Specification, RFC 1057, SUN Microsystems Incorporated

Network File System Protocol Specification, RFC 1094, SUN Microsystems Incorporated.

These RFCs are part of a distribution package contained in the **tcpip.arfc(rfc*nnnn*)** data set; where the *nnnn* in the member name is the number of the RFC. Other documents may be obtained from:

SRI International
DDN Network Information Center
Room EJ291
333 Ravenswood Avenue
Menlo Park, CA. 94025
1-800-235-3155

You may also obtain the RFCs from the Internet host with a domain name of **sri-nic.arpa**. Use the FTP command and the appropriate FTP subcommands to retrieve the files. Use a user ID of **guest** and supply a password of **anonymous** to the FTP server on that host. The files to receive are:

< rfc > rfc-index.txt

< rfc > rfc*nnnn*.txt

Glossary

This glossary defines the most common terms associated with TCP/IP communication in an internet environment.

A

access method. A mainframe data management routine that moves data between storage and an I/O device in response to requests made by a program.

active open. The state of a connection that is actively seeking a service.

address. The unique identifier assigned to each device or workstation connected to a network.

address space. The complete range of addresses in memory available to a computer program.

AIX. Acronym for Advanced Interactive Executive, IBM's licensed version of the UNIX operating system.

API. Acronym for Application Program Interface, the formally-defined programming language interface between an IBM system control program or licensed program and the user of the program.

ARP. Acronym for Address Resolution Protocol, a protocol used to dynamically bind an internet address to a hardware address. ARP is implemented on a single physical network, and is limited to networks that support broadcast addressing.

ARPA. Acronym for Advanced Research Projects Agency, the former name for DARPA. See DARPA.

ARPANET. A proprietary TCP/IP-based internetwork funded by United States Department of Defense.

ASCII. Acronym for American National Standard Code for Information Interchange, the standard code, using a coded character set consisting of 7-bit coded characters, used for information exchange among data processing systems, data communication systems, and associated equipment.

B

backbone. In a wide area network, a high speed link to which nodes or data switching exchanges are connected.

baseband. A frequency band that uses the complete bandwidth of a transmission. All the stations on the network must participate in every transmission. See also *broadband.*

block. A string of data elements recorded, processed or transmitted as a unit. The element may be characters, words or physical records.

bridge. A functional unit that connects two LANs that use the same logical link control procedure, but may use different medium access control procedures.

broadband. A frequency band divisible into several narrower bands that uses analog signals, carrier frequencies and multiplexing techniques to allow simultaneous communication by more than one process through a single connection.

broadcast. The transmission of data packets to all nodes on a network or subnetwork simultaneously.

broadcast address. An address that is recognized by all nodes on a network.

bus topology. A network configuration in which only one path is maintained between stations and any data transmitted by a station is available concurrently to all other stations on the link.

C

checksum. The sum of a group of data associated with the group and used for error-checking purposes.

Class A network. An internet network in which the high-order bit of the address is 0. The host number occupies the 3 low-order octets, allowing for 128 class A networks with 16 777 216 host numbers on each network.

Class B network. An internet network in which the high-order bit of the address is 1 and the next high-order bit is 0. The host number occupies the 2 low-order octets, allowing for 16 384 class B networks with 65 536 host numbers on each network.

Class C network. An internet network in which the 2 high-order bits of the address are 1 and the next high-order bit is 0. The host number occupies the low-order octets, allowing for 2 097 152 class C networks with 256 host numbers on each network.

client. (1) A function that requests services from a server, and makes them available to the user. (2) An address space in MVS that is using TCPIP services. (3) A term used in an environment to identify a machine that uses the resources of the network.

client-server relationship. Any process that provides resources to other processes on a network is a *server*. Any process that employs these resources is a *client*. A machine can run client and server processes at the same time.

connection. An association established between functional units for conveying information.

D

daemon. A background process usually started at system initialization that runs continuously and performs a function required by other processes.

DARPA. Acronym for Defense Advanced Projects Research Agency, the United States Department of Defense agency responsible for creating ARPANET. Formerly called ARPA.

datagram. The basic unit of information that is passed across an internet. It consists of one or more data packets.

data set. The major unit of data storage and retrieval in MVS, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

data set organization. The way in which data is arranged within a data set on a mainframe. Only sequential, direct, partitioned, and VSAM data set organizations are supported by the Network File System server feature of TCP/IP for MVS.

DDN. Acronym for Defense Data Network. It is sometimes used to refer to the collection of X.25 networks that include MILNET and ARPANET, but more accurately refers to MILNET and its interconnected military networks.

direct data set. A type of data set used in a mainframe environment for storing data on a random access device that is accessed using a record address.

Distributed Services. A facility that provides a stateful architecture for transparent file sharing, file-level remote mounts, inherited mounts, cross-system file locking, and for local/remote process transparency of Inter-Process Communications message queues. It is supported only on AIX RT and AIX PS/2 systems.

domain name. Part of the naming hierarchy used in an internet. It contains a sequence of names (labels) separated by periods (dots).

domain naming. A hierarchical system for naming network resources.

dotted-decimal notation. A representation for a 32-bit integer consisting of four 8-bit numbers, written in base 10, and separated by periods (dots). Dotted decimal notation is accepted by many Internet application programs (instead of machine names).

DS. See Distributed Services.

E

EBCDIC. Acronym for Extended Binary Coded Decimal Interchange Code, a coded character set consisting of 8-bit coded characters.

entry-sequenced data set. A type of data set used in a mainframe environment. The format consists of logical records sequenced by the time of their arrival. A particular record is located by using the relative byte address (RBA).

ESDS. Acronym for Entry-Sequenced Data Set.

