

z/VM
7.4

TCP/IP Programmer's Reference



Note:

Before you use this information and the product it supports, read the information in [“Notices” on page 359](#).

This edition applies to version 7, release 4 of IBM® z/VM® (product number 5741-A09) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2024-09-18

© **Copyright International Business Machines Corporation 1987, 2024.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures.....	xiii
Tables.....	xv
About This Document.....	xix
Intended Audience.....	xix
Conventions and Terminology.....	xix
How the Term “internet” Is Used in This Document.....	xix
How Numbers Are Used in This Document.....	xix
Syntax, Message, and Response Conventions.....	xx
Where to Find More Information.....	xxii
Links to Other Documents and Websites.....	xxiii
How to provide feedback to IBM.....	xxv
Summary of Changes for z/VM: TCP/IP Programmer's Reference.....	xxvii
SC24-6332-74, z/VM 7.4 (September 2024).....	xxvii
SC24-6332-73, z/VM 7.3 (September 2023).....	xxvii
SC24-6332-73, z/VM 7.3 (September 2022).....	xxvii
SC24-6332-03, z/VM 7.2 (December 2021).....	xxvii
SC24-6332-03, z/VM 7.2 (March 2021).....	xxviii
SC24-6332-03, z/VM 7.2 (September 2020).....	xxviii
Chapter 1. z/VM C Socket Application Programming Interface.....	1
TCP/IP Network Communication.....	2
Transport Protocols.....	2
What is a Socket?.....	2
Address Families.....	3
Socket Types.....	4
Domain-specific Socket Addresses.....	5
Client/Server Conversation.....	8
Server Perspective for AF_INET.....	9
Client Perspective for AF_INET.....	11
Typical TCP Socket Session.....	11
Typical UDP Socket Session.....	12
Locating the Server's Port.....	13
Network Application Example.....	14
z/VM C Socket Implementation.....	18
Header Files.....	18
Multithreading.....	19
POSIX Signals and Thread Cancellation.....	20
Sockets and Their Relationship to Other POSIX Functions.....	20
Secure Connection Considerations.....	21
Miscellaneous Implementation Notes.....	22
Incompatibilities with the VM TCP/IP C Sockets Library.....	23
Incompatibilities with z/OS and OS/390 C Sockets.....	25
Incompatibilities with the Berkeley Socket Implementation.....	25
Compiling and Linking a Sockets Program.....	26
Compiling and Linking a z/VM C Sockets Program.....	26

Compiling and Linking a TCP/IP C Sockets Program.....	28
Running a Sockets Program.....	29
Preparing to Run a Sockets Program.....	29
Using Environment Variables.....	30
Running a Program Residing in the BFS.....	31
Running a Program Residing on an Accessed Minidisk or SFS Directory.....	32
C Sockets Quick Reference.....	32
TCP Client Program.....	35
TCP Server Program.....	36
UDP Client Program.....	38
UDP Server Program.....	38
Chapter 2. TCP/UDP/IP API (Pascal Language).....	41
Software Requirements.....	41
Data Structures.....	41
Connection State.....	42
Connection Information Record.....	43
Socket Record.....	44
Notification Record.....	45
File Specification Record.....	53
Using Procedure Calls.....	53
Notifications.....	54
TCP/UDP Initialization Procedures.....	54
TCP/UDP Termination Procedure.....	55
Handling External Interrupts.....	55
TCP Communication Procedures.....	55
Ping Interface.....	57
Monitor Procedures.....	57
UDP Communication Procedures.....	57
Raw IP Interface.....	58
Timer Routines.....	58
Host Lookup Routines.....	58
Other Routines.....	59
Procedure Calls.....	60
AddUserNote.....	60
BeginTcpIp.....	60
ClearTimer.....	60
CreateTimer.....	61
DestroyTimer.....	61
EndTcpIp.....	61
GetHostNumber.....	62
GetHostResol.....	62
GetHostString.....	62
GetIdentity.....	63
GetNextNote.....	63
GetSmsg.....	64
Handle.....	64
IsLocalAddress.....	65
IsLocalHost.....	65
MonCommand.....	66
MonQuery.....	67
NotifyIo.....	68
PingRequest.....	69
QueryTLS.....	70
RawIpClose.....	70
RawIpOpen.....	71
RawIpReceive.....	72

RawIpSend.....	72
ReadXlateTable.....	73
RTcpExtRupt.....	74
RTcpVmcfRupt.....	74
SayCalRe.....	75
SaySslRe.....	75
SayConSt.....	75
SayIntAd.....	76
SayIntNum.....	76
SayNotEn.....	76
SayPorTy.....	77
SayProTy.....	77
SetTimer.....	77
StartTcpNotice.....	78
Tcp6Open and Tcp6WaitOpen.....	79
Tcp6Status.....	81
TcpAbort.....	81
TcpClose.....	82
TcpExtRupt.....	83
TcpFReceive, TcpReceive, and TcpWaitReceive.....	83
TcpFSend, TcpSend, and TcpWaitSend.....	86
TcpNameChange.....	88
TcpOpen and TcpWaitOpen.....	88
TcpOption.....	90
TcpSCertData.....	91
TcpSClient.....	94
TcpSClose.....	98
TcpSServer.....	98
TcpSStatus.....	99
TcpStatus.....	100
TcpVmcfRupt.....	101
Udp6Open.....	102
Udp6Send.....	102
UdpClose.....	103
UdpNReceive.....	104
UdpOpen.....	104
UdpReceive.....	105
UdpSend.....	106
Unhandle.....	107
UnNotifyIo.....	107
Sample Pascal Program.....	108

Chapter 3. Virtual Machine Communication Facility Interface..... 113

General Information.....	113
Data Structures.....	113
VMCF Functions.....	115
VMCF TCPIP Communication CALLCODE Requests.....	115
VMCF TCPIP Communication CALLCODE Notifications.....	117
TCP/UDP/IP Initialization and Termination Procedures.....	119
BEGINtcpIPservice.....	119
ENDtcpIPservice.....	119
HANDLEnotice.....	119
TCP CALLCODE Requests.....	120
CLOSEtcp.....	120
FRECEIVEtcp.....	120
OPENTcp.....	121
OPTIONtcp.....	122

RECEIVtcp.....	123
SENDtcp and FSENDtcp.....	123
STATUStcp.....	124
TLSSCERTDATAREQtcp	124
TLSSCLIENTtcp.....	124
TLSSCLOSEtcp.....	125
TLSSSERVERtcp.....	125
TLSSSTATUStcp.....	126
V6OPENTcp.....	126
V6STATUStcp.....	127
UDP CALLCODE Requests.....	127
CLOSEudp.....	127
NRECEIVEudp.....	128
OPENudp.....	128
SENDudp.....	128
V6OPENudp.....	129
V6SENDudp.....	129
IP CALLCODE Requests.....	130
CLOSErawip.....	130
OPENrawip.....	130
RECEIVERawip.....	130
SENDrawip.....	131
CALLCODE System Queries.....	131
IshostLOCAL.....	131
MONITORcommand.....	132
MONITORquery.....	132
PINGreq.....	133
TLSQuery.....	133
CALLCODE Notifications.....	134
ACTIVEprobe.....	134
BUFFERspaceAVAILABLE.....	134
CERTdataCOMPLETE.....	134
CLEARtextRESUMED.....	135
CONNECTIONstateCHANGED.....	135
DATAdelivered.....	136
DUMMYprobe.....	136
PINGresponse.....	136
QUERYtlsCOMPLETE.....	136
RAWIPpacketsDELIVERED.....	137
RAWIPspaceAVAILABLE.....	137
READYforHANDSHAKE.....	137
RESOURCESavailable.....	138
SECUREhandshakeCOMPLETE.....	138
UDPdatagramDELIVERED.....	138
UDPdatagramSPACEavailable.....	139
UDPresourcesAVAILABLE.....	139
URGENTpending.....	139

Chapter 4. Inter-User Communication Vehicle Sockets..... 141

Prerequisite Knowledge.....	141
Available Functions.....	141
Socket Programming with IUCV.....	141
Preparing to use the IUCV Socket API.....	142
Establishing an IUCV connection to TCP/IP.....	143
Initializing the IUCV Connection.....	143
Severing the IUCV Connection.....	144
Sever by the Application.....	145

Sever by TCP/IP.....	145
Issuing Socket Calls.....	145
Overlapping Socket Requests.....	146
TCP/IP Response to an IUCV Request.....	147
Encrypting Data on an IUCV Socket.....	147
Cancelling a Socket Request.....	147
IUCV Socket Call Syntax.....	148
IUCV Socket Calls.....	159
ACCEPT.....	159
BIND.....	160
CANCEL and CANCEL2.....	161
CLOSE.....	162
CONNECT.....	162
FCNTL.....	163
GETCLIENTID.....	164
GETHOSTID.....	165
GETHOSTNAME.....	165
GETPEERNAME.....	166
GETSOCKNAME.....	167
GETSOCKOPT.....	168
GIVESOCKET.....	169
IOCTL.....	170
LISTEN.....	173
MAXDESC.....	174
READ, READV.....	174
RECV, RECVFROM, RECVMSG.....	175
SELECT, SELECTEX.....	176
SEND.....	178
SENDMSG.....	179
SENDTO.....	180
SETSOCKOPT.....	181
SHUTDOWN.....	182
SOCKET.....	183
TAKESOCKET.....	184
WRITE, WRITEV.....	185
LASTERRNO.....	186

Chapter 5. Remote Procedure Calls..... 187

The RPC Interface.....	187
Portmapper.....	190
Contacting Portmapper.....	190
Target Assistance.....	190
RPCGEN Command.....	190
enum clnt_stat Structure.....	191
Porting.....	192
Accessing System Return Messages.....	192
Printing System Return Messages.....	192
Enumerations.....	192
Compiling, Linking, and Running an RPC Program.....	192
RPC Global Variables.....	193
rpc_createerr.....	193
svc_fds.....	193
svc_fdset.....	193
Remote Procedure Calls and External Data Representation.....	194
auth_destroy().....	194
authnone_create().....	194
authunix_create().....	194

authunix_create_default()	195
callrpc()	195
clnt_broadcast()	196
clnt_call()	197
clnt_control()	198
clnt_create()	198
clnt_destroy()	199
clnt_freeres()	199
clnt_geterr()	200
clnt_pcreateerror()	200
clnt_perrno()	201
clnt_perror()	201
clnt_spccreateerror()	201
clnt_sperrno()	202
clnt_sperror()	202
clntraw_create()	203
clnttcp_create()	203
clntudp_create()	204
get_myaddress()	205
getrpcport()	205
pmap_getmaps()	205
pmap_getport()	206
pmap_rmtcall()	206
pmap_set()	207
pmap_unset()	208
registerrpc()	208
svc_destroy()	209
svc_freeargs()	210
svc_getargs()	210
svc_getcaller()	210
svc_getreq()	211
svc_getreqset()	211
svc_register()	212
svc_run()	212
svc_sendreply()	213
svc_unregister()	213
svcerr_auth()	213
svcerr_decode()	214
svcerr_noproc()	214
svcerr_noprogram()	215
svcerr_progvers()	215
svcerr_systemerr()	215
svcerr_weakauth()	216
svcrw_create()	216
svctcp_create()	216
svcudp_create()	217
xdr_accepted_reply()	217
xdr_array()	218
xdr_authunix_parms()	218
xdr_bool()	219
xdr_bytes()	219
xdr_callhdr()	220
xdr_callmsg()	220
xdr_double()	220
xdr_enum()	221
xdr_float()	222
xdr_inline()	222
xdr_int()	223

xdr_long()	223
xdr_opaque()	224
xdr_opaque_auth()	224
xdr_pmap()	224
xdr_pmaplist()	225
xdr_pointer()	225
xdr_reference()	226
xdr_rejected_reply()	226
xdr_replymsg()	227
xdr_short()	227
xdr_string()	228
xdr_u_int()	228
xdr_u_long()	228
xdr_u_short()	229
xdr_union()	229
xdr_vector()	230
xdr_void()	231
xdr_wrapstring()	231
xdrmem_create()	231
xdrrec_create()	232
xdrrec_endofrecord()	232
xdrrec_eof()	233
xdrrec_skiprecord()	233
xdrstdio_create()	233
xprt_register()	234
xprt_unregister()	234
Sample RPC Programs	234
Running the Geneserv server and Genesend client	235
Running the Rawex program	235
RPC Genesend Client	236
RPC Geneserv Server	236
RPC Rawex Raw Data Stream	238

Chapter 6. SNMP Agent Distributed Programming Interface.....241

SNMP Agents and Subagents	241
Processing DPI Requests	241
Processing a GET Request	242
Processing a SET Request	243
Processing a GET_NEXT Request	243
Processing a REGISTER Request	244
Processing a TRAP Request	244
Compiling and Linking	244
SNMP DPI Reference	244
DPI Library Routines	245
DPIdebug()	245
fDPIparse()	245
mkDPIlist()	246
mkDPIregister()	246
mkDPIresponse()	247
mkDPIset()	248
mkDPItrap()	249
mkDPItrape()	249
Example of an Extended Trap	250
pDPIpacket()	251
query_DPI_port()	252
Sample SNMP DPI Client Program	252
The DPISAMPLE Program (Sample DPI Subagent)	253

DPISAMPLE TABLE.....	255
Client Sample Program.....	255
Compiling and Linking the DPISAMPLE.C Source Code.....	270
Chapter 7. SMTP Virtual Machine Interfaces.....	273
SMTP Transactions.....	273
SMTP Commands.....	273
HELO.....	274
EHLO.....	274
MAIL FROM.....	275
RCPT TO.....	276
DATA.....	276
RSET.....	277
QUIT.....	277
NOOP.....	277
HELP.....	277
QUEUE.....	278
VERFY.....	280
EXPN.....	280
VERB.....	281
TICK.....	281
SMTP Command Example.....	281
SMTP Command Responses.....	282
Path Address Modifications.....	283
Batch SMTP Command Files.....	283
Batch SMTP Examples.....	283
Sending Mail to a TCP Network Recipient.....	284
Querying SMTP Delivery Queues.....	284
SMTP Exit Routines.....	285
Client Verification Exit.....	285
Built-in Client Verification Function.....	285
Client Verification Exit Parameter Lists.....	286
Using the Mail Forwarding Exit.....	291
Mail Forwarding Exit Parameter Lists.....	292
Using the SMTP Command Exit.....	297
SMTP Command Exit Parameter Lists.....	298
Chapter 8. Telnet Exits.....	305
Telnet Session Connection Exit.....	305
Telnet Exit Parameter List.....	305
Sample Exit.....	306
Telnet Printer Management Exit.....	306
Telnet Printer Management Exit Parameter List.....	307
Sample Exit.....	307
Chapter 9. FTP Server Exit.....	309
The FTP Server Exit.....	309
Sample Exit.....	309
Audit Processing.....	309
Audit Processing Parameter List.....	310
Audit Processing Parameter Descriptions.....	311
Return Codes from Audit Processing.....	312
General Command Processing.....	312
General Command Processing Parameter List.....	313
General Command Processing Parameter Descriptions.....	313
Return Codes from General Command Processing.....	315
Change Directory Processing.....	315

Change Directory Processing Parameter List.....	316
Chapter 10. Remote authorization and auditing through LDAP.....	319
Using remote authorization and auditing.....	319
Setting up authorization for working with remote services.....	320
Remote authorization extended operation.....	320
Remote authorization extended operation response codes.....	322
Remote authorization audit controls.....	324
Remote auditing extended operation.....	324
Remote auditing extended operation response codes.....	328
Remote audit controls.....	330
Chapter 11. Building an LDAP Server Plug-in.....	333
Steps for writing an LDAP plug-in.....	333
Note about LDAP support on z/VM.....	334
Appendix A. TCPLOAD EXEC.....	335
Using TCPLOAD.....	335
Appendix B. Pascal Return Codes.....	337
Explanatory Notes.....	340
Appendix C. C API System Return Codes.....	341
Appendix D. Well-Known Port Assignments.....	345
TCP Well-Known Port Assignments.....	345
UDP Well-Known Port Assignments.....	346
Appendix E. Related Protocol Specifications.....	349
Appendix F. Abbreviations and acronyms.....	355
Notices.....	359
Programming Interface Information.....	360
Trademarks.....	360
Terms and Conditions for Product Documentation.....	361
IBM Online Privacy Statement.....	361
Bibliography.....	363
Where to Get z/VM Information.....	363
z/VM Base Library.....	363
z/VM Facilities and Features.....	364
Prerequisite Products.....	366
Related Products.....	366
Other TCP/IP Related Publications.....	366
Index.....	367

Figures

1. An Electrical Analogy Showing the Socket Concept.....	3
2. A Typical Stream Socket Session.....	12
3. A Typical Datagram Socket Session.....	13
4. An Application Using <code>socket()</code>	14
5. An Application Using <code>bind()</code>	14
6. An Application Using <code>listen()</code>	14
7. An Application Using <code>connect()</code>	15
8. A <code>connect()</code> Function Using <code>gethostbyname()</code>	15
9. An Application Using <code>accept()</code>	15
10. An Application Using <code>send()</code> and <code>recv()</code>	16
11. An Application Using <code>sendto()</code> and <code>recvfrom()</code>	16
12. An Application Using <code>select()</code>	17
13. An Application Using <code>ioctl()</code>	17
14. An Application Using <code>close()</code>	17
15. Pascal Declaration of Connection State Type.....	42
16. Pascal Declaration of Connection Information Record.....	43
17. IPv6 Pascal Declaration of Connection Information Record.....	44
18. Pascal Declaration of Socket Type.....	44
19. IPv6 Pascal Declaration of Socket Type.....	45
20. Notification Record (Part 1 of 2).....	46
21. Notification Record (Part 2 of 2).....	47
22. Pascal Declaration of File Specification Record.....	53
23. Monitor Query Record.....	68

24. Assembler Format of the VMCF Parameter List Fields.....	113
25. Equates for Notification Mask in the HANDLEnotice Call.....	120
26. Assembler Format of the Connection Information Record for VM.....	122
27. Miscellaneous Assembler Constants.....	122
28. Assembler Format of the IPv6 Connection Information Record for VM.....	126
29. Miscellaneous Assembler Constants.....	126
30. Pascal Format of the IPv6 Datagram Information Record for VM.....	129
31. Assembler Format of the SpecOfFileType Record for VM.....	132
32. Equates for MonQueryRecordType used in the MONITORquery Call.....	132
33. Assembler Format of the MonQueryRecordTypefor VM.....	132
34. Assembler format of the QueryRequest record for VM.....	133
35. Remote Procedure Call (Client).....	188
36. Remote Procedure Call (Server).....	189
37. SNMP DPI overview.....	242
38. DPISAMPLE Table MIB descriptions.....	255

Tables

1. Examples of Syntax Diagram Conventions.....	xx
2. TCP/IP TXTLIB Files and Applications.....	26
3. TCP/IP TXTLIB Files and Applications.....	29
4. C Sockets Quick Reference.....	33
5. TCP Connection States.....	42
6. Pascal Language Interface Summary—Notifications.....	54
7. Pascal Language Interface Summary—TCP/UDP Initialization.....	54
8. Pascal Language Interface Summary—TCP/UDP Termination.....	55
9. Pascal Language Interface Summary—Handling External Interrupts.....	55
10. Pascal Language Interface Summary—TCP Communication Procedures.....	55
11. Pascal Language Interface Summary—Ping Interface.....	57
12. Pascal Language Interface Summary—Monitor Procedures.....	57
13. Pascal Language Interface Summary—UDP Communication Procedures.....	57
14. Pascal Language Interface Summary—Raw IP Interface.....	58
15. Pascal Language Interface Summary—Timer Routines.....	58
16. Pascal Language Interface Summary—Host Lookup Routines.....	59
17. Pascal Language Interface Summary—Other Routines.....	59
18. Available VMCF Functions.....	115
19. VMCF TCPIP CALLCODE Requests.....	115
20. VMCF TCPIP CALLCODE Notifications.....	117
21. C Structures in Assembler Language Format.....	148
22. Values for cmd Argument in ioctl Call.....	171
23. Option name values for SETSOCKOPT.....	181

24. SNMP DPI Reference.....	245
25. Client Verification REXX Exit Parameter List.....	287
26. Client Verification ASSEMBLER Exit Parameter List.....	287
27. Client Verification Exit Return Codes.....	290
28. Mail Forwarding REXX Exit Parameter List.....	292
29. Mail Forwarding ASSEMBLER Exit Parameter List.....	293
30. Mail Forwarding Exit Return Codes.....	296
31. SMTP Commands REXX Exit Parameter List.....	298
32. SMTP Commands ASSEMBLER Exit Parameter List.....	299
33. SMTP Command Exit Return Codes.....	302
34. Telnet Session Connection Exit Parameter List.....	305
35. Telnet Exit Parameter List.....	307
36. FTP Exit Audit Parameter List	310
37. FTP Exit Parameter List.....	313
38. FTP Exit Parameter List.....	316
39. Remote authorization responseCodes.....	322
40. Remote authorization majorCodes.....	323
41. Remote authorization minorCodes.....	324
42. Remote auditing responseCodes.....	328
43. Remote auditing majorCodes.....	328
44. Remote auditing minorCodes.....	330
45. Remote audit event codes.....	331
46. Remote audit event code qualifiers.....	331
47. Event-specific fields for remote audit events	331
48. Pascal Language Return Codes.....	337

49. System Return Codes.....	341
50. TCP Well-Known Port Assignments.....	345
51. UDP Well-Known Port Assignments.....	346

About This Document

z/VM: TCP/IP Programmer's Reference describes the routines for application programming in IBM Transmission Control Protocol/Internet Protocol for z/VM 7.4.0.

This document contains information about the following application programming interfaces (APIs):

- C sockets
- Pascal
- Virtual Machine Communication Facility (VMCF)
- Inter-User Communication Vehicle sockets
- Remote Procedure Calls (RPCs)
- Simple Network Management Protocol (SNMP) agent distributed program interface
- Conversational Monitor System (CMS) command interface to the name server
- Simple Mail Transfer Protocol (SMTP)

The descriptive information in the chapters is supplemented with appendixes that contain sample programs and quick references.

For comments and suggestions about this document, use the Reader's Comment Form located at the back of this document. This form gives instructions on submitting your comments by mail, by FAX, or by electronic mail.

Intended Audience

This document is intended for users and programmers who are familiar with z/VM and the Control Program (CP) and the Conversational Monitor System (CMS) components. You should also be familiar with the C or Pascal programming language and the specific application programming interface (API) that you are using.

Before using this document, you should be familiar with z/VM, CP, and CMS. In addition, TCP/IP for z/VM at function level 740 should already be installed and customized for your network.

Conventions and Terminology

This topic describes important style conventions and terminology used in this document.

How the Term “internet” Is Used in This Document

In this document, an internet is a logical collection of networks supported by routers, gateways, bridges, hosts, and various layers of protocols, which permit the network to function as a large, virtual network.

Note: The term "internet" is used as a generic term for a TCP/IP network, and should not be confused with the Internet, which consists of large national backbone networks (such as MILNET, NSFNet, and CREN) and a myriad of regional and local campus networks worldwide.

How Numbers Are Used in This Document

In this document, numbers over four digits are represented in metric style. A space is used rather than a comma to separate groups of three digits. For example, the number sixteen thousand, one hundred forty-seven is written 16 147.





Syntax, Message, and Response Conventions

The following topics provide information on the conventions used in syntax diagrams and in examples of messages and responses.

How to Read Syntax Diagrams

Special diagrams (often called *railroad tracks*) are used to show the syntax of external interfaces.

To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The  symbol indicates the beginning of the syntax diagram.
- The  symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The  symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.
- The  symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the examples in [Table 1 on page xx](#).





Table 1. Examples of Syntax Diagram Conventions	
Syntax Diagram Convention	Example
Keywords and Constants A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown. In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.	 KEYWORD 
Abbreviations Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated. In this example, you can specify KEYWO, KEYWOR, or KEYWORD.	 KEYWOrd 

Table 1. Examples of Syntax Diagram Conventions (continued)

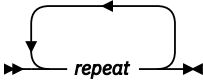
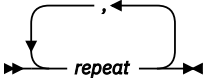
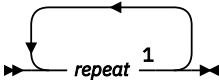
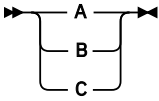
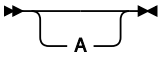
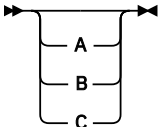
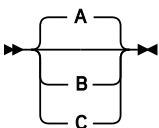
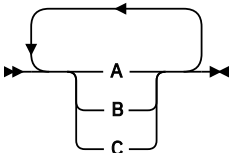
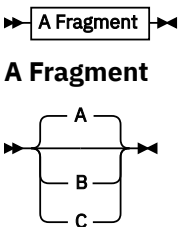
Syntax Diagram Convention	Example
Symbols You must specify these symbols exactly as they appear in the syntax diagram.	* Asterisk : Colon , Comma = Equal Sign - Hyphen () Parentheses . Period
Variables A variable appears in highlighted lowercase, usually italics. In this example, <i>var_name</i> represents a variable that you must specify following KEYWORD.	➤ KEYWORD — <i>var_name</i> ➤
Repetitions An arrow returning to the left means that the item can be repeated. A character within the arrow means that you must separate each repetition of the item with that character. A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated. Syntax notes may also be used to explain other special aspects of the syntax.	   Notes: ¹ Specify <i>repeat</i> up to 5 times.
Required Item or Choice When an item is on the line, it is required. In this example, you must specify A. When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.	➤ A ➤ 
Optional Item or Choice When an item is below the line, it is optional. In this example, you can choose A or nothing at all. When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.	 

Table 1. Examples of Syntax Diagram Conventions (continued)	
Syntax Diagram Convention	Example
<p>Defaults</p> <p>When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line.</p> <p>In this example, A is the default. You can override A by choosing B or C.</p>	
<p>Repeatable Choice</p> <p>A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item.</p> <p>In this example, you can choose any combination of A, B, or C.</p>	
<p>Syntax Fragment</p> <p>Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name.</p> <p>In this example, the fragment is named "A Fragment."</p>	

Examples of Messages and Responses

Although most examples of messages and responses are shown exactly as they would appear, some content might depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

xxx

Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.

[]

Brackets enclose optional text that might be displayed.

{ }

Braces enclose alternative versions of text, one of which will be displayed.

|

The vertical bar separates items within brackets or braces.

...

The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

Where to Find More Information

Appendix F, “Abbreviations and acronyms,” on page 355, lists the abbreviations and acronyms that are used throughout this document.

For more information about related publications, see the documents listed in the “Bibliography” on page 363.

Links to Other Online Documents

The online version of this document contains links to other online documents. These links are to editions that were current when this document was published. However, due to the nature of some links, if a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition. Also, a link from this document to another document works only when both documents are in the same directory.

Links to Other Documents and Websites

The PDF version of this document contains links to other documents and websites. A link from this document to another document works only when both documents are in the same directory or database, and a link to a website works only if you have access to the Internet. A document link is to a specific edition. If a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition.

How to provide feedback to IBM

We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information. See [How to send feedback to IBM](#) for additional information.

Summary of Changes for z/VM: TCP/IP Programmer's Reference

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line (|) to the left of the change.

SC24-6332-74, z/VM 7.4 (September 2024)

This edition supports the general availability of z/VM 7.4. Note that the publication number suffix (-74) indicates the z/VM release to which this edition applies.

Removal of support for LAN Channel Station (LCS) emulation

Support for the OSE CHPID type, which is used to provide LAN Channel Station (LCS) emulation, is discontinued. TCP/IP no longer supports the LCS device driver. LCS documentation is removed from z/VM publications.

This satisfies the Statement of Direction from the z/VM 7.3 product announcement.

The following topic is updated:

- [Appendix F, “Abbreviations and acronyms,” on page 355](#)

SC24-6332-73, z/VM 7.3 (September 2023)

This edition supports product changes that were provided or announced after the general availability of z/VM 7.3.

[PH56199, VM66698] System SSL z/OS 2.5 Equivalence

With the PTFs for APARs PH56199 (TCP/IP) and VM66698 (LE), z/VM 7.3 provides an update to the cryptographic services library, which includes certificate diagnostic enhancements and improved algorithmic support and allows for enablement of TLS 1.3, for secure connectivity to the z/VM platform.

SC24-6332-73, z/VM 7.3 (September 2022)

This edition supports the general availability of z/VM 7.3. Note that the publication number suffix (-73) indicates the z/VM release to which this edition applies.

Miscellaneous updates for z/VM 7.3

The following topic is updated:

- [“IUCV Socket Call Syntax” on page 148](#)

SC24-6332-03, z/VM 7.2 (December 2021)

This edition includes terminology, maintenance, and editorial changes.

The following topics are updated to clarify client certificate verification:

- [“Starting a Secure Connection” on page 21](#)
- [“Stopping a Secure Connection” on page 21](#)
- [“IUCV Socket Call Syntax” on page 148](#)

SC24-6332-03, z/VM 7.2 (March 2021)

This edition includes changes to support product changes provided or announced after the general availability of z/VM 7.2.

Miscellaneous updates for March 2021

The following topic is updated:

- [“IUCV Socket Call Syntax” on page 148](#)

SC24-6332-03, z/VM 7.2 (September 2020)

This edition includes changes to support the general availability of z/VM 7.2.

Chapter 1. z/VM C Socket Application Programming Interface

This chapter describes the z/VM C socket application programming interface (API). z/VM C sockets are C Language functions that closely correspond to the sockets used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.4.

z/VM C sockets are intended as replacements for VM TCP/IP C sockets (formerly documented in this chapter). Although TCP/IP C sockets are still supported for compatibility, the z/VM C socket API is preferred.

This chapter describes how to write, compile, and run applications that use z/VM C sockets. Existing applications that use the VM TCP/IP C sockets library may continue to do so without any modification. To use the z/VM C socket functions, existing TCP/IP C socket applications may need to be recompiled, but no source changes are required. Instructions are provided in this chapter.

Note:

1. To run programs that use z/VM C sockets, you must have Language Environment® (supplied with z/VM) installed on your system. Language Environment provides header files and the object code and run-time library for the z/VM C socket functions.

To compile programs that use the z/VM C socket API, you also need the IBM C for VM/ESA (C/VM) Compiler 3.1 (5654-033).

For specific program requirements, see the [*z/VM: General Information*](#).

2. This chapter provides a guide to using the z/VM C socket API. For complete reference information on the z/VM C socket functions, see the [*XL C/C++ for z/VM: Runtime Library Reference*](#).

This chapter contains the following sections:

- “TCP/IP Network Communication” on [page 2](#) defines some of the basic networking terms.
- “What is a Socket?” on [page 2](#) provides an overview of socket programming concepts.
- “Client/Server Conversation” on [page 8](#) shows how a client and server use sockets to exchange information.
- “Network Application Example” on [page 14](#) shows how sockets are used in a network application program.
- “z/VM C Socket Implementation” on [page 18](#) explains how z/VM has implemented the support for C sockets. This section also explains the incompatibilities between z/VM C sockets and VM TCP/IP C sockets.
- “Compiling and Linking a Sockets Program” on [page 26](#) describes how to compile and link programs to use the z/VM C sockets library.
- “Running a Sockets Program” on [page 29](#) describes how to run programs that use the z/VM C sockets library.
- “C Sockets Quick Reference” on [page 32](#) lists the z/VM C socket calls.
- “TCP Client Program” on [page 35](#) shows an example of a TCP client program using z/VM C sockets.
- “TCP Server Program” on [page 36](#) shows an example of a TCP server program using z/VM C sockets.
- “UDP Client Program” on [page 38](#) shows an example of a UDP client program using z/VM C sockets.
- “UDP Server Program” on [page 38](#) shows an example of a UDP server program using z/VM C sockets.

TCP/IP Network Communication

Network communication, or "internetworking", defines a set of protocols that allow application programs to talk with each other without regard to the hardware and operating systems where they are run. Internetworking allows application programs to communicate independently of their physical network connections.

TCP/IP is an internetworking technology and is named after its two main protocols: Transmission Control Protocol (TCP), and Internet Protocol (IP). You should also be familiar with the following basic internetworking terms:

client

A process that requests services on the network.

server

A process that responds to a request for service from a client.

datagram

A basic unit of information, consisting of one or more data packets, which are passed across an internet at the transport level.

packet

The unit or block of a data transaction between a computer and its network. A packet usually contains a network header, at least one high-level protocol header, and data blocks. Generally, the format of data blocks does not affect how packets are handled. Packets are the exchange medium used at the Internetwork layer to send data through the network.

Transport Protocols

There are two general types of transport protocols:

- A **connectionless protocol** treats each datagram as independent from all others. Each datagram must contain all the information required for its delivery.

An example of such a protocol is **User Datagram Protocol (UDP)**. UDP is a datagram-level protocol built directly on the IP layer and used for application-to-application programs on a TCP/IP host. UDP does not guarantee data delivery, and is therefore considered unreliable. Application programs that require reliable delivery of streams of data should use TCP.

- A **connection-oriented protocol** requires that hosts establish a logical connection with each other before communication can take place. This connection is sometimes called a "virtual circuit", although the actual data flow uses a packet-switching network. A connection-oriented exchange includes three phases:

1. Start the connection.
2. Transfer data.
3. End the connection.

An example of such a protocol is **Transmission Control Protocol (TCP)**. TCP provides a reliable vehicle for delivering packets between hosts on an internet. TCP breaks a stream of data 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 and re-sends the missing or damaged datagrams. The data stream that is received is therefore a reliable copy of the original.

These types of protocols are illustrated in [Figure 2 on page 12](#), and in [Figure 3 on page 13](#).

What is a Socket?

A **socket** can be thought of as an endpoint in a two-way communication channel. Socket routines create the communication channel, and the channel carries data between application programs either locally or over networks. Each socket open by a process — like any open file in a POSIX process — has a unique (within the process) number associated with it called a "file descriptor", an integer that designates a socket and allows the application program to refer to it when needed.

Using an electrical analogy, you can think of the communication channel as the electrical wire with its plug and the port, or socket, as the electrical socket or outlet, as shown in Figure 1 on page 3.

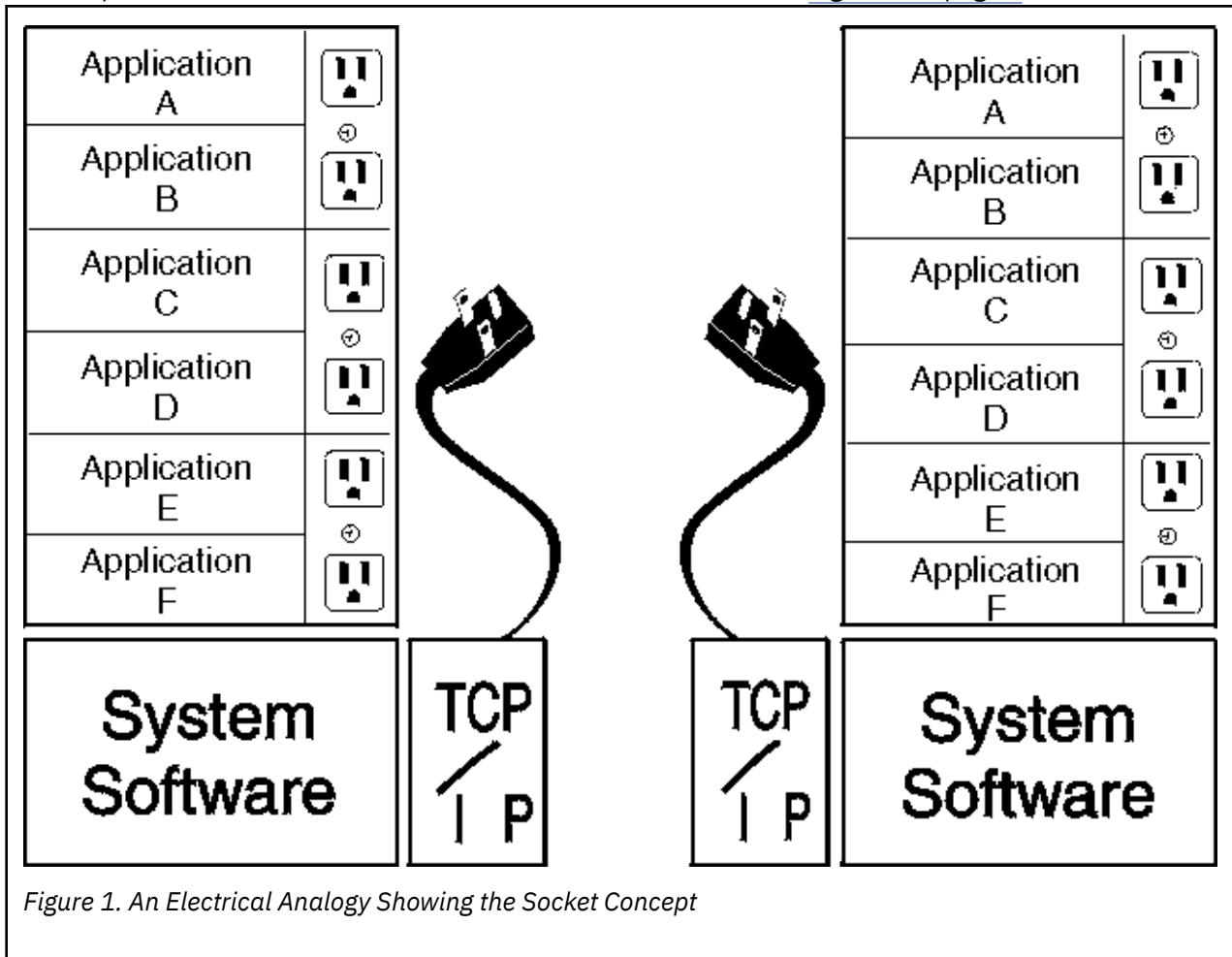


Figure 1 on page 3 shows many application programs running on a client and many application programs on a server. When the client starts a socket call, a socket connection is made between an application on the client and an application on the server.

Another analogy used to describe socket communication is a telephone conversation. Dialing a phone number from your telephone is similar to starting a socket connection. The telephone switching unit knows where to logically make the correct switch to complete the call at the remote location. During your telephone conversation, this connection is present and information is exchanged. After you hang up, the connection is broken and you must start it again. The client uses the `connect()` function call to start the logical switch mechanism to connect to the server.

User processes ask the sockets library to create a socket when one is needed. The sockets library returns an integer, the file descriptor that the application uses every time it wants to refer to that socket.

Sockets perform in many respects like UNIX files or devices, so they can be used with such traditional operations as `read()` or `write()`. For example, after two application programs create sockets and open a connection between them, one program can use `write()` to send a stream of data, and the other can use `read()` to receive it. Because each file or socket has a unique descriptor, the system knows exactly where to send and to receive the data.

Address Families

The z/VM C socket API supports four address families (also called domains):

AF_INET and AF_INET6

AF_INET and AF_INET6 (internet domain) sockets provide a means of communicating between application programs that are on different systems using the TCP and UDP transport protocols provided by a TCP/IP product. These address families support both stream and datagram sockets. TCP/IP for z/VM must be configured for you to be able to use these address families.

AF_IUCV

AF_IUCV (VM IUCV) sockets provide communication between processes on a single VM system, or on a group of systems that share IUCV connectivity. VM IUCV sockets allow interprocess communication within VM independent of TCP/IP. The AF_IUCV domain supports only stream sockets.

AF_UNIX

AF_UNIX sockets (also called local sockets) provide communication between processes on a single VM system, or on a group of systems that share a single Byte File System (BFS) server. UNIX domain sockets allow interprocess communication within VM independent of TCP/IP. On z/VM, the AF_UNIX domain supports only stream sockets.

The primary difference between VM IUCV sockets and UNIX sockets is how partners are identified (for example, how they are named).

Socket Types

The z/VM C socket API provides application programs with an interface that hides the details of the physical network. The API supports **stream** sockets, **datagram** sockets, and **raw** sockets, each providing different services for application programs. Stream and datagram sockets interface to the network layer protocols, and raw sockets interface to the network interface layers. You choose the most appropriate interface for an application.

Stream Sockets

The stream sockets interface provides a connection-oriented service. After the partner applications connect, the data sent on stream sockets acts like a stream of information. There are no boundaries between data, so communicating processes must agree on their own mechanism to distinguish information. For example, the process sending information could first send the length of the data, followed by the data itself. The process receiving information reads the length and then loops, reading data until all of it has been transferred. Stream sockets guarantee delivery of the data in the order it was sent and without duplication. The stream socket interface provides a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. Flow control is built in, to avoid data overruns. No boundaries are imposed on the data; the data is considered to be a stream of bytes.

Stream sockets are the most-commonly used, because the burden of transferring the data reliably is handled by the system rather than by the application.

Datagram Sockets

The datagram socket interface provides a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; datagrams can be lost, duplicated, and can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction.

Raw Sockets

The raw socket interface provides direct access to lower layer protocols, such as the **Internet Protocol (IP)** and **Internet Control Message Protocol (ICMP or ICMPv6)**. You can use raw sockets to test new protocol implementations. You can extend the socket interface; you can define new socket types to provide additional services. Because they isolate you from the communication functions of the different protocol layers, socket interfaces are largely independent of the underlying network. In the AF_INET address family, stream sockets interface to TCP, datagram sockets interface to UDP, and raw sockets interface to ICMP and IP. In the AF_INET6 address family, stream sockets interface to TCP, datagram sockets interface to UDP, and raw sockets interface to ICMPv6 and IP.

Guidelines for Using Socket Types

The following criteria will help you choose the appropriate socket type for an application program.

If you are communicating with an existing application program, you must use the same protocol as the existing application program. For example, if you want to communicate with an application that uses TCP, you must use AF_INET or AF_INET6 stream sockets. For new application programs, you should consider the following factors:

- **Reliability:** Stream sockets provide the most reliable connection. Datagram sockets are unreliable, because datagrams can be discarded, corrupted, or duplicated during transmission. This may be acceptable if the application program does not require reliability, or if the application program implements the reliability on top of the sockets interface. The trade-off is the improved performance available with datagram sockets.
- **Performance:** The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrade the performance of stream sockets in comparison with datagram sockets.
- **Data transfer:** Datagram sockets impose a limit on the amount of data transferred in a single transaction. If you send less than 2048 bytes at a time, use datagram sockets. As the amount of data in a single transaction increases, use stream sockets.