Ethernet. The name given to a local area packet-switched network technology invented in the early 1970s by the Xerox Corporation.

exit. A mechanism that provides an interface from a server application into a function. Exits are used in the Network File System server feature of TCP/IP for MVS to provide RPC services.

F

foreign host. Any host on the network other than the local host.

foreign network. In an internet, any other network interconnected to the local network by one or more intermediate gateways.

foreign node. See *foreign host.*

FTP. Acronym for File Transfer Protocol, a TCP/IP protocol used for transferring files to and from foreign hosts. FTP also provides the capability to access directories. Password protection is provided as part of the protocol.

G

gateway. A functional unit that interconnects computer networks of different architectures and protocols (at the IP layer).

H

hop count. The number of networks through which a datagram passes on the way to its destination node.

host. A computer connected to a network, and providing an access method to that network. A host provides end-user services.

I

ICMP. Acronym for Internet Control Message Protocol. It is included in IP, and handles error and control messages.

IEEE. Acronym for Institute of Electrical and Electronics Engineers.

internet. See *internetwork.*

internetwork. A collection of packet-switched networks that are connected by gateways. They function as single network.

Internet. A specific internetwork that includes ARPANET, MILNET and NSFnet. These networks use the TCP/IP protocol suite.

internet address. The unique 32-bit address identifying a node on an internetwork.

interoperability. The ability of hardware and software from multiple vendors to communicate on a network.

IP. Acronym for Internet Protocol, the TCP/IP layer between the higher-level host-to-host protocol and the local network protocols. IP uses local area network protocols to carry packets, in the form of datagrams, to the next gateway or destination host.

ISO. Acronym for International Standards Organization, an organization of national standards bodies from various countries established to promote development of standards to facilitate international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

IUCV. Acronym for Inter-User Communication Vehicle, a communication mechanism between address spaces.

J

JES. Acronym for Job Entry Subsystem, a system facility for spooling, job queuing, and managing I/O.

K

KB. Kilobyte; 1024 bytes.

kernel. A master program that manages all the physical resources of the computer, including file system management, virtual memory, reading and writing files to disks and tapes, scheduling of processes, printing, and communicating over a network.

key-sequenced data set. A type of data set used in a mainframe environment for sorting data on a random access device. The format consists of an index followed by one or more logical records.

KSDS. Acronym for Key-Sequenced Data Set.

L

LAN. Acronym for Local Area Network, a data network located on the user's premises in which serial transmission is used for direct data communication among data stations.

local host. The computer to which a user's terminal is directly connected.

local network. That portion of a network physically connected to the host without intermediate gateways.

M

MCH. Acronym for Multichannel Link.

mount. (1) The process of accessing a directory from a disk attached to the machine making the mount request (4.2 mount), or to the remote disk on a network (Network File System mount). (2) An operation that associates a group of files on a server with a directory (mount point) on a client to provide transparent access to the files through that directory. The files must be in a hierarchical arrangement.

mount point. A place established in a workstation or server local directory that is used during the transparent accessing of a remote file. Two entries must be created; first, an entry in the /ETC/FSTAB file and, second, an empty directory must be created in a local directory, quite often the /USR directory.

multichannel link. A means of enabling a data terminal equipment (DTE) to have several access channels to the data network over the single circuit.

N

name server. The server used for cross-referencing a name with its corresponding internet address.

national characters. The characters $, #, and @.

NCP. Acronym for Network Control Program, an IBM licensed program that provides communication controller support for single-domain, multiple-domain, and interconnected network capability.

network. An arrangement of nodes and connecting branches.

Network File System. (1) A generic term for a system based on the NFS 3.2 protocol. (2) A facility for sharing files in a heterogeneous environment of machines, operating systems, and networks.

NFS 3.2. A protocol developed by SUN Microsystems Incorporated. It allows computers on a network to access each other's file systems. Once accessed, the file system appears to reside on the local host. NFS 3.2 uses IP.

NJE. Acronym for Network Job Entry, a batch networking application that transmits data between IBM operating systems.

node. (1) In a network, a point at which one or more functional units connect channels or data circuits. (2) In a network topology, the point at an end of a branch.

NPSI. Acronym for NCP Packet Switching Interface, an IBM program product that provides NCP users with the capability of attaching IBM communications controllers to data transmission services that support X.25 interfaces.

O

obey list. The list of user IDs that is authorized to perform privileged functions in the TCPIP address space.

octet. A byte composed of eight binary elements.

OSI. (1) Acronym for Open Systems Interconnection, the interconnection of open systems in accordance with specific ISO standards. (2) The use of standardized procedures to enable the interconnection of data processing systems.

P

packet. A sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole.

passive open. The state of a connection that is prepared to provide a service on demand.

PDN. Acronym for Public Data Network, a network established and operated by a telecommunication administration or by a Recognized Private Operating Agency (RPOA) for the specific purpose of providing circuit-switched, packet-switched, and leased-circuit services to the public.

partitioned data set. See PDS.

PDS. A type of data set used in the mainframe environment. It must be on a direct access volume and consists of members. It has a directory that points to the locations of the various files stored in this data set. Often used to store libraries of programs and macro instructions.

peer. In network architecture, any functional unit that resides in the same layer as another entity.

PING. The process of sending an ICMP Echo Request packet to a host or gateway, with the expectation of receiving a reply.

POP. Acronym for Post Office Protocol, a protocol that allows an AIX RT or AIX PS/2 host to act as the receiver for mail destined for a user of TCP for the PS/2 computer.

portmapper. A server that converts RPC program numbers into port numbers acceptable to the protocol. This server must be running to make RPC calls.

port. (1) An endpoint for communication between devices, generally referring to a physical connection.

(2) A 16-bit number identifying a particular TCP or UDP resource within a given TCP/IP node.

process. (1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. (2) Any operation or combination of operations on data. (3) A function being performed or waiting to be performed. (4) A program in operation. For example, a daemon is a system process that is always running on the system. (If it stops running, you have to start it up.)

PROFS. Acronym for Professional Office Systems, IBM's proprietary integrated office management system used for sending, receiving, and filing electronic mail, and a variety of other office tasks.

protocol. A set of semantic and syntactic rules that defines the behavior of functional units in achieving communication.

R

RACF. Acronym for Resource Access Control Facility, a facility that allows access to data and system components based on authorization levels.

RBA. Acronym for Relative Byte Address, an address which may be used in accessing key sequenced or entry sequenced VSAM data sets.

relative record data set. A type of data set used in the mainframe environment. It must be on a direct access volume and the format consists of one logical record in a fixed-length slot. Each slot has a unique relative record number. Data is placed in a specific slot based on a user-supplied relative record number.

remote host. See *foreign host.*

remote spooling communications subsystem. A VM networking component that provides telecommunication facilities for the transmission of bulk files between VM users and remote stations.

resolver. A program or subroutine that obtains information from a name server for use by the calling program.

RFC. Acronym for Request For Comments, a series of documents that address a broad range of topics affecting internetwork communication. Some RFCs are established as internet standards.

ring topology. A network configuration in which devices are connected by unidirectional transmission links to form a closed path.

router. A device that connects networks at the physical network layer. It is protocol-dependent and connects only networks operating the same protocol. Routers do more than transmit data; they also select the best transmission paths and optimum sizes for packets.

routing table. A list of network numbers and the information needed to route packets to each.

RPC. Acronym for Remote Procedure Call, a facility that a client uses to have a server execute a procedure call. This facility is composed of a library of procedures plus an XDR.

RRDS. Acronym for Relative Record Data Set.

RSCS. Acronym for Remote Spooling Communications Subsystem.

S

segmentation. The process of dividing a unit of data into smaller units in order to send it across a network. Usually this is done at a gateway when the incoming data buffer is too large to be transmitted to the next network.

sequential data set. A type of data set used in the mainframe environment. It must be on a direct access volume and has the records stored and retrieved according to their physical order within the data set.

server. (1) A function that provides services for users. A machine may run client and server processes at the same time. (2) A machine that provides resources to the network. It provides a network service, such as disk storage and file transfer, or a program that uses such a service.

sharing. A term used in a computing environment to refer to utilizing a file on a remote system. It is done by mounting the remote file system, then reading or writing files in that remote system.

SMF. Acronym for System Management Facility, a facility used on the mainframe to log accounting information, which includes processor time, data transfer statistics, as well as user information.

SMTP. Acronym for Simple Mail Transfer Protocol, a TCP/IP application protocol used for transferring mail between users on different systems.

SNA. Acronym for Systems Network Architecture, the description of a logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

socket. (1) An endpoint for communication between processes or applications in the C Socket API of TCP/IP for MVS. (2) A pair consisting of TCP port and IP address, or UDP port and IP address.

star topology. A network configuration in which all nodes are connected to a central controller or computer that transfers data between nodes.

stream. A continuous sequence of data elements transmitted in character or binary-digit form using a defined format.

subnet. A networking scheme that divides a single logical network into smaller physical networks to simplify routing.

subnet address. The portion of the host address that identifies a subnetwork.

subnet mask. A mask used in the IP protocol layer to separate the subnet address from the host address.

SVC. Acronym for Supervisor Call, the macro instruction used by the mainframe to generate a software interrupt. Control is then transferred to a routine that will handle the interrupt processing.

switched virtual circuit. A virtual circuit that is requested by a virtual call. It is released when the virtual circuit is cleared.

system catalog. The highest level catalog on a mainframe that must exist so that the operating system can find data files. It is a VSAM data set and can contain pointers to VSAM data sets, VSAM user catalogs, OS data sets, and OS user catalogs.

T

TCB. Acronym for Transmission Control Block, an internal control block within the TCPIP address space.

TCF. Acronym for Transparent Computing Facility, a facility that allows a cluster of AIX/370 and AIX PS/2 systems to be constructed, allowing centralized administration of user logons, passwords, and system resources across the entire cluster.

TCP. Acronym for Transmission Control Protocol, a stream communication protocol that includes error recovery and flow control.

TCP/IP. Acronym for Transmission Control Protocol/Internet Protocol, a suite of protocols designed to allow communication between networks regardless of the technologies implemented in each network.

Telnet. Terminal Emulation Protocol, a TCP/IP application protocol that allows interactive access to foreign hosts.

token. (1) In a local network, the symbol of authority passed among data stations to indicate the station temporarily in control of the transmission medium. (2) In programming languages, a language construct that by convention represents an elemental unit of meaning.

TFTP. Acronym for Trivial File Transfer Protocol, the TCP/IP standard protocol for file transfer used primarily for communications among PS/2 computers. TFTP allows sending and receiving of files, but does not provide any password protection or directory capability.

TN3270. An informally defined protocol for transmitting 3270 data streams over Telnet.

Token-Ring network. A ring network that allows unidirectional data transmission between data stations by a token passing procedure over one transmission medium so that the transmitted data returns to the transmitting station.

U

UDP. Acronym for User Datagram Protocol, a connectionless datagram protocol that requires minimal overhead, but does not guarantee delivery.

user. Anyone who requires the services of a computing system.

USS. Acronym for Unformatted System Services.

V

VMCF. Acronym for Virtual Machine Communication Facility, a connectionless mechanism for communication between address spaces.

VSAM. Acronym for Virtual Storage Access Method, an access method used on a mainframe to organize data and maintain information about that data in a catalog. VSAM data sets cannot be accessed by any other access method.