If you are writing a new protocol on top of IP, or wish to use the ICMP protocol, then you must use raw sockets.

Domain-specific Socket Addresses

The following sections describe the different ways to address processes who communicate with each other using sockets.

Address Families

Each address family defines a different style of addressing. All hosts in the same address family use the same scheme for addressing socket endpoints. The AF_INET and AF_INET6 address families identify processes by IP address and port number. The AF_UNIX address family identifies processes by file name in the Byte File System. The AF_IUCV address family identifies processes by VM user ID and application name.

Socket Address

A socket address is defined by the *sockaddr* structure in the **sys/socket.h** header file. The structure has three fields, as shown in the following example:

```
struct sockaddr {
    unsigned char sa_len;
    unsigned char sa_family;
    char          sa_data[14];    /* variable length data */
};
```

The *sa_len* field contains the length of the entire *sockaddr* structure, in bytes. The *sa_family* field contains a value identifying the address family. It is AF_INET or AF_INET6 for the internet domain, AF_UNIX for the UNIX domain, and AF_IUCV for the IUCV domain. The *sa_data* field is different for each address family. Each address family defines its own structure, which can be overlaid on the *sockaddr* structure. See [“Addressing within the AF_INET and AF_INET6 Domains” on page 5](#), [“Addressing within the AF_UNIX Domain” on page 7](#), and [“Addressing within the AF_IUCV Domain” on page 8](#).

Addressing within the AF_INET and AF_INET6 Domains

Before discussing the contents of the AF_INET and AF_INET6 *sockaddr* structures, the following terms must be introduced:

Internet Addresses

Internet addresses represent a network interface. Every internet address within an administered domain is unique. On the other hand, it is not necessary that every host have a single internet address; in fact, a host has as many internet addresses as it has network interfaces.

Internet addresses can be in one of two formats: IPv4 (IP version 4) or IPv6 (IP version 6). Hosts can support either addressing format or both. IPv4 internet addresses are 32-bit quantities. The `AF_INET` address family communicates through IPv4 addresses. IPv6 internet addresses are 128-bit quantities. The `AF_INET6` address family communicates through IPv6 addresses.

Ports

A port distinguishes between different application programs using the same `AF_INET` network interface. It is an additional qualifier used by the system software to get data to the correct application program. Physically, a port is a 16-bit integer. Some ports are reserved for particular application programs or protocols and are called **well-known ports**.

Network byte order and host byte order

Ports and addresses are always specified in calls to the socket functions using the network byte order convention. This convention is a method of sorting bytes that is independent of specific machine architectures. Host byte order, on the other hand, sorts bytes in the manner which is most natural to the host software and hardware. There are two common host byte order methods:

- **Little-endian** byte ordering places the least significant byte first. This method is used in Intel microprocessors, for example.
- **Big-endian** byte ordering places the most significant byte first. This method is used in IBM z/Architecture® and S/390® mainframes and Motorola microprocessors, for example.

The network byte order is defined to always be big-endian, which may differ from the host byte order on a particular machine. Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information without confusion because of byte ordering. The following C functions allow the application program to switch numbers easily back and forth between the host byte order and network byte order without having to first know what method is used for the host byte order:

- `htonl()` translates an unsigned long integer into network byte order.
- `htons()` translates an unsigned short integer into network byte order.
- `ntohl()` translates an unsigned long integer into host byte order.
- `ntohs()` translates an unsigned short integer into host byte order.

See Figure 5 on page 14, Figure 7 on page 15, and Figure 8 on page 15 for examples of using the `htons()` call to put port numbers into network byte order.

The C functions `inet_ntop()` and `inet_pton()` are used to manipulate IPv6 addresses. For more information on these functions, see [XL C/C++ for z/VM: Runtime Library Reference](#).

AF_INET addresses

A socket address in the `AF_INET` address family is defined by the `sockaddr_in` structure, which is defined in the `netinet/in.h` header file:

```
typedef unsigned long in_addr_t;
struct in_addr {
    in_addr_t s_addr;
};
struct sockaddr_in {
    unsigned char sin_len; /* length of sockaddr struct */
    unsigned char sin_family; /* addressing family */
    unsigned short sin_port; /* port number */
    struct in_addr sin_addr; /* IP address */
    unsigned char sin_zero[8]; /* unassigned */
};
```

The *sin_len* field is set to either 0 or `sizeof(struct sockaddr_in)` when providing a *sockaddr_in* structure to the sockets library. Both values are treated the same. When the sockets library provides a *sockaddr_in* structure to the application, the *sin_len* field is set to `sizeof(struct sockaddr_in)`.

The *sin_family* field is set to `AF_INET`.

The *sin_port* field is set to the port to which the process is bound, in network byte order.

The *sin_addr* field is set to the internet address (IP address) of the interface to which the process is bound, in network byte order.

The *sin_zero* field is not used and must be set to all zeros.

AF_INET6 addresses

If the socket descriptor *socket* was created in the `AF_INET6` domain, the format of the name buffer is expected to be *sockaddr_in6*, as defined in the `netinet/in.h`:

```
struct sockaddr_in6 {
    uint8_t      sin6_len;          /* length of sockaddr structure */
    sa_family_t  sin6_family;       /* addressing family */
    in_port_t    sin6_port;         /* port number */
    uint32_t     sin6_flowinfo;     /* ignored */
    struct in6_addr sin6_addr;       /* IP address */
    uint32_t     sin6_scope_id;     /* scope ID */
};
```

The *sin6_len* field must be set to either 0 or `sizeof(struct sockaddr_in6)`. Both values are treated the same.

The *sin6_family* must be set to `AF_INET6`.

The *sin6_port* field is set to the port to which the socket is bound. It must be specified in network byte order.

The *sin6_flowinfo* field is currently unsupported so its contents are ignored.

The *sin6_addr.sin6_addr* field is set to the internet address of the interface to which the socket is bound. It must be specified in network byte order.

The *sin6_scope_id* field identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. For link local addresses, the *sin6_scope_id* can be used to specify the outgoing interface index. The z/VM stack supports *sin6_scope_id* for link local addresses only.

Addressing within the AF_UNIX Domain

A socket address in the `AF_UNIX` address family is defined by the *sockaddr_un* structure, which is defined in the **sys/un.h** header file:

```
struct sockaddr_un {
    unsigned char sun_len;          /* length of sockaddr struct */
    unsigned char sun_family;       /* addressing family */
    char sun_path[108];             /* file name */
};
```

When the application provides a *sockaddr_un* structure to the sockets library, the *sun_len* field should be set to either 0 or a value greater than or equal to `SUN_LEN(&sa)`, where *sa* is the name of the *sockaddr_un* variable, but less than or equal to `sizeof(struct sockaddr_un)`. The `SUN_LEN()` macro, which is defined in **sys/un.h**, evaluates to an expression which returns the total length of the used portion of the *sockaddr_un* structure, when *sun_path* has been filled in with a null-terminated file name. The length returned by `SUN_LEN()` does not include the terminating null character. If a 0 is specified for *sun_len*, the *sockaddr* length provided on the specific socket function call determines how long the path name is. If *sun_len* is nonzero, the lesser of *sun_len* and the provided length is used. In either case, if a null character appears in the string before the given length, the path name is considered to end there. When the sockets library provides a *sockaddr_un* structure to the application, the *sun_len* field is set to `SUN_LEN(&sa)+1`, where *sa* is the name of the *sockaddr_un* variable. This length thus includes the null byte which terminates the file name.

The `sun_family` field is set to `AF_UNIX`.

The `sun_path` field contains the name of the file which represents the open socket. It need not be null delimited, although it is recommended that it is, so that the `SUN_LEN()` macro can be used. A file by this name will be created in the Byte File System by the `bind()` function call, and must exist there for the `connect()` function call to succeed. Because the Byte File System contains the file, the form of the path name should follow the POSIX conventions. Generally, an absolute path name (one that begins with a slash) should be specified, so that the client and the server can both use the same path name to identify the file. If an `AF_UNIX` socket is not yet bound when a client calls the `connect()` function, it will be bound to the null path name string (for example, the string ""). In this case, no file is created in the Byte File System.

For more information about the Byte File System, see the [z/VM: OpenExtensions User's Guide](#).

Addressing within the AF_IUCV Domain

A socket address in the `AF_IUCV` address family is defined by the `sockaddr_iucv` structure, which is defined in the **saucv.h** header file:

```
struct sockaddr_iucv {
    unsigned char siucv_len;      /* length of sockaddr struct */
    unsigned char siucv_family;  /* addressing family */
    unsigned short siucv_port;   /* port number */
    unsigned long siucv_addr;    /* address */
    unsigned char siucv_nodeid[8]; /* nodeid to connect to */
    unsigned char siucv_userid[8]; /* userid to connect to */
    unsigned char siucv_name[8]; /* iucvname for connect */
};
```

The `siucv_len` field is set to either 0 or `sizeof(struct sockaddr_iucv)` when providing a `sockaddr_iucv` structure to the sockets library. Both values are treated the same. When the sockets library provides a `sockaddr_iucv` structure to the application, the `siucv_len` field is set to `sizeof(struct sockaddr_iucv)`.

The `siucv_family` field is set to `AF_IUCV`.

The `siucv_port`, `siucv_addr`, and `siucv_nodeid` fields are reserved for future use. The `siucv_port` and `siucv_addr` fields must be zeroed. The `siucv_nodeid` field must be set to exactly eight blank characters.

The `siucv_userid` field is set to the VM user ID of the application which owns the address. This field must be eight characters long, padded with blanks on the right. It cannot contain the null character.

The `siucv_name` field is set to the application name by which the socket is known. A server advertises a particular application name, and this is the name used by the client to connect to the server. The recommended form of the name contains eight characters, padded with blanks to the right.

For more information about IUCV, see [z/VM: CMS Application Development Guide for Assembler](#).

Client/Server Conversation

The client and server exchange data using a number of socket functions. They can send data using `send()`, `sendto()`, `sendmsg()`, `write()`, or `writv()`. They can receive data using `recv()`, `recvfrom()`, `recvmsg()`, `read()`, or `readv()`. The following is an example of the `send()` and `recv()` calls:

```
send(s, addr_of_data, len_of_data, 0);
recv(s, addr_of_buffer, len_of_buffer, 0);
```

The `send()` and `recv()` functions specify the socket `s` on which to communicate, the address in memory of the buffer that contains, or will contain, the data (`addr_of_data`, `addr_of_buffer`), the size of this buffer (`len_of_data`, `len_of_buffer`), and a flag that tells how the data is to be sent. Using the flag 0 tells TCP/IP to transfer the data normally. The server uses the socket that is returned from the `accept()` call. The client uses the socket that is returned from the `socket()` call.

These functions return the amount of data that was either sent or received. Because stream sockets send and receive information in streams of data, it can take more than one call to `send()` or `recv()` to transfer all

the data. It is up to the client and server to agree on some mechanism of signaling that all the data has been transferred.

When the conversation is over, both the client and server call the `close()` function to end the connection. The `close()` function deallocates the socket, freeing its space in the table of connections. To end a connection with a specific client, the server closes the socket returned by `accept()`. If the server closes its original socket, the "listening" socket, it can no longer accept new connections, but it can still converse with the clients it is connected to. The following is an example of the `close()` call:

```
close(s);
```

Server Perspective for AF_INET

Before the server can accept any connections with clients, it must register itself with TCP/IP and "listen" for client requests on a specific port.

socket()

The server must first allocate a socket. This socket provides an endpoint that clients connect to.

Opened sockets are identified by file descriptors, like any open files in a POSIX environment. The programmer calls the `socket()` function to allocate a new socket, as shown in the following example:

```
socket(AF_INET, SOCK_STREAM, 0);
```

The `socket()` function requires the address family (`AF_INET`), the type of socket (`SOCK_STREAM`), and the particular networking protocol to use (when `0` is specified, the system automatically uses the appropriate protocol for the specified socket type). A new socket is allocated and its file descriptor is returned.

bind()

At this point, an entry in the table of communications has been reserved for your application program. However, the socket has no port or IP address associated with it until you use the `bind()` function, which requires three parameters:

- The socket the server was just given
- The number of the port on which the server wishes to provide its service
- The IP address of the network connection on which the server is listening (to understand what is meant by "listening", see ["listen\(\)" on page 9](#)).

The server puts the port number and IP address into a `sockaddr_in` structure, passing it and the socket file descriptor to the `bind()` function. For example:

```
struct sockaddr_in sa;
...
bind(s, (struct sockaddr *) &sa, sizeof sa);
```

listen()

After the `bind`, the server has specified a particular IP address and port. Now it must notify the system that it intends to listen for connections on this socket. The `listen()` function puts the socket into passive open mode and allocates a backlog queue of pending connections. In passive open mode, the socket is open for clients to contact. For example:

```
listen(s, backlog_number);
```

The server gives the file descriptor of the socket on which it will be listening and the number of requests that can be queued (the *backlog_number*). If a connection request arrives before the server can process it, the request is queued until the server is ready.

The SOMAXCONN statement in the TCP/IP configuration file (PROFILE TCPIP) determines the maximum length that *backlog_number* can be. For more information on the SOMAXCONN statement, see [z/VM: TCP/IP Planning and Customization](#).

accept()

Up to this point, the server has allocated a socket, bound the socket to an IP address and port, and issued a passive open. The next step is for the server to actually establish a connection with a client. The `accept()` call blocks the server until a connection request arrives, or, if there are connection requests in the backlog queue, until a connection is established with the first client in the queue. The following is an example of the `accept()` call:

```
struct sockaddr_in sa;
int addrlen;
:
client_sock = accept(s, (struct sockaddr *) &sa, &addrlen);
```

The server passes the file descriptor of its socket to the `accept()` call. When the connection is established, the `accept()` call creates a new socket representing the connection with the client, and returns its file descriptor. When the server wishes to communicate with the client or end the connection, it uses the file descriptor of this new socket, *client_sock*. The original socket *s* is now ready to accept connections with other clients. The original socket is still allocated, bound, and opened passively. To accept another connection, the server calls `accept()` again. By repeatedly calling `accept()`, the server can establish many connections simultaneously.

select()

The server is now ready to start handling requests on this port from any client with the server's IP address and port number. If the server handles just one client at a time, it can just start sending or receiving data. A server is not limited to one active socket, though. Often a server processes requests from several clients at the same time, and additionally listens for new clients wanting to establish connections. For maximum performance, such a server should either create a new thread to handle each client request, or set all of its sockets to "nonblocking" mode, so that a delay in handling one client request does not affect other client requests. Using nonblocking mode allows a single-threaded server to operate only on those sockets that are ready for communication. The `select()` call allows an application program to test for activity on a group of sockets.

Note: The `select()` function can also be used with other descriptors, such as file descriptors, pipes, or character special files such as the tty.

To allow you to test any number of sockets with just a single call to `select()`, place the file descriptors of the sockets to test into a "bit set", passing the bit set to the `select()` call. A bit set is a string of bits where each possible member of the set is on or off. If the member's bit is off, the member is not in the set. If the member's bit is on, the member is in the set. If the socket with file descriptor 3 is a member of a bit set, then the bit that represents it is on.

The following macros are provided to manipulate the bit sets:

Macro

Description

FD_ZERO

Clears the whole bit set

FD_SET

Sets the bit corresponding to a particular file descriptor

FD_CLR

Clears the bit corresponding to a particular file descriptor

FD_ISSET

Tests whether the bit corresponding to a particular file descriptor is set or cleared

To be active, a socket must be ready for reading data or for writing data, or an exceptional condition must have occurred. Therefore, the server specifies three bit sets of file descriptors in its call to the `select()` function: one bit set for file descriptors on which to receive data, another for file descriptors on which to write data, and one for sockets with exception conditions. The `select()` call tests each file descriptor in each bit set for activity and returns only those file descriptors that are active.

A server that processes many clients simultaneously can be written so that it processes only those clients that are ready for activity.

Client Perspective for AF_INET

The client first issues the `socket()` function call to allocate a socket on which to communicate:

```
socket(AF_INET, SOCK_STREAM, 0);
```

To connect to the server, the client places the port number and the IP address of the server into a `sockaddr_in` structure, like the one used by the server of its `bind()` call. If the client does not know the server's IP address, but does know the server's host name, the `gethostbyname()` function may be called to translate the host name into its IP address. The client then calls `connect()`:

```
struct sockaddr_in sa;
:
connect(s, (struct sockaddr_in *) &sa, sizeof sa);
```

When the connection is established, the client uses its socket to communicate with the server.

Typical TCP Socket Session

You can use TCP sockets for both passive (server) and active (client) processes. Whereas some functions are necessary for both types, some are role-specific. After you make a connection, it exists until you close the socket. During the connection, data is either delivered or an error code is returned by TCP/IP.

See [Figure 2 on page 12](#) for the general sequence of calls to be followed for most socket routines using TCP, or stream sockets.

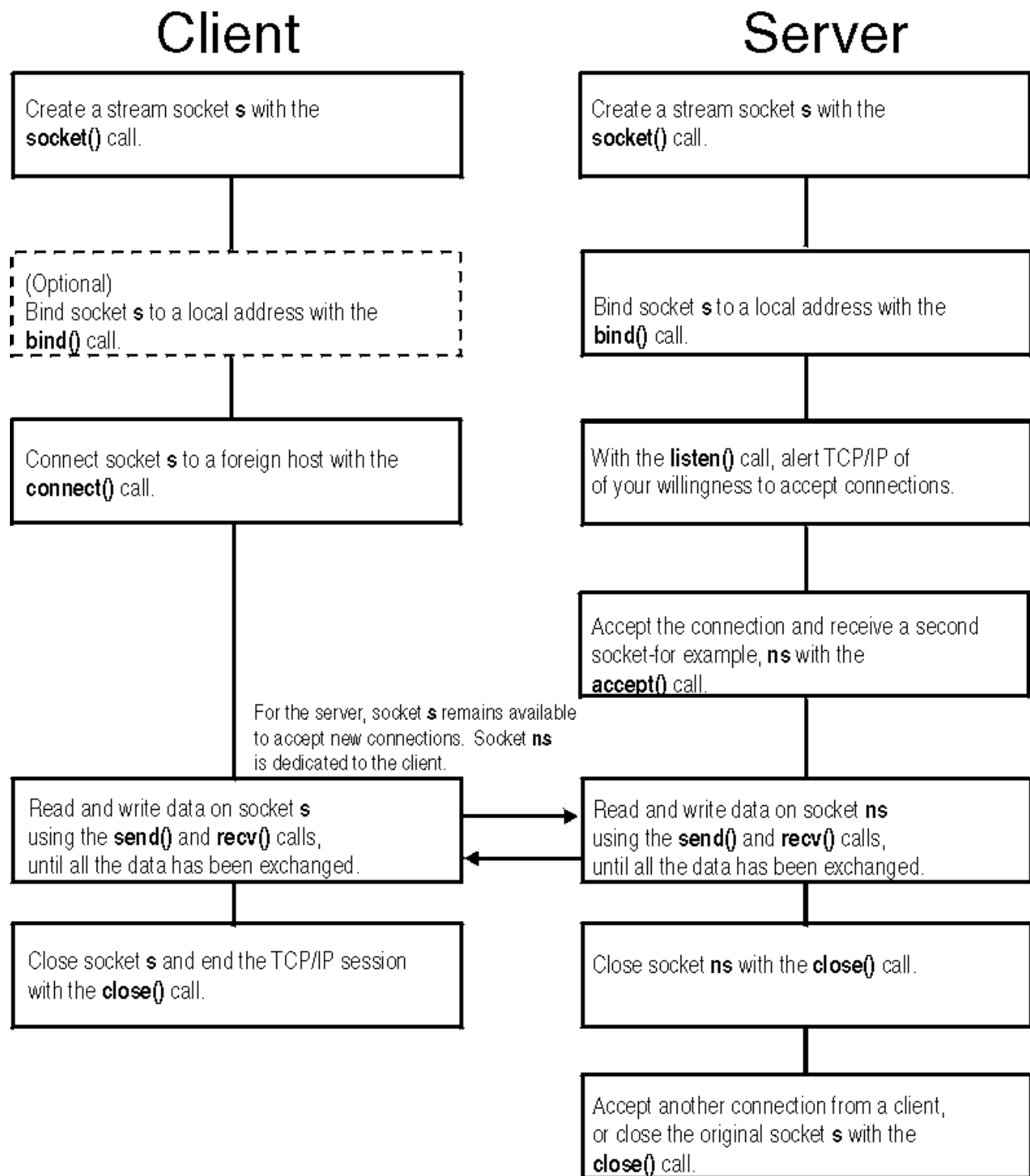


Figure 2. A Typical Stream Socket Session

Typical UDP Socket Session

UDP socket processes, unlike TCP socket processes, are not clearly distinguished by server and client roles. The distinction is between connected and unconnected sockets. An unconnected socket can be used to communicate with any host; but a connected socket, because it has a dedicated destination, can send data to, and receive data from, only one host.

Both connected and unconnected sockets send their data over the network without verification. Consequently, after a packet has been accepted by the UDP interface, the arrival and integrity of the packet cannot be guaranteed.

See Figure 3 on page 13 for the general sequence of calls to be followed for most socket routines using UDP, or datagram, sockets.

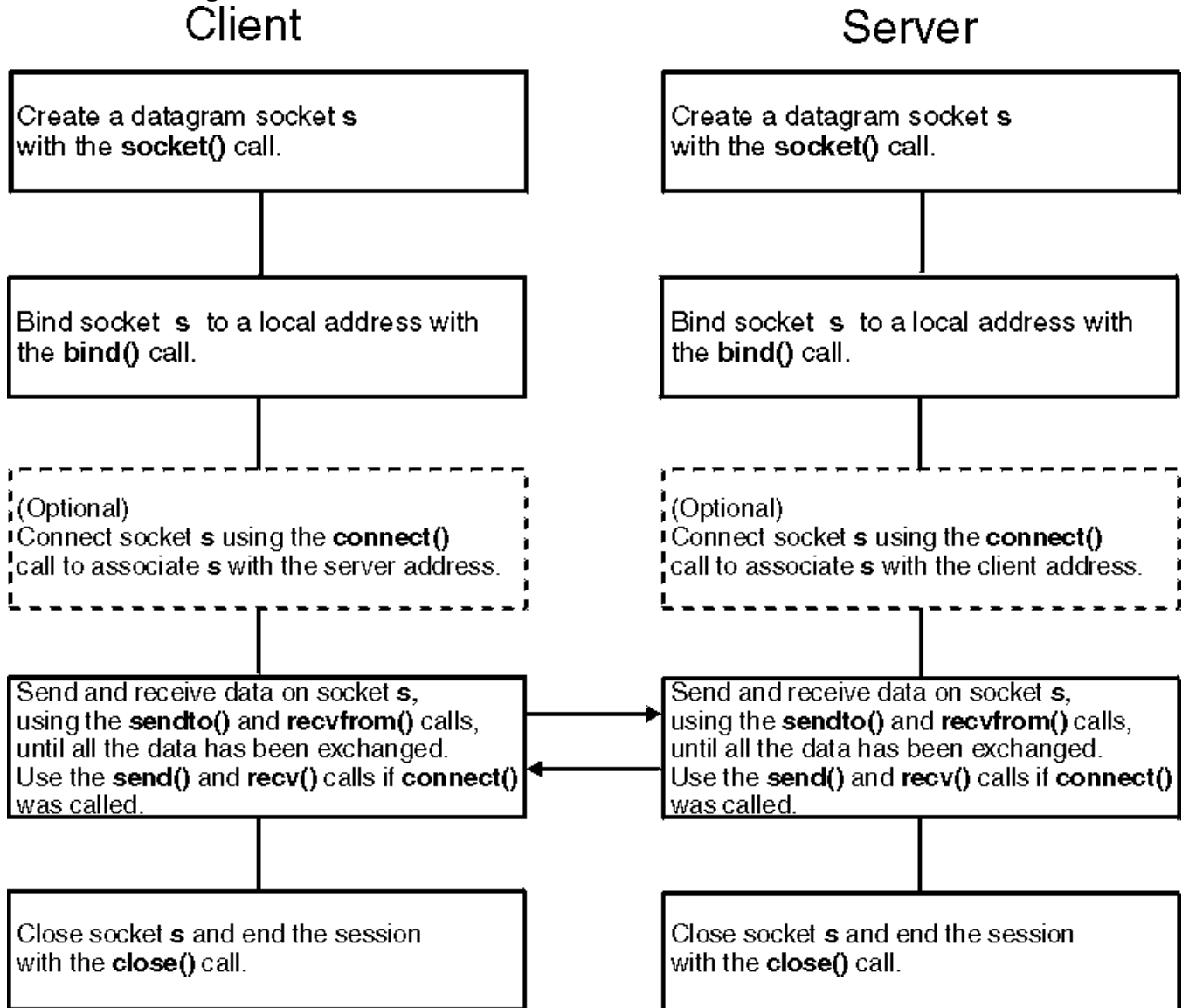


Figure 3. A Typical Datagram Socket Session

Locating the Server's Port

In the client/server model, the server provides a resource by listening for clients on a particular port. Such application programs as FTP, SMTP, and Telnet listen on a **well-known port**, a port reserved for use by a specific application program or protocol. However, for your own client/server application programs, you need a method of assigning port numbers to represent the services you intend to provide. One general method of defining services and their ports is to enter them into the ETC SERVICES file. The programmer uses the `getservbyname()` function to determine the port for a particular service. If the port number for a particular service changes, only the ETC SERVICES file must be modified.

Note: TCP/IP for z/VM is shipped with an ETC SERVICES file containing such well-known services as FTP, SMTP, and Telnet.

Network Application Example

The following steps illustrates using socket functions in an AF_INET network application program.

Note: Error checking has been omitted from the examples. Error checking is very important, and has been omitted only to avoid complicating the examples.

1. First, an application program must open a socket using the `socket()` call, as shown in [Figure 4 on page 14](#).

```
int s;  
...  
s = socket(AF_INET, SOCK_STREAM, 0);
```

Figure 4. An Application Using `socket()`

This example allocates a socket `s` in the AF_INET address family, with socket type SOCK_STREAM and protocol 0. Passing 0 for the protocol chooses the default, which for the AF_INET domain and SOCK_STREAM type is IPPROTO_TCP. The supported values for the socket domain, type, and protocol are defined in the **netinet/in.h** header file.

If successful, the `socket()` call returns a positive integer called a file descriptor that is used in subsequent function calls to identify the socket.

2. After an application program creates a socket, it can explicitly bind a unique address to the socket, as shown in [Figure 5 on page 14](#).

```
int rc;  
int s;  
struct sockaddr_in myname;  
  
/* Clear the structure to be sure that the sin_len and */  
/* sin_zero fields are clear */  
memset(&myname, 0, sizeof myname);  
myname.sin_family = AF_INET;  
myname.sin_addr.s_addr = INADDR_ANY; /* any interface */  
myname.sin_port = htons(5001);  
...  
rc = bind(s, (struct sockaddr *) &myname, sizeof myname);
```

Figure 5. An Application Using `bind()`

This example binds the socket with file descriptor `s` to port 5001, allowing it to accept connections from any interface available to the host in the internet domain. Servers must bind to an address and port to become accessible to the network. Also shown in this example is a handy utility routine called `htons()`, which takes a short integer (like a port number) in host byte order and returns it in network byte order.

3. After binding to a socket, a server that uses stream sockets must indicate its readiness to accept connections from clients. The server does this with the `listen()` call, as shown in [Figure 6 on page 14](#).

```
int s;  
int rc;  
...  
rc = listen(s, 5);
```

Figure 6. An Application Using `listen()`

This example tells TCP/IP that the server is ready to begin accepting connections, and that a maximum of five connection requests can be queued for the server. Additional requests are ignored.

4. Clients using stream sockets begin a connection request by calling `connect()`, as shown in [Figure 7](#) on [page 15](#).

```
int s;
struct sockaddr_in servername;
int rc;
:
memset(&servername, 0, sizeof servername);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = inet_addr("129.5.24.1");
servername.sin_port = htons(5001);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof servername);
```

Figure 7. An Application Using connect()

This example attempts to connect the socket with file descriptor `s` to the server with an address specified in the `servername` variable. This could be the server that was used in [Figure 5](#) on [page 14](#). After a successful return, the socket with file descriptor `s` is associated with the connection to the server. This example also uses another handy utility routine, `inet_addr()`, which takes an internet address in dotted-decimal form and returns it as a long integer in network byte order.

[Figure 8](#) on [page 15](#) shows another example of the `connect()` call. It uses the utility routine `gethostbyname()` to find the internet address of the host rather than using `inet_addr()` with a specific address.

```
int rc;
int s;
char *hostname = "jphhost.ibm.com";
struct sockaddr_in servername;
struct hostent *hp;

hp = gethostbyname(hostname);
:

/* Clear the structure to be sure that the sin_len and */
/* sin_zero fields are clear.                               */
memset(&servername, 0, sizeof servername);
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = *(in_addr_t *) hp->h_addr;
servername.sin_port = htons(5001);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof servername);
```

Figure 8. A connect() Function Using gethostbyname()

5. Servers using stream sockets accept a connection request with the `accept()` call, as shown in [Figure 9](#) on [page 15](#).

```
int clientsocket;
int s;
struct sockaddr_in clientaddress;
int addrlen;
:
addrlen = sizeof clientaddress;
clientsocket = accept(s, (struct sockaddr *) &clientaddress, &addrlen);
```

Figure 9. An Application Using accept()

If connection requests are not pending on the socket with file descriptor `s`, the `accept()` call blocks the server (unless `s` is in nonblocking mode). When a connection request is accepted, the socket, the name of the client, and length of the client name are returned, along with a file descriptor representing a new socket. The new socket is associated with the client that began the connection, and `s` is again available to accept new connections.

6. Clients and servers have many calls from which to choose for data transfer. The `send()` and `recv()`, `readv()` and `writew()`, and `read()` and `write()` calls can be used only on sockets that are in the connected state. The `sendto()` and `recvfrom()`, and `sendmsg()` and `recvmsg()` calls can be used at any time on datagram sockets. Figure 10 on page 16 illustrates the use of `send()` and `recv()`.

```
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
int s;
:
bytes_sent = send(s, data_sent, sizeof data_sent, 0);
:
bytes_received = recv(s, data_received, sizeof data_received, 0);
```

Figure 10. An Application Using `send()` and `recv()`

This example shows an application program sending data on a connected socket and receiving data in response. The flags field can be used to specify additional options to `send()` or `recv()`, such as sending out-of-band data. (In this case no flags are being used, so 0 is passed.)

7. If the socket is not in a connected state, additional address information must be passed to `sendto()` and can be optionally returned from `recvfrom()`. An example is shown in Figure 11 on page 16.

```
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr_in from;
int addrlen;
int s;
:
memset(&to, 0, sizeof to);
to.sin_family = AF_INET;
to.sin_addr.s_addr = inet_addr("129.5.24.1");
to.sin_port = htons(5001);
bytes_sent = sendto(s, data_sent, sizeof data_sent, 0,
                   (struct sockaddr *) &to, sizeof to);
:
addrlen = sizeof from; /* must be initialized */
bytes_received = recvfrom(s, data_received,
                         sizeof data_received, 0, (struct sockaddr *) &from, &addrlen);
```

Figure 11. An Application Using `sendto()` and `recvfrom()`

The `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()` calls take additional parameters that allow the caller to specify the recipient of the data or to be notified of the sender of the data. Usually, `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()` are used for datagram sockets, and `send()` and `recv()` are used for stream sockets.

8. The `writew()`, `readv()`, `sendmsg()`, and `recvmsg()` calls provide the additional features of "scatter" and "gather" buffers, two related operations where data is received and stored in multiple buffers (scatter data), and then taken from multiple buffers and transmitted (gather data). The `writew()` and `sendmsg()` calls gather the data and send it. The `readv()` and `recvmsg()` calls receive data and scatter it into multiple buffers.
9. Applications can handle multiple file descriptors. In such situations, use the `select()` call to determine the file descriptors that have data to be read, those that are ready for data to be written, and those that have pending exceptional conditions. Figure 12 on page 17 is an example of how the `select()` call is used.

```

fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_found;
:
/* set bits in read, write, and except bit masks */
FD_ZERO(&readsocks);
FD_ZERO(&writesocks);
FD_ZERO(&exceptsocks);

FD_SET(s, &readsocks);
FD_SET(s, &writesocks);
FD_SET(s, &exceptsocks);

timeout.tv_sec=5;      /* Wait up to 5 seconds for activity */
timeout.tv_usec=0;     /* No additional microseconds */

/* First argument is number of bits in masks to check */
number_found = select(s+1,
                      &readsocks, &writesocks, &exceptsocks, &timeout);

```

Figure 12. An Application Using select()

In this example, the application program uses bit sets to indicate that the sockets are being tested for certain conditions and also indicates a timeout. If the timeout parameter is a null pointer, the `select()` call blocks until a socket becomes ready. If the timeout parameter is nonnull, `select()` waits up to this amount of time for at least one socket to become ready on the indicated conditions. This is useful for application programs servicing multiple connections that cannot afford to block, waiting for data on one connection.

10. In addition to `select()`, application programs can use the `fcntl()` or `ioctl()` calls to help perform asynchronous (nonblocking) socket operations. An example of the use of the `ioctl()` call is shown in Figure 13 on page 17.

```

int s;
int dontblock;
char buf[256];
int rc;
:
dontblock = 1;
:
rc = ioctl(s, FIONBIO, &dontblock);
:
if (recv(s, buf, sizeof buf, 0) == -1 && errno == EWOULDBLOCK)
    /* no data available */
else
    /* either got data or some other error occurred */

```

Figure 13. An Application Using ioctl()

In this example, the socket with file descriptor `s` is placed into nonblocking mode. When this file descriptor is passed as a parameter to calls that would block, such as `recv()` when data is not present, it causes the call to return with an error code, and the global `errno` value is set to `EWOULDBLOCK` or `EAGAIN`. Setting the mode of the socket to be nonblocking allows an application program to continue processing without becoming blocked.

11. A socket with file descriptor `s` is deallocated with the `close()` call, as shown in Figure 14 on page 17.

```

int rc;
int s;
rc = close(s);

```

Figure 14. An Application Using close()

z/VM C Socket Implementation

The following sections describe some important implementation details of the z/VM C socket API.

Header Files

Most of the socket header files used by the z/VM C sockets library are shipped with Language Environment. The only header file that is unique to the z/VM library is **saiucv.h**, which contains the *sockaddr* structure definition for AF_IUCV sockets.

_OE_SOCKETS Preprocessor Symbol

In general, the Language Environment header files are sensitive to whether the **_OE_SOCKETS** preprocessor symbol has been defined. In order to use the Language Environment header files for sockets programming, you must define the **_OE_SOCKETS** preprocessor symbol before you include any Language Environment header files. You can do this in your program by placing a statement similar to the following at the top of each source file:

```
#define _OE_SOCKETS
```

Alternatively, you can cause the **_OE_SOCKETS** preprocessor symbol to be defined by the compiler, by using the **-D** option on the c89 command line. See [“Compiling and Linking a Sockets Program”](#) on page 26 for more information on compiling sockets programs.

For IPv6 sockets programming (AF_INET6 sockets), the symbol **_OPEN_SYS SOCK_IPV6** must also be defined.

Function Prototypes

Although they contain function prototypes for all of the POSIX.1 functions, the Language Environment header files do not contain prototypes for all of the socket functions. Specifically, when **_OE_SOCKETS** is defined, the following socket functions are available, but have no prototypes provided:

Header File

Functions

sys/socket.h

accept(), bind(), connect(), getpeername(), getsockname(), getsockopt(), listen(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(), setsockopt(), shutdown(), socket()

netdb.h

endhostent(), endnetent(), endprotoent(), endservent(), sethostent(), setnetent(), setprotoent(), setservent()

arpa/inet.h

inet_lnaof(), inet_netof()

sys/uio.h

readv(), writev()

Because the socket functions were designed to be useful for any networking interface, the types of the parameters declared for the functions do not always exactly match the types of the arguments provided. In one sense, therefore, it is convenient that prototypes are not always provided, as it reduces the number of possible compiler warning messages because of type mismatches. On the other hand, socket programs may contain subtle bugs because of misunderstandings about the type definitions of the function parameters, so care should be taken when coding a function call. One way to ensure care is to use prototypes and to explicitly cast function arguments when necessary (and only when necessary). For example, the connect() function call accepts a pointer to a *sockaddr* structure, but the *sockaddr* structure is a generic structure not associated with any particular address family. A program using AF_INET sockets might provide a pointer to a *sockaddr_in* structure for this connect() parameter. Because these two pointer types are not compatible, an explicit cast should be used on the function call to convert the *sockaddr_in* pointer into a generic *sockaddr* pointer.

Suppressing Function Prototypes

If you are porting a large program that you know is coded correctly, you may at first receive a lot of type-mismatch compiler errors if the program is not coded to explicitly cast function arguments to their proper types. This is common when porting code from other systems, because not all systems provide function prototypes for the socket functions. To avoid correcting the function calls to perform the explicit cast operations, you can define the **_NO_PROTO** preprocessor symbol before including any header files. If the **_NO_PROTO** preprocessor symbol is defined before including any Language Environment or z/VM header file, function prototypes will be suppressed, or at least modified to omit type declarations for the function arguments. Use of this preprocessor symbol will avoid the compiler warning messages and make porting easier, but be aware that it may also obscure coding errors in the program.

If you wish to define the **_NO_PROTO** preprocessor symbol, you can do so by placing a statement similar to the following at the top of each source file of your program:

```
#define _NO_PROTO
```

Alternatively, you can cause the **_NO_PROTO** preprocessor symbol to be defined by the compiler, by using the **-D** option on the c89 command line. See [“Compiling and Linking a Sockets Program” on page 26](#) for more information on compiling sockets programs.

Multithreading

The z/VM C sockets library is a multithreading sockets library. This means that you can write programs to exploit the z/VM multithreading capabilities provided for POSIX programs, and still use socket functions without worrying about socket calls by one thread interfering with calls by another thread, or about the entire process being blocked just because one thread is blocked.

The z/VM C sockets library protects its internal data structures with mutexes, and uses thread-local data areas, where necessary, to ensure that socket calls by different threads can occur "concurrently". The z/VM C sockets library is careful to never hold one of these internal mutexes when it might block for a substantial period of time, so multiple threads can use socket functions with as much concurrency as possible.

Some function calls in the z/VM C sockets library are not "primitive" socket function calls, however. For example, the `gethostbyname()` function call is really a procedure which tries to resolve a host name by reading data from local files and by communicating with Domain Name Servers in the network.

Multithreading versus Nonblocking Sockets

In a single-threaded program, a server that wants to handle concurrent requests from multiple clients usually sets all of its sockets to nonblocking mode, so that if a socket call on behalf of one client can not be processed immediately, other client requests are not delayed. In a multithreaded server, another approach is available. Instead of setting the sockets to be nonblocking, the server can create a separate thread to handle each client request. If a call to a socket function by one thread blocks, only that client request is affected; other threads are free to continue processing requests from other clients. Using multiple threads can therefore simplify the programming model, because each thread can concentrate on a single client without worrying about any other client. Either approach is available with the z/VM C sockets library.

Conflicts Between Socket Calls

When one thread of a multithreaded program is issuing a socket function call for a given socket, other threads are restricted from issuing certain socket function calls against that same socket. These restrictions are enforced by the TCP/IP service virtual machine. The following list describes the restrictions for each type of socket call:

- Multiple read-type calls (`read()`, `readv()`, `recv()`, `recvfrom()`, or `recvmsg()`) and multiple write-type calls (`write()`, `writen()`, `send()`, `sendto()`, or `sendmsg()`) for the same socket can be in progress simultaneously. The read-type calls are satisfied in the order they are issued, independently of the write-type calls.

Similarly, the write-type calls are satisfied in the order they are issued, independently of the read-type calls.

- Multiple `accept()` calls for the same listening stream socket can be in progress simultaneously. They are satisfied in the order they are issued.
- Multiple `select()` calls referring to any combination of sockets (or other file descriptors) may be in progress simultaneously. When the state of a socket or other file descriptor changes, all active `select()` calls are checked; any that are then satisfied will return.
- Calls other than the read-type, write-type, `accept()`, and `select()` calls may not be in progress simultaneously for the same socket. For example, your program must wait for a write-type call to complete (or interrupt it) before issuing a `close()` call for the same socket.

If your program violates one of these restrictions, the function call that violates it will fail with an `ECONFLICT` error, except a `close()` call, which will fail with an `EAGAIN` error.

POSIX Signals and Thread Cancellation

The POSIX.1 standard greatly enhances the ANSI C Language definition by defining and guaranteeing certain aspects of signal processing. For example, many POSIX.1 function calls are defined to unblock, returning the `EINTR` error code, if a signal is delivered to a thread while it is blocked in a function call. POSIX.1 also defines what function calls a program may safely make while in a signal handler. Similarly, the (draft) POSIX.1c threading standard defines the conditions under which a thread may be "cancelled" by a call to the `pthread_cancel()` function, and what function calls are considered to be "cancellation points". The intent of this section is to define these attributes for the z/VM C sockets library.

Any socket function call which blocks may return `EINTR` if interrupted by a signal. That is, if a signal is caught by a thread which is blocked in a call to a socket function, that socket function will unblock and return the `EINTR` error code when the signal handler returns. The z/VM C sockets library blocks signal delivery in places when it is holding any internal mutexes, so signal delivery will occur only when the z/VM C sockets library can tolerate it. However, the following strict restriction does exist: **It is not supported for a signal handler to use `longjmp()` or `siglongjmp()` to exit from a signal handler.** In order for the z/VM C sockets library to properly recover from being interrupted, the signal handler *must* return, allowing the interrupted function call to resume from the point of interruption. This restriction only exists for signal handlers which might run because the thread was interrupted in the middle of a call to a socket function.