Virtual Telecommunications Access Method. An IBM program product that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability. VTAM runs under MVS, VSE, and VM.

virtual circuit. (1) In packet switching, the facilities provided by a network that give the appearance to the

user of an actual connection. (2) A logical connection established between two DTEs.

VTAM. Acronym for Virtual Telecommunications Access Method.

W

WAN. Acronym for Wide Area Network, a network that provides communication services to a geographic area larger than that served by a local area network.

widget. (1) The fundamental data type of the X-Windows Toolkit. (2) An object providing a user-interface abstraction; for example, a Scrollbar widget. It is the combination of an X-Windows window (or subwindow) and its associated semantics.

working directory. A collection of files to be manipulated by an FTP operation.

X

X.25. A recommendation of the Consultative Committee on International Telephony and Telegraphy (CCITT) that defines the interface between data terminal equipment and packet switching networks.

XDR. Acronym for External Data Representation, a standard developed by SUN Microsystems Incorporated for representing data in machine independent format.

X-Windows API. An application program interface designed as a distributed, network-transparent, device independent, multitasking windowing and graphics system.

X Window System. An application developed by the Massachusetts Institute of Technology that incorporates the protocol also used in IBM's X-Windows API.

Index

A

accepting the C socket connection request 3-5
access control, manipulating (X-Windows) 4-14
access modes D-10, D-15
account exit D-28—D-37
  address of client system name D-30
  address of data set name D-31
  address of error message D-30
  address of global storage block D-30
  address of member name D-31
  address of MVS user ID D-30
  address of user storage block D-30
  APF-authorized library D-29
  APF-authorized task D-29
  client group ID number D-30
  client IP address D-30
  client user ID number D-30
  data set D-28
  data set bytes read D-31
  data set bytes written D-31
  data set read count D-31
  data set write count D-31
  logout D-35
  MVSLOGIN command D-28, D-32
  NFSXU4 D-29
  register contents D-29
    address of entry point D-29
    address of save area D-29
    address of the parameter list D-29
    return address D-29
  request active time D-31
  request bytes read from TCP/IP D-31
  request bytes written from TCP/IP D-31
  request code D-30
    system initialization D-30
    system termination D-30
    user data set usage D-30
    user initialization D-30
    user interval expiration D-30
    user request complete D-30
    user termination D-30
  request disk bytes read D-31
  request disk bytes written D-31
  request disk read count D-31
  request disk write count D-31
  request end time D-31
  request procedure number D-31
  request program number D-31
  request start time D-30
  request version number D-31
account exit *(continued)*
  resource utilization D-28
  return code D-30
    invalid request D-30
    processing error D-30
    processing successful D-30
  single thread access D-29
  start of new user session D-32
  storage block D-29, D-32, D-35
  system initialization D-32
  system shutdown D-28
  system startup D-28
  system termination D-35
  USB D-28
  user data set usage D-34
  User Interval Expiration D-34
  user request complete D-33
  user storage block D-32
  user termination D-35
ACF D-1
active open 2-23
Address Resolution Protocol (ARP) 1-4, 1-6
addressing 1-2, 1-4, 1-5—1-6
AddUserNote (Pascal) 2-55, C-2
APF-authorized library D-2
  account exit D-29
  archive exit D-17
  login exit D-2
  security exit D-10
APF-authorized task D-2
  account exit D-29
  archive exit D-17
  login exit D-2
  security exit D-10
archive exit D-16—D-28
  address of client system name D-19
  address of global storage block D-19
  address of message supplied D-19
  address of MVS data set member name D-19
  address of MVS data set name D-19
  address of MVS data set volume name D-19
  address of MVS user ID D-19
  address of user storage block D-19
  APF-authorized library D-17
  APF-authorized task D-17
  archive date D-19
  archive management package D-16
  archive package D-16
  Archive routine D-25
  archive time D-19
  client group ID number D-19

archive exit *(continued)*
client IP address D-19
client user ID number D-19
create request D-25
creation date D-19
data set D-16
archived D-16
create D-25
delete D-24
restoration D-16
unique processing D-24
data set block size D-19
data set organization D-19
direct access data set D-19
indexed sequential data set D-19
partitioned sequential data set D-19
sequential data set D-19
unknown D-19
VSAM D-19
VSAM entry sequenced data set D-19
VSAM key sequenced data set D-19
VSAM relative record data set D-19
data set record format D-19
data set record length D-19
delete request D-24
file size D-19
information from archive requested D-20
installation exit D-16
NFSXU3 D-17
read request D-23
register contents D-17
address of entry point D-17
address of save area D-17
address of the parameter list D-17
return address D-17
request code D-18
create request D-18
delete request D-18
information from archive requested D-18
read request D-18
retrieve from archive requested D-18
system initialization D-18
system termination D-18
write request D-18
retrieve D-23
retrieve from archive requested D-22
return code D-18
access privilege refused D-18
data set deleted from archives D-18
data set does not exist D-18
data set restored, re-issue lookup D-18
device not available D-18
information returned D-18
invalid request D-18
storage block D-17, D-20, D-26
archive exit *(continued)*
system initialization D-20
system shutdown D-16
system startup D-16
system termination D-26
unique processing D-24
unsigned archive time D-19
write request D-24
archive management package D-16
areas (X-Windows)
clearing 4-9
copying 4-9
filling 4-10
ARP 1-6
ARPANET 1-2
associating tables (X-Windows) 4-24
asynchronous communication 2-1
atoms (X-Windows) 4-7

B

batch mailing B-8
BeginTcpIp (Pascal) 2-20
bitmaps, manipulating (X-Windows) 4-19
broadcast address 1-6
BUFFERspaceAVAILABLE notification (Pascal) 2-8
BytesToRead information (Pascal) 2-4

C

C Language 3-1
C programming language 3-3
C socket application program interface 3-1—3-19
library calls 3-4—3-19
accept() 3-5
bind() 3-6
close() 3-8
connect() 3-8
gethostbyaddr() 3-9
gethostbyname() 3-10
gethostname() 3-11
getsockname() 3-11
listen() 3-12
readv() 3-12
read() 3-12
recvfrom() 3-13
recv() 3-13
select() 3-15
sendto() 3-16
send() 3-16
socket() 3-17
writev() 3-18
write() 3-18
quick reference 3-3
software requirements 3-3

character string sizes, querying (X-Windows) 4-11
ClearTimer (Pascal) 2-46
client defined 1-1
color cells, manipulating (X-Windows) 4-8
colormaps, manipulating (X-Windows) 4-7
communication procedures
  TCP (Pascal) 2-23
  UDP (Pascal) 2-38
Compartment information (Pascal) 2-4
Connection information (Pascal) 2-4
connection notification (Pascal) 2-7
CONNECTIONclosing state (Pascal) 2-3
ConnectionState (Pascal) 2-2
CONNECTIONstateCHANGED notification (Pascal) 2-8
CreateTimer (Pascal) 2-46
creating an application 4-3
cursors, manipulating (X-Windows) 4-12
cut and paste buffers, using (X-Windows) 4-18

D

DARPA Internet 3-1
DATA command (SMTP) B-3
data set D-10, D-12, D-14
  access D-10
  archived D-16
  create D-25
  MVS D-10, D-12, D-14
  permanent D-28
  restoration D-16
  unique processing D-24
data set names vii
data structures 3-4
DATAdelivered notification (Pascal) 2-8
datagram sockets 3-1
datagrams 1-4, 1-5
DatasetPassword file specification (Pascal) 2-12
DDName file specification (Pascal) 2-12
DestroyTimer (Pascal) 2-47
direct routing 1-6
display functions (X-Windows) 4-20
displays (X-Windows)
  closing 4-5
  defined 4-2
  opening 4-5
distributed computing 1-1
domain naming 1-5

E

electronic mail B-1
end point for communication, creating 3-17
EndTcpIp (Pascal) 2-21
error handling, default (X-Windows) 4-15
establishing a connection 3-6
events, handling (X-Windows) 4-14
exit routines
  account exit D-28—D-37
  archive exit D-16—D-28
  login exit D-1—D-9
  security exit D-10—D-16
extension routines (X-Windows) 4-23
External Data Representation (XDR) 5-1
external interrupt handling (Pascal) 2-21
EXTERNALinterrupt notification (Pascal) 2-9

F

file specification record (Pascal) 2-12
File Transfer Protocol (FTP) 1-5
fonts (X-Windows)
  freeing 4-10
  loading 4-10
foreign host 1-2
foreign network 1-2
ForeignSocket information (Pascal) 2-5
FRECEIVEerror notification (Pascal) 2-9
FSENDresponse notification (Pascal) 2-9
FullDatasetName file specification (Pascal) 2-12

G

gateways
  defined 1-2
GetHostNumber (Pascal) 2-47
GetHostResol (Pascal) 2-48
GetHostString (Pascal) 2-48
GetIdentity (Pascal) 2-49
GetNextNote (Pascal) 2-17
GetSmsg (Pascal) 2-51
global storage block
  account exit D-29, D-32, D-35
  archive exit D-17, D-20, D-26
  login exit D-2, D-4, D-8
  security exit D-10
graphics contexts, manipulating (X-Windows) 4-8

H

Handle (Pascal) 2-18
HELO command (SMTP) B-4
HELP command (SMTP) B-4
host
  foreign 1-2
  local 1-2
host lookup routines (Pascal) 2-47
host name, returning 3-11

hostent structure 3-9, 3-10
hosts, manipulating (X-Windows) 4-14

I

images, manipulating (X-Windows) 4-18
images, transferring (X-Windows) 4-12
indirect routing 1-6
initialization procedures, TCP/UDP (Pascal) 2-19
initiating a connection on a socket 3-8
installation exit D-16
Inter-User Communication Vehicle (IUCV) A-3
  disabling interrupts A-5—A-6
  external interrupt parameters A-5
  program call sequences A-3—A-4
interface
  TCP/UDP/IP interface (C Sockets) 3-1
internet
  addressing 1-2, 1-4, 1-5—1-6
  defined 1-2
  routing 1-6
Internet Control Message Protocol (ICMP) 1-4
Internet defined 1-2
Internet Protocol (IP) 1-4, 2-1
interoperability defined 1-2
Intrinsics (X-Windows) 4-25
IsLocalAddress (Pascal) 2-50
IsLocalHost (Pascal) 2-50
I/O status, checking 3-15

K

keyboard event functions (X-Windows) 4-17
keyboard settings, manipulating (X-Windows) 4-13