All z/VM C socket functions are async signal safe, and may thus be called without restriction from signal handlers.

All z/VM C socket functions are defined to *possibly* be thread cancellation points, as defined in the (draft) POSIX.1c threading standard, and *no* socket functions are defined to be async cancel safe. Any socket function which blocks will be a cancellation point. Although it is not supported by POSIX, it is safe for a cancellation cleanup handler to use `longjmp()` or `siglongjmp()` to exit, even if a function in the z/VM C sockets library was interrupted by the thread cancellation request.

Note: The difference between using `longjmp()` from a cancellation cleanup handler and using `longjmp()` from a signal handler is that in the case of cancellation, the z/VM C sockets library uses a cancellation cleanup handler of its own (which gets invoked before the application's cleanup handler) to clean up outstanding socket activity.

Sockets and Their Relationship to Other POSIX Functions

The z/VM C sockets library allocates file descriptors for sockets from the same pool of numbers that CMS uses for other open files in the POSIX environment. Among other things, this means that non-socket-specific POSIX functions can be called for file descriptors allocated to sockets. For example, the `fstat()` function can be called to retrieve information about an open socket. In the case of a socket, the `st_mode` field of the `stat` structure returned will indicate that the file descriptor is a socket. The `S_ISSOCK()` macro, defined in **sys/stat.h**, can be used to test the `st_mode` field. The `S_ISSOCK()` macro is analogous to the `S_ISREG()`, `S_ISFIFO()`, and other related macros provided in **sys/stat.h** to test for other file types.

Examples of other POSIX functions which can be called for socket file descriptors are `fchmod()`, `fchown()`, `dup()`, and `dup2()`. Note that using `fcntl()`, `dup()`, or `dup2()`, it is possible to open several file descriptors for

the same socket. When this is done, all of the file descriptors are considered to be equivalent, in the sense that none of them has any special status over the rest. A socket is closed when the last file descriptor which refers to a socket is closed.

When using AF_UNIX sockets, files are created by the bind() function call in the Byte File System, CMS's implementation of a POSIX-compliant file system. These files cannot be opened with the open() function, and are used only by the connect() function in the z/VM C sockets library. If stat() or lstat() is used on one of these files, the *st_mode* field in the return *stat* structure indicates that the file is a socket. The S_ISSOCK() macro can be used to test for this file type.

Note: Certain other function calls in the BPX layer of interfaces may report these files to be "external links".

Secure Connection Considerations

Applications can set up connections to be secure using the secure ioctl commands and data structures defined in [Table 21 on page 148](#) and [Table 22 on page 171](#). Secure connections flow through a TLS/SSL server. Once a connection is secure, any data that the application sends is encrypted by the TLS/SSL server before it is sent over the TCP connection. Any data that the application receives is decrypted by the TLS/SSL server before it is returned to the application.

Starting a Secure Connection

Use the SioCSecServer ioctl command to indicate to the TLS/SSL server that a connection is to be secure and that the TLS/SSL server must wait for an incoming handshake. Use the SioCSecClient ioctl command to indicate to the TLS/SSL server that a connection is to be secure and that the TLS/SSL server must initiate an outbound handshake on behalf of the application. The SecureDetail structure must be provided on these ioctl commands. Refer to [Table 21 on page 148](#) for details.

If non-blocking sockets are used, the application can wait for the handshake to complete by waiting for the socket to become writable or post an exception. In this case, the ioctl command completes with a return code of -1 and an ErrNo of EINPROGRESS. If the handshake fails for any reason, an exception is raised on the socket. The ErrNo presented on the subsequent read reflects the handshake error. If blocking sockets are used, the ioctl command blocks until the handshake completes.

Security can be negotiated by specifying data in the buffer field of the SecureDetail structure that gets passed on the SioCSecServer or SioCSecClient command. The buffer data can indicate to the partner application that the partner application must make the appropriate command (SioCSecServer or SioCSecClient) to secure the connection. The buffer data is sent in clear text to the partner:

1. The TLS/SSL server receives the ioctl command.
2. The TLS/SSL server immediately sends the buffer data in clear text to the partner application.
3. After sending the buffer data to the partner application, the TLS/SSL server waits for the handshake.

Stopping a Secure Connection

Use the SioCSecClose ioctl command and pass in the CloseReq structure to stop encrypted data from flowing on a TCP connection. The server application or the client application can initiate the return to a clear text connection. In the following steps, the server application initiates the process.

1. The server application issues an SioCSecClose ioctl command.
 - The SioCSecClose ioctl command can contain data in the CloseBuff buffer of the CloseReq structure. The buffer data is the last encrypted data that is sent before the SSL tunnel is closed. Any data that has not been delivered when the SioCSecClose ioctl command is issued is discarded.
2. After the server application issues the SioCSecClose ioctl command and before the client application issues the SioCSecClose ioctl command, the following events can occur:
 - The client application receives ErrNo EIBMCLRTEXT on the next read or write of the socket.
 - If the client application attempts to transmit data, the transmission fails with ErrNo of EIBMSCLSSIP, which indicates that a close is in progress.

3. When the client application issues the SioCSecClose ioctl command, the server application's SioCSecClose ioctl command completes and the connection returns to clear text.
 - If non-blocking sockets are used, the ioctl command completes with a return code of -1 and an ErrNo of EINPROGRESS. The return code and ErrNo indicate that the ioctl command was processed and is waiting for SSL to close the secure tunnel. When the client issues the SioCSecClose ioctl command, the secure tunnel is closed and the socket becomes writable.
 - If blocking sockets are used, data transmission is blocked until the SioCSecClose ioctl command completes.

Note: It is the server application's responsibility to process all data before the server application issues the SioCSecClose ioctl command. Any unprocessed data is discarded by the TLS/SSL server.

Requesting Details for a Secure Connection

Use the SIOCSECSTATUS ioctl command to request details about a session such as whether or not it is secure and the encryption suite being used. The SecStatus structure is returned. This structure provides the security level and cipher details.

Requesting Details from a Partner Certificate

Use the SIOGCERTDATA ioctl command to request specific fields from the local or partner certificate. The CertDataCompleteDetailType structure is returned. Refer to [Table 21 on page 148](#) for details.

If using blocking sockets, the ioctl will block until the certificate request completes.

If using non-blocking sockets, the application can wait for the certificate data to be returned by waiting for a read or an exception to be posted. In this case, the ioctl will complete with a return code of -1 and an ErrNo of EINPROGRESS. If the request completes with data returned, the socket will be woken up for read. The CertDataCompleteDetailType structure that is returned on the subsequent read will need to be parsed to determine the result of the request. If the certificate request fails and no data is returned, the socket will be woken up for exception. The error will be returned in the ErrNo field on the subsequent read. An ErrNo in the 40000 range indicates a System SSL error. Subtract 40000 from the ErrNo and refer to [Messages and codes in z/OS Cryptographic Services System Secure Sockets Layer Programming \(publibz.boulder.ibm.com/epubs/pdf/gsk2aa00.pdf\)](#) for details.

Determining if a TLS/SSL Server is Available

Use the SIOCTLSQUERY ioctl command, passing in the QueryTls structure, to determine if a TLS/SSL server is available, and if a label is specified, if it is known to the TLS/SSL server.

If using blocking sockets, the ioctl will block until the Query completes.

If using non-blocking sockets, the application can wait for the Query to complete by waiting for the socket to become writable or post an exception. In this case, the ioctl request will complete with a return code of -1 and an ErrNo of EINPROGRESS. If the Query fails for any reason, an exception will be raised on the socket. The ErrNo presented on the subsequent read will reflect the Query error.

Miscellaneous Implementation Notes

The following are some miscellaneous points to consider when writing socket programs for the z/VM C sockets library:

1. Most of the socket functions are defined to return an EFAULT error if an address is passed which cannot be used by the sockets library. In certain cases, your program may receive a signal such as SIGSEGV instead. A multithreading library such as the z/VM C sockets library has difficulty prechecking for all of the conditions that could cause an EFAULT error, so invalid addresses may sometimes be used instead of causing EFAULT. In the worst case, if using AF_INET sockets, the library's IUCV connection with the TCP/IP service virtual machine may be severed by TCP/IP when it receives an IUCV error because of an invalid address or length.

2. If you receive an error from a sockets function call, you may call the `perror()` or `strerror()` functions to translate the `errno` into an error message. These routines can decode socket error codes as well as error codes from non-socket functions. For compatibility with the VM TCP/IP C sockets library, the z/VM C sockets library defines a `tcperror()` function which invokes `perror()`. A small difference between `tcperror()` and `perror()` is that if the sockets library encountered an error during its most recent IUCV communication on the invoking thread, the `tcperror()` function will report that error as well as the one represented in the `errno` variable. It is not necessary to use `tcperror()` unless this additional information concerning IUCV errors is needed.
3. If you wish to use the `AF_UNIX` or `AF_IUCV` addressing domains, the virtual machines that connect to each other need authorization in their user directory entries to allow the connection to be established. This is usually handled by placing a statement in the user directory entry of the server virtual machine like the following:

```
IUCV ALLOW
```

This statement tells CP to allow any virtual machine to establish an IUCV connection to the server.

Another possibility is to place a statement like the following in the user directory entry for each client virtual machine:

```
IUCV ANY
```

This statement tells CP to allow the client virtual machine to establish an IUCV connection with any other virtual machine.

The requirement that virtual machines need authorization to connect through IUCV is true for the `AF_INET` and `AF_INET6` connections, but this is usually not a problem. `AF_INET` and `AF_INET6` connections connect through the TCP/IP server virtual machine, and that virtual machine is expected to have an `IUCV ALLOW` statement in its directory entry, which permits any client virtual machine to establish a connection with it.

Incompatibilities with the VM TCP/IP C Sockets Library

The goal of the z/VM C sockets library is to allow easier porting of UNIX programs that use sockets, and to provide a sockets API which can coexist with, and is more compatible with, the POSIX.1 API. To achieve this goal, it was often necessary to introduce incompatibilities with the TCP/IP C sockets library, because it has many incompatibilities with typical UNIX implementations. The following are some of the incompatibilities between the VM TCP/IP C sockets library and the z/VM C sockets library:

1. The names of the socket header files differ a great deal between the two libraries. For example, the z/VM C sockets library does not have a **manifest.h** header file, and you should not attempt to include one in your program. Another example is that the old **bsdtypes.h** header file has been replaced with a **sys/types.h** header file. The **time.h** header file has been replaced with two header files, **time.h** and **sys/time.h**. It is necessary for you to include the correct one (or both, if necessary) in your program. Be sure to use the header file names required by the functions as documented in this reference guide.

Do not omit the path name prefixes which are documented. For example, do not include **time.h** when you really should be including **sys/time.h**. The path name prefixes are significant.

2. Most of the header files to be used with the z/VM C sockets library are provided with Language Environment. Because those header files support several levels of socket functionality (on z/OS®), it is necessary for all z/VM C socket programs to declare the level of functionality they want by defining the **_OE_SOCKETS** preprocessor symbol *before* including any Language Environment header files. Failure to do so will usually cause several confusing compilation error messages. One way to define this preprocessor symbol is to place a statement similar to the following at the top of each source file of your program:

```
#define _OE_SOCKETS
```

Alternatively, you can cause the `_OE_SOCKETS` preprocessor symbol to be defined by the compiler, by using the `-D` option on the `c89` command line. See [“Compiling and Linking a Sockets Program”](#) on page 26 for more information on compiling sockets programs.

3. The BSD 4.4 UNIX system introduced a new field into the *sockaddr* structures used by many socket functions. For each socket address family, there is a *sockaddr_xx* structure which contains fields that define the address. For example, in the `AF_INET` address family, the structure is called *sockaddr_in* and primarily contains an IP address and port number. In the `AF_UNIX` address family, the structure is called *sockaddr_un* and primarily contains a file name. There is also a generic structure called *sockaddr*.

In the TCP/IP C sockets library definitions, such structures do not contain self-defining length fields. Each socket function that accepts a *sockaddr* structure also accepts a length, so there is really no need for them to contain lengths within the structure. In the 4.4 BSD UNIX implementation, however, there is now a length field in the *sockaddr* structure, so the Language Environment header files have them too. The z/VM C sockets library uses them, therefore, when processing those functions. When a *sockaddr* structure is given to the z/VM C sockets library, the length fields are handled as follows:

- For `AF_INET` and `AF_INET6` sockets, the library verifies that the length field is either 0 or `sizeof(struct sockaddr_in)`. If the library sees a zero length, it assumes that the application does not know about length fields, and uses `sizeof(struct sockaddr_in)` instead. If it sees any other length value, it rejects the socket request with `EINVAL`.
- For `AF_UNIX` sockets, if the length field is nonzero, the library uses it to limit how much of the file name is examined; a zero length field is ignored.
- For `AF_IUCV` sockets, if the length field is nonzero, the library checks to make sure it is equal to `sizeof(struct sockaddr_iucv)`.

If you do not explicitly initialize the *sockaddr* length field, then, depending on how the storage is allocated, you might have an unintended value there, and get unexpected `EINVAL` errors. This is more of a problem for `AF_UNIX` sockets than for `AF_INET` and `AF_INET6` sockets. The reason is that an `AF_INET` or `AF_INET6` *sockaddr* structure already contains a field which must be zeros, so most robust applications use `memset()` to zero the entire *sockaddr* structure before filling it in. Because the z/VM C sockets library treats a zero *sockaddr_in* length field the same as if `sizeof(struct sockaddr_in)` were specified, robust `AF_INET` and `AF_INET6` applications need no changes to deal with *sockaddr* length fields. `AF_UNIX` *sockaddr* structures have no fields which must be zero, however, so it is less likely that the structure will be cleared before filling it in, especially since the full size is so much bigger. Having un-initialized data in that length field might cause the socket library to use less of the file name than you intend.

A simple method to check code you are porting for proper length-field handling is to search for places that initialize the "family" field, which is called *sin_family* for `AF_INET`, *sin6_family* for `AF_INET6` sockets, and *sun_family* for `AF_UNIX` sockets. If there is a call to `memset()` just before this code to clear the entire structure, you are probably safe. If not, you should fill in the length field. For `AF_INET` sockets, either fill it in with 0 or `sizeof(struct sockaddr_in)`. For `AF_INET6` sockets, either fill it in with 0 or `sizeof(struct sockaddr_in6)`. For `AF_UNIX` sockets, either fill it in with 0 or a value greater than or equal to `SUN_LEN(&sa)`, where *sa* is the name of the *sockaddr_un* variable, but less than or equal to `sizeof(struct sockaddr_un)`. For more information and examples of how to initialize the *sockaddr* length fields, see the `bind()` and `connect()` functions in [XL C/C++ for z/VM: Runtime Library Reference](#).

4. The z/VM C sockets library supports the `selectex()` function call. No `WAITECBs` are done, because the CMS OS `WAIT` and `POST` are not multitasking-aware and are not interruptable. Instead, the `ECB` post bits are checked directly during a polling loop that processes socket and file descriptors. If a set post bit is found, then `selectex()` stops processing and returns to its caller.

Consider replacing the usage of `ECBs` with POSIX constructs such as condition variables. You can then create a thread that waits on the condition variable, and when the condition being waited for has really occurred, it could signal a thread blocked in a `select()` call, if necessary, to cause it to exit the `select()`. In many cases, the signal is not even necessary; the thread that waited on the condition variable could process the event itself without disturbing the other threads.

5. If your program calls `tcperror()` instead of `perror()`, you must define the constant `_OPEN_SOURCE` as follows:

```
#define _OPEN_SYS_SOCKET_EXT
```

This will cause `tcperror()` to be mapped to `perror()`.

6. If you used the TCP/IP C sockets library, and did not call the `maxdesc()` function to increase the number of sockets you could open, your program was guaranteed that no socket descriptor would be greater than 49. This guarantee may have been exploited by programs, allowing them to set the `FD_SETSIZE` preprocessor symbol to a very small value. The `FD_SETSIZE` preprocessor symbol is used by the **sys/time.h** header file to control how much storage it takes to hold an *fd_set*, as used by the `select()` function call. When using the z/VM C sockets library, there is no relationship between file descriptor numbers as allocated by CMS and the value of the `FD_SETSIZE` preprocessor symbol. Another change to keep in mind is that the default `FD_SETSIZE` in the Language Environment **sys/time.h** header file is 2048, much larger than the default of 256 in the TCP/IP **bsdtypes.h** header file.
7. Programs compiled with the TCP/IP C sockets library header files must be recompiled before they can be link edited to the z/VM C sockets library. The two sets of header files do not produce object-compatible code. For example, the external symbol names associated with socket functions have changed, and the `errno` mapping is quite different.
8. If your program includes BSD header files (**bsdtypes.h**, **bsdtime.h**, **bsdtoCMS.h**), you must remove those includes. The z/VM C sockets library covers BSD functions, but it does not provide those header files.
9. In the TCP/IP C sockets library, `getdtablesize()` returns the maximum number of socket descriptors. In the z/VM C sockets library, `getdtablesize()` functions as it really should, returning the maximum number of file descriptors. The z/VM C sockets library provides `getstablesz()` for determining the maximum number of socket descriptors. If an existing TCP/IP application that uses `getdtablesize()` is being rebuilt with the z/VM C sockets library, to get the same results as before you must either use the `APPTYPE` environment variable or change the `getdtablesize()` call to `getstablesz()`.
10. The TCP/IP remote procedure calls library (RPCLIB) cannot be used with the z/VM C sockets library. Use the VMRPC library instead. For more information about RPC, see [Chapter 5, "Remote Procedure Calls,"](#) on page 187.

Incompatibilities with z/OS and OS/390 C Sockets

The z/VM C socket API is equivalent to the OS/390® Language Environment 2.5 sockets subset, except the following functions have not been implemented in z/VM:

- `accept_and_recv()`
- `aio_read()`
- `aio_write()`
- `poll()`
- `send_file()`
- `socketpair()`
- `srx_np()`

Incompatibilities with the Berkeley Socket Implementation

The following list summarizes some of the differences between the z/VM C socket implementation and the Berkeley socket implementation:

1. The z/VM `ioctl()` implementation may be different from the current Berkeley `ioctl()` implementation.
2. The z/VM `getsockopt()` and `setsockopt()` calls support only a subset of the options available, and only for the `AF_IUCV`, `AF_INET`, and `AF_INET6` address families.

3. In the z/VM C socket API, the AF_UNIX address family does not support datagram sockets or nonblocking mode.
4. In the z/VM socket API, select() is used to determine when an asynchronous secure ioctl call completes.

Compiling and Linking a Sockets Program

This section describes how to compile and link-edit C Language programs that use the z/VM C sockets library.

Note: An existing application that currently uses the VM TCP/IP C sockets library (COMMTXT) may continue to do so in exactly the same way it did before, without any modification. Also, the application may also continue to use the RPCLIB TXTLIB without modification. See [“Running a Sockets Program”](#) on page 29. However, to use the z/VM C socket functions, the application may need to be recompiled. See [“Compiling and Linking a TCP/IP C Sockets Program”](#) on page 28.

Compiling and Linking a z/VM C Sockets Program

To compile z/VM C socket programs, you must have the IBM C for VM/ESA Compiler 3.1 (5654-033) and IBM Language Environment (supplied with z/VM) installed on your z/VM system. In order to use AF_INET sockets, you must have TCP/IP installed and running.

To compile and link-edit a z/VM C sockets application program, use the c89 utility. You must make sure that c89 has access to the files it needs to compile and link-edit. The VM-unique header files reside on the CMS S-disk. The Language Environment object code and header files reside on the Y-disk.

Another aspect to ensuring that c89 has all required files available is to make sure that you have a Byte File System mounted and available. The files and directories in this Byte File System must be arranged in the manner done by the BFS installation procedures. Specifically, the `/usr/include/sys/time.h` file is assumed to be an external link of type CMSDATA to the SYS_TIME H file.

The c89 program can be run from the CMS command line (or equivalent) or from within a POSIX command shell, if you are running with a command shell. The syntax is the same in both cases. For example, if you wish to compile the `testprog.c` file in your current Byte File System (BFS) directory and bind the socket function stubs to it, use a command like the following:

```
c89 -o testprog -D_OE_SOCKETS testprog.c
```

Depending on the TCP/IP functions your application uses, additional libraries (listed in [Table 2 on page 26](#)) may be required. For example, if `testprog.c` uses RPC functions, the command would be:

```
c89 -o testprog -D_OE_SOCKETS testprog.c -l//VMRPC
```

Table 2. TCP/IP TXTLIB Files and Applications

TXTLIB File	Application
VMRPC	Remote procedure calls
SCEELKED	C APIs
SCEE OBJ	C writable static variables

The previous examples assume that `testprog.c` is the only source file in the program, and that the `_OE_SOCKETS` preprocessor symbol is not defined in the source file itself. If it is, then do not specify the `-D` option. To avoid having to type the `-D` option all the time, or including it in your make file, put the following statement in the beginning of every source file of your program, before it includes its first header file:

```
#define _OE_SOCKETS
```

If you want to use the Language Environment extended socket and bulkmode support, define the feature test macro `_OPEN_SYS SOCK_EXT` using a preprocessor directive, either on the c89 compile command line:

```
c89 filename -D_OPEN_SYS SOCK_EXT
```

or in the source code before including any header files:

```
#define _OPEN_SYS SOCK_EXT
```

For more information about this macro, see [XL C/C++ for z/VM: Runtime Library Reference](#).

Sometimes defining `_OE_SOCKETS` in the source program itself is inconvenient because, for example, you are porting many source files from another system, and you would rather not change them all. In this case, define `_OE_SOCKETS` on the c89 command line with the `-D` option, either by hand or in your make file, if you are using the make utility. For more information about the make utility, see [z/VM: OpenExtensions Advanced Application Programming Tools](#).

The `-o` option in the example above tells c89 to store the final executable file with the name `testprog`. This overrides the default name of `a.out`.

When you compile, be very careful *not* to have the disk containing the header files for TCP/IP accessed ahead of the disk containing the Language Environment header files. Because both TCP/IP and Language Environment ship socket header files with the same names, it is important to use the correct Language Environment header files, and not the TCP/IP files. No TCP/IP header files are needed to compile z/VM C socket programs. However, if you are using RPC functions, then header files on the TCP/IP disk are required.

If you would rather have c89 produce a MODULE file on a CMS minidisk or accessed SFS directory, then specify something like the following on your c89 command:

```
c89 -o //testprog -D_OE_SOCKETS testprog.c
```

This will cause c89 to create a CMS file called `TESTPROG MODULE A` instead of an executable file in the BFS.

If the source file itself is on a CMS minidisk or accessed SFS directory, then use a c89 command like the following:

```
c89 -D_OE_SOCKETS //testprog.c
```

This example adds `//` to the front of the name of the source file, and removes the `-o` option. It adds the `//` because the source file resides on a CMS minidisk. The `-o` option is removed because when the `testprog.c` source file is on a minidisk, the c89 default name for the executable file is `//testprog.module.a`, so there is no need to specify it explicitly.

Note: It is not necessary to use uppercase letters in the name, type, or mode of a CMS file when the file ID is preceded with `//`. The file ID is converted to uppercase automatically.

If your program is composed of several source files, for example `progfile1.c` and `progfile2.c`, you can use either of the two following sequences to produce an executable file.

```
c89 -c -D_OE_SOCKETS progfile1.c
c89 -c -D_OE_SOCKETS progfile2.c
c89 -o testprog progfile1.o progfile2.o
```

or

```
c89 -o testprog -D_OE_SOCKETS progfile1.c progfile2.c
```

In the first sequence, the source files are first compiled (the `-c` option prevents c89 from trying to link edit them) and then link-edited in a separate c89 command. In the second sequence, the source files are compiled and link-edited in one command. The point being demonstrated in this example is that the `-D` option is needed only for the compilation step.

Many other variations of the c89 command are possible. See [z/VM: OpenExtensions Commands Reference](#) for a complete description of the c89 command.

After you have created an executable program, you can use the OPENVM GET and OPENVM PUT commands to move it back and forth between a CMS minidisk or accessed SFS directory and the Byte File System. See [z/VM: OpenExtensions Commands Reference](#) for information on those commands.

Compiling and Linking a TCP/IP C Sockets Program

If you want to recompile and relink an existing application that was built with VM TCP/IP C sockets, you have three choices:

- Convert the program to use the z/VM C sockets library
- Recompile and relink using the z/VM C sockets library with minimal changes to the program source
- Recompile using the TCP/IP C sockets library

Converting Your Program to Use z/VM C Sockets

To convert a TCP/IP C sockets program to use z/VM C sockets:

1. Go to [“Incompatibilities with the VM TCP/IP C Sockets Library”](#) on page 23. Make the necessary changes to your program to resolve the incompatibilities.
2. Go to [“Compiling and Linking a z/VM C Sockets Program”](#) on page 26 and follow the instructions.

Using the z/VM C Sockets Library with Minimal Changes to Program Source

You can recompile and relink your VM TCP/IP C sockets program to use the z/VM C sockets library (SCEELKED) with little or no source code modification:

- If your program uses remote procedure calls, you must use the VMRPC library instead of RPCLIB.
- If your program includes BSD header files (**bsdtypes.h**, **bsdtime.h**, **bsdto cms.h**), you must remove those includes. Language Environment covers BSD functions, but it does not provide those header files.
- Define the feature test macro `_TCPVM_SOCKETS` using a preprocessor directive, either on the c89 command line:

```
c89 filename -D_TCPVM_SOCKETS
```

or in the source code before including any header files:

```
#define _TCPVM_SOCKETS
```

Recompiling with the TCP/IP C Sockets Library

To recompile with the TCP/IP C sockets library:

1. Access the TCP/IP Client-code minidisk (usually TCPMAINT 592) *ahead* of the disk that contains the Language Environment header files (usually the Y-disk) to avoid conflicts.
2. Establish the C development environment:
 - a. Access the C compiler.
 - b. Issue `GLOBAL LOADLIB SCEERUN`.
3. Compile your program. Make sure that the preprocessor symbol VM is defined; if it is not already defined in your program, you can specify it on the compile command:

```
CC myprog (DEF(VM))
```

With OpenExtensions, you can also use the c89 command or the make utility.

4. Select the link libraries your application needs and put them on a `GLOBAL TXTLIB` command. `COMMTXT` and `SCEELKED` are the minimum required:

```
GLOBAL TXTLIB COMMTXT SCEELKED
```

Additional libraries (listed in Table 3 on page 29) may be required, depending on the functions your application uses. For example, programs that use RPC must issue:

```
GLOBAL TXTLIB COMMTXT RPCLIB SCEELKED
```

Note that the Language Environment text library, SCEELKED, should always be listed last.

Table 3. TCP/IP TXTLIB Files and Applications

TXTLIB File	Application
COMMTXT	TCP/IP C sockets and Pascal API
RPCLIB	Remote procedure calls
X11LIB	Xlib, Xmu, Xext, and Xau routines
OLDXLIB	X Release 10 compatibility routines
XTLIB	X Intrinsics
XAWLIB	Athena widget set
XMLIB	OSF/Motif-based widget set
DPILIB	SNMP DPI
SCEELKED	C APIs
SCEE OBJ	C writable static variables

5. Link-edit your programs into an executable module. The sample applications in this book are built using the TCPLOAD utility. Your own applications should be built using the CMOD command. For example,

```
TCPLOAD sample@c c
```

or

```
CMOD myprog1 myprog2 (AUTO
```

Complete information on compiling and link-editing C programs can be found in the [z/OS: Language Environment Programming Guide \(https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf\)](https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf). For information about TCPLOAD, see [Appendix A, “TCPLOAD EXEC,”](#) on page 335.

Running a Sockets Program

After building your executable sockets program, the next step is to run the program. Before you do, however, some preparation may be necessary. In addition, you may want to consider using environment variables to affect certain aspects of the execution. There are also differences between running a program from the BFS and running it from an accessed minidisk or SFS directory.

Preparing to Run a Sockets Program

If your program uses AF_INET sockets, then you should access the TCP/IP "client" minidisk or SFS directory that contains the TCP/IP configuration files. Usually you can LINK to TCPIP 592 to access the disk. Your installation may have assigned a VMLINK nickname to this minidisk (for example, TCPIP). Issue a VMLINK command (with no arguments) to see if one has been assigned. In the compilation step, it was noted that this disk contains some header files with the same names as Language Environment and VM-unique header files. If you might compile your program again after running it, be sure to access the TCP/IP client disk at a mode *after* the disk that contains the Language Environment header files.

To run your POSIX program, it should reside on an accessed minidisk or SFS directory, or in a mounted BFS file system. As described below, if it resides on an accessed minidisk or SFS directory, you can run it by either typing the name of the executable module on the CMS command line, or by using the OPENVM RUN command. If it resides in the BFS, you can run it by either typing the name of the executable file on a POSIX shell command line, or by using the OPENVM RUN command. Before running a module that resides on an accessed minidisk or SFS directory, either by using the OPENVM RUN command or by simply typing the name of the module on the CMS command line, you must establish the proper run-time Language Environment load library with the following CMS command:

```
GLOBAL LOADLIB SCEERUN
```

Note: If you have a TCP/IP C sockets application that you have recompiled and relinked with the z/VM C sockets library, with no source changes, but you want the `maxdesc()` and `getdtablesize()` functions to operate the same way they did in the TCP/IP C socket API, you must set the APPTYPE environment variable (to the value OLDAPP) before running your program. For example:

```
GLOBALV SELECT CENV SETLP APPTYPE OLDAPP
```

This will cause the `maxdesc()` default to be 50 and `getdtablesize()` to return a maximum of 50.

Using Environment Variables

Environment variables can be used to affect certain aspects of the execution of a z/VM C sockets program. If the z/VM C sockets program is executed from the OpenExtensions shell, the shell controls the contents of the program's environment. For example, the following shell command could be used to set the APPTYPE environment variable:

```
export APPTYPE=OLDAPP
```

If the z/VM C sockets program is being run from the CMS command line (or equivalent), then the global variables existing in the CENV group managed by the GLOBALV command are used as the environment variables for the process. In this case, a CMS command like the following could be used to temporarily set the APPTYPE environment variable:

```
GLOBALV SELECT CENV SET APPTYPE OLDAPP
```

Note: Be aware, however, that some of the environment variables described below accept values which are case sensitive, and which will often be set to lowercase values. It can be difficult, using the GLOBALV command, to set lowercase values, because commands typed in from the CMS command line are automatically uppercased by CMS before processing. One way to set the variable to a mixed-case value is to issue the GLOBALV command from a REXX exec with "Address Command" in effect.

Some of the environment variables described below are set to values which represent file names. For these environment variables, the given file names are interpreted as POSIX-style file names, which means that case is significant, and that the file name is interpreted as residing in the Byte File System unless you precede the file name with two slashes. To specify the name of a file which resides on a minidisk or accessed SFS directory instead of in the BFS, precede the name of the file with two slashes, and separate the CMS file name and type (and mode, if specified) with a period.

The following environment variables can be used to affect the execution of the Language Environment sockets library:

Variable	Description
----------	-------------

APPTYPE	This environment variable forces the z/VM C socket functions <code>maxdesc()</code> and <code>getdtablesize()</code> to return the same values that those functions returned in the TCP/IP C socket API.
----------------	--

HOSTALIASES	This environment variable tells the socket library resolver code to use the named file when searching for aliases for AF_INET host names. For example, setting the variable to the string <code>/etc/aliases</code>
--------------------	---

tells the resolver to use the `/etc/aliases` file when needed. By default, no aliases file is used by the resolver. If you intend to transfer an aliases file into the BFS using the `OPENVM PUTBFS` command, be sure to specify the `BFSLINE NL` option, or let it default.

X_ADDR

This environment variable tells the socket library resolver code to use the named file in place of the `HOSTS ADDRINFO` file, which contains information about `AF_INET` networks known to this host. For example, setting the variable to the string `/etc/addr` tells the resolver to use the `/etc/addr` file in place of the default file, which is `//HOSTS.ADDRINFO`. If you intend to transfer the `HOSTS ADDRINFO` file into the BFS using the `OPENVM PUTBFS` command, be sure to specify the `BFSLINE NONE` option. This environment variable is used by the `gethostbyaddr()` function call, the `getnetent()` function call, and several others.

Note: When using the TCP/IP C sockets library, the format of this environment variable is **X-ADDR**.

X_SITE

This environment variable tells the socket library resolver code to use the named file in place of the `HOSTS SITEINFO` file, which contains information about `AF_INET` hosts known to this host. For example, setting the variable to the string `/etc/hosts` tells the resolver to use the `/etc/hosts` file in place of the default file, which is `//HOSTS.SITEINFO`. If you intend to transfer the `HOSTS SITEINFO` file into the BFS using the `OPENVM PUTBFS` command, be sure to specify the `BFSLINE NONE` option. This environment variable is used by the `gethostbyname()` function call, the `gethostent()` function call, and several others.

Note: When using the TCP/IP C sockets library, the format of this environment variable is **X-SITE**.

X_XLATE

This environment variable tells the socket library resolver code to use the named file in place of the `STANDARD TCPXLBIN` file, which contains ASCII to EBCDIC and EBCDIC to ASCII translation tables for use by the resolver when sending or receiving information from an `AF_INET` network. For example, setting the variable to the string `/etc/xlate` tells the resolver to use the `/etc/xlate` file in place of the default file, which is `//STANDARD.TCPXLBIN`. If you intend to transfer the `STANDARD TCPXLBIN` file into the BFS using the `OPENVM PUTBFS` command, be sure to specify the `BFSLINE NONE` option. This environment variable is used by the `gethostbyname()` and `gethostbyaddr()` function calls.

Note: When using the TCP/IP C sockets library, the format of this environment variable is **X-XLATE**.

Running a Program Residing in the BFS

If you are using a POSIX command shell, and the executable file is in your path, then simply type the name of the program to run it:

```
testprog
or
/dirname/dirname/.../testprog
```

In this scenario, the shell will spawn a process to run your program. Any program which is spawned is automatically considered by CMS to be a POSIX program, and will have access to OpenExtensions services.

If you are not using a POSIX command shell, then use the `OPENVM RUN` command to execute the program. For example:

```
OPENVM RUN /dirname/dirname/.../testprog
```

In this scenario, the `OPENVM RUN` command will spawn a process to run your program. As before, your program will automatically have access to OpenExtensions services. Be aware that path name arguments to the `OPENVM RUN` command are case sensitive.

Running a Program Residing on an Accessed Minidisk or SFS Directory

If the executable file is on an accessed minidisk or SFS directory, then there are several ways to execute it.

The OPENVM RUN command, which was mentioned earlier to run a program residing in the BFS, can also run a MODULE file. To do so, uppercase the file name on the command line, as follows:

```
OPENVM RUN TESTPROG
```

If the executable file is on an accessed minidisk or SFS directory, you can type the name of the module on the CMS command line to run it, but you must also tell CMS that the program is a POSIX program and should be given access to the OpenExtensions services.

The following are the two techniques for establishing your program as a POSIX program when you run it:

1. Specify the Language Environment POSIX(ON) run-time option on the CMS command line. In order to be able to pass run-time options to a program, the EXECOPS compiler option must be in effect when the program is compiled. Because it is the default setting, EXECOPS is in effect unless overridden with the NOEXECOPS option. Specify the run-time options by separating them from the program arguments with a slash (/) as you run your program:

```
testprog runopt1 runopt2 ... / arg1 arg2 arg3 ...
```

To specify the POSIX(ON) run-time option, use a command like the following:

```
testprog posix(on)/arg1 arg2 arg3 ...
```

2. Specify the Language Environment POSIX(ON) run-time option by putting it in the source file containing the main function. If you plan to run the program often by simply typing its name on the CMS command line, the most convenient way to get the program recognized as a POSIX program is to place a pragma like the following in the source file which contains the main function:

```
#pragma runopts(posix(on))
```

With this pragma, you never need to type the POSIX(ON) run-time option.

When you run a program by typing its name on the CMS command line, and the EXECOPS compiler option was in effect when you compiled your program (it is by default), then everything before the first slash, if there is a slash on the command line, will be interpreted as a run-time option. Because POSIX path names often contain slashes, this can cause program arguments to be misinterpreted as run-time options if your program accepts a POSIX path name as an argument. To avoid this, consider placing a pragma like the following in the source file containing the main function:

```
#pragma runopts(noexecops,posix(on))
```

This will prevent POSIX path names from accidentally being interpreted as run-time options, and cause the POSIX(ON) run-time option to always be in effect.

Using pragma statements such as the ones discussed above is necessary only when you intend to run your program from the CMS command line by typing its name. If you use OPENVM RUN to run the program, or run it from a POSIX command shell, all operands are interpreted as program arguments, and the program is automatically treated as a POSIX program.

C Sockets Quick Reference

This section provides brief descriptions of the z/VM C socket calls. For additional information about these socket functions, see the *XL C/C++ for z/VM: Runtime Library Reference*.

Table 4. C Sockets Quick Reference

Socket() Call	Description
accept()	Accepts a connection request from a foreign host.
bind()	Assigns a local address to a socket.
close()	Closes a socket.
connect()	Requests a connection to a foreign host.
endhostent()	Closes the HOSTS SITEINFO file.
endnetent()	Closes the HOSTS ADDRINFO file.
endprotoent()	Closes the ETC PROTO file.
endservent()	Closes the ETC SERVICES file.
fcntl()	Controls socket operating characteristics.
freeaddrinfo()	Frees one or more addrinfo structures returned by getaddrinfo()
gai_strerror()	Returns information on errors returned by getaddrinfo() and getnameinfo()
getaddrinfo()	Resolves IP addresses (IPv4 or IPv6)
getclientid()	Returns the identifier by which the calling application is known to the TCPIP virtual machine.
gethostbyaddr()	Returns information about a host specified by an address.
gethostbyname()	Returns information about a host specified by a name.
gethostent()	Returns the next entry in the HOSTS SITEINFO file.
gethostid()	Returns the unique identifier of the current host.
gethostname()	Returns the standard name of the current host.
getnameinfo()	Translates a socket address to a node name and service location
getnetbyaddr()	Returns the network entry specified by address.
getnetbyname()	Returns the network entry specified by name.
getnetent()	Returns the next entry in the HOSTS ADDRINFO file.
getpeername()	Returns the name of the peer connected to a socket.
getprotobyname()	Returns a protocol entry specified by name.
getprotobynumber()	Searches the ETC PROTO file for a specified protocol number.
getprotoent()	Returns the next entry in the ETC PROTO file.
getservbyname()	Returns a service entry specified by name.
getservbyport()	Returns a service entry specified by port number.
getservent()	Returns the next entry in the SERVICES file.
getsockname()	Obtains the local socket name.
getsockopt()	Gets options associated with sockets in the AF_IUCV, AF_INET, and AF_INET6 domains.
givesocket()	Tells TCPIP to make the specified socket available to a takesocket() call issued by another application.

Table 4. C Sockets Quick Reference (continued)

Socket() Call	Description
htonl()	Translates host byte order to network byte order for a long integer.
htons()	Translates host byte order to network byte order for a short integer.
if_freenameindex()	Frees storage allocated by if_nameindex()
if_indextoname()	Maps a network interface index to its corresponding name
if_nameindex()	Returns all network interface names and indexes
if_nametoindex()	Maps a network interface name to its corresponding index
inet_addr()	Constructs an internet address from character strings set in standard dotted-decimal notation.
inet_lnaof()	Returns the local network portion of an internet address.
inet_makeaddr()	Constructs an internet address from a network number and a local address.
inet_netof()	Returns the network portion of the internet address in network byte order.
inet_network()	Constructs a network number from character strings set in standard dotted-decimal notation.
inet_ntoa()	Returns a pointer to a string in dotted-decimal notation.
inet_ntop()	Converts a binary IP address (IPv4 or IPv6) into string format
inet_pton()	Converts an IP address (IPv4 or IPv6) in string format to binary format
ioctl()	Performs special operations on a socket.
listen()	Indicates that a stream socket is ready for a connection request from a foreign host.
maxdesc()	Allows socket numbers to extend beyond default range of 0 - 49.
ntohl()	Translates network byte order to host byte order for a long integer.
ntohs()	Translates network byte order to host byte order for a short integer.
read()	Reads a set number of bytes into a buffer.
readv()	Obtains data from a socket and reads this data into specified buffers.
recv()	Receives messages from a connected socket.
recvfrom()	Receives messages from a datagram socket, regardless of its connection status.
recvmsg()	Receives messages on a socket into an array of buffers.
select()	Monitors activity on a set of sockets.
selectex()	Monitors activity on a set of different sockets.
send()	Transmits messages to a connected socket.
sendmsg()	Sends messages on a socket from an array of buffers.
sendto()	Transmits messages to a datagram socket, regardless of its connection status.
sethostent()	Opens the HOSTS SITEINFO file at the beginning.

Table 4. C Sockets Quick Reference (continued)

Socket() Call	Description
setnetent()	Opens the HOSTS ADDRINFO file at the beginning.
setprotoent()	Opens the ETC PROTO file at the beginning.
setservent()	Opens the ETC SERVICES file at the beginning.
setsockopt()	Sets options associated with a socket in the AF_IUCV, AF_INET, and AF_INET6 domains.
shutdown()	Shuts down all or part of a full-duplex connection.
socket()	Requests that a socket be created.
takesocket()	Acquires a socket from another application.
write()	Writes a set number of bytes from a buffer to a socket.
writev()	Writes data in the buffers specified by an array of iovec structures.