L

lines, drawing (X-Windows) 4-10
LISTENING state (Pascal) 2-3
local host 1-2
local network 1-2
LocalSocket information (Pascal) 2-4
login exit D-1—D-9
  ACF D-1
  address of client system name D-4
  address of GSB D-4
  address of message supplied D-4
  address of MVS group name D-4
  address of MVS new user password D-4
  address of MVS user ID D-4
  address of MVS user password D-4
  address of USB D-4
  APF-authorized library D-2
  APF-authorized task D-2
  authentication type D-4
    DES credentials D-4
login exit *(continued)*
  authentication type *(continued)*
    short hand UNIX credentials D-4
    UNIX type credentials D-4
  cleanup D-1
  client group ID number D-4
  client IP address D-4
  client user ID number D-4
  logout D-1, D-7
    forced D-1
  logout requested D-8
  mvslogin command D-1, D-5
  mvslogout command D-1
  new password D-6
  new password processing D-1
  new password supplied D-6
  NFSXU1 D-2
  password checking D-1
  RACF D-1
  register contents D-2
    address of entry point D-2
    address of save area D-2
    address of the parameter list D-2
    return address D-2
  request code D-3
    logout requested D-3
    new password D-3
    new user session D-3
    system initialization D-3
    system termination D-3
    user login request D-3
    user timed out D-3
  return code D-3
    forced logout D-3
    invalid password D-3
    invalid user ID D-3
    login successful D-3
    password expired D-3
    password required D-3
    request invalid D-3
    timeout interval extended D-3
    unauthorized user D-3
    user ID required D-3
  session timeout value D-4
  start of new user session D-5
    return code D-5
  storage block D-2, D-4, D-5, D-8
  system initialization codes and fields D-4
  system termination D-8
  time-out interval D-1, D-7
  user login request D-6
  user time delta D-4
  user time out D-7
  user verification D-1

logout D-1, D-7
  forced D-1

M

MAIL FROM command (SMTP) B-5
MemberName file specification (Pascal) 2-12
MILNET 1-2
MonCommand (Pascal) 2-35
monitor procedures (Pascal) 2-35
MonQuery (Pascal) 2-36
MVS defined iv
mvslogin command D-5
  account exit D-28, D-32
  login exit D-1
mvslogout command
  login exit D-1

N

network
  classes 1-6
  defined 1-1
  foreign 1-2
  gateways 1-2
  local 1-2
  peer-to-peer 1-1
  protocols 1-3
  software 1-3
  topology examples 1-1
NFSXU1 D-2
NFSXU2 D-10
NFSXU3 D-17
NFSXU4 D-29
node defined 1-1
NONEXISTENT state (Pascal) 2-3
NOOP command (SMTP) B-5
notifications (Pascal) 2-17
NotificationTag (Pascal) 2-7
NSFnet 1-2

O

OPEN state (Pascal) 2-3
OpenAttemptTimeout information (Pascal) 2-4

P

packets 1-3, 1-4
Pascal API assembler calls C-1—C-2
Pascal APIs 2-1—2-57
  data structures 2-2
    connection information record 2-4
    ConnectionState 2-2
    file specification record 2-12
    notification record 2-5
Pascal APIs *(continued)*
  procedure calls 2-12—2-55
    handling external interrupts 2-21
    host lookup routines 2-47
    monitor procedures 2-35
    notifications 2-17
    other routines 2-51
    Ping interface 2-34
    raw IP interface 2-42
    summary 2-13—2-17
    TCP communication procedures 2-23
    TCP/UDP initialization 2-19
    TCP/UDP termination 2-21
    timer routines 2-45
    UDP communication procedures 2-38
  return codes 2-55—2-57
  software requirements 2-2
passive open 2-23
peer-to-peer networks 1-1
permission D-10, D-15
Ping interface (Pascal) 2-34
PingRequest (Pascal) 2-34
PINGresponse notification (Pascal) 2-9
pixmaps (X-Windows)
  creating 4-8
  freeing 4-8
pointers to objects, returning 3-9, 3-10
Precedence information (Pascal) 2-4
program call numbers A-2
properties (X-Windows) 4-7
protocol notification (Pascal) 2-7
protocols, network
  Address Resolution Protocol 1-4
  defined 1-3
  File Transfer Protocol (FTP) 1-5
  Internet Control Message Protocol (ICMP) 1-4
  Internet Protocol (IP) 1-4
  Simple Mail Transfer Protocol (SMTP) 1-5
  Telnet Protocol (Telnet) 1-5
  Transmission Control Protocol (TCP) 1-4
  User Datagram Protocol (UDP) 1-5

Q

QUEU command (SMTP) B-5
queuing a connection request 3-12
QUIT command (SMTP) B-6

R

RACF D-1
raw IP interface (Pascal) 2-42
RawIpClose (Pascal) 2-45
RawIpOpen (Pascal) 2-42

RAWIPpacketsDELIVERED notification (Pascal) 2-10
RawIpReceive (Pascal) 2-43
RawIpSend (Pascal) 2-44
RAWIPspaceAVAILABLE notification (Pascal) 2-10
RCPT TO command (SMTP) B-6
ReadXlateTable (Pascal) 2-51
receiving messages from a socket 3-13
RECEIVINGonly state (Pascal) 2-3
regions, manipulating (X-Windows) 4-17
register contents
  account exit D-29
  archive exit D-17
  login exit D-2
  security exit D-11
related protocol specifications F-1
Remote Procedure Calls (RPCs) 5-1—5-32
  interface described 5-1
  library calls 5-5—5-32
    authnone_create 5-6
    authunix_create 5-6
    authunix_create_default 5-6
    auth_destroy 5-5
    callrpc 5-6
    clntraw_create 5-10
    clnttcp_create 5-10
    clntudp_create 5-11
    clnt_call 5-7
    clnt_destroy 5-8
    clnt_freeres 5-8
    clnt_geterr 5-8
    clnt_pcreateerror 5-9
    clnt_perrno 5-9
    clnt_perror 5-9
    get_myaddress 5-11
    mvs_xdr_enum 5-11
    pmap_getmaps 5-12
    pmap_getport 5-12
    pmap_rmtcall 5-13
    pmap_set 5-13
    pmap_unset 5-14
    registerrpc 5-14
    rpc_createerr 5-15
    svcerr_auth 5-18
    svcerr_decode 5-18
    svcerr_noproc 5-19
    svcerr_noprog 5-19
    svcerr_progvers 5-19
    svcerr_systemerr 5-19
    svcerr_weakauth 5-20
    svcraw_create 5-20
    svctcp_create 5-20
Remote Procedure Calls (RPCs) *(continued)*
  library calls *(continued)*
    svcudp_create 5-21
    svc_destroy 5-15
    svc_fds 5-15
    svc_freeargs 5-15
    svc_getargs 5-16
    svc_getcaller 5-16
    svc_getreq 5-16
    svc_register 5-17
    svc_run 5-17
    svc_sendreply 5-17
    svc_unregister 5-18
    xdr_accepted_reply 5-21
    xdr_array 5-21
    xdr_authunix_parms 5-22
    xdr_bool 5-22
    xdr_bytes 5-22
    xdr_callhdr 5-23
    xdr_callmsg 5-23
    xdr_double 5-23
    xdr_enum 5-24
    xdr_float 5-25
    xdr_inline 5-25
    xdr_int 5-26
    xdr_long 5-26
    xdr_opaque 5-26
    xdr_opaque_auth 5-27
    xdr_pmap 5-27
    xdr_pmaplist 5-27
    xdr_reference 5-28
    xdr_rejected_reply 5-28
    xdr_replymsg 5-28
    xdr_short 5-29
    xdr_string 5-29
    xdr_union 5-30
    xdr_u_int 5-29
    xdr_u_long 5-30
    xdr_u_short 5-30
    xdr_void 5-31
    xdr_wrapstring 5-31
    xprt_register 5-31
    xprt_unregister 5-31
  quick reference 5-1—5-4
  software requirements 5-1
Requests For Comment (RFCs) F-1
resource manager, using (X-Windows) 4-19
resource sharing 1-1

resource utilization D-28
RESOURCESavailable notification (Pascal) 2-10
retrieve D-23, D-24
return codes, Pascal 2-55—2-57
routing 1-4
  direct 1-6
  indirect 1-6
RSET command (SMTP) B-6
RTcpExtRupt (Pascal) C-1
RTcpVmcfRupt (Pascal) C-1
Running an Application 4-4

S

saving a socket name 3-11
SayCalRe (Pascal) 2-52
SayConSt (Pascal) 2-52
SayIntAd (Pascal) 2-53
SayIntNum (Pascal) 2-53
SayNotEn (Pascal) 2-54
SayPorTy (Pascal) 2-54
SayProTy (Pascal) 2-54
screen saver, controlling (X-Windows) 4-14
security exit D-10—D-16
  access allowed D-11
    access denied D-11
    invalid request D-11
    permissions returned D-11
  access modes D-10, D-15
    allocate D-10
    getting access mode D-10
    read D-10
    write D-10
  address of catalog data set name D-12
  address of catalog volume name D-12
  address of client system name D-11
  address of data set volume name D-12
  address of GSB D-12
  address of MVS data set member name D-12
  address of MVS data set name D-12
  address of MVS user ID D-11
  address of USB D-12
  APF-authorized library D-10
  APF-authorized task D-10
  client group ID number D-11
  client IP address D-11
  client user ID number D-11
  data set D-10, D-12, D-14
    access D-10
    MVS D-10, D-12, D-14
  NFSXU2 D-10
  permission D-10, D-15
  permissions returned D-12
    allocate allowed D-12
    none D-12
    read allowed D-12
security exit *(continued)*
  permissions returned *(continued)*
    write allowed D-12
  register contents D-11
    address of entry point D-11
    address of save area D-11
    address of the parameter list D-11
    return address D-11
  request code D-11
    return security permissions D-11
    validate allocate request D-11
    validate read request D-11
    validate write request D-11
  return security permissions D-15
  storage block D-10
  validate allocate request D-13
  validate read request D-14
  validate write request D-14
Security information (Pascal) 2-4
sending mail to TCP network recipient B-8
sending messages to a socket 3-16
SENDINGonly state (Pascal) 2-3
server defined 1-1
SETSMSG command A-1
SetTimer (Pascal) 2-46
Simple Mail Transfer Protocol (SMTP) 1-5
SMSGreceived notification (Pascal) 2-10
SMTP address space interface B-1—B-8
  batch SMTP command file format B-1
  command responses B-1
  commands B-3—B-8
    DATA B-3
    HELO B-4
    HELP B-4
    MAIL FROM B-5
    NOOP B-5
    QUEU B-5
    QUIT B-6
    RCPT TO B-6
    RSET B-6
    TICK B-7
    VERB B-7
    VRFY B-7
  example B-8
  JES spool interface B-1
  network interface B-1
  path_address syntax B-2
  unimplemented commands B-8
  verbose mode B-2
SMTPNOTE command B-1
socket 4-1
socket API protocol 4-3
sockets 3-1—3-19
  datagram 3-1
  defined 3-1

sockets *(continued)*
stream 3-1
software requirements
C Socket interface 3-3
RPC interface 5-1
TCP/UDP/IP interface (Pascal) 2-2
X-Windows interface 4-1
software, network 1-3
Special Messages (Smsgs) A-1
specifications
protocol F-1
SRI International 1-5
StartTcpNotice (Pascal) 2-20
stream sockets 3-1
structures
hostent 3-9, 3-10
subnets 1-6
synchronization (X-Windows)
disabling 4-15
enabling 4-15
syntax diagrams, how to read v—vii