TCP Client Program

The following is an example of a C socket TCP client program.

```

/*
 * Include Files.
 */
#define VM
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

/*
 * Client Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;          /* port client will connect to */
    char buf[12];                 /* data buffer for sending and receiving */
    struct hostent *hostnm;        /* server host name information */
    struct sockaddr_in server;     /* server address */
    int s;                        /* client socket */

    /*
     * Check Arguments Passed. Should be hostname and port.
     */
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
        exit(-1);
    }

    /*
     * The host name is the first argument. Get the server address.
     */
    hostnm = gethostbyname(argv[1]);
    if (hostnm == (struct hostent *) 0)
    {
        fprintf(stderr, "Gethostbyname failed\n");
        exit(-1);
    }

    /*
     * The port is the second argument.
     */
    port = (unsigned short) atoi(argv[2]);

```

```

/*
 * Put a message into the buffer.
 */
strcpy(buf, "the message");

/*
 * Put the server information into the server structure.
 * The port must be put into network byte order.
 */
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);

/*
 * Get a stream socket.
 */
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket()");
    exit(-1);
}

/*
 * Connect to the server.
 */
if (connect(s, &server, sizeof(server)) < 0)
{
    perror("Connect()");
    exit(-1);
}

if (send(s, buf, sizeof(buf), 0) < 0)
{
    perror("Send()");
    exit(-1);
}

/*
 * The server sends back the same message. Receive it into the buffer.
 */
if (recv(s, buf, sizeof(buf), 0) < 0)
{
    perror("Recv()");
    exit(-1);
}

/*
 * Close the socket.
 */
close(s);

printf("Client Ended Successfully\n");
exit(0);
}

```

TCP Server Program

The following is an example of a C socket TCP server program.

```

/*
 * Include Files.
 */
#define VM
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <stdio.h>

/*
 * Server Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;          /*port server binds to */

```

```

char buf[12];           /*buffer for sending and receiving data */
struct sockaddr_in client; /*client address information */
struct sockaddr_in server; /*server address information */
int s;                 /*socket for accepting connections */
int ns;                /*socket connected to client */
int namelen;           /*length of client name */

/*
 * Check arguments. Should be only one: the port number to bind to.
 */

if (argc != 2)
{
    fprintf(stderr, "Usage:%s port\n", argv[0]);
    exit(-1);
}
/*
 * First argument should be the port.
 */
port = (unsigned short)atoi(argv[1]);

/*
 * Get a socket for accepting connections.
 */
if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket()");
    exit(-1);
}

/*
 * Bind the socket to the server address.
 */
server.sin_family = AF_INET;
server.sin_port = htons(port);
server.sin_addr.s_addr = INADDR_ANY;

if (bind(s, &server, sizeof(server)) < 0)
{
    perror("Bind()");
    exit(-1);
}

/*
 * Listen for connections. Specify the backlog as 1.
 */
if (listen(s, 1) != 0)
{
    perror("Listen()");
    exit(-1);
}

/*
 * Accept a connection.
 */
namelen = sizeof(client);
if ((ns = accept(s, &client, &namelen)) == -1)
{
    perror("Accept()");
    exit(-1);
}

/*
 * Receive the message on the newly connected socket.
 */
if (recv(ns, buf, sizeof(buf), 0) == -1)
{
    perror("Recv()");
    exit(-1);
}

/*
 * Send the message back to the client.
 */
if (send(ns, buf, sizeof(buf), 0) < 0)
{
    perror("Send()");
    exit(-1);
}

close(ns);
close(s);

```

```

    printf("Server ended successfully\n");
    exit(0);
}

```

UDP Client Program

The following is an example of a C socket UDP client program.

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

main(argc, argv)
int argc;
char **argv;
{
    int s;
    unsigned short port;
    struct sockaddr_in server;
    char buf[32];

    /* argv[1] is internet address of server argv[2] is port of server.
     * Convert the port from ascii to integer and then from host byte
     * order to network byte order.
     */
    if(argc != 3)
    {
        printf("Usage: %s <host address> <port> \n",argv[0]);
        exit(1);
    }
    port = htons(atoi(argv[2]));

    /* Create a datagram socket in the internet domain and use the
     * default protocol (UDP).
     */
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket()");
        exit(1);
    }

    /* Set up the server name */
    server.sin_family = AF_INET; /* Internet Domain */
    server.sin_port = port; /* Server Port */
    server.sin_addr.s_addr = inet_addr(argv[1]); /* Server's Address */

    strcpy(buf, "Hello");

    /* Send the message in buf to the server */
    if (sendto(s, buf, (strlen(buf)+1), 0, &server, sizeof(server)) < 0)
    {
        perror("sendto()");
        exit(2);
    }

    /* Deallocate the socket */
    close(s);
}

```

UDP Server Program

The following is an example of a C socket UDP server program.

```

#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
#include <in.h>
#include <socket.h>
#include <netdb.h>
#include <stdio.h>

```

```

main()
{
    int s, namelen, client_address_size;
    struct sockaddr_in client, server;
    char buf[32];

    /*
     * Create a datagram socket in the internet domain and use the
     * default protocol (UDP).
     */
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket()");
        exit(1);
    }

    /*
     * Bind my name to this socket so that clients on the network can
     * send me messages. (This allows the operating system to demultiplex
     * messages and get them to the correct server)
     *
     * Set up the server name. The internet address is specified as the
     * wildcard INADDR_ANY so that the server can get messages from any
     * of the physical internet connections on this host. (Otherwise we
     * would limit the server to messages from only one network
     * interface.)
     */
    server.sin_family      = AF_INET; /* Server is in Internet Domain */
    server.sin_port        = 0;       /* Use any available port */
    server.sin_addr.s_addr = INADDR_ANY; /* Server's Internet Address */

    if (bind(s, &server, sizeof(server)) < 0)
    {
        perror("bind()");
        exit(2);
    }

    /* Find out what port was really assigned and print it */
    namelen = sizeof(server);
    if (getsockname(s, (struct sockaddr *) &server, &namelen) < 0)
    {
        perror("getsockname()");
        exit(3);
    }

    printf("Port assigned is %d\n", ntohs(server.sin_port));

    /*
     * Receive a message on socket s in buf of maximum size 32
     * from a client. Because the last two parameters
     * are not null, the name of the client will be placed into the
     * client data structure and the size of the client address will
     * be placed into client_address_size.
     */
    client_address_size = sizeof(client);

    if (recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *) &client,
                &client_address_size) < 0)
    {
        perror("recvfrom()");
        exit(4);
    }

    /*
     * Print the message and the name of the client.
     * The domain should be the internet domain (AF_INET).
     * The port is received in network byte order, so we translate it to
     * host byte order before printing it.
     * The internet address is received as 32 bits in network byte order
     * so we use a utility that converts it to a string printed in
     * dotted decimal format for readability.
     */
    printf("Received message %s from domain %s port %d internet"
           "address %s\n",
           buf,
           (client.sin_family == AF_INET ? "AF_INET" : "UNKNOWN"),
           ntohs(client.sin_port),
           inet_ntoa(client.sin_addr));

    /*
     * Deallocate the socket.
     */
}

```

```
    close(s);  
}
```

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

This chapter describes the Pascal language application program interface (API) provided with TCP/IP for z/VM. 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 to use the Pascal language API.

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

The general sequence of operations is:

1. Start 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 virtual machine (TcpSend, TcpFSend, TcpWaitSend, TcpReceive, TcpFReceive, TcpWaitReceive, UdpSend, UdpNReceive, RawIpSend, RawIpReceive).

Note: TcpWaitReceive and TcpWaitSend are synchronous calls.

5. Check the status returned from the TCPIP virtual machine in the form of notifications (GetNextNote).
6. Repeat the data transfer operations (steps “4” on page 41 and “5” on page 41) 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 file.

A sample program is supplied with the TCP/IP program, see “[Sample Pascal Program](#)” on page 108.

Software Requirements

To develop programs in Pascal that interface directly to the TCP, UDP, and IP protocol boundaries, you require the 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 COPY and CMCLIEN COPY. The CMCOMM and CMCLIEN are included in the ALLMACRO MACLIB shipped with TCP/IP. To include these files in your program source, enter:

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

Additional include statements are required in programs that use certain calls. The following list shows the members of the ALLMACRO MACLIB that need to be included for the various calls.

- CMRESGLB for GetHostResol
- CMINTER for GetHostNumber, GetHostString, IsLocalAddress, and IsLocalHost.

The load modules are in the TCPIP COMMTXT file. Include this file in your GLOBAL TXTLIB command when you are creating a load module to link an application program.

Connection State

ConnectionState is the current state of the connection. For the Pascal declaration of the ConnectionStateType data type, see Figure 15 on page 42. 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. For the mapping between TCP states and ConnectionStateType, see Table 5 on page 42.

```

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

```

Figure 15. 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 5. TCP Connection States

TCP State	ConnectionStateType
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

Table 5. TCP Connection States (continued)

TCP State	ConnectionStateType
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 program to exchange information about the connection. There are two types of records, one used for IPv4 calls and one used for IPv6. The IPv4 Pascal declaration is shown in Figure 16 on page 43. For more information about the use of each field, see “[TcpOpen and TcpWaitOpen](#)” on page 88 and “[TcpStatus](#)” on page 100. The IPv6 declaration is shown in Figure Figure 17 on page 44. For more information about the use of each field, see “[Tcp6Open and Tcp6WaitOpen](#)” on page 79 and “[Tcp6Status](#)” on page 81.

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

Figure 16. Pascal Declaration of Connection Information Record

Connection

Specifies a number identifying the connection that is described. This connection number is different from the connection number displayed by the NETSTAT command. For more information about the NETSTAT command, see [z/VM: TCP/IP User's Guide](#).

OpenAttemptTimeout

Specifies 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

Specifies entries used only when working within a multilevel secure environment.

BytesToRead

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

UnackedBytes

Specifies 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

Specifies the local internet address and local port. Together, these form one end of a connection. The foreign socket forms the other end. For the Pascal declaration of the SocketType record, see [Figure 18](#) on page 44.

ForeignSocket

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

```

Status6InfoType =
  record
    Connection: ConnectionType;
    OpenAttemptTimeout: integer;
    Security: SecurityType;
    Compartment: CompartmentType;
    Precedence: PrecedenceType;
    BytesToRead: integer;
    UnackedBytes: integer;
    ConnectionState: ConnectionStateType;
    LocalSocket: Socket6Type;
    ForeignSocket: Socket6Type;
  end;

```

Figure 17. IPv6 Pascal Declaration of Connection Information Record

Connection

Specifies a number identifying the connection that is described. This connection number is different from the connection number displayed by the NETSTAT command. For more information about the NETSTAT command, see [z/VM: TCP/IP User's Guide](#).

OpenAttemptTimeout

Specifies 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

Specifies entries used only when working within a multilevel secure environment. These fields only have meaning when returned on a Tcp6Status call for an IPv4 connection. When specified for IPv6 connection requests, they will be ignored.

BytesToRead

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

UnackedBytes

Specifies 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

Specifies the local internet address, in IPv6 format, and local port. Together, these form one end of a connection. The foreign socket forms the other end. For the Pascal declaration of the Socket6Type record, see [Figure 19 on page 45](#).

ForeignSocket

Specifies the foreign, or remote, internet address, in IPv6 format, and its associated port. These form one end of a connection. The local socket forms the other end.

Socket Record

```

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

```

Figure 18. Pascal Declaration of Socket Type

Field	Description
-------	-------------

Address

Specifies the internet address.

Port

Specifies the port.

```
IPv6AddressType = packed array (.1..16.) of char;  
Socket6Type =  
  record  
    Address6: IPv6AddressType;  
    Port: PortType;  
    Flow: UnsignedIntegerType;  
    Scope: UnsignedIntegerType;  
  end; { Socket6Type }
```

Figure 19. IPv6 Pascal Declaration of Socket Type

Field**Description****Address6**

Specifies the internet address. This can be specified as an IPv6 address or an IPv6 mapped IPv4 address (refer to RFC 4291).

Port

Specifies the port.

Flow

This field is ignored.

Scope

The scope field identifies a set of interfaces as appropriate for the scope of the address carried in the Socket6Type record address field. For link local address, the scope can be used to specify the outgoing interface index. The z/VM stack supports scope for link local addresses only.

Notification Record

The notification record is used to provide event information. You receive this information by using the GetNextNote call. For more information, see [“GetNextNote” on page 63](#). It is a variant record; the number of fields is dependent on the type of notification. For the Pascal declaration of this record, see [Figure 20 on page 46](#).

```

NotificationInfoType =
record
  Connection: ConnectionType;
  Protocol: ProtocolType;
  case NotificationTag: NotificationEnumType of
    BUFFERspaceAVAILABLE:
      (
        AmountOfSpaceInBytes: integer
      );
    CertDataComplete
      (
        CertDataPtr: CertDataCmplPtrType;
      );
    CLEARtextRESUMED
      (
        Connection: ConnectionType;
      );
    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
      );
  end case;
end record

```

Figure 20. Notification Record (Part 1 of 2)

```

QueryTLSComplete:
(
    QTLSTurnCode: CallReturnCodeType;
);
RAWIPpacketsDELIVERED:
(
    RawIpDataLength: integer;
    RawIpFullLength: integer;
);
RAWIPspaceAVAILABLE:
(
    RawIpSpaceInBytes: integer;
);
READYforHANDSHAKE:
(
    HSTurnCode: CallReturnCodeType;
);
RESOURCESavailable: ();
SecureHandshakeComplete:
(
    SecHSCompleteDetail: SecureHSCompleteDetailType;
);
MSGreceived: ();
TIMERexpired:
(
    Datum: integer;
    AssociatedTimer: TimerPointerType;
);
UDPdatagramDELIVERED:
(
    DataLength: integer;
    ForeignSocket: SocketType;
    FullLength: integer;
);
UDPdatagramSPACEavailable: ();
UDPresourcesAVAILABLE: ();
URGENTpending:
(
    BytesToRead: integer;
    UrgentSpan: integer;
);
USERdefinedNOTIFICATION:
(
    UserData: UserNotificationDataType;
);
end;

```

Figure 21. Notification Record (Part 2 of 2)

Connection

Indicates 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 previously returned NObufferSPACE. For more information about these procedures, see ["TcpFSend, TcpSend, and TcpWaitSend"](#) on page 86.

AmountOfSpaceInBytes

Indicates 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.

CertDataComplete

The results of the certificate data request.

```

CertDataCmplPtrType = @ CertDataCompleteDetailType
CertDataCompleteDetailType =

```

```

packed record
  CDComp: CertDataCompleteHdrType;
  CDData: packed array (. 1..CDDDataLen.) of char;
end;
CertDataCompleteHdrType =
  packed record
    CDRetCode: integer;
    CDRetCnt: integer;
    CDDataLen: UnsignedHalfwordType;
    CDRes: UnsignedHalfwordType;
  end;

```

CDData

Is requested data from the certificate. The format is as follows:

```

+-----+
| Len | Code | CertData .... | Len | Code | CertData..... |
+-----+

```

where:

Len

Is a halfword field that contains the total length of the item (Len+Code+CertData). The total of all of the Len fields in the buffer is returned in CDDDataLen.

Code

Is a halfword that contains the certificate field code (600-677).

CertData

Is the certificate data that corresponds to the requested code. Note that a single field could appear multiple times in the returned buffer if more than one "answer" is valid.

CDRetCode

Indicates the return code from the certificate request. Possible values are:

0 - No errors.

4021 - The partner value is not valid.

4023 - The partner certificate is not available.

4024 - The certificate does not contain any values.

4025 - The buffer length passed is too large.

4026 - The returned data will not fit in the provided buffer. Partial data is returned.

4027 - The passed buffer pointer is null.

4028 - The number of certificate fields requested (CDReqNum) is 0.

4029 - The number of certificate fields requested (CDReqNum) is greater than 64.

4030 - The requested certificate field is not found.

4031 - The requested certificate field is not valid.

4032 - Both of these errors exist in the return data: A requested certificate field is not found *and* a requested certificate field is not valid.

CDRetCnt

Is the number of certificate fields returned in CDData.

CDDDataLen

Is the length of the returned certificate data.

CDRes

Is reserved - will be 0.

Usage Notes

- Certificate fields will be placed in the CDData buffer in the order in which they appear in the CertReqCodes input structure.
- The CDData buffer will contain as many certificate fields as will fit completely. If a requested certificate field does not fit in the buffer, it will not be returned and subsequent fields in the

CertReqCodes input structure will also fail. CDRetCode will indicate that not all of the data will fit in CDDData. CDRetCnt will reflect the number of completed requests.

- If the requested field cannot be found in the certificate, CDDData will contain a Len of 4 along with the requested Code. No data will be returned. CDRetCode will be updated to indicate that one or more fields are not present in the certificate.

CLEARtextRESUMED

Notification given when a Close_Notify command is received on the connection.

Connection

Indicates the connection number which received the Close_notify command.

CONNECTIONstateCHANGED

Indicates that a TCP connection has changed state.

NewState

Indicates the new state for this connection.

Reason

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

Note:

1. The following is the sequence of state notifications for a connection.

- 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 can 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:

- 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.

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

Indicates the number of bytes of data delivered to you.

LastUrgentByte

Indicates the 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 virtual machine. 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

Specifies 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 [“2”](#) on page 49). 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 because of connection closing.

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

Indicates 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 because of connection closing.

IOinterrupt

Notification given when a simulated I/O interrupt occurs in your virtual machine. The Connection and Protocol fields are not applicable.

DeviceAddress

This address corresponds to the DEVICE statement.

UnitStatus

Specifies the status returned by the device.

ChannelStatus

Specifies the status returned by the channel.

IUCVInterrupt

Notification given when a simulated IUCV interrupt occurs in your virtual machine. The Connection and Protocol fields are not applicable.

IUCVResponseBuf

Contains the information returned from the application.

PINGresponse

Notification given when a PINGresponse is received.

PingTurnCode

Specifies the status of the ping operation.

ElapsedTime

Indicates 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 virtual machine. This field is valid only if PingTurnCode has a value of OK.

QUERYtlsCOMPLETE

Notification given when the SSL server has completed verification of the label passed on the QueryTLS command.

ReturnCode

Indicates the status of the QUERYtlsCOMPLETE operation. READYforHANDSHAKE to read: 'Any other return code indicates a handshake failure.'

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

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

RawIpFullLength

Specifies the 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

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

READYforHANDSHAKE

Notification given when a TcpSServer command is issued with a null data buffer. It indicates that the connection is now waiting for a handshake.

ReturnCode

Indicates status of the handshake. A return code of OK indicates that the connection is waiting for a handshake. Any other return code indicates that there was a problem and the handshake cannot be done.

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.

SECUREhandshakeCOMPLETE

Notification given when SSL has completed the handshake (either inbound or outbound).

```
SecureHSCompleteDetailType =
    record
        ReturnCode:      SecureTurnCodeType;
        AlertLevel:      SecureAlertLevelType;
```

```
AlertDescription: SignedHalfwordType;  
end
```

ReturnCode

Indicates the status of the handshake.

```
SecureTurnCodeType = (NOALERT, ALERT, TIMEOUT)
```

NOALERT

The handshake completed successfully.

ALERT

Problems were encountered during the handshake.

TIMEOUT

The handshake did not complete within the time allotted.

AlertLevel

When the ReturnCode is ALERT, this classifies the level of the alert.

```
SecureAlertLevelType = ( AlertOK, Warning, Fatal )
```

AlertDescription

When ReturnCode is ALERT, this field contains the details of the failure. An AlertDescription value in the 4000 range indicates an SSL server error as follows:

- 4001** - The type is not valid.
- 4002** - The integer format of the IP address is not valid.
- 4003** - ValidationBuffer is too long.
- 4004** - Len is either too big or extends beyond the buffer.
- 4005** - The maximum number of validation fields has been exceeded.
- 4006** - The dotted decimal format of the IPv4 address is not valid.
- 4007** - The dotted decimal format of the IPv6 address is not valid.
- 4008** - Validation of a host name or fully-qualified domain name failed.
- 4009** - Validation of an IPv4 or IPv6 address failed.
- 4010** - Validation failed.

An AlertDescription value in the 40000 range indicates a System SSL error. Subtract 40000 from the AlertDescription value and refer to [Messages and codes in z/OS Cryptographic Services System Secure Sockets Layer Programming \(publibz.boulder.ibm.com/epubs/pdf/gsk2aa00.pdf\)](http://publibz.boulder.ibm.com/epubs/pdf/gsk2aa00.pdf) for details.

SMSGreceived

Notification given when one or more Special Messages (Smsgs) arrive. The GetSmsg call is used to retrieve queued Smsgs. For information on the SMSG command, see [z/VM: TCP/IP User's Guide](#).

TIMERexpired

Notification given when a timer set through SetTimer expires.

Datum

Indicates the data specified when SetTimer was called.

AssociatedTimer

Specifies 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.

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

Specifies the length of the data delivered to your buffer.

ForeignSocket

Specifies the source of the datagram.

FullLength

Specifies 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 because of 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

Indicates the number of incoming bytes not yet delivered to the client.

UrgentSpan

Indicates the 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.

File Specification Record

The file specification record is used to fully specify a file. The Pascal declaration is shown in [Figure 22 on page 53](#).

```
SpecOfFileType =
  record
    Owner: DirectoryNameType;
    Case SpecOfSystemType of
      VM:
        (
          VirtualAddress:VirtualAddressType;
          NewVirtualAddress:VirtualAddressType;
          DiskPassword: DirectoryNameType;
          Filename: DirectoryNameType;
          Filetype: DirectoryNameType;
          Filemode: FilemodeType
        );
      MVS:
        (
          (* The MVS declaration is listed here. *)
        );
    end;
```

Figure 22. Pascal Declaration of File Specification Record

Using Procedure Calls

Your program uses procedure calls to initiate communication with the TCPIP virtual machine. Most of these procedure calls return with a code, which indicates success or the type of failure incurred by the call. For an explanation of the return codes, see [Table 48 on page 337](#).

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 can send you. To terminate the TCP/UDP/IP service, use the `EndTcpIp` procedure.

Notifications

The TCPIP virtual machine sends you notifications to inform you of asynchronous events. Also, some notifications are generated in your virtual machine 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. For more information (see [Figure 20 on page 46](#)).

The following table provides a short description of the Notification procedure calls and gives the page number where each call's detailed description is located.

<i>Table 6. Pascal Language Interface Summary—Notifications</i>		
Procedure Call	Description	Location
<code>GetNextNote</code>	Retrieves the next notification.	“GetNextNote” on page 63
<code>Handle</code>	Specifies the types of notifications that your program can process.	“Handle” on page 64
<code>NotifyIo</code>	Requests that an <code>IOinterrupt</code> notification be sent to you when an I/O interrupt occurs on a given virtual address.	“NotifyIo” on page 68
<code>Unhandle</code>	Specifies notification types that your program can no longer process.	“Unhandle” on page 107
<code>UnNotifyIo</code>	Indicates that you no longer wish to be sent a notification when an I/O interrupt occurs on a given virtual address.	“UnNotifyIo” on page 107

TCP/UDP Initialization Procedures

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

The following table provides a short description of the TCP/UDP Initialization procedure calls and gives the page number where each call's detailed description is located.

<i>Table 7. Pascal Language Interface Summary—TCP/UDP Initialization</i>		
Procedure Call	Description	Location
<code>TcpNameChange</code>	Identifies the name of the virtual machine running the TCP/IP program when the virtual machine has a name other than TCPIP.	“TcpNameChange” on page 88
<code>BeginTcpIp</code>	Establishes communication with the TCP/IP services.	“BeginTcpIp” on page 60
<code>StartTcpIpNotice</code>	Establishes communication with the TCP/IP services.	“StartTcpNotice” on page 78

TCP/UDP Termination Procedure

The Pascal API has one termination procedure call. You should use the EndTcpIp call when you have finished with the TCP/IP services.

The following table provides a short description of the TCP/UDP Termination procedure call and gives the page number where the call's detailed description is located.

Table 8. Pascal Language Interface Summary—TCP/UDP Termination		
Procedure Call	Description	Location
EndTcpIp	Terminates communication with the TCP/IP services.	“EndTcpIp” on page 61

Handling External Interrupts

The handling external interrupts 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.

The following table provides a short description of the Handling External Interrupts and gives the page number where each call's detailed description is located.

Table 9. Pascal Language Interface Summary—Handling External Interrupts		
Procedure Call	Description	Location
TcpExtRupt	Notifies the TCP interface of the arrival of a simulated external interrupt.	“TcpExtRupt” on page 83
RTcpExtRupt	A version of TcpExtRupt.	“RTcpExtRupt” on page 74
TcpVmcfRupt	Notifies the TCP interface of the arrival of a simulated external VMCF interrupt.	“TcpVmcfRupt” on page 101
RTcpVmcfRupt	A version of TcpVmcfRupt.	“RTcpVmcfRupt” on page 74

TCP Communication Procedures

The TCP communication 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.

The following table provides a short description of the TCP communication procedures and gives the page number where each call's detailed description is located.

Table 10. Pascal Language Interface Summary—TCP Communication Procedures		
Procedure Call	Description	Location
Tcp6Open	Initiates a TCP IPv6 connection.	“Tcp6Open and Tcp6WaitOpen” on page 79
Tcp6Status	Obtains the current status of a TCP IPv6 connection.	“Tcp6Status” on page 81
Tcp6WaitOpen	Initiates a TCP IPv6 connection and waits for the establishment of the connection.	“Tcp6Open and Tcp6WaitOpen” on page 79

<i>Table 10. Pascal Language Interface Summary—TCP Communication Procedures (continued)</i>		
Procedure Call	Description	Location
TcpNameChange	Is used if the virtual machine running the TCP/IP program is not using the default name, TCPIP, and is not the same as specified in the TCPIPUSERID statement of the TCPIP DATA file.	“TcpNameChange” on page 88
TcpOpen	Initiates a TCP IPv4 connection.	“TcpOpen and TcpWaitOpen” on page 88
TcpOption	Sets an option for a TCP connection.	“TcpOption” on page 90
TcpSClient	Indicates to the SSL server that the connection is to be secure and that the SSL server needs to initiate an outbound handshake.	“TcpSClient” on page 94
TcpSClose	Performs Close_Notify on a TLS connection but leave the TCP session up.	“TcpSClose” on page 98
TcpSServer	Indicates to the SSL server that the connection is to be secure and that the SSL server needs to wait for an incoming handshake.	“TcpSServer” on page 98
TcpSStatus	Returns details about a session, such as whether or not it is secure and the encryption suite.	“TcpSStatus” on page 99.
TcpWaitOpen	Initiates a TCP IPv4 connection and waits for the establishment of the connection.	“TcpOpen and TcpWaitOpen” on page 88
TcpFSend	Sends TCP data.	“TcpFSend, TcpSend, and TcpWaitSend” on page 86
TcpSend	Sends TCP data.	“TcpFSend, TcpSend, and TcpWaitSend” on page 86
TcpWaitSend	Sends TCP data and waits until TCPIP accepts it.	“TcpFSend, TcpSend, and TcpWaitSend” on page 86
TcpFReceive	Establishes a buffer to receive TCP data.	“TcpFReceive, TcpReceive, and TcpWaitReceive” on page 83
TcpReceive	Establishes a buffer to receive TCP data.	“TcpFReceive, TcpReceive, and TcpWaitReceive” on page 83
TcpWaitReceive	Establishes a buffer to receive TCP data and waits for the reception of the data.	“TcpFReceive, TcpReceive, and TcpWaitReceive” on page 83
TcpClose	Begins the TCP one-way closing sequence.	“TcpClose” on page 82
TcpAbort	Shuts down a TCP connection immediately.	“TcpAbort” on page 81

<i>Table 10. Pascal Language Interface Summary—TCP Communication Procedures (continued)</i>		
Procedure Call	Description	Location
TcpStatus	Obtains the current status of a TCP IPv4 connection.	“TcpStatus” on page 100

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.

The following table provides a short description of the Ping interface procedures and gives the page number where each call’s detailed description is located.

<i>Table 11. Pascal Language Interface Summary—Ping Interface</i>		
Procedure Call	Description	Location
PingRequest	Sends an Internet Control Message Protocol (ICMP) echo request.	“PingRequest” on page 69

Monitor Procedures

Two monitor procedures, MonCommand and MonQuery, provide a mechanism for querying and controlling the TCPIP virtual machine.

The following table provides a short description of the Monitor procedures and gives the page number where each call’s detailed description is located.

<i>Table 12. Pascal Language Interface Summary—Monitor Procedures</i>		
Procedure Call	Description	Location
MonCommand	Instructs TCP to read a specific file and execute the commands that it contains.	“MonCommand” on page 66
MonQuery	Performs control functions and retrieves internal TCPIP control blocks.	“MonQuery” on page 67

UDP Communication Procedures

The UDP communication procedures describe the programming interface for the User Datagram Protocol (UDP) provided in the TCP/IP product.

The following table provides a short description of the UDP communication procedures and gives the page number where each call’s detailed description is located.

<i>Table 13. Pascal Language Interface Summary—UDP Communication Procedures</i>		
Procedure Call	Description	Location
Udp6Open	Requests communication with UDP on a specified socket using IPv4 or IPv6 protocols.	“Udp6Open” on page 102
Udp6Send	Sends a UDP datagram to a specified foreign socket.	“Udp6Send” on page 102
UdpOpen	Requests communication with UDP on a specified socket using IPv4 protocols.	“UdpOpen” on page 104
UdpSend	Sends a UDP datagram to a specified foreign socket.	“UdpSend” on page 106
UdpNReceive	Notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.	“UdpNReceive” on page 104

<i>Table 13. Pascal Language Interface Summary—UDP Communication Procedures (continued)</i>		
Procedure Call	Description	Location
UdpReceive	Notifies the TCPIP virtual machine that you are willing to receive UDP datagram data.	“UdpReceive” on page 105
UdpClose	Terminates use of a UDP socket.	“UdpClose” on page 103

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 about the obey list, see [z/VM: TCP/IP Planning and Customization](#).

The following table provides a short description of the Raw IP interface procedures and gives the page number where each call's detailed description is located.

<i>Table 14. Pascal Language Interface Summary—Raw IP Interface</i>		
Procedure Call	Description	Location
RawIpOpen	Informs the TCPIP virtual machine that the client wants to send and receive IP packets of a specified protocol.	“RawIpOpen” on page 71
RawIpReceive	Specifies a buffer to receive raw IP packets of a specified protocol.	“RawIpReceive” on page 72
RawIpSend	Sends raw IP packets of a specified protocol.	“RawIpSend” on page 72
RawIpClose	Informs the TCPIP virtual machine that the client no longer handles the protocol.	“RawIpClose” on page 70

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.

The following table provides a short description of the Timer routines and gives the page number where each call's detailed description is located.

<i>Table 15. Pascal Language Interface Summary—Timer Routines</i>		
Procedure Call	Description	Location
CreateTimer	Allocates a timer.	“CreateTimer” on page 61
ClearTimer	Resets a timer.	“ClearTimer” on page 60
SetTimer	Sets a timer to expire after a specified interval.	“SetTimer” on page 77
DestroyTimer	Deallocates a timer.	“DestroyTimer” on page 61

Host Lookup Routines

The host lookup routines (with the exception of GetHostResol) are declared in the CMINTER member of the ALLMACRO MACLIB. The host lookup routine GetHostResol is declared in the CMRESGLB member of

the ALLMACRO MACLIB. Any program using these procedures must include CMINTER or CMRESGLB after the include statements for CMCOMM and CMCLIEN.

The following table provides a short description of the host lookup routines and gives the page number where each call's detailed description is located.

<i>Table 16. Pascal Language Interface Summary—Host Lookup Routines</i>		
Procedure Call	Description	Location
GetHostNumber	Converts a host name to an internet address using static tables.	“GetHostNumber” on page 62
GetHostResol	Converts a host name to an internet address using a domain name resolver.	“GetHostNumber” on page 62
GetHostString	Converts an internet address to a host name using static tables.	“GetHostString” on page 62
GetIdentity	Returns environment information.	“GetIdentity” on page 63
IsLocalAddress	Determines if an internet address is local.	“IsLocalAddress” on page 65
IsLocalHost	Determines if a host name is local, remote, loopback, or unknown.	“IsLocalHost” on page 65

Other Routines

The following table provides a short description of these procedure calls and gives the page number where the detailed description is located.

<i>Table 17. Pascal Language Interface Summary—Other Routines</i>		
Procedure Call	Description	Location
AddUserNote	Adds a USERdefinedNOTIFICATION notification to the note queue.	“AddUserNote” on page 60
GetSmsg	Retrieves one queued special message (Smsg).	“GetSmsg” on page 64
QueryTLS	Determines if the security server is available, and if the label is specified, is it known to the security server.	“QueryTLS” on page 70.
ReadXlateTable	Reads a binary translation table file.	“ReadXlateTable” on page 73
SayCalRe	Converts a return code into a descriptive message.	“SayCalRe” on page 75
SayConSt	Converts a connection state into a descriptive message.	“SayConSt” on page 75
SayIntAd	Converts an internet address into a name or dotted-decimal form.	“SayIntAd” on page 76
SayIntNum	Converts an internet address into its dotted-decimal form.	“SayIntNum” on page 76
SayNotEn	Converts a notification enumeration type into a descriptive message.	“SayNotEn” on page 76
SayPorTy	Converts a port number into a descriptive message or into EBCDIC.	“SayPorTy” on page 77
SayProTy	Converts the protocol type into a descriptive message or into EBCDIC.	“SayProTy” on page 77

Table 17. Pascal Language Interface Summary—Other Routines (continued)

Procedure Call	Description	Location
SaySslRe	Returns a printable string describing the AlertDescription returned when a handshake completes. The AlertDescription is passed in CallReturn.	“SaySslRe” on page 75
TcpSCertData	For a secure connection, use this function to request specific fields from the local or partner certificate.	“TcpSCertData” on page 91

Procedure Calls

This section provides the syntax, operands, and other appropriate information for each Pascal procedure call supported by TCP/IP for VM.

AddUserNote

The AddUserNote procedure can be called to add a USERdefinedNOTIFICATION notification to the note queue and wake up GetNextNote if it is waiting for a notification. For more information, see [“RTcpExtRupt” on page 74](#) and [“RTcpVmcfRupt” on page 74](#).

BeginTcpIp

The BeginTcpIp procedures inform the TCPIP virtual machine that you want to start using its services. If your program handles simulated external interrupts itself, use StartTcpNotice rather than BeginTcpIp. For information about simulated external interrupt support, see [Chapter 3, “Virtual Machine Communication Facility Interface,” on page 113](#).

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

Operand Description

ReturnCode

Indicates success or failure of call. Possible return values are:

- OK
- ABNORMALcondition
- fatalerror
- NOTtcpIPservice
- TCPIPALREADYstarted
- TCPIPshutdown
- VIRTUALmemoryTOOsmall

If ReturnCode is OK, you must call EndTcpIp when you have finished with the TCP/IP services.

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,” on page 337](#).

ClearTimer

The ClearTimer procedure resets the timer to prevent it from timing out.

```

procedure ClearTimer
(
    T: TimerPointerType
);
external;

```

Operand Description

T

Specifies a timer pointer, as returned by a previous CreateTimer call.

CreateTimer

The CreateTimer procedure allocates a timer. The timer is not set in any way. For the procedure to activate the timer, see [“SetTimer” on page 77](#).

```

procedure CreateTimer
(
    var T: TimerPointerType
);
external;

```

Operand Description

T

Sets to a timer pointer that can be used in subsequent SetTimer, ClearTimer, and DestroyTimer calls.

DestroyTimer

The DestroyTimer procedure deallocates or *fre*es a timer that you created.

```

procedure DestroyTimer
(
    var T: TimerPointerType
);
external;

```

Operand Description

T

Specifies a timer pointer, as returned by a previous CreateTimer call.

EndTcpIp

The EndTcpIp 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. Use EndTcpIp when you have finished with the TCP/IP services.

It is safe to call EndTcpIp even if BeginTcpIp or StartTcpNotice did not previously succeed.

```

procedure EndTcpIp;
external;

```

The EndTcpIp procedure has no operands.

GetHostNumber

The GetHostNumber procedure resolves 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'. For information about how to create host lookup tables, see [z/VM: TCP/IP Planning and Customization](#).

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

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

Operand Description

Name

Specifies the name or dotted-decimal number to be converted.

HostNumber

Set to the converted address, or NOhost if conversion fails.

GetHostResol

The GetHostResol procedure resolves 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;
```

Operand Description

Name

Specifies the name or dotted-decimal number to be converted.

HostNumber

Set to the converted address, or NOhost if conversion fails.

GetHostString

The GetHostString procedure 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;
```

Operand Description

Address

Specifies the address to be converted.

Name

Set to the corresponding host, gateway, or network name, or to null string if no match found.

GetIdentity

The GetIdentity procedure returns the following information:

- The user ID of the VM user
- The host machine name
- The network domain name
- The user ID of the TCPIP virtual machine.

The host machine name and domain name are extracted from the HOSTNAME and DOMAINORIGIN statements, respectively, in the *user_id* DATA file. If the *user_id* DATA file does not exist, the TCPIP DATA file is used. If a HOSTNAME statement is not specified, then the default host machine name is the name specified by the TCP/IP installer during installation. See *z/VM: TCP/IP Planning and Customization*. The TCPIP virtual machine user ID is extracted from the TCPIPUSERID statement in the *user_id* DATA file; 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;
```

Operand Description

UserId

Specifies the user ID of the VM user or the job name of a batch job that has invoked GetIdentity.

HostName

Specifies the host machine name.

DomainName

Specifies the network domain name.

TcpIpServiceName

Specifies the user ID of the TCPIP virtual machine.

GetNextNote

The GetNextNote procedure retrieves notifications from the queue. This procedure returns the next notification queued for you.

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

Operand	Description
---------	-------------

Note

Indicates that the next notification is stored here when ReturnCode is OK.

ShouldWait

Sets 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 the success or failure of the call. Possible return values are:

- OK
- NOoutstandingNOTIFICATIONS
- NOTyetBEGUN

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

GetSmsg

The GetSmsg procedure is called by your program 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.

For information about the SMSG command, see *z/VM: TCP/IP User's Guide*

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

Operand	Description
---------	-------------

Smsg

Set to the returned Smsg if Success is set to TRUE.

Success

TRUE if Smsg returned, otherwise FALSE.

Handle

The Handle procedure specifies 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 can contain any of the NotificationEnumType values shown in [Figure 20 on page 46](#).

```

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

```

Operand
Description

Notifications

Specifies the set of notification types to be handled.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- NOTyetBEGUN
- TCPIPshutdown
- ABNORMALcondition
- FATALerror

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

IsLocalAddress

The IsLocalAddress procedure queries the TCPIP virtual machine 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;

```

Operand
Description

HostAddress

Specifies the host address to be tested.

ReturnCode

Indicates whether the host address is local, or may indicate an error. Possible return values are:

- OK
- NONlocalADDRESS
- TCPIPshutdown
- ABNORMALcondition
- FATALerror

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

IsLocalHost

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

The host classes are:

MonCommand

Host Class

Description

HOSTlocal

Specifies an internet address for the local host.

HOSTloopback

Specifies one of the dummy internet addresses used to designate various levels of loopback testing.

HOSTremote

Specifies a known host name for some remote host.

HOSTunknown

Specifies an unknown host name (or other error).

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

Operand

Description

Name

Specifies the host name.

Class

Specifies the host class.

MonCommand

The MonCommand procedure instructs the TCPIP virtual machine to read a specific file and execute the commands found there. This procedure updates TCPIP internal tables and parameters while the TCPIP virtual machine 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. For more information about the OBEYFILE command, see [z/VM: TCP/IP Planning and Customization](#). You must be in the TCPIP obey list to use the MonCommand procedure.

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

Operand

Description

FileSpec

Specifies a file in a manner that allows access to that file. The TCPIP virtual machine must be authorized to access the file.

The SpecOfFileType record is listed in [Figure 22 on page 53](#).

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- ERRORinPROFILE
- HASnoPASSWORD

- INCORRECTpassword
- INVALIDuserID
- INVALIDvirtualADDRESS
- MINIDISKinUSE
- MINIDISKnotAVAILABLE
- NOTyetBEGUN
- PROFILEnotFOUND
- SOFTWAREerror
- TCPipSHUTDOWN
- UNAUTHORIZEDuser
- UNIMPLEMENTEDrequest

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

MonQuery

The MonQuery procedure obtains status information, or requests TCPIP to perform certain actions. This procedure is used by the NETSTAT command. For more information about the NETSTAT command, see *z/VM: TCP/IP User's Guide*.

```

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

```

Operand Description

Buffer

Specifies the address of the buffer to receive data.

BufSize

Specifies the size of the buffer.

ReturnCode

Indicates the success or failure of the call.

Length

Specifies the length of the data returned in the buffer.

QueryRecord

Sets up a QueryRecord to specify the type of status information to be retrieved. The MonQueryRecordType is shown in [Figure 23 on page 68](#).

```

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 23. Monitor Query Record

The only QueryType values available for your use are:

QUERYhomeONLY

Used to obtain a list of the home internet addresses recognized by your TCPIP virtual machine. 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 size of (InternetAddressType) to get the number of the home addresses that are returned.

COMMANDdropCONNECTION

Used to instruct the TCPIP virtual machine 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. For information about the NETSTAT command, see *z/VM: TCP/IP User's Guide*. The virtual machine running your program that uses COMMANDdropCONNECTION must be in the TCPIP virtual machine.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- NOTyetBEGUN
- TCPipSHUTDOWN
- UNAUTHORIZEDuser
- UNIMPLEMENTedrequest

For a description of Pascal ReturnCodes, see [Appendix B, "Pascal Return Codes,"](#) on page 337.

NotifyIo

The NotifyIo procedure is used to request that an IOinterrupt notification be sent to you when an I/O interrupt occurs on a given virtual address. You can specify that you wish notifications on up to 10 different virtual device addresses (by means of individual NotifyIo calls). This notification is intended for unsolicited interrupts, not for interrupts showing the completion of a channel program.

```

procedure NotifyIo
(
    DeviceAddress: integer;
var    ReturnCode: integer;
);
external;

```

Operand Description

DeviceAddress

Specifies the address of the device for which IOinterrupt notifications are to be generated.