T

TcpAbort (Pascal) 2-32
TcpClose (Pascal) 2-31
TcpExtRupt (Pascal) 2-22
TcpFReceive (Pascal) 2-28
TcpFSend (Pascal) 2-25
TcpNameChange (Pascal) 2-19
TcpOpen (Pascal) 2-23
TcpReceive (Pascal) 2-28
TcpSend (Pascal) 2-25
TcpStatus (Pascal) 2-33
TcpVmcfRupt (Pascal) 2-22
TcpWaitOpen (Pascal) 2-23
TcpWaitReceive (Pascal) 2-28
TcpWaitSend (Pascal) 2-25
TCP/IP
See Transmission Control Protocol/Internet Protocol (TCP/IP)
Telnet Protocol (Telnet) 1-5
termination procedure, TCP/UDP (Pascal) 2-21
text, drawing (X-Windows) 4-11
TICK command (SMTP) B-7
time-out interval D-1, D-7
timer routines (Pascal) 2-45
TIMERexpired notification (Pascal) 2-10
toolkit (X-Windows) 4-25
topology, network 1-1
Transmission Control Protocol (TCP) 1-4, 2-1
Transmission Control Protocol/Internet Protocol (TCP/IP)
addressing 1-2, 1-4, 1-5—1-6
example of using 1-7
Transmission Control Protocol/Internet Protocol (TCP/IP) *(continued)*
protocols 1-3—1-5
routing 1-6
software 1-3
TRYINGtoOPEN state (Pascal) 2-3

U

UdpClose (Pascal) 2-41
UDPdatagramDELIVERED notification (Pascal) 2-10
UDPdatagramSPACEavailable notification (Pascal) 2-11
UdpNReceive (Pascal) 2-40
UdpOpen (Pascal) 2-38
UdpReceive (Pascal) 2-41
UDPresourcesAVAILABLE notification (Pascal) 2-11
UdpSend (Pascal) 2-39
UnackedBytes information (Pascal) 2-4
Unhandle (Pascal) 2-18
URGENTpending notification (Pascal) 2-11
User Datagram Protocol (UDP) 1-5, 2-1
user storage block D-32
account exit D-29
archive exit D-17
login exit D-2, D-5
security exit D-10
USERdefinedNOTIFICATION (Pascal) 2-11
USERdeliversLINE notification (Pascal) 2-11
USERwantsATTENTION notification (Pascal) 2-11

V

VERB command (SMTP) B-7
Virtual Machine Communication Facility (VMCF) A-1—A-28
aborting a TCP connection A-16
closing a TCP connection A-15
closing a UDP port A-16
command files A-18
determining whether an address is local A-18
disabling interrupts A-5—A-6
dropping an IP protocol A-21
ending TCP/IP communication A-12
external interrupt parameters A-5
identifying your IP protocol A-20
interrupt header fields A-10
interrupts described A-22—A-28
locating the CVT A-3
obtaining TCP connection status A-16
opening a TCP/IP connection A-12
opening a UDP port A-17
parameter list fields A-10
parameter list structure A-7—A-10
CALLCODE values (initiation) A-8
CALLCODE values (notification) A-8

Virtual Machine Communication Facility (VMCF) *(continued)*
parameter list structure *(continued)*
connection information record format A-9
connection state values A-9
fields A-7
miscellaneous constants A-10
notification mask values A-9
program call numbers A-2
program call sequences A-3—A-4
receiving raw IP packets A-22
receiving TCP data (FRECEIVEtcp) A-14
receiving TCP data (RECEIVEtcp) A-15
receiving UDP data A-18
sending an ICMP echo request A-20
sending raw IP packets A-21
sending TCP data A-13
sending UDP data A-17
Special Messages (Smsgs) A-1
specifying notifications to receive A-11
starting TCP/IP communication A-11
status information A-19
when to use A-1
visual types, querying (X-Windows) 4-18
VRFY B-7

W

well-known application protocol 3-1
well-known port numbers 3-2
well-known services 3-2
widgets (X-Windows) 4-27
window manager functions, handling (X-Windows) 4-12
window property data structures (X-Windows) 4-24
windows (X-Windows)
changing attributes 4-6
communicating with window managers 4-16
creating 4-5
destroying 4-5
manipulating 4-5
manipulating properties 4-7
obtaining information 4-6
setting selections 4-7

X

X Defaults 4-3
X Protocol 4-2
X Server 4-1, 4-2
X-Windows interface 4-1—4-27
creating an application 4-3
EBCDIC-ASCII Translation 4-3
example of an application E-13
how it works 4-1
X-Windows interface *(continued)*
quick reference 4-4—4-27
associating tables 4-24
atoms 4-7
changing window attributes 4-6
clearing areas 4-9
closing displays 4-5
communicating with window managers 4-16
controlling the screen saver 4-14
copying areas 4-9
creating pixmaps 4-8
creating windows 4-5
destroying windows 4-5
disabling synchronization 4-15
display functions 4-20
drawing lines 4-10
drawing text 4-11
enabling synchronization 4-15
extension routines 4-23
filling areas 4-10
freeing fonts 4-10
freeing pixmaps 4-8
handling events 4-14
handling window manager functions 4-12
keyboard event functions 4-17
loading fonts 4-10
manipulating access control 4-14
manipulating bitmaps 4-19
manipulating color cells 4-8
manipulating colormaps 4-7
manipulating cursors 4-12
manipulating graphics contexts 4-8
manipulating hosts 4-14
manipulating images 4-18
manipulating keyboard settings 4-13
manipulating regions 4-17
manipulating window properties 4-7
manipulating windows 4-5
obtaining window information 4-6
opening displays 4-5
properties 4-7
querying character string sizes 4-11
querying visual types 4-18
setting window selections 4-7
toolkit 4-25
transferring images 4-12
using cut and paste buffers 4-18
using default error handling 4-15
using the resource manager 4-19
running an application 4-4
software requirements 4-1
X Defaults 4-3
XDR 5-1
XID (X-Windows) 4-24

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s): Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name _____

Company or Organization _____

Address _____

_____

SC09-1261-00

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 6R1T
180 Kost Road
Mechanicsburg, Pennsylvania 17055

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

SC09-1261-00

Printed in U.S.A.