ReturnCode

Indicates success or failure of the call. Possible return values are:

- OK
- TOOManyOPENS
- SOFTWAREerror

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

PingRequest

The PingRequest procedure sends an ICMP echo request to a foreign host. When a response is received or the time-out limit is reached, you receive a PingResponse notification.

The PingRequest procedure is used by the PING user command. For more information about the PING command, see [z/VM: TCP/IP Planning and Customization](#).

```

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

```

Operand Description

ForeignAddress

Specifies the address of the foreign host to be pinged.

Length

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

Timeout

Specifies how long to wait for a response, in seconds.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- BADlengthARGUMENT
- CONNECTIONalreadyEXISTS
- NObufferSPACE
- NOTyetBEGUN

QueryTLS

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

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

QueryTLS

The QueryTLS with an optional label determines if the security server is available, and if the label is specified, is it known to the security server.

```
Procedure QueryTLS
(
    TLSLabel:    DirectoryNameType;
    TLSKeyring:  KeyringType;
    var ReturnCode: CallReturnCodeType
);
KeyringType = packed array(. 1..KEYRINGlength .) of char;
KEYRINGlength = 50;
```

Operand

Description

TLSLabel

The certificate label passed along to the security server for verification.

TLSKeyring

Specifies the group that the label resides in. This capability is not yet available. The value must be blank.

ReturnCode

Indicates success or failure of the call. Possible return code values are:

- OK
- BACKlevelSSL
- KEYRINGnotPERMITTED
- KEYRINGnotRECOGNIZED
- LABELnotPERMITTED
- LABELnotRECOGNIZED
- NOTyetBEGUN
- SOFTWAREError
- SSLnotAVAILABLE
- SSLnotRESPONDING
- TCPipSHUTDOWN
- TLSnotAVAILABLE
- UNAUTHORIZEDuser

RawIpClose

The RawIpClose procedure tells the TCPIP virtual machine 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;

```

Operand Description

ProtocolNo

Specifies the number of the IP protocol.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREError
- TCPIPshutdown
- UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

RawIpOpen

The RawIpOpen procedure tells the TCPIP virtual machine that the client wants to send and receive packets of the specified protocol.

You cannot use protocols 6 and 17. They specify the TCP (6) and UDP (17) protocols. When you specify 6, 17, or a protocol that has been opened by another virtual machine, you receive the LOCALportNOTavailable return code.

```

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

```

Operand Description

ProtocolNo

Specifies the number of the IP protocol.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- LOCALportNOTavailable
- NOTyetBEGUN
- SOFTWAREError
- TCPIPshutdown
- UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

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

RawIpReceive

The RawIpReceive procedure specifies 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;
```

Operand

Description

ProtocolNo

Specifies the number of the IP protocol.

Buffer

Specifies the address of your buffer.

BufferLength

Specifies the length of your buffer. If you specify a length greater than 8492 bytes, only the first 8492 bytes are used.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREError
- TCPIPshutdown
- UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see [Appendix B, "Pascal Return Codes,"](#) on page 337.

RawIpSend

The RawIpSend procedure sends IP packets of the given protocol number. The entire packet, including the IP header, must be in the buffer. The TCPIP virtual machine uses the total length field of the IP header to determine where each packet ends. Subsequent packets begin at the next doubleword (8-byte) boundary within the buffer.

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

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

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 8K bytes.
- NumPackets is 0.

- A packet is greater than 2048 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;

```

Operand Description

ProtocolNo

Specifies the number of the IP protocol.

Buffer

Specifies the address of your buffer containing packets to send.

DataLength

Specifies the total length of data in your buffer.

NumPackets

Specifies the number of packets in your buffer.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- BADlengthARGUMENT
- NObufferSPACE
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREError
- TCPipSHUTDOWN
- UNAUTHORIZEDuser

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

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

ReadXlateTable

The ReadXlateTable procedure reads the binary translation table file 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;

```

Operand
Description

TableName
Specifies the name of the translate table. ReadXlateTable tries to read TableName TCPXLBIN. If that file exists but it has a bad format, ReadXlateTable returns with a ReturnCode FILEformatINVALID. If *user_id* TCPXLBIN does not exist, ReadXlateTable tries to read TCPIP TCPXLBIN. ReturnCode reflects the status of reading that file.

AtoETable
Contains an ASCII-to-EBCDIC table if the return code is OK.

EtoATable
Contains an EBCDIC-to-ASCII table if the return code is OK.

TranslateTableSpec
If ReturnCode is OK, TranslateTableSpec contains the complete specification of the file that ReadXlateTable used. If the ReturnCode is not OK, TranslateTableSpec contains the complete specification of the last file that ReadXlateTable tried to use.

ReturnCode
Indicates the success or failure of the call. Possible return values are:

- OK
- ERRORopeningORreadingFILE
- FILEformatINVALID

RTcpExtRupt

The RTcpExtRupt procedure is a version of the TcpExtRupt Pascal procedure and can be called directly from an assembler interrupt handler.

Note: The content of this section is Internal Product Information and must not be used as programming interface information.

The following is a sample of the assembler 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 with an environment address. Pass it unchanged in subsequent calls to RTCPEXTR.
EXTPARM	DC	A(ENV)	
	DC	A(RUPTCODE)	

The RTcpExtRupt procedure has no operands.

RTcpVmcfRupt

The RTcpVmcfRupt procedure is a version of the TcpVmcfRupt Pascal procedure and can be called directly from an assembler interrupt handler.

Note: The content of this section is Internal Product Information and must not be used as programming interface information.

The following is a sample assembler 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 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.

The RTcpVmcfRupt procedure has no operands.

SayCalRe

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

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

Operand
Description

CallReturn
Specifies the return code to be described.

SaySslRe

The SaySslRe function returns a printable string describing the AlertDescription returned when a handshake completes. The AlertDescription is passed in CallReturn.

```
function SaySslRe
)
    CallReturn: SignedHalfwordType
):
WordType;
external;
```

Operand
Description

CallReturn
Specifies the return code to be described.

SayConSt

The SayConSt 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;
```

Operand
Description

State
Specifies the connection state to be described.

SayIntAd

The SayIntAd function converts the internet address specified by InternetAddress to a printable string. The address is looked up in HOSTS ADDRINFO file, 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;
```

Operand
Description

InternetAddress
Specifies the internet address to be converted.

SayIntNum

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

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

Operand
Description

InternetAddress
Specifies the internet address to be converted.

SayNotEn

The SayNotEn 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;
```

Operand
Description

Notification

Specifies the notification enumeration type to be described.

SayPorTy

The SayPorTy 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;
```

Operand
Description

Port

Specifies the port number to be described.

SayProTy

The SayProTy 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;
```

Operand
Description

Protocol

Specifies the number of the protocol to be described.

SetTimer

The SetTimer 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.

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

```

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

```

Operand Description

T

Specifies a timer pointer, as returned by a previous CreateTimer call.

AmountOfTime

Specifies the time interval in seconds.

Data

Specifies an integer value to be returned with the TIMERexpired notification.

StartTcpNotice

The StartTcpNotice procedure establishes your own external interrupt handler. Use this procedure rather than BeginTcpIp when you want to handle simulated external interrupts yourself.

If your program does not use simulated VMCF, set the ClientDoesVmcF parameter to FALSE. For more information about the simulated Virtual Machine Communication Facility interface, see [Chapter 3, “Virtual Machine Communication Facility Interface,”](#) on page 113. 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 previously.

```

procedure StartTcpNotice
(
    ClientDoesVmcF: Boolean;
    var ReturnCode: integer
);
external;

```

Operand Description

ClientDoesVmcF

Set to FALSE if your program does not use simulated VMCF. Otherwise, set to TRUE.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- ALREADYclosing
- NOTcpIPservice
- TCPIPALREADYstarted
- VIRTUALmemoryTOOsmall
- FATALerror

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

If ReturnCode is OK, you must call EndTcpIp when you have finished with the TCP/IP services.

Tcp6Open and Tcp6WaitOpen

The Tcp6Open or Tcp6WaitOpen procedures initiate a TCP connection. The Tcp6Open or Tcp6WaitOpen should be used for IPv6 connections or for a mix of IPv4 and IPv6 connections. Tcp6Open 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. Tcp6WaitOpen does not return until the connection is established, or until an error occurs.

```
procedure Tcp6Open
(
  var ConnectionInfo: Status6Info;
  var ReturnCode: integer;
);
external;

procedure Tcp6WaitOpen
(
  var ConnectionInfo: Status6Info;
  var ReturnCode: integer;
);
external;
```

Operand Description

ConnectionInfo

Specifies a connection information record.

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 Tcp6Open, 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 Tcp6WaitOpen, 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: You can use an address of UNSPECIFIEDip6address (the TCPIP virtual machine uses the home address corresponding to the network interface used to route to the foreign address) and port of UNSPECIFIEDport (the TCPIP virtual machine assigns a port number, in the range of 1024 to 65 534). You can specify the address, the port, or both if particular values are required by your application. The address must be a valid home address for your node and must be specified as an IPv6 address. IPv6 mapped IPv4 addresses are acceptable, refer to RFC 4291 for details. The port must not be reserved via a PORT statement in Profile TCPIP or in use by another application.

Passive Open: A predetermined port number is specified which allows other programs to connect to your program using the port. Alternatively, you can use UNSPECIFIEDport to let the TCPIP virtual machine assign a port number, obtain the port number through Tcp6Status for Tcp6Open or using the NETSTAT CONN command for Tcp6WaitOpen, and transmit it to the other program through an existing TCP connection or manually. (For more information about the NETSTAT CONN command, see *z/VM: TCP/IP User's Guide*). Generally, an address of UNSPECIFIEDipv6address is specified so that the active open to the port succeeds regardless of the home addresses to which it was sent.

ForeignSocket

Active Open: The address and port must both be specified, because the TCPIP virtual machine cannot actively initiate a connection without knowing the destination address and port. The address must be specified as an IPv6 address. IPv6 mapped IPv4 addresses are acceptable, refer to RFC 4291 for details.

Note: Attempting to specify an IPv6 link local address will produce unpredictable results.

Passive Open: If your program is offering a service to anyone who wants it, specify an address of UNSPECIFIEDipv6address 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 the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- CONNECTIONalreadyEXISTS
- DROPPEDbyOPERATOR (TcpWaitOpen only)
- IPv6connection
- LOCALportNOTavailable
- MIXEDaddresses
- 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

Tcp6Status

The Tcp6Status procedure obtains 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 Tcp6Status
(
  var ConnectionInfo: Status6Info;
  var ReturnCode: integer;
);
external;
```

Operand

Description

ConnectionInfo

If ReturnCode is OK, the following fields are returned:

Field

Description

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

Specifies the number of bytes of incoming data queued for your program (waiting for TcpReceive, TcpFReceive, or TcpWaitReceive).

UnackedBytes

Specifies 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

Specifies the current connection state.

LocalSocket

Specifies the local socket, consisting of a local address and a local port. The local address will be returned as an IPv6 address. If the address is an IPv4 address, it will be returned as a mapped IPv6 address.

ForeignSocket

Specifies the foreign socket, consisting of a foreign address and a foreign port. The foreign address will be returned as an IPv6 address. If the address is an IPv4 address, it will be returned as a mapped IPv6 address.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN

TcpAbort

The TcpAbort procedure shuts down a specific connection immediately. Data sent by your application on the aborted connection can 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;
```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record for IPv4 connections or by Tcp6Open or Tcp6WaitOpen in the connection field of the Status6InfoType record for IPv6 connections.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- NOSuchCONNECTION
- NOTyetBEGUN
- SSLcloseINprogress
- 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 [Appendix B, “Pascal Return Codes,”](#) on page 337.

TcpClose

The TcpClose procedure begins the TCP one-way closing sequence. During this closing sequence, you, the local client, cannot send any more data. Data can 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;
```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record for IPv4 connections or by Tcp6Open or Tcp6WaitOpen in the connection field of the Status6InfoType record for IPv6 connections.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- ALREADYclosing

- NOsuchCONNECTION
- NOTyetBEGUN
- SSLcloseINprogress
- SSLhandshakeINprogress
- TCPIPshutdown

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

Note:

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.

TcpExtRupt

Use the TcpExtRupt 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. For the handling of VMCF interrupts, see [“TcpVmcfRupt”](#) on page 101.
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. For more information, see [“RTcpExtRupt”](#) on page 74 and [“RTcpVmcfRupt”](#) on page 74.

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

**Operand
Description**

RuptCode

Specifies the external interrupt code you received.

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 can receive an FRECEIVEerror notification rather than DATAdelivered, indicating that the receive request was rejected, or that it was initially accepted but was later canceled because of 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. Therefore, it is 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 virtual machine followed by REJECT by the TCPIP virtual machine) to tell the TCPIP virtual machine that your program is ready to receive, and another complete VMCF transaction (SEND by TCPIP virtual machine followed by RECEIVE by your virtual machine) 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 virtual machine does a VMCF REPLY, completing the transaction.

TcpFReceive requires fewer 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 (for each virtual machine), thus 50 outstanding TcpFReceives.

With TcpReceive, you are not subject to the limit of 50 outstanding receives (for each virtual machine). 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;

```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record for IPv4 connections or by Tcp6Open or Tcp6WaitOpen in the connection field of the Status6InfoType record for IPv6 connections.

Buffer

Specifies the address of the buffer to contain the received data.

BytesToRead

Specifies 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 8192 bytes. Specifying BytesToRead of more than 8192 bytes may not cause an error return, but only 8192 bytes of the buffer are used.

Note: The order of TcpFReceive or TcpReceive calls on multiple connections, and the order of DATA delivered notifications among the connections, are not necessarily related.

BytesRead

Indicates a value 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 values are:

- OK⁺
- ABNORMALcondition
- FATALerror
- TIMEOUTopen⁺
- UNREACHABLEnetwork⁺
- BADlengthARGUMENT
- NOSuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- OPENrejected⁺
- RECEIVEstillPENDING
- REMOTEreset⁺
- UNEXPECTEDsyn⁺
- WRONGsecORprc⁺
- DROPPEDbyOPERATOR⁺
- FATALerror⁺
- KILLEDbyCLIENT⁺
- TCPipSHUTDOWN⁺
- TIMEOUTconnection⁺
- REMOTEclose

ReturnCode:

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- BEGUNlengthARGUMENT
- fatalerror
- NOSuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- RECEIVEstillPENDING
- REMOTEclose
- TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

Note:

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 “2” on page 49). 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 can be returned either because the connection initiation has failed, or because the connection has been terminated, because of shutdown. In either case, your program should not issue any more TCP/IP calls.

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 virtual machine has insufficient space to accept the data being sent.

TcpFSend and TcpSend differ in the way that they handle VMCF when the TCPIP virtual machine 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 virtual machine 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 TcpSends 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;

```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record for IPv4 connections or by Tcp6Open or Tcp6WaitOpen in the connection field of the Status6InfoType record for IPv6 connections.

Buffer

Specifies the address of the buffer containing the data to send.

BufferLength

Specifies the length of data in the buffer. Maximum is 8192.

PushFlag

Forces the data, and previously queued data, to be sent immediately to the foreign application.

UrgentFlag

Marks 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 the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- BADlengthARGUMENT
- CANNOTsendDATA

TcpNameChange

- FATALerror
- FSENDstillpending
- NObufferSPACE (TcpSend only)
- NOSuchCONNECTION
- NOTyetBEGUN
- NOTyetOPEN
- SSLcloseINprogress
- SSLhandshakeINprogress
- TCPIPshutdown

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

Note:

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 gives 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.

TcpNameChange

The TcpNameChange procedure is used if the virtual machine running the TCP/IP program is not using the default name, TCPIP, and is not the same as specified in the TCPIPUSERID statement of the TCPIP DATA file. For more information, see [z/VM: TCP/IP Planning and Customization](#).

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

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

Operand Description

NewNameOfTcp

Specifies the name of the virtual machine running TCP/IP.

TcpOpen and TcpWaitOpen

The TcpOpen or TcpWaitOpen procedures initiate a TCP IPv4 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;
```

Operand

Description

ConnectionInfo

Specifies a connection information record.

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: You can use an address of UNSPECIFIEDaddress (the TCPIP virtual machine uses the home address corresponding to the network interface used to route to the foreign address) and a port of UNSPECIFIEDport (the TCPIP virtual machine assigns a port number, in the range of 1024 to 65 534). You can specify the address, the port, or both 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: A predetermined port number is specified which allows other programs to connect to your program using the port. Alternatively, you can use UNSPECIFIEDport to let the TCPIP

TcpOption

virtual machine assign a port number, obtain the port number through TcpStatus for TcpOpen or using the NETSTAT CONN command for TcpWaitOpen, and transmit it to the other program through an existing TCP connection or manually. (For more information about the NETSTAT CONN command, see [z/VM: TCP/IP User's Guide](#)). Generally, an address of UNSPECIFIEDaddress is specified so that the active open to the port succeeds regardless of the home addresses to which it was sent.

ForeignSocket

Active Open: The address and port must both be specified, because the TCPIP virtual machine 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 the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- CONNECTIONalreadyEXISTS
- DROPPEDbyOPERATOR (TcpWaitOpen only)
- IPv6connection
- 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 [Appendix B, "Pascal Return Codes,"](#) on page 337.

TcpOption

The TcpOption procedure sets an option for a TCP connection.

```

procedure TcpOption
(
    Connection: ConnectionType;
    OptionName: integer;
    OptionValue: integer;
    var ReturnCode: integer
);
external;

```

Operand
Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record for IPv4 connections or by Tcp6Open or Tcp6WaitOpen in the connection field of the Status6InfoType record for IPv6 connections.

OptionName

Specifies the code for the option.

OPTIONtcpKEEPAIVE

If OptionValue is zero, the keep-alive mechanism is deactivated for the connection. If OptionValue is nonzero, the keep-alive mechanism is activated for the connection. This mechanism sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet, the connection state will be changed to NONEXISTENT with TIMEOUTconnection as the reason.

OptionValue

Specifies the value for the option.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPipSHUTDOWN
- INVALIDrequest

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

TcpSCertData

For a secure connection, use this function to request specific fields from the local or partner certificate.

```

procedure TcpSCertData
(
    Connection: ConnectionType;
    Wait: boolean;
    CertReqDetail: CertReqDetailType;
    var ReturnCode: CallReturnCodeType;
);
external;

```

Operand
Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record.

Wait

Is set to true if your program will wait for the certificate data request to complete. This is set to false if your program will process the CERTdataCOMPLETE notification.

CertReqDetail

Sets up the details of the certificate request.

```

CertReqDetailType = packed record
  CertReqNum:      UnsignedByteType;
  CertReqSide:     CertReqSideType;
  CertReqRes1:     UnsignedHalfwordType;
  CertReqRes2:     integer;
  CertReqLen:      integer;
  CertReqPtr:      integer;
  CertReqCodes:    array (. 1..64 .) of UnsignedHalfwordType;
end;

CertReqNum - Number of certificate fields requested in CertReqCodes

CertReqSideType =
(
  Local,           { Field in local certificate }
  Partner          { Field in partner certificate }
);

CertReqRes1 and CertReqRes2 are reserved fields and must be
set to 0.

CertReqLen - Length of the data buffer pointed to by CertReqPtr.

CertReqPtr - Pointer to a data buffer. This buffer must be large
enough to contain the CertDataCompleteDetailType structure that
is returned. Note that CDDDataLen is limited to 16K.

CertReqCodes - List of certificate fields to be returned. See
below for valid field codes.

```

The fields that can be requested from a certificate along with the request code to specify in the CertReqCodes field are as follows. More information about the structure and additional information about the fields in an x.509 certificate can be found in RFC 5280.

600 - CERT_BODY_DER
601 - CERT_BODY_BASE64
602 - CERT_SERIAL_NUMBER
610 - CERT_COMMON_NAME
611 - CERT_LOCALITY
612 - CERT_STATE_OR_PROVINCE
613 - CERT_COUNTRY
614 - CERT_ORG
615 - CERT_ORG_UNIT
616 - CERT_DN_PRINTABLE
617 - CERT_DN_DER
618 - CERT_POSTAL_CODE
619 - CERT_EMAIL
620 - CERT_DOMAIN_COMPONENT
621 - CERT_SURNAME
622 - CERT_STREET
623 - CERT_TITLE
650 - CERT_ISSUER_COMMON_NAME
651 - CERT_ISSUER_LOCALITY
652 - CERT_ISSUER_STATE_OR_PROVINCE
653 - CERT_ISSUER_COUNTRY
654 - CERT_ISSUER_ORG
655 - CERT_ISSUER_ORG_UNIT
656 - CERT_ISSUER_DN_PRINTABLE
657 - CERT_ISSUER_DN_DER

658 - CERT_ISSUER_POSTAL_CODE
659 - CERT_ISSUER_EMAIL
660 - CERT_ISSUER_DOMAIN_COMPONENT
661 - CERT_ISSUER_SURNAME
662 - CERT_ISSUER_STREET
663 - CERT_ISSUER_TITLE
664 - CERT_NAME
665 - CERT_GIVENNAME
666 - CERT_INITIALS
667 - CERT_GENERATIONQUALIFIER
668 - CERT_DNQUALIFIER
669 - CERT_MAIL
670 - CERT_SERIALNUMBER
671 - CERT_ISSUER_NAME
672 - CERT_ISSUER_GIVENNAME
673 - CERT_ISSUER_INITIALS
674 - CERT_ISSUER_GENERATIONQUALIFIER
675 - CERT_ISSUER_DNQUALIFIER
676 - CERT_ISSUER_MAIL
677 - CERT_ISSUER_SERIALNUMBER

Upon return, the ReturnCode field will be set indicating that there was an error on the call, or, if Wait=TRUE, the data will be returned in the buffer that was provided.

ReturnCode

Is one of the following:

- CERTdataNOTavail
- CONNECTIONnotSECURE
- ENOBUFS
- INVALIDrequest
- SSLcloseINprogress
- SSLnotRESPONDING
- TCPIPshutdown
- TLSnotAVAILABLE

Return Data

When the wait flag is set to true, the results of the certificate request will be provided in the buffer pointed to by CertReqPtr. The format of the buffer is below. If the wait flag is set to false, the results of the certificate request will be reflected in the CertDataComplete notification (see CertDataComplete).

```

CertDataCmplPtrType = @ CertDataCompleteDetailType
CertDataCompleteDetailType =
  packed record
    CDComp: CertDataCompleteHdrType;
    CDData: packed array (. 1..CDDataLen.) of char;
  end;
CertDataCompleteHdrType =
  packed record
    CDRetCode: integer;
    CDRetCnt: integer;
    CDDataLen: UnsignedHalfwordType;
    CDRes: UnsignedHalfwordType;
  end;
  
```

CDData

Is requested data from the certificate. The format is as follows:

```

+-----+
  
```

```
| Len | Code | CertData .... | Len | Code | CertData..... |
+-----+-----+-----+-----+-----+-----+
```

where:

Len

Is a halfword field that contains the total length of the item (Len+Code+CertData). The total of all of the Len fields in the buffer is returned in CDDDataLen.

Code

Is a halfword that contains the certificate field code (600-677).

CertData

Is the certificate data that corresponds to the requested code. Note that a single field could appear multiple times in the returned buffer if more than one "answer" is valid.

CDRetCode

Indicates the return code from the certificate request. Possible values are:

0 - No errors.

4021 - The partner value is not valid.

4023 - The partner certificate is not available.

4024 - The certificate does not contain any values.

4025 - The buffer length passed is too large.

4026 - The returned data will not fit in the provided buffer. Partial data is returned.

4027 - The passed buffer pointer is null.

4028 - The number of certificate fields requested (CDReqNum) is 0.

4029 - The number of certificate fields requested (CDReqNum) is greater than 64.

4030 - The requested certificate field is not found.

4031 - The requested certificate field is not valid.

4032 - Both of these errors exist in the return data: A requested certificate field is not found *and* a requested certificate field is not valid.

CDRetCnt

Is the number of certificate fields returned in CDDData.

CDDDataLen

Is the length of the returned certificate data.

CDRes

Is reserved - will be 0.

Usage Notes

- Certificate fields will be placed in the CDDData buffer in the order in which they appear in the CertReqCodes input structure.
- The CDDData buffer will contain as many certificate fields as will fit completely. If a requested certificate field does not fit in the buffer, it will not be returned and subsequent fields in the CertReqCodes input structure will also fail. CDRetCode will indicate that not all of the data will fit in CDDData. CDRetCnt will reflect the number of completed requests.
- If the requested field cannot be found in the certificate, CDDData will contain a Len of 4 along with the requested Code. No data will be returned. CDRetCode will be updated to indicate that one or more fields are not present in the certificate.

TcpSClient

Indicates to the SSL server that the connection is to be secure and that the SSL server needs to initiate an outbound handshake.

```

procedure TcpSClient
(
    Connection:           ConnectionType;
    Wait:                 boolean;
    SecureClientDetail:    SecureDetailType;
    var HandshakeCompleteDetail: SecureHSCompleteDetailType;
    var ReturnCode:        CallReturnCodeType
);
external;

```

Operand
Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record.

Wait

Set to true if your program should wait for the Close_Notify to complete. Set to false otherwise.

SecureClientDetail

Sets up the details of the request to be sent to the SSL server.

```

SecureDetailType =
record
    TLSLabel:           DirectoryNameType;
    TLSTimeout:         integer;
    RequestClientCert:  boolean;
    ValidatePeerCert:   ValidateCertType;
    CipherRequest:      CipherSuiteType;
    Version:            UnsignedByteType;
    Keyring:            KeyringType;
    Buffer:              string[255];
    SecDetailExt:       SecureDetailExtensionType;
end;

ValidateCertType =
(
    Full_Check,
    No_Check
);

CipherSuiteType =
(
    Default,
    NoV2
);

KeyringType = packed array (. 1.. KEYRINGlength .) of char;
KEYRINGlength = 50;

SecureDetailExtensionType =
packed record
    ValidationFlags: integer;
    ValidationLen: integer;
    ValidationBuffer array (.1..512.) of char;
end;

```

Operand
Description

TLSLabel

The label associated with the certificate in the certificate database.

TLSTimeout

This capability is not yet available. The value must be 0.

RequestClientCert

See **ValidatePeerCert**.

ValidatePeerCert

The RequestClientCert and ValidatePeerCert flags are used in combination to determine the level of client certificate checking that will be done during a secure server call. The level and the flag settings are as follows:

None

A client certificate will not be requested.

```
RequestClientCert = 0
ValidatePeerCert  = 1  (No_Check)
```

Preferred

A client certificate is requested. If a client certificate is not received, the connection will proceed without it. If a client certificate is received, it will be authenticated. If the client certificate is not valid, the failure will be logged in the SSL console log and the connection will continue as a secure connection protected by the server certificate.

```
RequestClientCert = 1
ValidatePeerCert  = 1  (No_Check)
```

Required

A client certificate will be authenticated. If a client certificate is not received, the connection will be terminated with a fatal TLS error. If the certificate fails authentication, the handshake will fail.

```
RequestClientCert = 1
ValidatePeerCert  = 0  (Full_Check)
```

Note: For a secure client call, the server certificate is always validated. Set these flags to indicate a level of None.

CipherRequest

This field is set to NoV2 for clients that do not want to use SSL V2. When set to Default, default cipher suite values will be used.

Version

When set to 0, the SecDetailExt is not passed on the call. When set to 1, the SecDetailExt is filled in and passed on the call to tell the SSL/TLS server to compare the passed-in host name, domain name, or IP address against the server certificate. A value of 1 is valid only when securing the client side of the connection.

Keyring

This capability is not yet available. The value must be blank.

Buffer

Contains the string that the SSL server will send out on the connection before waiting for the handshake. After this command is sent, the initiation of the handshake is expected on the connection. If an empty buffer is sent, a READYforHANDSHAKE notification will be sent to indicate that this side of the connection is waiting for the handshake.

ValidationFlags

Possible values:

0 indicates not required. If the validation text does not match what is in the server certificate, the mismatch will be logged and the handshake will continue.

1 indicates required. At least one of the specified validation items must match what is in the server certificate. If there are no matching items, the handshake will fail.

ValidationLen

The total length of ValidationBuffer.

ValidationBuffer

Contains multiple items to validate against the certificate. Each item has the following format:

```

+-----+
| Len | Type | Text |
+-----+

```

The total length of all items (Len+Type+Text) must not exceed 512 bytes.

Len

A halfword field that contains the total length of the item (Len+Type+Text). The total of all of the Len fields in the buffer should equal ValidationLen.

Type

A halfword field that contains the type of the Text data:

- 0** indicates an IPv4 address in integer format with 4-byte hexadecimal representation. For example: 093C1C66.
- 1** indicates an IPv6 address in integer format with 16-byte hexadecimal representation. For example: 50C6 C2C1 0000 0000 0009 0060 0028 0102.
- 2** indicates a fully-qualified domain name (FQDN) in EBCDIC format.
- 3** indicates a host name in EBCDIC format.
- 4** indicates an IPv4 address in dotted decimal format. For example: 9.60.28.102.
- 5** indicates an IPv6 address in dotted decimal format. For example: 50C6:C2C1::9.60.28.102.

Text

The string that is compared to the common name, domain name, or in a subject alternate name extension marked as an IP address in the server certificate.

Note: When Version is 1, the caller must allocate and send the full length of the ValidationBuffer (512 bytes) even though it might be partially filled in.

HandshakeCompleteDetail

When the wait flag is set to true, the results of the handshake will be returned here. If the wait flag is set to false, the results of the handshake will be reflected in the SecureHandshakeComplete notification (see [SECUREhandshakeCOMPLETE](#)).

ReturnCode

Indicates the success or failure of the call. Possible return values include:

- OK
- BACKlevelSSL
- KEYRINGnotPERMITTED
- KEYRINGnotRECOGNIZED
- LABELnotPERMITTED
- LABELnotRECOGNIZED
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- SSLcloseINprogress
- TCPIPshutdown
- TLSnotAVAILABLE
- UNAUTHORIZEDuser
- ALREADYsecured
- STATICALLYsecured
- INVALIDrequest
- ALREADYclosing

- ZeroResources

TcpSClose

Perform Close_Notify on a TLS connection but leave the TCP session up.

```
procedure TcpSClose
(
    Connection:      ConnectionType;
    Wait:            boolean;
    Buffer:           string[255];
    var ReturnCode:  CallReturnCodeType
);
```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record.

Wait

Set to true if your program should wait for the Close_Notify to complete. Set to false otherwise.

Buffer

Contains a string of data that the SSL server will send out on the connection prior to switching the connection to clear text.

ReturnCode

Indicates the success or failure of the call. Possible return codes include:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREError
- SSLcloseINprogress
- TCPIPshutdown
- TLSnotAVAILABLE
- UNAUTHORIZEDuser

TcpSServer

The TcpSServer procedure indicates to the SSL server that the connection is to be secure and that the SSL server needs to wait for an incoming handshake.

```
procedure TcpSServer
(
    Connection:      ConnectionType;
    SecureServerDetail: SecureDetailType;
    var ReturnCode:  CallReturnCodeType
);
external;
```

Operand Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record.

SecureServerDetail

Sets up the details of the request to be sent to the SSL server. See [SecureDetailType](#).

ReturnCode

Indicates the success or failure of the call. Possible return values include:

- OK
- BACKlevelSSL
- KEYRINGnotPERMITTED
- KEYRINGnotRECOGNIZED
- LABELnotPERMITTED
- LABELnotRECOGNIZED
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- SSLcloseINprogress
- TCPIPshutdown
- TLSnotAVAILABLE
- UNAUTHORIZEDuser
- ALREADYsecured
- STATICALLYsecured
- INVALIDrequest
- ALREADYclosing
- ZeroResources

TcpSSStatus

The TcpSSStatus procedure returns details about a session such as whether or not it is secure and the encryption suite.

```

procedure TcpSSStatus
(
    Connection:      ConnectionType;
    var Secure:      SecureType;
    var CipherDetails: CipherDetailsType;
    var ReturnCode:  CallReturnCodeType
);
external;

```

Operand

Description

Connection

Specifies the connection number, as returned by TcpOpen or TcpWaitOpen in the connection field of the StatusInfoType record.

Secure

Describes how the connection is secured.

```
SecureType = (SecNone, SecStatic, SecDynamic)
```

SecNone

The connection is not secure.

SecStatic

The connection is statically secured.

TcpStatus

SecDynamic

The connection is dynamically secured.

CipherDetails

When a connection is secure, this field describes the cipher details that are in effect.

```
CipherDetailsType =
  record
    CipherClass:      CipherClassType;
    CipherHash:       CipherHashType;
    CipherAlgorithm:  CipherSymmetricAlgorithmType;
    CipherPkAlgorithm: CipherPkAlgorithmType;
    CipherKeyLength: integer;
  end;
CipherClassType = ( NULLclass, SSLV2, SSLV3, TLS,
                    TLS10, TLS11, TLS12);
CipherSymmetricAlgorithmType = ( NULLalgorithm,
                                RC2,           {deprecated}
                                RC4,           {deprecated}
                                DES,           {deprecated}
                                DES3,          {deprecated}
                                FIPSDDES,     {deprecated}
                                FIPS3DES,     {deprecated}
                                AES,
                                AESGCM,
                                AES128,
                                AES128GCM,
                                AES256,
                                AES256GCM);
CipherPkAlgorithmType = ( NULLpkAlgorithm, RSA, DH_DSS,
                          DH_RSA, DHE_DSS, DHE_RSA,
                          ECDH_ECDSA, ECDHE_ECDSA,
                          ECDH_RSA, ECDHE_RSA );
CipherHashType = ( SHA1, MD5, NULLhash, SHA2, SHA256,
                   SHA384 );
```

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREError
- TCPIPshutdown
- UNAUTHORIZEDDuser

TcpStatus

The TcpStatus procedure obtains 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;
```

Operand

Description

ConnectionInfo

If ReturnCode is OK, the following fields are returned:

Field

Description

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

Specifies the number of bytes of incoming data queued for your program (waiting for TcpReceive, TcpFReceive, or TcpWaitReceive).

UnackedBytes

Specifies 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

Specifies the current connection state.

LocalSocket

Specifies the local , consisting of a local address and a local port.

ForeignSocket

Specifies the foreign , consisting of a foreign address and a foreign port.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- IPv6Connection
- NOsuchCONNECTION
- NOTyetBEGUN
- TCPIPshutdown

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

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.

TcpVmcfRupt

The TcpVmcfRupt procedure is used when you initiate 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. For more information, see “RTcpExtRupt” on page 74 and “RTcpVmcfRupt” on page 74.

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

Operand

Description

VmcfHeaderAddress

Indicates the address of your VMCF interrupt buffer as specified in the VMCF AUTHORIZE function that your program issued at initialization.

Udp6Open

The Udp6Open procedure requests acceptance of UDP datagrams on the specified socket and allows datagrams to be sent from the specified socket. Udp6Open should be used for IPv6 connections, or for a mix of IPv4 and IPv6 connections. 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.

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 Udp6Open
(
  var LocalSocket: Socket6Type;
  var ConnIndex: ConnectionIndexType;
  var ReturnCode: CallReturnCodeType
);
external;
```

Operand Description

LocalSocket

Specifies the local socket (address and port pair).

ConnIndex

Specifies the ConnIndex value returned from UdpOpen.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- LOCALportNOTavailable
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UDPlocalADDRESS
- UDPzeroRESOURCES

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

Note: If a Udp6Open call returns UDPzeroRESOURCES, and your application handles UDPresourcesAVAILABLE notifications, you receive a UDPresourcesAVAILABLE notification when sufficient resources are available to process a Udp6Open call. The first Udp6Open your program issues after a UDPresourcesAVAILABLE notification is guaranteed not to get the UDPzeroRESOURCES return code.

Udp6Send

The Udp6Send procedure sends a UDP datagram to the specified foreign socket. Udp6send should be used for IPv6 connections, or for a mix of IPv4 and IPv6 connections. The source socket is the local socket selected in the Udp6Open that returned the ConnIndex value that was used. The buffer does not include the UDP header. This header is supplied by the TCPIP virtual machine.

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

```

procedure Udp6Send
(
    ConnIndex: ConnectionIndexType;
    ForeignSocket: Socket6Type;
    BufferAddress: integer;
    Length: integer;
var    ReturnCode: CallReturnCodeType
);
external;

```

Operand Description

ConnIndex

Specifies the ConnIndex value returned from UdpOpen.

ForeignSocket

Specifies the foreign socket (address and port) to whom the datagram is to be sent.

BufferAddress

Specifies the address of your buffer containing the UDP datagram to be sent, excluding UDP header.

Length

Specifies the length of the datagram to be sent, excluding UDP header. Maximum is 8192 bytes.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- BADlengthARGUMENT
- MIXEDaddress
- NObufferSPACE
- NOSuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UDPunspecADDRESS
- UDPunspecPORT

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

UdpClose

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

```

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

```

Operand Description

ConnIndex

Specifies the ConnIndex value returned from UdpOpen.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

UdpNReceive

- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPIPshutdown

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

UdpNReceive

The UdpNReceive procedure notifies the TCPIP virtual machine that you can 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;
```

Operand Description

ConnIndex

Specifies the ConnIndex value returned from UdpOpen.

BufferAddress

Specifies the address of your buffer that will be filled with a UDP datagram.

BufferLength

Specifies the length of your buffer. If you specify a length larger than 8192 bytes, only the first 8192 bytes are used.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- NOsuchCONNECTION
- NOTyetBEGUN
- RECEIVEstillPENDING
- TCPIPshutdown

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

UdpOpen

The UdpOpen 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.

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;

```

Operand Description

LocalSocket

Specifies the local socket (address and port pair).

ConnIndex

Specifies the `ConnIndex` value returned from `UdpOpen`.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- LOCALportNOTavailable
- NOTyetBEGUN
- SOFTWAREerror
- TCPipSHUTDOWN
- UDPlocalADDRESS
- UDPzeroRESOURCES

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

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.

UdpReceive

The `UdpReceive` procedure notifies the TCPIP virtual machine 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 can put a datagram of up to 2012 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 virtual machine 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;
```

Operand	Description
---------	-------------

ConnIndex	Specifies the ConnIndex value returned from UdpOpen.
DatagramAddress	Specifies the address of your buffer that will be filled with a UDP datagram.
ReturnCode	Indicates the success or failure of the call. Possible return values are: <ul style="list-style-type: none">• OK• ABNORMALcondition• FATALerror• NOSuchCONNECTION• NOTyetBEGUN• SOFTWAREerror• TCPipSHUTDOWN

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,” on page 337](#).

UdpSend

The UdpSend 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 the TCPIP virtual machine.

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;
```

Operand	Description
---------	-------------

ConnIndex	Specifies the ConnIndex value returned from UdpOpen.
ForeignSocket	Specifies the foreign socket (address and port) to whom the datagram is to be sent.
BufferAddress	Specifies the address of your buffer containing the UDP datagram to be sent, excluding UDP header.
Length	Specifies the length of the datagram to be sent, excluding UDP header. Maximum is 8192 bytes.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- BADlengthARGUMENT
- NObufferSPACE
- NOsuchCONNECTION
- NOTyetBEGUN
- SOFTWAREerror
- TCPIPshutdown
- UDPUnderspecADDRESS
- UDPUnderspecPORT

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

Unhandle

The Unhandle procedure specifies that 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;

```

Operand**Description****Notifications**

Specifies the set of notifications that you no longer want to receive.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- ABNORMALcondition
- FATALerror
- INVALIDrequest
- NOTyetBEGUN
- TCPIPshutdown

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

UnNotifyIo

The UnNotifyIo routine is used to indicate that you no longer wish to be sent a notification when an I/O interrupt occurs on the specified virtual address.

```

procedure UnNotifyIo
(
    DeviceAddress: integer;
var   ReturnCode: integer
);
external;

```

Operand Description

DeviceAddress

Specifies the address of the device for which IOinterrupt notifications are no longer to be generated.

ReturnCode

Indicates the success or failure of the call. Possible return values are:

- OK
- NOsuchCONNECTION
- SOFTWAREerror

For a description of Pascal ReturnCodes, see [Appendix B, “Pascal Return Codes,”](#) on page 337.

Sample Pascal Program

The following is an example of a sample Pascal program.

```

%UHEADER 5741-A05 (C) COPYRIGHT 1991, 2024 BY IBM, PSAMPLE.

{
    Licensed Materials - Property of IBM
    This product contains "Restricted Materials of IBM"
    5741-A05 (C) Copyright IBM Corp. - 1991, 2024
    All rights reserved.
    US Government Users Restricted Rights -
    Use, duplication or disclosure restricted by GSA ADP Schedule
    Contract with IBM Corp.
    See IBM Copyright Instructions.
}

{
    Change Activity
    VREBA - IPv6 Stage 1 line item
}

{*****}
{ * }
{ * 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 PSAMPLE;

#include CMALLCL
#include CMINTER
#include CMRESGLB

const
    BUFFERlength = 8192;           { same as MAXdataBUFFERsize }
    PORTnumber   = 9876;           { 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  : IPAddressType;
  AddrSpec    : IPv6AddrSpecType;
  Lookup      : LookupSetType;
  IbmSeconds   : integer;
  Ignored      : integer;
  Line        : string(80);
  Note        : NotificationInfoType;
  PushFlag    : boolean;
  RealRate     : real;
  ReturnCode   : integer;
  SendFlag     : boolean;
  StartingTime : TimeStampType;
  Thousandths  : integer;
  TotalBytes   : integer;
  UrgentFlag   : boolean;

var RoundRealRate : integer;

{*****}
{* 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('');
    Msg1(Output,1, addr(SayCalRe(ReturnCode)) )
  else Msg0(Output,2);

  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)); *}
    Msg1(Output,4, addr(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)'); *}
  Msg0(Output,5);
  ReadLn (Line);
  if (Substr(Ltrim(Line),1,1) = 's')
  or (Substr(Ltrim(Line),1,1) = 'S') then
    SendFlag := TRUE
  else
    SendFlag := FALSE;

  {* Writeln('Host Name or Internet Address :'); *}
  Msg0(Output,6);
  ReadLn (Line);
  Lookup := [IPv4];
  if not (GetIPAddr(Trim(Ltrim(Line)), HostAddress,

```

```

                                AddrSpec, Lookup)) then                                { @VRFBAP }
begin                                                                    { @VRFBAP }
    Restore ('GetIPAddr failed. ', OK);                                    { @VRFBAP }
    return;                                                                { @VRFBAP }
end;                                                                      { @VRFBAP }

{ 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.Ipv4Addr; { @VRFBAP }
        ForeignSocket.Port := PORTnumber;
    end
    else begin
        ConnectionState := LISTENING;
        LocalSocket.Address := HostAddress.Ipv4Addr; { @VRFBAP }
        LocalSocket.Port := PORTnumber;
        ForeignSocket.Address := UNSPECIFIEDaddress;
        ForeignSocket.Port := UNSPECIFIEDport;
    end
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 }

{ 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)); * }
    Msg1(Output, 7, addr(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
            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.');

```


Chapter 3. Virtual Machine Communication Facility Interface

The Virtual Machine Communication Facility (VMCF) is part of the Control Program (CP) of VM. VMCF enables virtual machines to send data to and receive data from any other virtual machine.

You can communicate directly with the TCPIP virtual machine using VMCF calls, rather than Pascal API or C socket calls. You can use VMCF calls when:

- You want to write your program in assembler.
- You add TCP/IP communication to an existing complex program, and it can be difficult or impossible for your program to monitor TCP/IP events through the Pascal GetNextNote interface.

If your program drives the VMCF interface directly, do not link any of the TCP interface library modules with your program. Consequently, you cannot use any of the auxiliary routines, such as the Say functions and timer routines. (You must use VM timer support, or support provided by your existing program). VMCF consists of data transfer functions, control functions, a special external interrupt for pending messages, and an external interrupt message header to pass control information and data to another virtual machine.

For more information about the VMCF interface, see *VM/ESA: CP Programming Services*.

General Information

The following section describes the data structure of the VMCF interrupt header used by TCP/IP for VM. The section also lists the VMCF functions available with TCP/IP for VM. Tables summarizing the CALLCODE for making VMCF requests and receiving notifications from TCPIP virtual machine are provided. The remainder of the chapter describes these CALLCODE calls in details.

Data Structures

VMCF is implemented with functions invoked using DIAGNOSE X'68' and a special 40-byte parameter list. A VMCF function is requested by a particular function subcode in the FUNC field in the parameter list.

Your program uses the standard 40-byte VMCF parameter list to submit VMCF requests to the TCPIP virtual machine. The TCPIP virtual machine returns VMCF interrupts results in the similar 40-byte VMCF parameter list. The parameter list is the interrupt header being stored in your virtual machine. In this chapter, fields in the parameter list and interrupt header are referred to using the data structure header names in [Figure 24 on page 113](#).

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 24. Assembler Format of the VMCF Parameter List Fields

VMCF Parameter List Fields

The following describes the VMCF parameter list fields.

V1

Used for security and data integrity. You can enable your virtual machine for VMCF communication to the TCPIP virtual machine by executing the AUTHORIZE control function. The AUTHORIZE control function is set by issuing a DIAGNOSE Code X'68' Subcode X'0000' assembler call. If you do not set the AUTHORIZE function in V1, check the JOBNAME field when processing each interrupt to ensure that interrupts from other virtual machines are not misinterpreted as coming from TCPIP. V1 must be zero for all VMCF functions other than AUTHORIZE. To terminate VMCF activities for a virtual machine, issue the UNAUTHORIZE control function. The UNAUTHORIZE control function is set by issuing a DIAGNOSE Code X'68' Subcode X'0001' assembler call.

FUNC

The IUCV operation.

V2

Reserved for IBM use, and should be X'00' initially.

MSGID

Contains a unique message identifier associated with a transaction. You must use a unique, even number for each outstanding transaction. A simple method is to use consecutive, even numbers for each transaction.

JOBNAME

Specifies the user ID of the virtual machine making VMCF requests. You must set this field to the user ID of the TCPIP virtual machine.

VADA

Contains the address of the source or destination address depending on the VMCF function requested.

LENA

Contains the length of the data sent by a user, the length of a RECEIVE buffer, or the length of an external interrupt buffer, whichever is specified in the VADA field.

VADB

Contains the address of a source virtual machine's REPLY buffer for VMCF request.

LENB

Specifies the length of the source virtual machine's REPLY buffer.

The use of each field is described individually for each TCP/IP function.

VMCF Interrupt Header Fields

The following describes the VMCF parameter list fields for the interrupt header.

V1

Sets the VMCMRESP flag, which is the interrupt in response to a transaction initiated by your virtual machine. If the TCPIP virtual machine 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 completion status code in the RETCODE field, as described for each function.

ANINTEGR

Checks the status of VMCF transactions. It is a field, of fullword length (four bytes), used to check the status of VMCF transactions. The field is described for each function.

CONN

Establishes a TCP connection. If a connection between your virtual machine and TCPIP virtual machine was established successfully and the RETCODE field indicates OK, the connection number of the new connection is stored in this field.

CALLCODE

Calls instructions to be passed by your program when initiating a VMCF function to interface with TCPIP virtual machine. If the interrupt is in response to a transaction initiated by your virtual machine

(VMCMRESP flag set in V1), the CALLCODE value is the same as the value set by your program when it initiated the transaction.

RETCODE

Contains the completion status codes of a transaction. Return codes reported in this field are taken from the same set used by Pascal programs (see [Appendix B, “Pascal Return Codes,”](#) on page 337). Further information is given in the description of each function.

VMCF Functions

Table 18 on page 115 lists the available VMCF functions, with descriptions, to communicate with the TCPIP virtual machine.

Table 18. Available VMCF Functions

Function	Code	Description
AUTHORIZE	Control	Initializes VMCF for a given virtual machine. Once AUTHORIZE is executed, the virtual machine can execute other VMCF functions and receive messages or requests from other users.
UNAUTHORIZE	Control	Terminates VMCF activity.
SEND	Data	Directs a message or block of data to another virtual machine.
SEND/RECV	Data	Directs a message or block of data to another virtual machine, and requests a reply.
RECEIVE	Data	Allows you to accept selective messages or data sent using the SEND or SEND/RECV functions.
REPLY	Data	Allows you to direct data back to the originator of a SEND/RECV function, simulating duplex communication.
REJECT	Data	Allows you to reject specific SEND or SEND/RECV requests pending for your virtual machine.

Note:

Data

Indicates a data transfer

Control

Indicates a VMCF control function

VMCF TCPIP Communication CALLCODE Requests

Table 19 on page 115 lists the equate values and available calls for initiating a VMCF TCPIP request; it also includes a description of each CALLCODE request.

Table 19. VMCF TCPIP CALLCODE Requests

Call Code	Equates	Description
CONNECTIONclosing	00	Data may no longer be transmitted on this connection since the TCP/IP service is in the process of closing down the process.
LISTENING	01	Waiting for a foreign site to open a connection.
NONEXISTENT	02	The connection no longer exists.
OPEN	03	Data can go either way on the connection.

Table 19. VMCF TCPIP CALLCODE Requests (continued)

Call Code	Equates	Description
RECEIVINGonly	04	Data can be received but not sent on this connection, because the client has done a one-way close.
SENDINGonly	05	Data can be sent out but not received on this connection. This means that the foreign site has done a one way close.
TRYINGtoOPEN	06	Trying to contact a foreign site to establish a connection.
V6OPENtcp	33	Initiates a TCP connection (IPv4 or IPv6).
V6STATUStcp	34	Obtains the IPv6 Connection Information Record giving the current status of a TCP connection (IPv4 or IPv6).
ABORTtcp	100	Terminates a TCP connection.
BEGINtcpIPservice	101	Initializes a TCP/IP connection between your program and the TCPIP virtual machine.
CLOSEtcp	102	Initiates the closing of a TCP connection.
CLOSEudp	103	Initiates the closing of a UDP connection.
ENDtcpIPservice	104	Terminates the use of TCPIP services. All existing TCP connections are reset, all open UDP ports are canceled, and all IP protocols are released.
HANDLEnotice	105	Specifies the types of notifications to be received from TCPIP.
IShostLOCAL	106	Determines whether a given internet address is one of your host's local addresses.
MONITORcommand	107	Instructs TCPIP to obey a file of commands.
MONITORquery	108	Obtains status information from the TCPIP virtual machine or requests that it performs certain functions.
OPENtcp	110	Initiates a TCP connection for IPv4 only.
OPENudp	111	Initiates a UDP connection for IPv4 only.
OPTIONtcp	112	Sets an option for a TCP connection.
RECEIVEtcp	113	Tells TCPIP that you are ready to receive data on a specified TCP connection.
NRECEIVEudp	115	Tells TCPIP that your program is ready to receive a UDP datagram on a particular port.
SENDtcp	118	Sends data on a TCP connection. The SENDtcp transaction is unsuccessful if the receiving TCPIP virtual machine has insufficient buffer space to receive the data.
SENDudp	119	Sends a UDP datagram.
STATUStcp	120	Obtains a Connection Information Record giving the current status of a TCP IPv4 connection.
FRECEIVEtcp	121	Tells TCPIP virtual machine that you are ready to receive data on a specified TCP connection. TCPIP does not respond or send a notification until the data has been placed in the receiving buffer or the connection has been closed.

Table 19. VMCF TCPIP CALLCODE Requests (continued)

Call Code	Equates	Description
FSENDtcp	122	Sends data to a TCP connection. FSENDtcp waits for available receiving buffer space in the TCPIP virtual machine before completing the VMCF transaction.
CLOSErawIP	123	Tells TCPIP that your program does not handle the protocol any longer. Any queued incoming packets are discarded.
OPENrawIP	124	Initiates a connection and tells TCPIP virtual machine that your program is ready to send and receive packets of a specified IP protocol.
RECEIVERawIP	125	Tells 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.
SENDrawIP	126	Tells TCPIP virtual machine to send raw IP packets of a given protocol number.
PINGreq	127	Sends an ICMP echo request to a specified host and wait a specified time for a response.
TLSQuery	128	Sends a query to determine if the SSL server is available and, if so, if the label specified is known.
TLSSCLOSEtcp	132	Indicates to the SSL Server that secure communication on this connection should stop and communication should continue in the clear.
TLSSSTATUStcp	129	Returns details about a session, such as whether or not it is secure and the encryption suite.
TLSSSERVERtcp	130	Indicates to the SSL server that the connection is to be secure and that the SSL server needs to wait for an incoming handshake.
TLSSCLIENTtcp	131	Indicates to the SSL server that the connection is to be secure and that the SSL server needs to initiate an outbound connection.
V6OPENudp	135	Initiates a UDP connection (IPv4 or IPv6).
V6SENDudp	136	Sends a UDP datagram (IPv4 or IPv6).
TLSSCERTDATAReqtcp	137	Requests specific fields from the partner or local certificate.

VMCF TCPIP Communication CALLCODE Notifications

Table 20 on page 117 lists the equate values for the CALLCODE field when VMCF TCPIP sends a notification to your program. The table includes a description of each CALLCODE response.

Table 20. VMCF TCPIP CALLCODE Notifications

Notification Code	Equates	Description
BUFFERspaceAVAILABLE	10	Notification that there is space available to send data on this connection. The space is currently set to 8192 bytes of buffer space.

Table 20. VMCF TCPIP CALLCODE Notifications (continued)

Notification Code	Equates	Description
CONNECTIONstateCHANGED	11	Notification that the state of the connection between the TCPIP virtual machine and your program has changed.
DATAdelivered	12	Notification that the TCPIP virtual machine data was delivered to your program, after issuing a RECEIVEtcp or FRECEIVEtcp call.
URGENTpending	15	Notification that there is queued data on a TCP connection not yet received by your program.
UDPdatagramDELIVERED	16	Notification that UDP datagram has been delivered to your program after issuing a NRECEIVEudp call to the TCPIP virtual machine.
UDPdatagramSPACEavailable	17	Notification that buffer space is available to process the data, after an error occurred performing a SENDudp call.
RAWIPpacketsDELIVERED	24	Notification that your buffer has received the raw IP packets.
RAWIPspaceAVAILABLE	25	Notification that buffer space is available to process the data. This notification is sent after the SENDrawip call was rejected by TCPIP virtual machine.
RESOURCESavailable	28	Notification that the resources needed to initiate a TCP connection are now available. This notification is sent only if a previous OPENtcp call received a ZEROresources return code.
UDPresourcesAVAILABLE	29	Notification that the resources needed to initiate a UDP connection are now available. This notification is sent only if a previous OPENudp call received a UDPzeroRESOURCES return code.
PINGresponse	30	Notification that your ping request from the PINGreq call has been received or that the time-out limit or your request has been reached.
DUMMYprobe	32	Notification that the TCPIP virtual machine is monitoring your machine
ACTIVEprobe	33	Notification that the TCPIP virtual machine is monitoring your machine for responsiveness
CLEARtextRESUMED	34	Notification that the SSL Server has stopped secure communication on the connection
QUERYtlsCOMPLETE	35	Notification that the SSL server has completed verification of the label
SECUREhandshakeCOMPLETE	36	Notification that the Inbound or Outbound handshake has completed.
READYforHANDSHAKE	37	Notification that the server side is set up to receive a secure handshake.

Table 20. VMCF TCPIP CALLCODE Notifications (continued)

Notification Code	Equates	Description
CERTdataCOMPLETE	38	Notification that the requested certificate data is available.

TCP/UDP/IP Initialization and Termination Procedures

This section contains information about procedures for initializing and terminating TCP/UDP/IP connections.

BEGINtcpIPservice

Your program performs the BEGINtcpIPservice call after doing a VMCF AUTHORIZE function, but before performing any other TCP/IP functions. The BEGINtcpIPservice call informs TCPIP that your virtual machine uses TCPIP services. An ENDtcpIPservice call is logically performed first, in the case where your virtual machine already has TCPIP resources allocated.

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      0 or, if your application supports probe messages (see the
            descriptions of the DUMMYprobe and ACTIVEprobe CALLCODE
            notifications), X'80000000'
LENB:      0
CONN:      0 or, if your application does not provide the client level in
            ANINTEGR, any non-zero value
ANINTEGR:  the client level, in the form X'vvrl0000', where 'vv' is the
            version number, 'r' is the release number, and 'l' is the
            level number
CALLCODE:  BEGINtcpIPservice

```

The TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header, stored in your virtual machine 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 (see “BeginTcpIp” on page 60).

The VMCF interrupt header also includes values in the CONN and ANINTEGR fields that reflect the level information for the TCPIP virtual machine. If CONN is zero, then ANINTEGR contains the TCPIP virtual machine's level; otherwise, no level information is returned.

ENDtcpIPservice

Your program performs the ENDtcpIPservice call when it has finished using TCPIP services. All existing TCP connections are reset (aborted), all open 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 virtual machine responds using the VMCF REJECT function. The VMCF interrupt header indicates a return code of OK in the RETCODE field.

HANDLEnotice

Your program performs the HANDLEnotice call to specify the types of notifications to be received from TCPIP. The VADB field in the VMCF parameter 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 25 on page 120](#).

Figure 25 on page 120 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 25. Equates for Notification Mask in the HANDLEnotice Call

Each HANDLEnotice call must specify all the notification types to be handled. Notification types specified in previous HANDLEnotice calls are not stored.

```

FUNC:    SEND
VADA:    0
LENA:    1
VADB:    Note mask
LENB:    0
CALLCODE: HANDLEnotice

```

The TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header contains a return code in the RETCODE field. The return code can be any of those listed for the Handle Pascal procedure (see [“Handle” on page 64](#)).

TCP CALLCODE Requests

The following sections describe the VMCF interrupt headers that are stored in your virtual machine for CALLCODE calls used to make TCP requests.

CLOSEtcp

The CLOSEtcp call initiates the closing of a TCP connection. For more information about the close connection call, see the Pascal procedure, [“TcpClose” on page 82](#).

```

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 contains the return code. The return code is one of those listed for the TcpClose Pascal procedure, see [“TcpClose” on page 82](#).

FRECEIVEtcp

The FRECEIVEtcp call tells TCPIP that you are ready to receive data on a specified TCP connection. TCPIP does not respond or send a notification notice 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 for each virtual machine; 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 does not need to wait for a response from FRECEIVetcp. It can issue functions involving other connections, before receiving a response from FRECEIVetcp.

For general information about receiving TCP data, see the TcpFReceive Pascal procedure under “TcpFReceive, TcpReceive, and TcpWaitReceive” on page 83.

```

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 receives 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 are:

LENB

Indicates 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

Contains a value where 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

Specifies the connection number.

RETCODE

OK

If TCPIP responds with the VMCF REJECT function (VMCFRJCT flag set in the VMCF interrupt header), then one of the following occurred:

- TCPIP did not accept the request, in which case the ANIntegerFLAGrequestERR bit in ANINTEGR is on.
- TCPIP 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. For more information, see [“2” on page 49](#).

Note: 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.

OPENTcp

The OPENTcp call initiates a TCP connection for IPv4 only. Your program sends a Connection Information Record to TCPIP. [Figure 26 on page 122](#) gives the assembler format of the record. [Figure 27 on page 122](#) gives the equates for the assorted constants used to set up the record. For more information about the usage of the fields of the Connection Information Record, see [“TcpOpen and TcpWaitOpen” on page 88](#).

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 26. Assembler Format of the Connection Information Record for VM

UNSPECIFIEDconnection	EQU	-48
DEFAULTsecurity	EQU	0
DEFAULTcompartment	EQU	0
DEFAULTprecedence	EQU	0
UNSPECIFIEDaddress	EQU	0
UNSPECIFIEDport	EQU	X'FFFF'
ANIntegerFLAGrequestERR	EQU	X'80000000'

Figure 27. Miscellaneous Assembler Constants

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 virtual machine responds using the VMCF REJECT function. The VMCF interrupt header, contains a return code in the RETCODE field. The return code can be any of those listed for the Tcp6Open Pascal procedure.

If the OPENTcp call was rejected because not enough TCPIP resources were available, a ZEROresources code is returned. When the TCPIP resources are available, a notice of RESOURCESavailable is sent to your program.

If the open attempt is not immediately rejected, the TCPIP virtual machine uses 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 is set as described in the previous paragraph.

If the open was successfully initiated, the TCPIP virtual machine responds using the VMCF REPLY function to send back the updated Connection Information Record. The Connection field of the Connection Information Record contains the connection number of the new connection. The RETCODE field in the VMCF interrupt header indicates 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. For more information about NotificationInfoType, see the section on the Pascal under “Notification Record” on page 45 and see also “CALLCODE Notifications” on page 134.

OPTIONtcp

The OPTIONtcp call sets an option for a TCP connection for IPv4 only. For more information about the connection options, see the Pascal procedure, “TcpOption” on page 90.

```

FUNC:    SEND
VADA:    0
LENA:    1
VADB:    Option name
LENB:    Option value
CONN:    Connection number from open
CALLCODE: OPTIONtcp

```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code. The return code is one of those listed for the Pascal TcpOption procedure.

RECEIVEtcp

The RECEIVEtcp call tells 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 receive requests pending using RECEIVEtcp without exceeding the limit of 50 outstanding VMCF transactions.

For more information about receiving TCP data, see the TcpReceive Pascal procedure under “[TcpFReceive, TcpReceive, and TcpWaitReceive](#)” on page 83.

```

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 indicates whether the request was successful. If the request was successful, with a RETCODE of OK, the data is delivered to your buffer and a notification of DATAdelivered is sent to your program. If the request was not successful, then the return code is one of those listed for the TcpReceive Pascal procedure.

SENDtcp and FSENDtcp

The SENDtcp or FSENDtcp calls send data on a TCP connection. For the advantages and disadvantages of using each function, and for information about sending TCP data, see “[TcpFSend, TcpSend, and TcpWaitSend](#)” on page 86.

```

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 indicates a RETCODE of OK.

If TCPIP cannot queue the data for sending, it responds with the VMCF REJECT function. The RETCODE field indicates the type of error. The return code can be any of those listed for the TcpSend Pascal procedure. For a list of the return codes, see “[TcpFSend, TcpSend, and TcpWaitSend](#)” on page 86.

If the SENDtcp transaction is unsuccessful, because of insufficient space in the buffer of the receiving TCPIP virtual machine, a return code of NObufferSPACE is placed in the RETCODE field. A notification of BUFFERspaceAVAILABLE is sent, on this connection, when the space is available to process data.

TcpFSend is the same as FSENDtcp. If TCPIP cannot accept the data, because of 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 previously. In summary, TCPIP may not respond immediately to FSENDtcp, and the response, after waiting, may indicate either success or failure. If TCPIP responds with REJECT, your program can check the ANIntegerFLAGrequestERR bit in the ANINTEGR field to determine if the request was rejected during initial or retry processing (bit on) or because of connection closing (bit off).

Your program does not need to wait for a response from SENDtcp or FSENDtcp VMCF transaction. It can issue functions involving other connections, before receiving a response from making a SENDtcp or FSENDtcp VMCF transaction.

There is a limit of 50 outstanding VMCF transactions for each virtual machine; therefore, your program can have FSENDtcp functions outstanding on only 50 connections at a time. If your application needs more outstanding sends, use the SENDtcp function.

STATUStcp

The STATUStcp call obtains a Connection Information Record giving the current status of a TCP connection for IPv4 only. For the assembler format of the Connection Information Record, see [Figure 26 on page 122](#). For more information about the connection status call, see the Pascal procedure, “TcpStatus” on page 100.

```

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, 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.

TLSSCERTDATAREQtcp

The TLSSCERTDATAREQtcp call indicates to the SSL server that certificate data is being requested for the local or partner certificate. For more information about the certificate data request call, see “TcpSCertData” on page 91.

```

FUNC:    SEND/RECEIVE
VADA:    Address of CertReqDetailType record
LENA:    Size of CertReqDetailType record
VADB:    ANYoldADDRESS
LENB:    0
CONN:    Connection number from open
CALLCODE: TLSSCERTDATAREQtcp

```

TCP/IP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code from the preprocessing of this command. The return code is one of those listed for the TcpSCertData procedure. The results of the actual certificate data request will be returned with the CERTdataCOMPLETE notification.

TLSSCLIENTtcp

The TLSSCLIENTtcp call indicates to the SSL server that the connection is to be secure and that the SSL server needs to initiate an outbound connection. For more information about the secure client call, see the Pascal Procedure , “TcpSClient” on page 94.

```

FUNC:      SEND/RECV
VADA:      Address of SecureDetailType record
LENA:      Length of SecureDetailType record
VADB:      0
LENB:      1
CONN:      Connection number from open
CALLCODE:  TLSSCLIENTtcp

```

See [“TcpSClient” on page 94](#) for details of the SecureDetailType structure.

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code from the preprocessing of this command. The return code is one of those listed for the TcpSClient procedure. The results of the actual handshake will be returned with the SECUREhandshakeCOMPLETE notification.

TLSSCLOSEtcp

The TLSSCLOSEtcp call indicates to the SSL Server that the connection should no longer be secure. The SSL server issues a Close_Notify command on the connection and sends a notification to indicate that data transmission can continue in the clear. For more information about the secure close call, see the Pascal Procedure, [“TcpSClose” on page 98](#).

```

FUNC:      SEND
VADA:      0
LENA:      1
VADB:      0
LENB:      1
CONN:      Connection number from open
CALLCODE:  TLSSCLOSEtcp
FUNC:      SEND/RECV

```

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code from the pre-processing of this command. The return code is one of those listed for the TcpSClose procedure. The results of the actual Close_Notify command will be returned with the CLEARtextRESUMED notification.

TLSSSERVERtcp

The TLSSSERVERtcp call indicates to the SSL server that the connection is to be secure and that the SSL server needs to wait for an incoming handshake. For more information about the secure server call, see [“TcpSClient” on page 94](#).

```

FUNC:      SEND/RECV
VADA:      Address of SecureDetailType record
LENA:      Length of SecureDetailType record
VADB:      0
LENB:      1
CONN:      Connection number from open
CALLCODE:  TLSSSERVERtcp

```

See [“TcpSServer” on page 98](#) for details of the SecureDetailType structure.

TCPIP responds with the VMCF REJECT function. The RETCODE field of the VMCF interrupt buffer contains the return code from the preprocessing of this command. The return code is one of those listed for the TcpSServer procedure. The results of the actual handshake will be returned with the SECUREhandshakeCOMPLETE notification.

TLSSSTATUStcp

The TLSSSTATUStcp call returns details about a session, such as whether or not it is secure and the encryption suite. For more information about the connections security status, see the Pascal Procedure, [“TcpSStatus”](#) on page 99.

```
FUNC:      SEND/RECV
VADA:      0
LENA:      1
VADB:      Address of CipherDetails Record to be filled in
LENB:      Length of CipherDetails Record to be filled in
CONN:      Connection number from open
CALLCODE:  TLSSSTATUStcp
```

See [“TcpSStatus”](#) on page 99 for details of the CipherDetails structure. TCPIP responds with the VMCF REPLY function, filling in the record whose address you supplied in VADB. The record is only valid if the return code, in the RETCODE field of the VMCF interrupt header, returns OK; otherwise, the return code is one of those listed for the TcpSStatus Pascal Procedure.

V6OPENtcp

The V6OPENtcp call initiates a TCP connection. Your program sends an IPv6 Connection Information Record to TCPIP. [Figure 28 on page 126](#) gives the assembler format of the record. [Figure 29 on page 126](#) gives the equates for the assorted constants used to set up the record. For more information about the usage of the fields of the Connection Information Record, see [“Tcp6Open and Tcp6WaitOpen”](#) on page 79.

```
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  XL16
LocalSocket.Port      DS  H
ForeignSocket.Address DS  XL16
ForeignSocket.Port    DS  H
```

Figure 28. Assembler Format of the IPv6 Connection Information Record for VM

```
UNSPECIFIEDconnection EQU  -48
DEFAULTsecurity        EQU   0
DEFAULTcompartment     EQU   0
DEFAULTprecedence      EQU   0
UNSPECIFIEDipv6address EQU   0
UNSPECIFIEDport        EQU  X'FFFF'
ANintegerFLAGrequestERR EQU  X'80000000'
```

Figure 29. Miscellaneous Assembler Constants

```

FUNC:    SEND/RECV
VADA:    Address of the IPv6 Connection Information Record initialized
         by your program
LENA:    Length of IPv6 Connection Information Record
VADB:    Address of the IPv6 Connection Information Record to be filled
         in with TCPIP reply
LENB:    Length of IPv6 Connection Information Record
CONN:    UNSPECIFIEDconnection
CALLCODE: V6OPENtcp

```

If the open attempt cannot be initiated, the TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header, contains a return code in the RETCODE field. The return code can be any of those listed for the TcpOpen Pascal procedure.

If the V6OPENtcp call was rejected because not enough TCPIP resources were available, a ZEROresources code is returned. When the TCPIP resources are available, a notice of RESOURCESavailable is sent to your program.

If the open attempt is not immediately rejected, the TCPIP virtual machine uses 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 is set as described in the previous paragraph.

If the open was successfully initiated, the TCPIP virtual machine responds using the VMCF REPLY function to send back the updated IPv6 Connection Information Record. The Connection field of the Connection Information Record contains the connection number of the new connection. The RETCODE field in the VMCF interrupt header indicates 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. For more information about NotificationInfoType, see the section on the Pascal under “Notification Record” on page 45 and see also “CALLCODE Notifications” on page 134.

V6STATUStcp

The V6STATUStcp call obtains an IPv6 Connection Information Record giving the current status of a TCP connection. For the assembler format of the IPv6 Connection Information Record, see Figure 28 on page 126. For more information about the connection status call, see the Pascal procedure, “Tcp6Status” on page 81.

```

FUNC:    SEND/RECV
VADA:    0
LENA:    1
VADB:    Address of the IPv6 Connection Information Record to fill in
LENB:    Length of IPv6 Connection Information Record to fill in
CONN:    Connection number from open
CALLCODE: V6STATUStcp

```

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, in the RETCODE field of the VMCF interrupt header, returns OK. Otherwise, the return code is one of those listed for the Tcp6Status Pascal procedure.

UDP CALLCODE Requests

The following sections describe the VMCF interrupt headers, which are stored in your virtual machine, for CALLCODE calls used to make UDP requests.

CLOSEudp

The CLOSEudp call closes a UDP port. For more information about the CLOSEudp call, see the Pascal procedure, “UdpClose” on page 103.

```

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 the value of 32 768 to the value in the CONN field, using unsigned arithmetic, to compute the port number.

NRECEIVEudp

The NRECEIVEudp call tells 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 TCPIP has accepted your request, your program receives a UDPdatagramDELIVERED notification when a datagram arrives. For more information about receiving UDP datagrams, see the Pascal procedure, [“UdpNReceive” on page 104](#).

```

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 contains one of the return codes listed for the UdpNReceive Pascal procedure.

OPENudp

The OPENudp call opens a UDP port. For more information about the UDP open function, see the Pascal procedure, [“UdpOpen” on page 104](#).

```

FUNC:    SEND
VADA:    0
LENA:    1
VADB:    Local port number or UNSPECIFIEDport
LENB:    Local address
CONN:    Connection number: An arbitrary number, which your program
        uses in subsequent actions involving this port.
CALLCODE: OPENudp

```

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 UdpOpen Pascal procedure. If the OPENudp call was rejected, because not enough TCPIP resources were available, a UDPzeroRESOURCES code is returned. When the TCPIP resources are available, a notice of UDPresourcesAVAILABLE is sent to your program.

SENDudp

The SENDudp call sends a UDP datagram. For more information about the UDP send function, see the Pascal procedure, [“UdpSend” on page 106](#).

```

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 contains a return code of 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. When the buffer space is not available to process the data, an error is returned. The notification message of UDPdatagramSPACEavailable is sent to your program when the buffer space is available to process data.

V6OPENudp

The V6OPENudp call opens a UDP port. For more information about the UDP open function, see the Pascal procedure, “Udp6Open” on page 102.

```

FUNC:      SEND/RECV
VADA:      Address of the local socket
LENA:      Length of the local socket
VADB:      0
LENB:      0
CONN:      Connection number: An arbitrary number, which your program
           uses in subsequent actions involving this port.
CALLCODE:  V6OPENudp

```

If the open attempt cannot be initiated, the TCPIP virtual machine responds using the VMCF REJECT function. The VMCF interrupt header, contains a return code in the RETCODE field. The return code can be any of those listed for the Udp6Open Pascal procedure.

If the V6OPENudp call was rejected, because not enough TCPIP resources were available, a UDPzeroRESOURCES code is returned. When the TCPIP resources are available, a notice of UDPresourcesAVAILABLE is sent to your program.

If the open attempt is not immediately rejected, the TCPIP virtual machine uses the VMCF RECEIVE function to receive the local socket information. If the connection then cannot be initiated, TCPIP responds using the VMCF REJECT function. The RETCODE field in the VMCF interrupt header is set as described in the previous paragraph.

V6SENDudp

The V6SENDudp call sends a UDP datagram. Your program sends an IPv6 Datagram Information Record to TCPIP. Figure 30 on page 129 gives the pascal format of the record. It includes foreign socket, datagram data, and length of datagram data. For more information about the UDP send function, see the Pascal procedure, “Udp6Send” on page 102.

```

UdpSendPacket6InfoType = record
  Socket: Socket6Type;
  BufferLen: integer;
  Packet: packed array (.1..65535.) of char ;
end;

```

Figure 30. Pascal Format of the IPv6 Datagram Information Record for VM

```

FUNC:    SEND/RECEIVE
VADA:    Address of IPv6 Datagram Information Record initialized
          by your program
LENA:    Length of IPv6 Datagram Information Record
VADB:    0
LENB:    0
CONN:    Connection number
CALLCODE: V6SENDudp

```

TCPIP virtual machine will first respond with the VMCF RECEIVE function to get the IPv6 Datagram Information Record. Then if TCPIP can send the datagram, it responds with the VMCF REJECT function and the RETCODE field in the VMCF interrupt header contains a return code of 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 Udp6Send Pascal procedure. When the buffer space is not available to process the data, an error is returned. The notification message of UDPdatagramSPACEavailable is sent to your program when the buffer space is available to process data.

IP CALLCODE Requests

The following sections describe the VMCF interrupt headers, which are stored in your virtual machine, for CALLCODE calls used to make IP requests.

CLOSErawip

The CLOSErawip call tells TCPIP that your program is ready to cease sending and receiving packets of a specified IP protocol. For more information, see the Pascal procedure, [“RawIpClose”](#) on page 70.

```

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 contains one of the return codes listed for the RawIpClose Pascal procedure.

OPENrawip

The OPENrawip call tells TCPIP that your program is ready to send and receive packets of a specified IP protocol. For more information, see the Pascal procedure, [“RawIpOpen”](#) on page 71.

```

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 contains one of the return codes listed for the RawIpOpen Pascal procedure.

RECEIVERawip

The RECEIVERawip call tells 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. For information about the RAWIPpacketsDELIVERED notification record, see the Pascal procedure,

“RawIpReceive” on page 72, and the section on the Pascal NotificationInfoType under “Notification Record” on page 45.

```
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 contains one of the return codes listed for the RawIpReceive Pascal procedure.

SENDrawip

The SENDrawip call sends raw IP packets of a given protocol number. For more information, see the Pascal procedure, “RawIpSend” on page 72.

```
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 contains one of the return codes listed for the RawIpSend Pascal procedure. If TCPIP virtual machine is out of buffers, the data is rejected and a return code of NObufferSPACE is returned. When buffer space is available, the notification of RAWIPspaceAVAILABLE is sent to your program.

CALLCODE System Queries

The following sections describe the VMCF interrupt headers, which are stored in your virtual machine, for CALLCODE calls used to make system queries.

IShostLOCAL

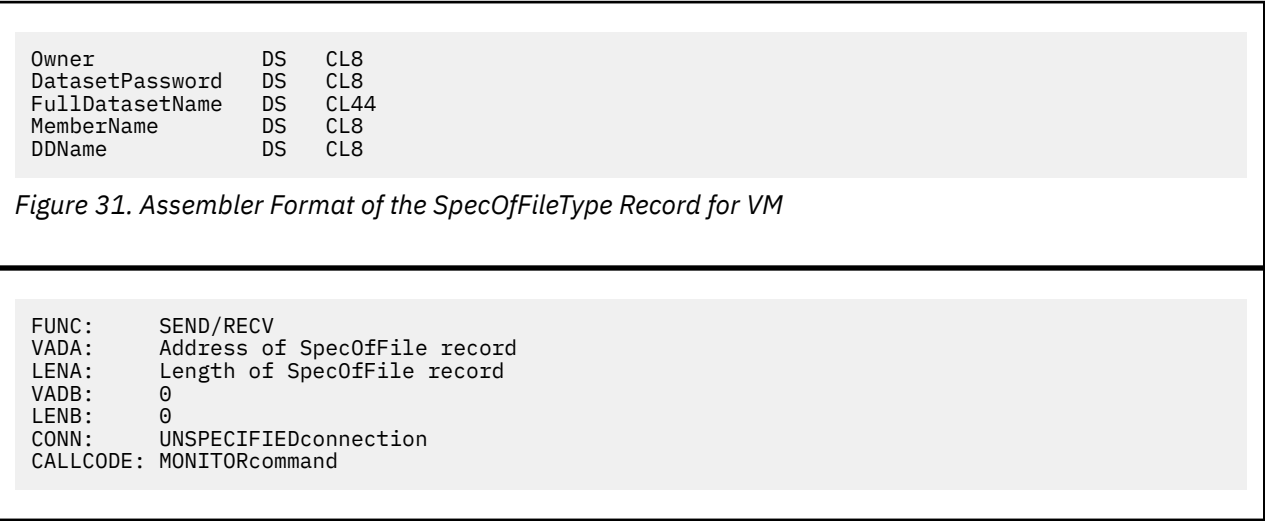
The IShostLOCAL call determines whether a given internet address is one of your host’s local addresses. For more information about this procedure, see the Pascal procedure “IsLocalAddress” on page 65.

```
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.

MONITORcommand

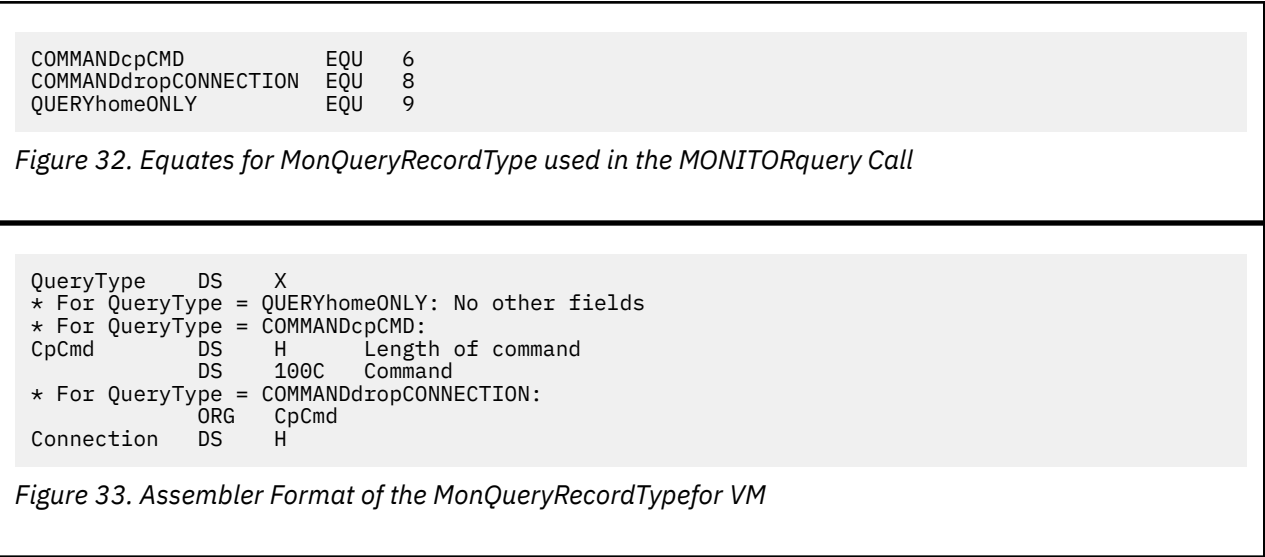
The MONITORcommand call instructs TCPIP to obey a file of commands. For more information, see the Pascal procedure, “MonCommand” on page 66, and for more information about the OBEYFILE command, which uses the MonCommand procedure, see *z/VM: TCP/IP Planning and Customization*.



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.

MONITORquery

The MONITORquery call obtains status information from the TCPIP virtual machine or to request that it performs certain functions. For more information, see the Pascal procedure, “MonQuery” on page 67. Assembler formats of constants and records used with MONITORquery are:



The Pascal type HomeOnlyListType is an array of 64 InternetAddressType elements found in the COMMMAC MACLIB file. The size of 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. For information about the return codes and the data returned (if any), see the Pascal procedure, [“MonQuery”](#) on page 67.

PINGreq

The PINGreq call sends an ICMP echo request (PING request) to a specified host and wait a specified time for a response. For more information, see the Pascal procedure [“PingRequest”](#) on page 69.

```

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 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.

TLSQuery

The TLSQuery call sends a query to determine if the SSL server is available, and if so, if the label specified is known. For more information, see the Pascal Procedure, [“QueryTLS”](#) on page 70.

```

QueryLabel  DS    CL8
QueryKeyring DS    CL50

```

Figure 34. Assembler format of the QueryRequest record for VM

```

FUNC:      SEND/RECV
VADA:      Address of QueryRequest record
LENA:      Length of QueryRequest record
VADB:      0
LENB:      0
CIBB:      UNSPECIFIED
CALLCODE:  TLSQuery

```

TCPIP responds with the VMCF REPLY function. The RETCODE field of the VMCF interrupt buffer contains the return code from the preprocessing of this command. The return code is one of those listed for the QueryTLS procedure. The results of the actual query will be returned with the QUERYtlsCOMPLETE notification.

CALLCODE Notifications

The following sections describe the VMCF interrupt headers that are stored in your virtual machine for the various types of notifications. The action that your program should take is also indicated.

For more information about the various notification types, see the Pascal NotificationInfoType record under “Notification Record” on page 45.

The VMCF transaction for a notification must be completed before TCPIP sends your program another notification. You must ensure that your program takes the VMCF actions in the following sections, or TCPIP cannot communicate further with your program.

ACTIVEprobe

This interrupt header notifies you that the TCPIP virtual machine is monitoring your machine so it can determine if it is still responsive.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
CALLCODE:  ACTIVEprobe
RETCODE:   OK

Your program should issue the VMCF REJECT function, with the VMCF
parameter list copied from the interrupt header and with the
following fields changed:

V1:        0
V2:        0
FUNC:      REJECT

The response to this message must be made within one minute after the
associated interrupt is received.
```

BUFFERspaceAVAILABLE

This interrupt header notifies you that there is space available to send data on this connection. The space is currently set to 8192 bytes of buffer space. The notification is sent after making a SENDtcp call and receiving an unsuccessful return code of NObufferSPACE in the RETCODE field.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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
```

CERTdataCOMPLETE

This interrupt header notifies you that certificate data is available from a TcpSCertData call.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
LENB:      Length of the data being delivered
CONN:      Connection Number
CALLCODE:  CERTdataCOMPLETE
RETCODE:   OK

```

Your program should issue the VMCF RECEIVE function, with the VMCF parameter 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 LENB.

```

CLEARtextRESUMED

The interrupt header notifies you that the Close_Notify on the connection is complete and data transmission is now in the clear.

```

FUNC:      SEND
JOBNAME:   Name of TCPIP virtual machine
VADB:      0
CONN:      Connection number
CALLCODE:  CLEARtextRESUMED
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

```

CONNECTIONstateCHANGED

This interrupt header notifies you that the state of the connection between the TCPIP virtual machine and your program has changed.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
VADB:      New connection state
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

```

DATAdelivered

This interrupt header notifies you that the TCPIP virtual machine data was delivered to your program, after issuing a RECEIVEtcp or FRECEIVEtcp call.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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 is no
            larger than the buffer length you specified using the
            RECEIVEtcp function.

```

DUMMYprobe

This interrupt header notifies you that the TCPIP virtual machine is monitoring your machine so it can determine if it logs off or resets unexpectedly.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
CALLCODE:  DUMMYprobe
RETCODE:   OK

Your program should leave this message pending.

```

PINGresponse

This interrupt header notifies you that your ping request from the PINGreq call has been received or that the time-out limit or your request has been reached.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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

```

QUERYtlsCOMPLETE

The interrupt header notifies you that the Query is complete and returns the status of the query.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
VADB:      0
CONN:      UNSPECIFIEDconnection.
CALLCODE:  QUERYtlisCOMPLETE
RETCODE:   return code. If 0, indicates that the label is ok.

```

Your program should issue the VMCF REJECT function, with the VMCF parm list copied from the interrupt header, with the following fields changed:

```

V1:        0
V2:        0
FUNC:      REJECT

```

RAWIPpacketsDELIVERED

This interrupt header notifies you that your buffer has received the raw IP packets.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
ANINTEGR:  Total length of datagram being delivered (including IP header)
LENA:      Length of data (including IP header) that TCPIP
           delivers 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.

```

RAWIPspaceAVAILABLE

This interrupt header notifies you that buffer space is available to process the data. This notification is sent after the SENDrawip call was rejected by TCPIP virtual machine because of insufficient buffer space.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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

```

READYforHANDSHAKE

The interrupt header notifies you whether or not this connection is ready for the initiation of an SSL handshake.

```
FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
VADB:      0
CONN:      Connection Number
CALLCODE:  REAQDYforHANDSHAKE
RETCODE:   Return code. If 0, indicates that a handshake can be
           initiated on this connection
```

Your program should issue the VMCF REJECT function, with the VMCF parm list copied from the interrupt header, with the following fields changed:

```
V1:        0
V2:        0
FUNC:      REJECT
```

RESOURCESavailable

This interrupt header notifies you that the resources needed to initiate a TCP connection are now available. This notification is sent only if a previous `OPENtcp` call received a `ZEROresources` return code.

```
FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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
```

SECUREhandshakeCOMPLETE

The interrupt header notifies you that the SSL handshake on this connection is complete and indicates the status of that handshake.

```
FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
VADB:      SecureHSCompleteDetail
CONN:      Connection Number
CALLCODE:  SECUREhandshakeCOMPLETE
RETCODE:   OK
```

Refer to [“Notification Record” on page 45](#) for details of `SecureHSCompleteDetail`.

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
```

UDPdatagramDELIVERED

This interrupt header notifies you that the UDP datagram has been delivered to your program after issuing a `NRECEIVEudp` call to the TCPIP virtual machine.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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.

```

UDPdatagramSPACEavailable

This interrupt header notifies you that buffer space is available to process the data, after an error occurred performing a SENDudp call.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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

```

UDPresourcesAVAILABLE

This interrupt header notifies you that the resources needed to initiate a UDP connection are now available. This notification is sent only if a previous OPENudp call received a UDPzeroRESOURCES return code.

```

FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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

```

URGENTpending

This interrupt header notifies you that there is queued incoming data on a TCP connection not yet received by your program.

```
FUNC:      SEND
JOBNAME:   Name of the TCPIP virtual machine
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
```

Chapter 4. Inter-User Communication Vehicle Sockets

The Inter-User Communication Vehicle (IUCV) socket API is an assembler language application programming interface that can be used with TCP/IP. While not every C socket library function is provided, all of the basic operations necessary to communicate with other socket programs are present.

Prerequisite Knowledge

This chapter assumes you have a working knowledge of IUCV, as documented in *z/VM: CP Programming Services*.

You must also know how and when to use the CMS CMSIUCV macro or the GCS IUCVCOM macro, depending on the execution environment, as documented in *z/VM: CMS Application Development Guide for Assembler* or *z/VM: Group Control System*, respectively.

You should also have a working knowledge of TCP/IP sockets.

Available Functions

Only these functions are available when you use the IUCV socket interface:

ACCEPT	READ
BIND	READV
CLOSE	RECV
CONNECT	RECVFROM
FCNTL	RECVMSG
GETCLIENTID	SELECT
GETHOSTID	SELECTEX
GETHOSTNAME	SEND
GETPEERNAME	SENDMSG
GETSOCKNAME	SENDTO
GETSOCKOPT	SETSOCKOPT
GIVESOCKET	SHUTDOWN
IOCTL	SOCKET (AF_INET sockets only)
LISTEN	TAKESOCKET
MAXDESC	WRITE
	WRITEV

Socket Programming with IUCV

TCP/IP sockets are manipulated by using the following assembler macros:

Macro	Library	Description
IUCV	HCPGPI	Provides the mechanisms for setting values in the IUCV input parameter list and for executing the IUCV instruction

Macro	Library	Description
IPARML	HCPGPI	Mapping macro for the IUCV parameter list and the external interrupt buffer.
HNDIUCV	DMSGPI	Informs CMS that your program wishes to handle IUCV or APPC/VM interrupts. Only those interrupts occurring on IUCV paths that your application created will be routed to your program.
CMSIUCV	DMSGPI	Used to perform IUCV CONNECT and SEVER functions. It enables multiple IUCV or APPC/VM applications to run at the same without interference.
IUCVINI	GCTGPI	Similar to HNDIUCV, but for the GCS execution environment.
IUCVCOM	GCTGPI	Similar to CMSIUCV, but for the GCS execution environment. In addition to providing multiple application support, it provides a way for GCS programs running in problem state to use IUCV services.

A typical socket application uses only four IUCV operations: CONNECT, SEND (with reply), PURGE, and SEVER. CONNECT establishes the IUCV connection with the TCP/IP virtual machine, SEND performs initialization and socket operations, PURGE cancels an outstanding socket operation, and SEVER deletes the IUCV connection.

If an IUCV operation completes with condition code 0, the requested operation was successfully started. An IUCV interrupt will be received when the operation completes. When your interrupt routine receives control, it receives a pointer to the *external interrupt buffer* which contains information about the IUCV function that completed. The IPTYPE field of the external interrupt buffer (mapped by IPARML) identifies the interrupt:

IPTYPE	Interrupt Name	Description
X'02'	Connection Complete	Acknowledgement that TCP/IP has accepted your request to establish an IUCV connection (IUCV CONNECT)
X'03'	Connection Severed	Your IUCV connection has been deleted by TCP/IP
X'07'	Message complete	The requested socket function has completed

Note: IPTYPE is byte 3 of the external interrupt buffer.

While there are other types of IUCV interrupts, they are not normally seen on TCP/IP IUCV socket paths. [z/VM: CP Programming Services](#) has a complete description of each interrupt type.

If an IUCV operation completes with condition code 1, the requested function could not be performed. The exact cause of the error is stored in byte 3 of the IUCV parameter list (IPRCODE). See the description of each IUCV function in [z/VM: CP Programming Services](#) for the possible return codes.

Note: CMSIUCV and IUCVCOM use return codes in general register 15 to indicate the success or failure of the operation. Refer to [z/VM: CMS Application Development Guide for Assembler](#) or [z/VM: Group Control System](#) for details on these system services.

If an IUCV PURGE operation completes with condition code 2, it means that TCP/IP has already finished processing the socket request.

Preparing to use the IUCV Socket API

Before the socket functions can be used, an IUCV socket API environment must be established. This is done in two steps:

1. Establish an IUCV connection to the TCP/IP service virtual machine.
2. Send an initialization message to TCP/IP, identifying your application and defining how the IUCV connection will be used.

Establishing an IUCV connection to TCP/IP

To create an IUCV connection to the TCP/IP service virtual machine, issue IUCV CONNECT with the following parameters:

Keyword
Value

USERID
The user ID of the TCP/IP virtual machine.

PRTY
NO

PRMDATA
YES

QUIESCE
NO

MSGLIM
If this IUCV connection may have more than one outstanding socket function on it at the same time, set MSGLIM to the maximum number of socket calls that may be outstanding simultaneously on this path. Otherwise, set it to zero.

USERDTA
Binary zeros

CONTROL
NO

If IUCV CONNECT returns condition code 0, you subsequently receive either a Connection Complete external interrupt or a Connection Severed external interrupt. If you receive a Connection Severed interrupt now or later, see [“Severing the IUCV Connection”](#) on page 144 for more information.

To ensure that your program does not interfere with other IUCV or APPC/VM applications, your program should use the HNDIUCV and CMSIUCV macros in CMS, or the IUCVINI and IUCVCOM macros in GCS.

Initializing the IUCV Connection

If you receive a Connection Complete interrupt in response to IUCV CONNECT, then TCP/IP has accepted the connection request.

Your program responds by sending an *initialization message* using IUCV SEND to TCP/IP, identifying your application and the way that it will use the IUCV socket interface.

When the IUCV SEND completes, then, if the IPAUDIT field shows no error, the reply buffer has been filled. The *maxsock* field indicates that maximum number of sockets you can open on this IUCV path at the same time.

Your program can now issue any supported socket call. See [“Issuing Socket Calls”](#) on page 145.

The initialization message is sent using an IUCV SEND with the following parameters:

Keyword
Value

TRGCLS
0

DATA
BUFFER

BUFLen

20

TYPE

2WAY

ANSLen

8

PRTY

NO

BUFFER

Points to a buffer in the following format:

Offset	Name	Length	Comments
0		8	Constant 'IUCVAPI '. The trailing blank is required.
8		2	Halfword integer. Maximum number of sockets that can be established on this IUCV connection. minimum: 50, Default: 50.
10	<i>apitype</i>	2	X'0002'. Provided for compatibility with prior implementations of TCP/IP. Use X'0003' instead. X'0003'. Any number of socket requests may be outstanding on this IUCV connection at the same time. For AF_INET sockets only. X'0004'. Any number of socket requests may be outstanding on this IUCV connection at the same time. For AF_INET6 sockets only. For more information, see “Overlapping Socket Requests” on page 146.
12	<i>subtaskname</i>	8	Eight printable characters. The combination of your user ID and <i>subtaskname</i> uniquely identifies the TCP/IP client using this path. This value is displayed by the NETSTAT CLIENT command.

Keyword

Value

ANSBUF

Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0		4	Reserved
4	<i>maxsock</i>	4	The maximum socket number that your application can use on this path. The minimum socket number is always 0. Your application chooses a socket number for the accept, socket, and takesocket calls.

Note: A single virtual machine can establish more than one IUCV path to TCP/IP, but a different *subtaskname* must be specified on each IUCV path. If the same *subtaskname* is specified for more than one IUCV path, TCP/IP severs the existing path with that *subtaskname*.

Severing the IUCV Connection

An IUCV connection to TCP/IP can be severed (deleted) by your application or by TCP/IP at any time.

Sever by the Application

Your application can sever a socket API IUCV path at any time by calling IUCV SEVER with USERDTA specified as 16 bytes of binary zeros. TCP/IP cleans up all sockets associated with the IUCV path.

Clean-Up of Stream Sockets

The TCP connection corresponding to each stream socket associated with the IUCV path is reset. In the case of a listening socket, all connections in the process of opening, or already open and in the accept queue, are reset.

If your program closed a stream socket earlier, the corresponding TCP connection might still be in the process of closing. Such connections, which are no longer associated with any socket, are **not** reset when your program severs the IUCV path.

Sever by TCP/IP

TCP/IP severs a socket API IUCV path only in case of shutdown or an unexpected error. The 16-byte IPUSER field in the SEVER external interrupt indicates the reason for the sever. The reason is coded in EBCDIC. The following are possible reason codes and explanations:

Reason Code

Explanation

BAD API TYPE

The *apitype* field in your initialization message contained an incorrect value.

BAD INIT MSG LEN

Your program sent an initialization message that was not of the expected length.

BAD PATH ID

An attempt was made to exceed the maximum number of IUCV connections support by the target TCPIP virtual machine.

IPV6 NOT ENABLED

Your program sent an initialization message with *apitype* 4, but TCP/IP is not enabled for IPv6 communications.

IUCVCHECKRC

IUCV error detected. This code is used only before or during processing of the initialization message.

NO CCB!!!!

A software error occurred in TCP/IP. Contact your system support personnel or the IBM Support Center.

NO MORE CCBS

Your IUCV path cannot be accepted because there are no more client control blocks available in the TCPIP virtual machine.

NULL SAVED NAME

A software error occurred in TCP/IP. This code is used only before or during processing of the initialization message.

REQUIREDCONSTANT

The first 8 bytes of your initialization message were not "IUCVAPI ".

RESTRICTED

Your virtual machine is not permitted to use TCP/IP.

SHUTTINGDOWN

TCP/IP service is being shut down. This code is used only in response to the Pending Connection interrupt.

Issuing Socket Calls

The following section describes how to issue an IUCV socket call.

All socket calls are invoked by issuing an IUCV SEND with the following parameters:

Keyword

Value

TRGCLS

The high-order halfword specifies the socket call. For most calls, the low-order halfword specifies the socket descriptor.

DATA

BUFFER or PRMSG, depending on call

BUFLIST

If DATA=BUFFER, then either YES or NO as desired. If DATA=PRMSG, not applicable.

BUFFER

If DATA=BUFFER, points to the buffer (or buffer list) in the format required by the call. If DATA=PRMSG, not applicable.

BUFLN

If DATA=BUFFER, length of buffer. If DATA=PRMSG, not applicable.

PRMSG

If DATA=PRMSG, data as required by the call. DATA=PRMSG is not allowed when ANSLIST=YES. If DATA=BUFFER, not applicable.

TYPE

2WAY

ANSLIST

Either YES or NO as desired. DATA=PRMSG is not allowed when ANSLIST=YES.

ANSBUF

Points to a buffer to contain the reply from TCP/IP.

ANSLEN

Length of the reply buffer

PRTY

NO

SYNC

YES or NO as desired. Applications that need to serve multiple clients at the same time should specify SYNC=NO. SYNC=YES will block the entire virtual machine from execution until the function is complete.

Overlapping Socket Requests

Your program may have more than one socket call outstanding on the same IUCV path. There are some restrictions on the types of calls that are queued simultaneously for the same socket descriptor.

The following list describes the restrictions for each type of socket call:

- Multiple read-type calls (READ, READV, RECV, RECVFROM, RECVMSG) and multiple write-type calls (WRITE, WRITEV, SEND, SENDTO, SENDMSG), for the same socket, can be queued simultaneously. The read-type calls are satisfied in order, independently of the write-type calls. Similarly, the write-type calls are satisfied in order, independently of the read-type calls.
- Multiple ACCEPT calls, for the same listening stream socket, can be queued simultaneously. They are satisfied in order.
- Multiple SELECT calls, referring to any combination of sockets, can be queued simultaneously on an IUCV path. TCP/IP checks all queued SELECT calls when an event occurs and responds to any that are satisfied.
- Calls other than the read-type, write-type, ACCEPT, and SELECT calls, cannot be queued simultaneously for the same socket. For example, your program must wait for TCP/IP's response to a write-type call before issuing a CLOSE call for the same socket.

TCP/IP Response to an IUCV Request

TCP/IP's response to your socket call is signaled by the Message Complete external interrupt. When the Message Complete external interrupt is received, if the IPAUDIT field shows no error, your program's reply buffer has been filled. The IPBFLN2F field indicates how many bytes of the reply buffer were not used.

If the IPADRJCT bit of the IPAUDIT field is set, then TCP/IP was unable to use IUCV REPLY to respond, and instead used IUCV REJECT. Your program issues the special LASTERRNO function (see [“LASTERRNO” on page 186](#)) to retrieve the return code and *errno* for the rejected call. TCP/IP's use of IUCV REJECT does not necessarily mean the socket call failed.

The following *errno* values (shown in decimal) are seen only by a program using the IUCV socket interface.

Errno Value	Description
-------------	-------------

1000	An unrecognized socket call constant was found in the high-order halfword of the Target Message Class.
-------------	--

1001	A request or reply length field is incorrect
-------------	--

1002	The socket number assigned by your program for ACCEPT, SOCKET, or TAKESOCKET is out of range.
-------------	---

1003	The socket number assigned by your program for ACCEPT, SOCKET, or TAKESOCKET is already in use.
-------------	---

1008	This request conflicts with a request already queued on the same socket (see “Overlapping Socket Requests” on page 146).
-------------	---

1009	The request was canceled by the CANCEL call (see “CANCEL and CANCEL2” on page 161).
-------------	--

1011	The user ID issuing this request does not match the user ID specified on the SSLServerID statement in the configuration file.
-------------	---

1025	The local or partner certificate was not requested on the SSL handshake, therefore, no fields can be obtained from that certificate.
-------------	--

Encrypting Data on an IUCV Socket

Once a connection has been established, `ioctl()` commands can be used to direct the data through the TLS/SSL server so that it is encrypted when sent and decrypted when received. Refer to the section, [“Secure Connection Considerations” on page 21](#) for details.

Cancelling a Socket Request

Your socket program can use the CANCEL call to cancel a previously issued socket call. Read-type calls, write-type calls, ACCEPT calls, and SELECT calls can be canceled using this function. See [“CANCEL and CANCEL2” on page 161](#) for more information about using the CANCEL call.

IUCV PURGE can also be used to cancel a call, but it does not stop TCP/IP processing the same way as the CANCEL call.

Each IUCV SEND operation that completes with condition code zero is assigned a unique message identification number. This number is placed in the IUCV parameter list. To use the CANCEL or IUCV PURGE functions, your program must keep track of the message ID numbers assigned to each socket request.

IUCV Socket Call Syntax

Each of the IUCV Socket calls described includes the C language syntax for the call. IUCV SEND parameters and buffer contents are described using variable names from the C syntax. Call types are in capital letters. For example, the accept call is ACCEPT.

The parameter lists for some C language socket calls include a pointer to a data structure defined by a C *structure*. When using the IUCV socket interface, the contents of the data structure are passed in the send buffer, the reply buffer, or both. [Table 21 on page 148](#) shows the C structures used, and the corresponding assembler language syntax.

Table 21. C Structures in Assembler Language Format

C Structure	Assembler Language Equivalent
<pre>structure CertDataCompleteDetailType { int CDRetCode; int CDRetCnt; short CDDDataLen; short CDRes; char CDDData[CDDDataLen] };</pre>	<pre>CDRetCode DS F CDRetCnt DS F CDDDataLen DS H CDRes DS H CDDData DS x</pre>

where:

CDRetCode

Indicates the return code from the certificate request. Possible values:

- 0** - No errors
- 4021** - The partner value is not valid.
- 4023** - The partner certificate is not available.
- 4024** - The certificate does not contain any values.
- 4025** - The buffer length passed is too large.
- 4026** - The returned data will not fit in the provided buffer. Partial data is returned.
- 4027** - The passed buffer pointer is null.
- 4028** - The number of certificate fields requested (CDReqNum) is 0.
- 4029** - The number of certificate fields requested (CDReqNum) is greater than 64.
- 4030** - The requested certificate field is not found.
- 4031** - The requested certificate field is not valid.
- 4032** - Both of these errors exist in the return data: A requested certificate field is not found *and* a requested certificate field is not valid.

CDRetCnt

Is the number of certificate fields returned in CDDData.

CDDDataLen

Is the length of the returned certificate data.

CDRes

Is reserved (will be 0).

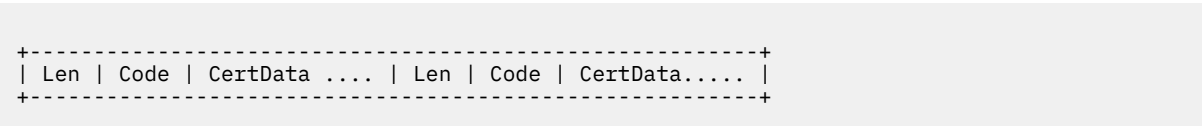
Table 21. C Structures in Assembler Language Format (continued)

C Structure

Assembler Language Equivalent

CDData

Is requested data from the certificate. The format is as follows:



where:

Len

Is a halfword field that contains the total length of the item (Len+Code+CertData). The total of all of the Len fields in the buffer is returned in CDDataLen.

Code

Is a halfword that contains the certificate field code (600-677).

CertData

Is the certificate data that corresponds to the requested code. Note that a single field could appear multiple times in the returned buffer if more than one "answer" is valid.

x

Is the value that is specified for CDDataLen.

Usage Notes:

- Certificate fields will be placed in the CDData buffer in the order in which they appear in the CertReqCodes input structure.
- The CDData buffer will contain as many certificate fields as will fit completely. If a requested certificate field does not fit in the buffer, it will not be returned and subsequent fields in the CertReqCodes input structure will also fail. CDRetCode will indicate that not all of the data will fit in CDData. CDRetCnt will reflect the number of completed requests.
- If the requested field cannot be found in the certificate, CDData will contain a Len of 4 along with the requested Code. No data will be returned. CDRetCode will be updated to indicate that one or more fields are not present in the certificate.

<pre>structure CertReqDetailType { char CertReqNum; char CertReqSide; short CertReqRes1; int CertReqRes2; int CertReqLen; int CertReqPtr; short CertReqCodes[64]; };</pre>	<pre>CertReqNum DS X CertReqSide DS X CertReqRes1 DS H CertReqRes2 DS F CertReqLen DS F CertReqPtr DS F CertReqCodes DS 64H</pre>
--	---

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
where:	
CertReqNum	Is the number of certificate fields requested.
CertReqSide	Is 0 for local or 1 for partner.
CertReqRes1	Is reserved for future use and must be 0.
CertReqRes2	Is reserved for future use and must be 0.
CertReqLen	Is the length of the buffer (not to exceed 16K + 12).
CertReqPtr	Is a pointer to the returned CertDataCompleteDetailType structure for C programs. In the Assembler case, this field is ignored.

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
-------------	-------------------------------

CertReqCodes

Is a list of requested certificate fields. The valid codes are:

- 600 - CERT_BODY_DER
 - 601 - CERT_BODY_BASE64
 - 602 - CERT_SERIAL_NUMBER
 - 610 - CERT_COMMON_NAME
 - 611 - CERT_LOCALITY
 - 612 - CERT_STATE_OR_PROVINCE
 - 613 - CERT_COUNTRY
 - 614 - CERT_ORG
 - 615 - CERT_ORG_UNIT
 - 616 - CERT_DN_PRINTABLE
 - 617 - CERT_DN_DER
 - 618 - CERT_POSTAL_CODE
 - 619 - CERT_EMAIL
 - 620 - CERT_DOMAIN_COMPONENT
 - 621 - CERT_SURNAME
 - 622 - CERT_STREET
 - 623 - CERT_TITLE
 - 650 - CERT_ISSUER_COMMON_NAME
 - 651 - CERT_ISSUER_LOCALITY
 - 652 - CERT_ISSUER_STATE_OR_PROVINCE
 - 653 - CERT_ISSUER_COUNTRY
 - 654 - CERT_ISSUER_ORG
 - 655 - CERT_ISSUER_ORG_UNIT
 - 656 - CERT_ISSUER_DN_PRINTABLE
 - 657 - CERT_ISSUER_DN_DER
 - 658 - CERT_ISSUER_POSTAL_CODE
 - 659 - CERT_ISSUER_EMAIL
 - 660 - CERT_ISSUER_DOMAIN_COMPONENT
 - 661 - CERT_ISSUER_SURNAME
 - 662 - CERT_ISSUER_STREET
 - 663 - CERT_ISSUER_TITLE
 - 664 - CERT_NAME
 - 665 - CERT_GIVENNAME
 - 666 - CERT_INITIALS
 - 667 - CERT_GENERATIONQUALIFIER
 - 668 - CERT_DNQUALIFIER
 - 669 - CERT_MAIL
 - 670 - CERT_SERIALNUMBER
 - 671 - CERT_ISSUER_NAME
 - 672 - CERT_ISSUER_GIVENNAME
 - 673 - CERT_ISSUER_INITIALS
 - 674 - CERT_ISSUER_GENERATIONQUALIFIER
 - 675 - CERT_ISSUER_DNQUALIFIER
 - 676 - CERT_ISSUER_MAIL
 - 677 - CERT_ISSUER_SERIALNUMBER
-

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
<pre> struct clientid { int domain; char name[8]; char subtaskname[8]; char reserved[20]; }; </pre>	<pre> DOMAIN DS F NAME DS CL8 SUBTASK DS CL8 RESERVED DC XL20'00' </pre>
<pre> struct CloseReq [short CloseLen; char CloseBuff[255];]; </pre>	<pre> CloseLen DS H CloseBuff DS CL255 </pre>
<p>where:</p> <p>CloseLen Is the length of the message in the CloseBuff buffer.</p> <p>CloseBuff Specifies a message to be sent to the partner over the encrypted connection before the SSL tunnel is closed. The message indicates that the partner's very next step must be to issue the SioCSecClose ioctl call to close the partner's side of the SSL tunnel and return to unencrypted communication.</p>	
<pre> struct ifconf { int ifc_len; union { caddr_t ifcu_buf; struct ifreq *ifcu_req; } ifc_ifcu; }; </pre>	<pre> IFCLEN DS F IGNORED DS F </pre>
<pre> struct ifreq { #define IFNAMSIZ 16 char ifr_name[IFNAMSIZ]; union { struct sockaddr ifru_addr; struct sockaddr ifru_dstaddr; struct sockaddr ifru_broadaddr; short ifru_flags; int ifru_metric; caddr_t ifru_data; } ifr_ifru; }; </pre>	<pre> NAME DS CL16 ADDR.FAMILY DS H ADDR.PORT DS H ADDR.ADDR DS F ADDR.ZERO DC XL8'00' ORG ADDR.FAMILY DST.FAMILY DS H DST.PORT DS H DST.ADDR DS F DST.ZERO DC XL8'00' ORG ADDR.FAMILY BRD.FAMILY DS H BRD.PORT DS H BRD.ADDR DS F BRD.ZERO DC XL8'00' ORG ADDR.FAMILY FLAGS DS H ORG ADDR.FAMILY METRIC DS F </pre>
<pre> struct linger { int l_onoff; int l_linger; }; </pre>	<pre> ONOFF DS F LINGER DS F </pre>

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
<pre>struct QueryTls { char TLSLabel[8]; char TLSKeyring[50]; };</pre>	<pre>TLSLabel DS CL8 TLSKeyring DS CL50</pre>
<p>The QueryTLS call can determine whether the security server is available and if the security server recognizes the TLSLabel. The call can include the following parameters:</p> <p>TLSLabel If the optional TLSLabel is specified, the call determines whether the security server recognizes the TLSLabel.</p> <p>TLSKeyring Is not yet available. The value must be blank.</p>	
<pre>struct SecStatus { int SecLevel; char CipherClass; char CipherHash; char CipherAlgorithm; char CipherPKAlgorithm; int CipherKeyLength; };</pre>	<pre>SecLevel DS F CipherClass DS X CipherHash DS X CipherAlgorithm DS X CipherPKAlgorithm DS X CipherKeyLength DS F</pre>

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
-------------	-------------------------------

where:

SecLevel

- 0** - Not Secure
- 1** - Statically Secured
- 2** - Dynamically Secured

CipherClass

- 0** - NULLclass
- 1** - SSLV2
- 2** - SSLV3
- 3** - TLS
- 4** - TLS10
- 5** - TLS11
- 6** - TLS12

CipherHash

- 0** - SHA1
- 1** - MD5
- 2** - NULL
- 3** - SHA2
- 4** - SHA256
- 5** - SHA384

CipherAlgorithm

- 0** - NULL
- 2** - RC4
- 4** - DES3
- 7** - AES
- 8** - AESGCM
- 9** - AES128
- 10** - AES128GCM
- 11** - AES256
- 12** - AES256GCM

CipherPKAlgorithm

- 0** - NULL
 - 1** - RSA
 - 2** - DH_DSS
 - 3** - DH_RSA
 - 4** - DHE_DSS
 - 5** - DHE_RSA
 - 6** - ECDH_ECDSA
 - 7** - ECDHE_ECDSA
 - 8** - ECDH_RSA
 - 9** - ECDHE_RSA
-

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
<pre> struct SecureDetail { char TLSLabel[8]; int TLStimeout; char requestClientCert; char validatePeerCert; char cipher_request; char version; char keyring[50]; short buflen; char buffer[255]; struct SecureDetailExtension; }; struct SecureDetailExtension { int validationFlags int validationLen char validationBuffer[512]; }; </pre>	<pre> TLSLabel DS CL8 TLStimeout DS F requestClientCert DS X validatePeerCert DS X cipher_request DS X version DS X keyring DS CL50 buflen DS H buffer DS CL255 DS XL3 validationFlags DS F validationLen DS F validationBuffer DS CL512 </pre>

Table 21. C Structures in Assembler Language Format (continued)

C Structure

Assembler Language Equivalent

where:

TLSLabel

Is the label associated with the certificate in the certificate database.

TLSTimeout

Is not yet available. The value must be 0.

requestClientCert

See **validatePeerCert**.

validatePeerCert

The requestClientCert and validatePeerCert flags are used in combination to determine the level of client certificate checking that will be done during a secure server call. The level and the flag settings are as follows:

None

A client certificate will not be requested.

```
requestClientCert = 0
validatePeerCert  = 1
```

Preferred

A client certificate is requested. If a client certificate is not received, the connection will proceed without it. If a client certificate is received, it will be authenticated. If the client certificate is not valid, the failure will be logged in the SSL console log and the connection will continue as a secure connection protected by the server certificate.

```
requestClientCert = 1
validatePeerCert  = 1
```

Required

A client certificate will be authenticated. If a client certificate is not received, the connection will be terminated with a fatal TLS error. If the certificate fails authentication, the handshake will fail.

```
requestClientCert = 1
validatePeerCert  = 0
```

Note: For a secure client call, the server certificate is always validated. Set these flags to indicate a level of None.

cipher_request

Indicates whether SSL V2 will be used. Possible values:

- 0** - The default cipher suite values will be used.
- 1** - The client does not want to use SSL V2.

version

When set to 0, the SecDetailExt is not passed on the call.

When set to 1, the SecDetailExt is filled in and passed on the call to tell the SSL/TLS server to compare the passed-in host name, domain name, or IP address against the server certificate. A value of 1 is valid only when securing the client side of the connection.

keyring

Is not yet available. The value must be blank.

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
buffer	Contains the string that the SSL server will send out on the connection before waiting for the handshake. After this command is sent, the initiation of the handshake is expected on the connection. If an empty buffer is sent, a READYforHANDSHAKE notification will be sent to indicate that this side of the connection is waiting for the handshake.
ValidationFlags	Possible values: 0 indicates not required. If the validation text does not match what is in the server certificate, the mismatch will be logged and the handshake will continue. 1 indicates required. At least one of the specified validation items must match what is in the server certificate. If there are no matching items, the handshake will fail.
ValidationLen	Is the total length of the validation buffer.
ValidationBuffer	Contains multiple items to validate against the certificate. Each item is in the following format:
	<pre> +-----+ Len Type Text +-----+</pre>
	The total length of all items (Len+Type+Text) must not exceed 512 bytes.
Len	A halfword field that contains the total length of the item (Len+Type+Text). The total of all of the Len fields in the buffer should equal ValidationLen.
Type	A halfword field that contains the type of the Text data. Possible values: 0 indicates an IPv4 address in integer format with 4-byte hexadecimal representation. For example: 093C1C66. 1 indicates an IPv6 address in integer format with 16-byte hexadecimal representation. For example: 50C6 C2C1 0000 0000 0009 0060 0028 0102. 2 indicates a fully-qualified domain name (FQDN) in EBCDIC format. 3 indicates a host name in EBCDIC format. 4 indicates an IPv4 address in dotted decimal format. For example: 9.60.28.102. 5 indicates an IPv6 address in dotted decimal format. For example: 50C6:C2C1::9.60.28.102.
Text	The string that is compared to the common name, domain name, or in a subject alternate name extension marked as an IP address in the server certificate.
Note:	When Version is 1, the caller must allocate and send the full length of the ValidationBuffer (512 bytes) even though it might be partially filled in.
Handshake Complete	The SecureHSCompleteDetailType structure contains the result of the handshake request. For a blocking socket, the SecureHSCompleteDetailType structure is returned in the ErrNo field. An ErrNo of 0 indicates a successful completion. A non-blocking socket is woken up for write or exception. If the socket is woken up for write, it is assumed that the SecureHSCompleteDetailType structure contains all 0's and is not returned. If the socket is woken up for exception, the SecureHSCompleteDetailType structure is returned in the ErrNo field on the subsequent read and provides the details of the handshake failure. Refer to the SecureHSCompleteDetailType structure for details.

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
<pre>struct SecureHSCompleteDetailType { char ReturnCode; char AlertLevel; short AlertDescription; };</pre>	<pre>ReturnCode DS X AlertLevel DS X AlertDescription DS XL2</pre>

where:

ReturnCode

Indicates the status of the handshake.

- 0** - NOALERT - The handshake completed successfully.
- 1** - ALERT - Problems were encountered during the handshake.
- 2** - TIMEOUT - The handshake did not complete within the time allotted.

AlertLevel

When the ReturnCode is ALERT, this classifies the level of the alert:

- 0** - AlertOK
- 1** - Warning
- 2** - Fatal

AlertDescription

When ReturnCode is ALERT, this field contains the details of the failure. An AlertDescription value in the 4000 range indicates an SSL server error as follows:

- 4001** - The type is not valid.
- 4002** - The integer format of the IP address is not valid.
- 4003** - ValidationBuffer is too long.
- 4004** - Len is either too big or extends beyond the buffer.
- 4005** - The maximum number of validation fields has been exceeded.
- 4006** - The dotted decimal format of the IPv4 address is not valid.
- 4007** - The dotted decimal format of the IPv6 address is not valid.
- 4008** - Validation of a host name or fully-qualified domain name failed.
- 4009** - Validation of an IPv4 or IPv6 address failed.
- 4010** - Validation failed.

An AlertDescription value in the 40000 range indicates a System SSL error. Subtract 40000 from the AlertDescription value and refer to Messages and codes in *z/OS Cryptographic Services System Secure Sockets Layer Programming* (publibz.boulder.ibm.com/epubs/pdf/gsk2aa00.pdf) for details.

<pre>struct sockaddr_in { short sin_family; ushort sin_port; struct in_addr sin_addr; char sin_zero[8]; };</pre>	<pre>FAMILY DS H PORT DS H ADDR DS F ZERO DC XL8'00'</pre>
<pre>struct sockaddr_in6 { short sin6_family; ushort sin6_port; uint sin6_flowinfo; struct in6_addr sin6_addr; uint sin6_scope_id; };</pre>	<pre>FAMILY DS H PORT DS H FLOWINFO DS F ADDR6 DS 4F SCOPEID DS F</pre>

Table 21. C Structures in Assembler Language Format (continued)

C Structure	Assembler Language Equivalent
<pre>struct timeval { long tv_sec; long tv_usec; };</pre>	<pre>TVSEC DS F TVUSEC DS F</pre>

IUCV Socket Calls

This section provides the C language syntax, parameters, and other information about each IUCV socket call supported by TCP/IP. For information about C socket calls, see [Chapter 1, “z/VM C Socket Application Programming Interface,”](#) on page 1.

Note: In the following socket descriptions, structures labelled **For AF_INET:** are for the AF_INET address family and structures labelled **For AF_INET6:** are for the AF_INET6 address family.

ACCEPT

The ACCEPT call is issued when the server receives a connection request from a client. ACCEPT points to a socket that was created with a socket call and marked by a LISTEN call. ACCEPT can also be used as a blocking call. Concurrent server programs use the ACCEPT call to pass connection requests to child servers.

When issued, the ACCEPT call:

1. Accepts the first connection on a queue of pending connections
2. Creates a new socket with the same properties as the socket used in the call and returns the address of the client for use by subsequent server calls. The new socket cannot be used to accept new connections, but can be used by the calling program for its own connection. The original socket remains available to the calling program for more connection requests.
3. Returns the new socket descriptor to the calling program.

For AF_INET:

```
ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr_in *addr;
int *addrlen;
```

For AF_INET6:

```
ns = accept(s, addr, addrlen)
int ns, s;
struct sockaddr_in6 *addr;
int *addrlen;
```

Keyword

Value

TRGCLS

High-order halfword = 1

Low-order halfword = s

DATA

PRMMSG

PRMMSG

High-order fullword = 0

Low-order fullword = socket number for the new socket, chosen by your program, in the range 0 through *maxsock*. If you wish the stack to choose an available socket number for you, specify any

BIND

negative value (bit 0 is 1). See [“Initializing the IUCV Connection” on page 143](#) for more information on *maxsock*.

ANSLEN

For AF_INET: 24

For AF_INET6: 36

ANSBUF

Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>ns</i>	4	The new socket number assigned to this connection. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>ns</i> is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*addr</i>	For AF_INET: 16 For AF_INET6: 28	The remote address and port of the new socket. See Table 21 on page 148 for format.

BIND

In a typical server program, the BIND call follows a SOCKET call and completes the new socket creation process.

The BIND call can either specify the port or let the system choose the port. A listener program should always bind to the same well-known port so that clients know what socket address to use when issuing a CONNECT call.

For AF_INET:

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

For AF_INET6:

```
rc = bind(s, name, namelen)
int rc, s;
struct sockaddr_in6 *name;
int namelen;
```

Keyword

Value

TRGCLS

High-order halfword = 2

Low-order halfword = s

DATA

BUFFER

BUFLEN

For AF_INET: 16

For AF_INET6: 28

BUFFER

Points to a buffer in the following format:

Offset	Name	Length	Comments
0	<i>*name</i>	For AF_INET: 16	The local address and port to which the socket is to be bound. See Table 21 on page 148 for format.
		For AF_INET6: 28	

ANSLEN

8

ANSBUF

Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the BIND call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

CANCEL and CANCEL2

The CANCEL and CANCEL2 calls are used to cancel a previously issued socket call. For the CANCEL call, TCP/IP responds to the canceled call with a return code of -1 and an *errno* value of 1009. For the CANCEL2 call, TCP/IP does not send a response to the canceled call.

Keyword

Value

TRGCLS

High-order halfword = 42 (CANCEL)

High-order halfword = 43 (CANCEL2)

Low-order halfword = Low-order halfword of TRGCLS from call to be canceled.

DATA

PRMMSG

PRMMSG

High-order fullword = High-order halfword of TRGCLS from call to be canceled.

Low-order fullword = IUCV message ID of call to be canceled.

ANSLEN

8

ANSBUF

Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the CANCEL call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.

CLOSE

Offset	Name	Length	Comments
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Possible <i>errno</i> values are: 3 Specifies that the call cannot be found. TCP/IP might have already responded to it. 22 Specifies that the call is not a type that may be canceled.

CLOSE

The CLOSE call shuts down the socket and frees the resources that are allocated to the socket.

```
rc = close(s)
int rc, s;
```

Keyword
Value

TRGCLS
High-order halfword = 3
Low-order halfword = s

DATA
PRMMSG

PRMMSG
Binary zeros

ANSLEN
8

ANSBUF
Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the CLOSE call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

CONNECT

The CONNECT call is used by a client to establish a connection between a local socket and a remote socket.

For stream sockets, the CONNECT call:

- Completes the binding process for a stream socket if a BIND call has not been previously issued.
- Attempts a connection to a remote socket. This connection must be completed before data can be transferred.

For datagram sockets, a CONNECT call is not essential, but you can use it to send messages without including the destination.

For AF_INET:

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

For AF_INET6:

```
rc = connect(s, name, namelen)
int rc, s;
struct sockaddr_in6 *name;
int namelen;
```

Keyword**Value****TRGCLS**

High-order halfword = 4

Low-order halfword = s

DATA

BUFFER

BUFLen**For AF_INET:** 16**For AF_INET6:** 28**BUFFER**

Points to a buffer in the following format:

Offset	Name	Length	Comments
0	<i>*name</i>	For AF_INET: 16 For AF_INET6: 28	The remote address and port to which the socket is to be connected. See Table 21 on page 148 for format.

ANSLEN

8

ANSBUF

The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the CONNECT call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

FCNTL

The blocking mode for a socket can be queried or set using the FNDELAY flag described in the FCNTL call.

See [“IOCTL” on page 170](#) for another way to control blocking for a socket.

```
retval = fcntl(s, cmd, arg)
int retval;
int s, cmd, arg;
```

GETCLIENTID

Keyword
Value

TRGCLS
High-order halfword = 5
Low-order halfword = s

DATA
PRMMSG

PRMMSG
High-order fullword:

F_GETFL	(X '00000003')
F_SETFL	(X '00000004')

The low-order fullword is used only for the F_SETFL command:

Zero	(X '00000000') Socket will block
FNDELAY	(X '00000004') Socket is non-blocking

ANSLEN
8

ANSBUF
Points to a buffer that is filled with a reply in the format described as follows:

Offset	Name	Length	Comments
0	<i>retval</i>	4	For F_SETFL, the return code. A value of zero indicates FNDELAY flag was set. For F_GETFL, the value of the FNDELAY flag. Zero means the socket will block. A value of FNDELAY (4) means the socket is non-blocking. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

GETCLIENTID

The GETCLIENTID call returns the identifier by which the calling application is known to the TCPIP address space. The client ID structure that is returned is used in the GIVESOCKET and TAKESOCKET calls.

```
rc = getclientid(domain, clientid)
int rc, domain;
struct clientid *clientid;
```

Keyword
Value

TRGCLS
High-order halfword = 30
Low-order halfword = 0

DATA
PRMMSG

PRMMSG
Binary zeros

ANSLEN

48

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the GETCLIENTID call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*clientid</i>	40	See Table 21 on page 148 for format.

Note: *domain* is not passed to TCP/IP. It is implicitly AF_INET.

GETHOSTID

The GETHOSTID call gets the unique 32-bit identifier for the current host. This value is the default home internet address.

```
hostid = gethostid
unsigned long hostid;
```

Keyword**Value****TRGCLS**

High-order halfword = 7

Low-order halfword = 0

DATA

PRMMSG

PRMMSG

Binary zeros

ANSLEN

8

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>hostid</i>	4	The default home internet address.
4		4	Your program should ignore this field.

GETHOSTNAME

The GETHOSTNAME call returns the name of the host processor on which the program is running. Up to *namelen* characters are copied into the name field.

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

GETPEERNAME

Keyword
Value

TRGCLS
High-order halfword = 8
Low-order halfword = 0

DATA
PRMMSG

PRMMSG
Binary zeros

ANSLEN
namelen + 8

ANSBUF
Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the GETHOSTNAME call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*name</i>	<i>namelen</i>	The host name, not null-terminated.

GETPEERNAME

The GETPEERNAME call returns the name of the remote socket to which the local socket is connected.

For AF_INET:

```
rc = getpeername(s, name, namelen)
int rc, s;
struct sockaddr_in *name;
int *namelen;
```

For AF_INET6:

```
rc = getpeername(s, name, namelen)
int rc, s;
struct sockaddr_in6 *name;
int *namelen;
```

Keyword
Value

TRGCLS
High-order halfword = 9
Low-order halfword = *s*

DATA
PRMMSG

PRMMSG
Binary zeros

ANSLEN
For AF_INET: 24

For AF_INET6: 36

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the GETPEERNAME call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*name</i>	For AF_INET: 16 For AF_INET6: 28	The remote address and port to which the socket is connected. See Table 21 on page 148 for format.

GETSOCKNAME

The GETSOCKNAME call stores the name of the socket into the structure pointed to by the *name* parameter and returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the *family* field completed and the rest of the structure set to zeros.

For AF_INET:

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

For AF_INET6:

```
rc = getsockname(s, name, namelen)
int rc, s;
struct sockaddr_in6 *name;
int *namelen;
```

Keyword
Value

TRGCLS
High-order halfword = 10
Low-order halfword = s

DATA
PRMMSG

PRMMSG
Binary zeros

ANSLEN
For AF_INET: 24
For AF_INET6: 36

ANSBUF
Points to the buffer that is filled with a reply in the following format:

GETSOCKOPT

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the GETSOCKNAME call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*name</i>	For AF_INET: 16 For AF_INET6: 28	The local address and port to which the socket is bound. See Table 21 on page 148 for format.

GETSOCKOPT

The GETSOCKOPT call returns the current setting of an option for a specific socket. Some of these options are under program control and can be changed using the SETSOCKOPT call.

```
rc = getsockopt(s, level, optname, optval, &optlen)
int rc, s, level, optname, optlen;
char *optval;
```

Keyword

Value

TRGCLS

High-order halfword = 11

Low-order halfword = s

DATA

PRMMSG

PRMMSG

High-order fullword = *level*. Possible values are:

Value	C Symbol	Comments
X'FFFF'	SOL_SOCKET	Socket option
X'0006'	IPPROTO_TCP	TCP protocol option

Low-order fullword = *optname*. Possible values are:

Value	Option Name	Returned Value
X'0001'	SO_DEBUG	Returns current setting.
X'0004'	SO_REUSEADDR	Returns current setting.
X'0008'	SO_KEEPALIVE	Returns current setting.
X'0010'	SO_DONTROUTE	Returns current setting.
X'0020'	SO_BROADCAST	Returns current setting.
X'0080'	SO_LINGER	Returns current setting in a C language <i>struct linger</i> . See Table 21 on page 148 for the assembler language equivalent.

Value	Option Name	Returned Value
X'0100'	SO_OOBLINE	Returns current setting.
X'1001'	SO_SNDBUF	Returns the size of the TCP/IP send buffer.
X'1007'	SO_ERROR	Returns any pending error code and clears any error status conditions.
X'1008'	SO_TYPE	Socket type is returned: <div> Value Type 1 Stream 2 Datagram 3 Raw </div>
X'0001'	TCP_NODELAY	Returns current setting. Note: This option applies only to <i>level</i> =IPPROTO_TCP

ANSLEN

16 for option SO_LINGER, 12 for all other options

ANSBUF

Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the GETSOCKOPT call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*optval</i>	4 or 8	The value of the requested option. If the option SO_LINGER was requested, 8 bytes are returned. For all other options, 4 bytes are returned.

GIVESOCKET

The GIVESOCKET call makes the socket available for a TAKESOCKET call issued by another program. The GIVESOCKET call can specify any connected stream socket. Typically, the GIVESOCKET call is issued by a concurrent server program that creates sockets to be passed to a child server.

The GIVESOCKET sequence is:

- To pass a socket, the concurrent server first calls GIVESOCKET. If the optional parameters, name of the child server's virtual machine and subtask ID are specified in the GIVESOCKET call, only a child with a matching virtual machine and subtask ID can take the socket.
- The concurrent server then starts the child server and passes it the socket descriptor and concurrent server's ID that were obtained from earlier SOCKET and GETCLIENTID calls.
- The child server calls TAKESOCKET, with the concurrent server's ID and socket descriptor.

IOCTL

- The concurrent server issues the `select` call to test the socket for the exception condition, `TAKESOCKET` completion.
- When the `TAKESOCKET` has successfully completed, the concurrent server issues the `CLOSE` call to free the socket.

```
rc = givesocket(s, clientid)
int rc, s;
struct clientid *clientid;
```

Keyword
Value

TRGCLS

High-order halfword = 31

Low-order halfword = s

DATA

BUFFER

BUFLen

40

BUFFER

Points to the message in the following format:

Offset	Name	Length	Comments
0	<i>*clientid</i>	40	See Table 21 on page 148 for format.

ANSLEN

8

ANSBUF

The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the <code>GIVESOCKET</code> call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

IOCTL

The `IOCTL` call is used to control certain operating characteristics for a socket.

Before you issue an `IOCTL` call, you must load a value representing the characteristic that you want to control into the *cmd* field.

```
rc = ioctl(s, cmd, arg)
int rc, s;
unsigned long cmd;
char *arg;
```

Keyword
Value

TRGCLS

High-order halfword = 12

Low-order halfword = s

DATA

BUFFER

BUFLENRequest *arg* length + 4**BUFFER**

The pointer to the message in the format described in the following format:

Offset	Name	Length	Comments
0	<i>cmd</i>	4	The type of request. See Table 22 on page 171 for values.
4	<i>*arg</i>	See Table 22 on page 171 .	The request data, if any.

ANSLENReply *arg* length + 8**ANSBUF**

The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*arg</i>	See Table 22 on page 171 .	The response data, if any.

Table 22. Values for cmd Argument in ioctl Call

C Symbol	Value	Request <i>arg</i> Length	Reply <i>arg</i> Length	Comments
FIONBIO	X'8004A77E'	4	0	Request <i>arg</i> data is a fullword integer.
FIONREAD	X'4004A77F'	0	4	Reply <i>arg</i> data is a fullword integer.
SIOCADDRT	X'8030A70A'	48	0	For IBM use only.
SIOCATMARK	X'4004A707'	0	4	Reply <i>arg</i> data is a fullword integer.
SIOCDELRT	X'8030A70B'	48	0	For IBM use only.

Table 22. Values for *cmd* Argument in *ioctl* Call (continued)

C Symbol	Value	Request <i>arg</i> Length	Reply <i>arg</i> Length	Comments
SIOCGCERTDATA	X'C090DA07'	144	*	arg data is the C language struct <code>CertReqDetailType</code> for the request and <code>CertDataCompleteDetailType</code> for the reply. See Table 21 on page 148 for the assembler language equivalent. For more information, see “Requesting Details from a Partner Certificate” on page 22. * Length of <code>CertDataCompleteDetailType</code>
SIOCGIBMIFMTU	X'C020D903'	32	32	For IBM use only.
SIOCGIBMOP	X'C048D900'	72	*	For IBM use only.
SIOCGIFADDR	X'C020A70D'	32	32	arg data is the C language struct <i>ifreq</i> . See Table 21 on page 148 for the assembler language equivalent.
SIOCGIFBRDADDR	X'C020A712'	32	32	arg data is the C language struct <i>ifreq</i> . See Table 21 on page 148 for the assembler language equivalent.
SIOCGIFCONF	X'C008A714'	8	*	Request arg data is the C-language struct <i>ifconf</i> . See Table 21 on page 148 for the assembler language equivalent. Your program sets <i>ifc_len</i> to the reply length. The other field is ignored. Response arg data is an array of C language struct <i>ifreq</i> structures, one for each defined interface. Note: * = the maximum number of interfaces multiplied by 32.
SIOCGIFDSTADDR	X'C020A70F'	32	32	arg data is the C language struct <i>ifreq</i> . See Table 21 on page 148 for the assembler language equivalent.
SIOCGIFFLAGS	X'C020A711'	32	32	arg data is the C language struct <i>ifreq</i> . See Table 21 on page 148 for the assembler language equivalent.
SIOCGIFMETRIC	X'C020A717'	32	32	For IBM use only.

Table 22. Values for cmd Argument in ioctl Call (continued)

C Symbol	Value	Request arg Length	Reply arg Length	Comments
SIOCGIFNETMASK	X'C020A715'	32	32	arg data is the C language struct <i>ifreq</i> . See Table 21 on page 148 for the assembler language equivalent.
SIOCSECCLIENT	X'8143DA01'	323	0	arg data is the C language struct <i>SecureDetail</i> . See Table 21 on page 148 for assembler language equivalent. ¹
SIOCSECCLOSE	X'8101DA04'	257	0	arg data is the C language struct <i>CloseReq</i> . See Table 21 on page 148 for assembler language equivalent. ¹
SIOCSECERT	X'8008DA06'	8	*	For IBM use only.
SIOCSECSEVER	X'8143DA02'	323	0	arg data is the C language struct <i>SecureDetail</i> . See Table 21 on page 148 for assembler language equivalent. ¹
SIOCSECSTATUS	X'400CDA05'	0	12	arg data is the C language struct <i>SecStatus</i> . See Table 21 on page 148 for assembler language equivalent. ¹
SIOCSIBMOPT	X'8048D900'	*	0	For IBM use only.
SIOCSIFDSTADDR	X'8020A70E'	32	0	For IBM use only.
SIOCSIFFLAGS	X'8020A710'	32	0	For IBM use only.
SIOCSIFMETRIC	X'8020A718'	32	0	For IBM use only.
SIOCTLSQUERY	X'803ADA03'	58	0	arg data is the C language struct <i>QueryTLS</i> . See Table 21 on page 148 for assembler language equivalent. ¹

LISTEN

The LISTEN call:

- Completes the bind, if BIND has not already been called for the socket.
- Creates a connection-request queue of a specified length for incoming connection requests.

The LISTEN call is typically used by a concurrent server to receive connection requests from clients. When a connection request is received, a new socket is created by a later ACCEPT call. The original socket continues to listen for additional connection requests. The LISTEN call converts an active socket to a passive socket and configures it to accept connection requests from client programs. If a socket is passive it cannot initiate connection requests.

¹ For additional information on using secure ioctls, refer to “Secure Connection Considerations” on page 21.

MAXDESC

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

Keyword

Value

TRGCLS

High-order halfword = 13

Low-order halfword = s

DATA

PRMMSG

PRMMSG

High-order fullword = 0

Low-order fullword = *backlog*

ANSLEN

8

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the LISTEN call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

MAXDESC

Your program specifies the maximum number of Internet domain (AF_INET and AF_INET6) sockets in the *initialization message*. For more information about the initialization message, see [“Initializing the IUCV Connection”](#) on page 143.

READ, READV

From the point of view of TCP/IP, the READ and READV calls are identical. From the point of view of the application, they differ only in that the buffer for READ is contiguous in storage, while the buffer for READV might not be contiguous.

Your program, utilizing the direct IUCV socket interface, can use the ANSLIST=YES parameter on IUCV SEND to specify a noncontiguous READ buffer. You can choose to use ANSLIST=YES even if your READ buffer is contiguous, so that the reply area for *cc* and *errno* need not adjoin the READ buffer in storage.

This section does not distinguish between READ and READV. IUCV usage is described in terms of variable names from the C language syntax of READ.

```
cc = read(s, buf, len)
int cc, s;
char *buf;
int len;
```

Keyword

Value

TRGCLS

High-order halfword = 14

Low-order halfword = s

DATA

PRMMSG

PRMMSG

Binary zeros

ANSLEN

For AF_INET: *len* + 24

For AF_INET6: *len* + 36

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes read. A value of zero means the partner has closed the connection. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8		For AF_INET: 16 For AF_INET6: 28	Your program should ignore this field.
24	<i>*buf</i>	<i>len</i>	The received data.

RECV, RECVFROM, RECVMSG

From the point of view of TCP/IP, the RECV, RECVFROM, and RECVMSG calls are identical.

From the point of view of the application, RECVFROM differs from RECV in that RECVFROM additionally provides the source address of the message. Your program, using the direct IUCV socket interface, must provide space to receive the source address of the message, even if the source address is not required.

From the point of view of the application, RECVMSG differs from RECVFROM in that RECVMSG additionally allows the buffer to be in noncontiguous storage. Your program, utilizing the direct IUCV socket interface, can use the ANSLIST=YES parameter on IUCV SEND to specify a noncontinuous read buffer. You can choose to use ANSLIST=YES even if your read buffer is contiguous, so that the reply area for *cc* and *errno*, and the space to receive the source address of the message, need not adjoin the read buffer in storage.

For AF_INET:

```
cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr_in *from;
int *fromlen;
```

For AF_INET6:

```
cc = recvfrom(s, buf, len, flags, from, fromlen)
int cc, s;
char *buf;
int len, flags;
struct sockaddr_in6 *from;
int *fromlen;
```

SELECT, SELECTEX

Keyword
Value

TRGCLS
High-order halfword = 16
Low-order halfword = s

DATA
PRMMSG

PRMMSG
High-order fullword = 0.
Low-order fullword = *flags*:

MSG_OOB	(X'00000001')
MSG_PEEK	(X'00000002')

ANSLEN
For AF_INET: *len* + 24
For AF_INET6: *len* + 36

ANSBUF
Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes read. A value of zero indicates that communication is closed. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code. Note: The rest of the reply buffer is filled only if the call was successful.
8	<i>*from</i>	For AF_INET: 16 For AF_INET6: 28	The source address and port of the message. See Table 21 on page 148 for format.
24	<i>*buf</i>	<i>len</i>	The received data.

SELECT, SELECTEX

From the point of view of the TCP/IP, the SELECT and SELECTEX calls are identical. From the point of view of the application, they differ in that return from SELECTEX can be triggered by the posting of an ECB as well as the selection of a descriptor or a time-out.

Multiple SELECT calls, referring to any combination of sockets, can be queued simultaneously on an IUCV path.

```
nfound = select(nfds, readfds, writefds, exceptfds, timeout)
int nfound, nfds;
fd_set *readfds, *writefds, *exceptfds;
struct timeval *timeout;
```

Descriptor Sets

A descriptor set is an array of fullwords. The following is the required array size in integer arithmetic:

```
number_of_fullwords = (nfds + 31) / 32
number_of_bytes = number_of_fullwords * 4
```

DESCRIPTOR_SET, FD_CLR, FD_ISSET Calls

The following describes how to perform the function of these C language calls, which set, clear, and test the bit in the specified descriptor set corresponding to the specified descriptor number.

You can compute the offset of the fullword containing the bit (integer arithmetic) as follows:

```
offset = (descriptor_number / 32) * 4
```

Compute a mask to locate the bit within the fullword by:

```
bitmask = X'00000001' << (descriptor_number modulo 32)
```

("<<" is the left-shift operator).

Then use the mask, or a complemented copy of the mask, to set, clear, or test the bit, as appropriate.

The IUCV SEND parameters particular to select are:

Keyword

Value

TRGCLS

High-order halfword = 19

Low-order halfword = descriptor set size in bytes (*fdsize*). See [“Descriptor Sets” on page 176](#).

DATA

BUFFER

BUFLEN

(3**fdsize*)+28

BUFFER

The pointer to the message in the following format:

Offset	Name	Length	Number of file descriptors
0	<i>nfds</i>	4	To improve processing efficiency, <i>nfds</i> should be no greater than one plus the largest descriptor number actually in use.
4		4	Set this field to zero if you want select to block. Otherwise set this field to any nonzero value and fill in <i>*timeval</i> .
8		4	If any descriptor bits are set in <i>readfds</i> , your program sets this field to a nonzero value. If no descriptor bits are set in <i>readfds</i> , your program can set this field to zero, to improve processing efficiency.
12		4	If any descriptor bits are set in <i>writefds</i> , your program sets this field to a nonzero value. If no descriptor bits are set in <i>writefds</i> , your program can set this field to zero to improve processing efficiency.
16		4	If any descriptor bits are set in <i>exceptfds</i> , your program sets this field to a nonzero value. If no descriptor bits are set in <i>exceptfds</i> your program can set this field to zero to improve processing efficiency.
20	<i>*timeval</i>	8	See Table 21 on page 148 for format. If field at offset 4 is zero, then set this field to binary zeros.

SEND

Offset	Name	Length	Number of file descriptors
28	<i>*readfds</i>	<i>fdsizes</i>	If field at offset 8 is zero, then this field is not used.
28 + <i>fdsizes</i>	<i>*writefds</i>	<i>fdsizes</i>	If field at offset 12 is zero, then this field is not used.
28 + (2 * <i>fdsizes</i>)	<i>*exceptfds</i>	<i>fdsizes</i>	If field at offset 16 is zero, then this field is not used.

ANSLEN

(3**fdsizes*)+16

ANSBUF

The pointer to the buffer that is filled in with a reply in the following format:

Offset	Name	Length	Comments
0	<i>nfound</i>	4	The total number of ready sockets (in all bit masks). A value of zero indicates an expired time limit. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>nfound</i> is -1, this field contains a reason code.
8		8	Your program ignores this field. Note: The rest of the reply buffer is filled only if the call was successful.
16	<i>*readfds</i>	<i>fdsizes</i>	If field at offset 8 in request data was zero, then your program ignores this field.
16 + <i>fdsizes</i>	<i>*writefds</i>	<i>fdsizes</i>	If field at offset 12 in request data was zero, then your program ignores this field.
16 + (2 * <i>fdsizes</i>)	<i>*exceptfds</i>	<i>fdsizes</i>	If field at offset 16 in request data was zero, then your program ignores this field.

SEND

The SEND call sends datagrams on a specified connected socket.

The *flags* field allows you to:

- Send out-of-band data, for example, interrupts, aborts, and data marked urgent.
- Suppress use of local routing tables. This implies that the caller takes control of routing, writing network software.

For datagram sockets, the entire datagram is sent if the datagram fits into the buffer. Excess data is discarded.

For stream sockets, data is processed as streams of information with no boundaries separating data the data. For example, if a program is required to send 1000 bytes, each call to this function can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in *errno*. Therefore, programs using stream sockets should place this call in a loop, reissuing the call until all data has been sent.

```
cc = send(s, msg, len, flags)
int cc, s;
char *msg;
int len, flags;
```

Keyword

Value

TRGCLS

High-order halfword = 20

Low-order halfword = s

BUFLEN**For AF_INET:** *len* + 20**For AF_INET6:** *len* + 32**DATA**

BUFFER

BUFFER

The pointer to the message in the following format:

Offset	Name	Length	Comments
0	<i>flags</i>	4	MSG_OOB (X'00000001') MSG_DONTROUTE (X'00000004')
4		For AF_INET: 16 For AF_INET6: 28	Your program should set this field to binary zeros.
20	<i>*msg</i>	<i>len</i>	The data to be sent.

ANSLEN

8

ANSBUF

The pointer to the buffer that is filled in with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes sent. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code.

SENDMSG

From the point of view of TCP/IP, the SENDMSG call with a null *msg->msg_name* parameter is identical to the SEND call. Similarly, the SENDMSG call with a non-null *msg->msg_name* parameter is identical to the SENDTO call.

From the point of view of the application, SENDMSG differs from SEND and SENDTO in that SENDMSG additionally allows the write buffer to be in noncontiguous storage.

Your program, using the direct IUCV socket interface can use the BUFLIST=YES parameter on IUCV SEND to specify a noncontiguous write buffer. You can choose to use BUFLIST=YES even if your write buffer is contiguous, so that the fields preceding the write data in the request format need not adjoin the write data in storage.

See [“SEND” on page 178](#) and [“SENDTO” on page 180](#) for more information.

SENDTO

SENDTO is similar to SEND, except that it includes the destination address parameter. You can use the destination address on the SENDTO call to send datagrams on a UDP socket that is connected or not connected.

Use the *flags* parameter to :

- Send out-of-band data such as, interrupts, aborts, and data marked as urgent.
- Suppress the local routing tables. This implies that the caller takes control of routing, which requires writing network software.

For datagram sockets, the SENDTO call sends the entire datagram if the datagram fits into the buffer.

For stream sockets, data is processed as streams of information with no boundaries separating the data. For example, if a program is required to send 1000 bytes, each SENDTO call can send any number of bytes, up to the entire 1000 bytes, with the number of bytes sent returned in *errno*. Therefore, programs using stream sockets should place SENDTO in a loop that repeats the call until all data has been sent.

For AF_INET:

```
cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr_in *to;
int tolen;
```

For AF_INET6:

```
cc = sendto(s, msg, len, flags, to, tolen)
int cc, s;
char *msg;
int len, flags;
struct sockaddr_in6 *to;
int tolen;
```

Keyword
Value

TRGCLS
High-order halfword = 22
Low-order halfword = s

DATA
BUFFER

BUFLen
For AF_INET: len + 20
For AF_INET6: len + 32

BUFFER
The pointer to the message in the following format:

Offset	Name	Length	Comments
0	flags	4	MSG_OOB (X'00000001') MSG_DONTROUTE (X'00000004')
4	*to	For AF_INET: 16 For AF_INET6: 28	See Table 21 on page 148 for format.

Offset	Name	Length	Comments
20	<i>*msg</i>	<i>len</i>	The data to be sent.

ANSLEN

8

ANSBUF

The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes sent. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code.

SETSOCKOPT

The SETSOCKOPT call sets the options associated with a socket.

```
rc = setsockopt(s, level, optname, optval, optlen)
int rc, s, level, optname;
char *optval;
int optlen;
```

Keyword**Value****TRGCLS**

High-order halfword = 23

Low-order halfword = s

DATA

BUFFER

BUFLen

16 for option SO_LINGER, 12 for all other options

BUFFER

Points to a buffer in the following format:

Offset	Name	Length	Comments
0	<i>level</i>	4	X'FFFF' - SOL_SOCKET - Socket option X'0006' - IPPROTO_TCP - TCP protocol option
4	<i>optname</i>	4	Option to set. See Table 23 on page 181 for values.
8	<i>*optval</i>	4 or 8	The value of the specified option. If the option SO_LINGER is specified, 8 bytes are needed. For all other options, 4 bytes are needed.

Table 23. Option name values for SETSOCKOPT

Value	Option Name	Option Value
X'0001'	SO_DEBUG	On (1) or Off (0). Option may be set, but has no effect.
X'0004'	SO_REUSEADDR	Yes (1) or No (0).
X'0008'	SO_KEEPALIVE	Yes (1) or No (0).
X'0010'	SO_DONTROUTE	Yes (1) or No (0). Option may be set, but has no effect. Use MSG_DONTROUTE on write-type calls instead.

SHUTDOWN

Table 23. Option name values for SETSOCKOPT (continued)

Value	Option Name	Option Value
X'0020'	SO_BROADCAST	Yes (1) or No (0).
X'0080'	SO_LINGER	Value is a C language <i>struct linger</i> . See Table 21 on page 148 for the assembler language equivalent.
X'0100'	SO_OOINLINE	Yes (1) or No (0). Note: The following option applies only to <i>level</i> =IPPROTO_TCP
X'0001'	TCP_NODELAY	Yes (1) or No (0).

ANSLEN

8

ANSBUF

Points to a buffer to contain the reply from TCP/IP:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the SETSOCKOPT call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

SHUTDOWN

The normal way to terminate a network connection is to issue the CLOSE call which attempts to complete all outstanding data transmission requests prior to breaking the connection. The SHUTDOWN call can be used to close one-way traffic while completing data transfer in the other direction. The how parameter determines the direction of the traffic to shutdown.

A client program can use the SHUTDOWN call to reuse a given socket with a different connection.

```
rc = shutdown(s, how)
int rc, s, how;
```

Keyword

Value

TRGCLS

High-order halfword = 24

Low-order halfword = s

DATA

PRMMSG

PRMMSG

High-order fullword = 0

Low-order fullword = *how*:

0 = receive

1 = send

2 = both

ANSLEN

8

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the SHUTDOWN call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

SOCKET

The SOCKET call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

```
s = socket(domain, type, protocol)
int s, domain, type, protocol
```

Keyword

Value

TRGCLS

High-order halfword = 25

Low-order halfword = 0

DATA

BUFFER

BUFLN

16

BUFFER

The pointer to the message in the following format:

Offset	Name	Length	Comments
0	<i>domain</i>	4	Values are: AF_INET (X'00000002') AF_INET6 (X'00000013')
4	<i>type</i>	4	Fullword integer: SOCK_STREAM (X'00000001') SOCK_DGRAM (X'00000002') SOCK_RAW (X'00000003')
8	<i>protocol</i>	4	Fullword integer: IPPROTO_ICMP (X'00000001') IPPROTO_TCP (X'00000006') IPPROTO_UDP (X'00000011') IPPROTO_RAW (X'000000FF')
12	<i>s</i>	4	Socket number for the new socket, chosen by your program, in the range 0 through <i>maxsock</i> . See “Initializing the IUCV Connection” on page 143 .

ANSLEN

8

TAKESOCKET

ANSBUF

Points to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	s	4	The socket number assigned to this communications end point. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	errno	4	When s is -1, this field contains a reason code.

TAKESOCKET

The TAKESOCKET call acquires a socket from another program and creates a new socket. Typically, a child server issues this call using client ID and socket descriptor data which it obtained from the concurrent server. When TAKESOCKET is issued, a new socket descriptor is returned in *errno*. You should use this new socket descriptor in later calls such as GETSOCKOPT, which require the s (socket descriptor) parameter.

Note: Both concurrent servers and iterative servers are used by this interface. An iterative server handles one client at a time. A concurrent server receives connection requests from multiple clients and creates child servers that process the client requests. When a child server is created, the concurrent server gets a new socket, passes the new socket to the child server, and dissociates itself from the connection. The TCP/IP Listener program is an example of a concurrent server.

```
s = takesocket(clientid, hisdesc)
int s;
struct clientid *clientid;
int hisdesc;
```

Keyword

Value

TRGCLS

High-order halfword = 32

Low-order halfword = 0

DATA

BUFFER

BUFLEN

48

BUFFER

The pointer to the message in the following format:

Offset	Name	Length	Comments
0	*clientid	40	See Table 21 on page 148 for format.
40	hisdesc	4	
44	s	4	Socket number for the new socket, chosen by your program, in the range 0 through <i>maxsock</i> . See “Initializing the IUCV Connection” on page 143 .

ANSLEN

8

ANSBUF

The pointer to the buffer that is filled with a reply in the following format:

Offset	Name	Length	Comments
0	<i>s</i>	4	The socket number assigned to this communications end point. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>s</i> is -1, this field contains a reason code.

WRITE, WRITEV

From the point of view of TCP/IP, the WRITE and WRITEV calls are identical. From the point of view of the application, WRITEV differs from WRITE in that WRITEV additionally allows the write buffer to be in noncontiguous storage.

Your program, using the direct IUCV socket interface, can use the BUFLIST=YES parameter on IUCV SEND to specify a noncontiguous write buffer. You can choose to use BUFLIST=YES even if your write buffer is contiguous, so that the 20-byte prefix need not adjoin the write buffer in storage.

This section does not distinguish between WRITE and WRITEV. IUCV usage is described in terms of variable names from the C language syntax of WRITE.

```
cc = write(s, buf, len)
int cc, s;
char *buf;
int len;
```

Keyword

Value

TRGCLS

High-order halfword = 26

Low-order halfword = *s*

DATA

BUFFER

BUFLLEN

For **AF_INET**: *len* + 20

For **AF_INET6**: *len* + 32

BUFFER

The pointer to the message in the following format:

Offset	Name	Length	Comments
0		For AF_INET: 20	Your program sets this parameter to binary zeros.
		For AF_INET6: 32	
20	<i>*buf</i>	<i>len</i>	The data to be sent.

ANSLEN

8

ANSBUF

Points to the buffer that is filled with a reply in the following format:

LASTERRNO

Offset	Name	Length	Comments
0	<i>cc</i>	4	The number of bytes sent. A value of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When <i>cc</i> is -1, this field contains a reason code.

LASTERRNO

As explained in “TCP/IP Response to an IUCV Request” on page 147, if TCP/IP uses IUCV REJECT to respond to a socket request, your program uses the LASTERRNO special request to retrieve the return code and *errno*.

Keyword

Value

TRGCLS

High-order halfword = 29

Low-order halfword = 0

DATA

PRMMSG

PRMMSG

Binary zeros

ANSLEN

8

ANSBUF

Points to the buffer that is filled in with a reply in the following format:

Offset	Name	Length	Comments
0	<i>rc</i>	4	The return code from the last rejected call. A return code of 0 indicates that the call was successful. A return code of -1 indicates that the function could not be completed and that <i>errno</i> contains a reason code.
4	<i>errno</i>	4	When the return code is -1, this field contains a reason code.

Chapter 5. Remote Procedure Calls

This chapter describes the high-level remote procedure calls (RPCs) implemented in TCP/IP, including the RPC programming interface to the C language, and communication between processes.

The 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. RPCs can communicate between processes on the same or different hosts.

The RPC Interface

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

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

When you use RPCs, the client communicates 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.

For sample RPC client, server, and raw data stream programs, see [“Sample RPC Programs” on page 234](#). [Figure 35 on page 188](#) and [Figure 36 on page 189](#) provide an overview of the high-level RPC client and server processes from initialization through cleanup.

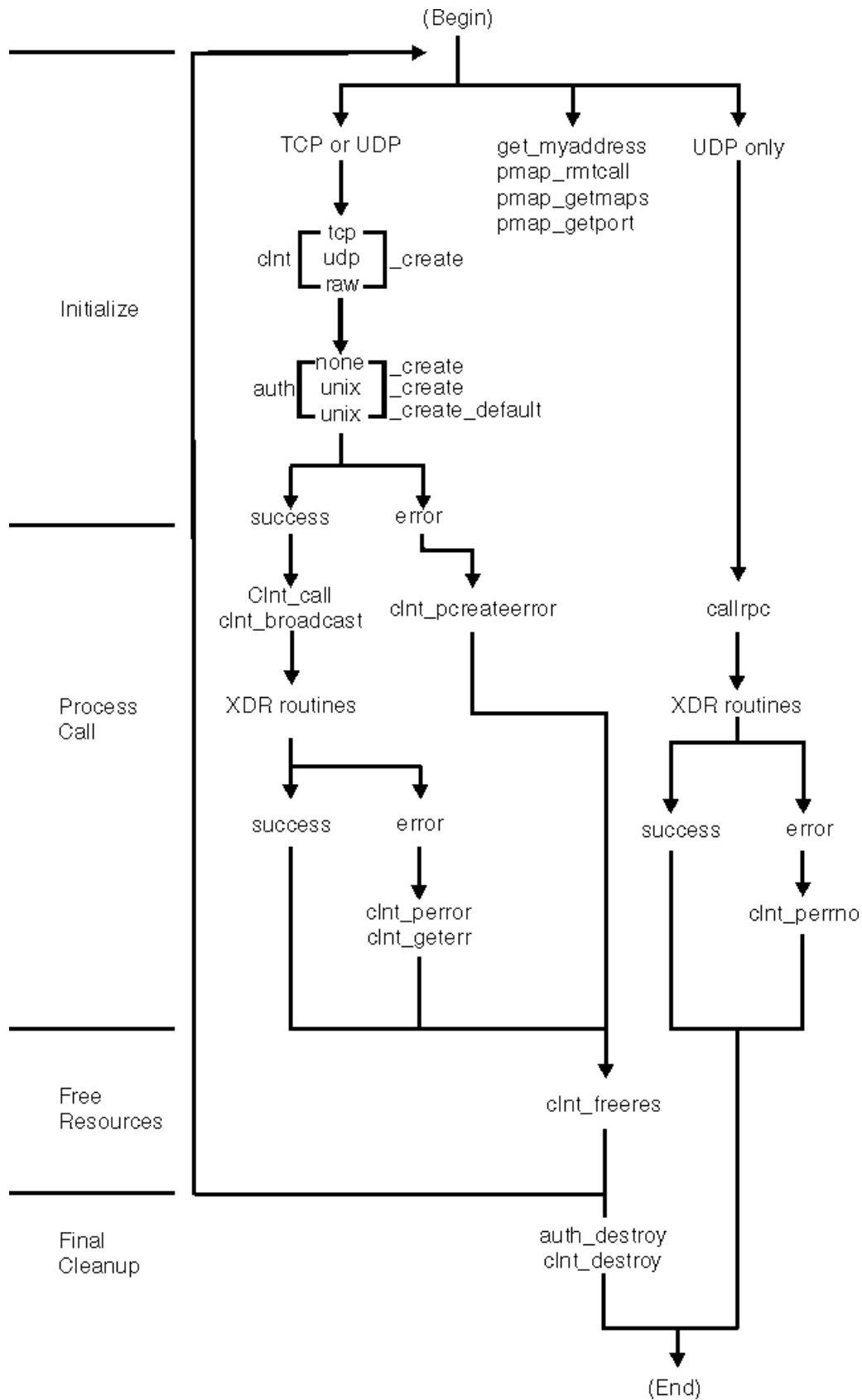


Figure 35. Remote Procedure Call (Client)

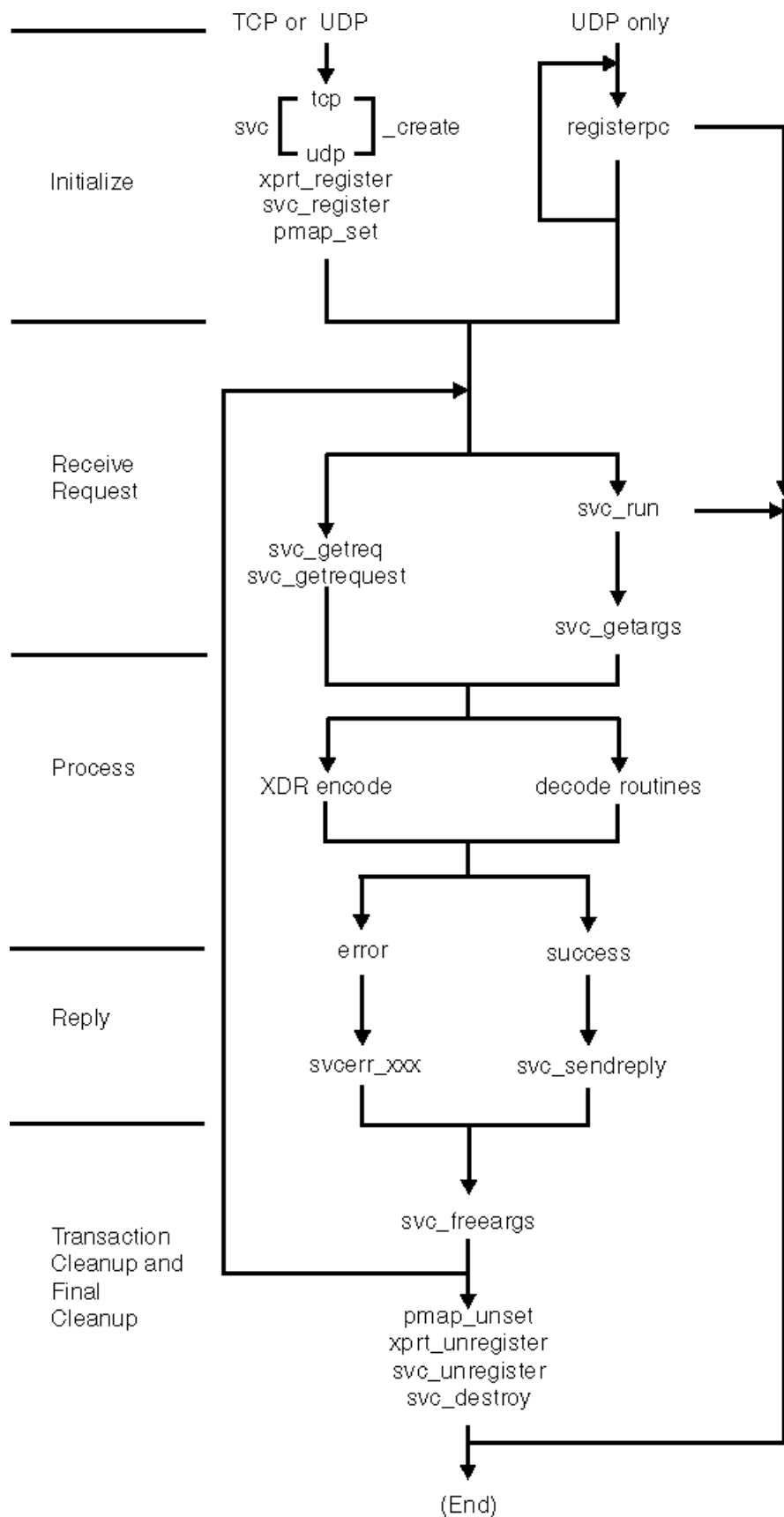


Figure 36. Remote Procedure Call (Server)

Portmapper

Portmapper is the software that supplies client programs with the port numbers of server programs.

You can communicate between different computer operating systems when messages are directed to port numbers rather than to targeted remote programs. Clients contact server programs by sending messages to the port numbers where receiving processes receive the message. Because you make requests to the port number of a server rather than directly to a server program, client programs need a way to find the port number of the server programs they wish to call. Portmapper standardizes the way clients locate the port number of the server programs supported on a network.

Portmapper resides on all hosts on well-known port 111.

The port-to-program information maintained by Portmapper is called the portmap. Clients ask Portmapper about entries for servers on the network. Servers contact Portmapper to add or update entries to the portmap.

Contacting Portmapper

To find the port of a remote program, the client sends an RPC to well-known port 111 of the server's host. If Portmapper has a portmap entry for the remote program, Portmapper provides the port number in a return RPC. The client then requests the remote program by sending an RPC to the port number provided by Portmapper.

Clients can save port numbers of recently called remote programs to avoid having to contact Portmapper for each request to a server.

To see all the servers currently registered with Portmapper, use the `RPCINFO` command in the following manner:

```
RPCINFO -p host_name
```

For more information about Portmapper and `RPCINFO`, see [z/VM: TCP/IP User's Guide](#) and [z/VM: TCP/IP Planning and Customization](#).

Target Assistance

Portmapper offers a program to assist clients in contacting server programs. If the client sends Portmapper an RPC with the target program number, version number, procedure number, and arguments, Portmapper searches the portmap for an entry, and passes the client's message to the server. When the target server returns the information to Portmapper, the information is passed to the client, along with the port number of the remote program. The client can then contact the server directly.

RPCGEN Command

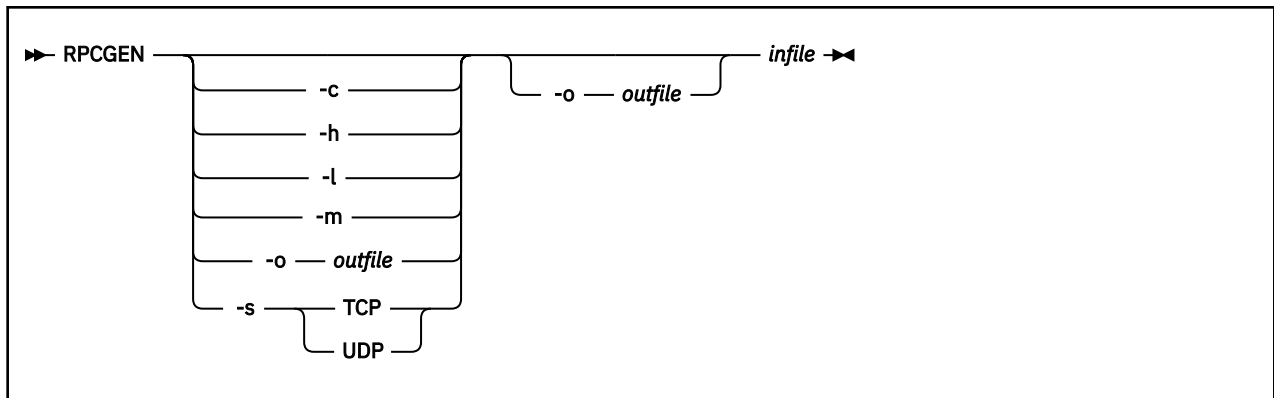
RPCGEN is a tool that generates C code to implement an RPC protocol. The input to RPCGEN is a language similar to C, known as RPC language. For RPCGEN to work correctly you must have access to the CC EXEC that is a part of the C compiler and have accessed the TCPIP Client-code minidisk (usually the TCPMAINT 592).

RPCGEN *infile* is normally used when you want to generate all four of the following output files. For example, if the *infile* is named *proto.x*, RPCGEN generates:

- A header file called `PROTO.H`
- XDR routines called `PROTOX.C`
- Server-side stubs called `PROTOS.C`
- Client-side stubs called `PROTOS.C`

Note: A temporary file called `PROTO.EXPANDED` or `PROTO.EXPAND` is created by the RPCGEN command. During normal operation, this file is also subsequently erased by the RPCGEN command.

For additional information about the RPCGEN command, see the Sun Microsystems publication, *Network Programming*.



Operand Description

-c

Compiles into XDR routines.

-h

Compiles into C data definitions (a header file).

-l

Compiles into client-side stubs.

-m

Compiles into server-side stubs without generating a main routine. This option is useful for call-back routines and for writing a main routine for initialization.

-s TCP/UDP

Compiles into server-side stubs using the given transport. The TCP option supports the TCP transport protocol. The UDP option supports the UDP transport protocol.

-o outfile

Specifies the name of the output file. If none is specified, standard output is used for -c, -h, -l, -m, and -s modes.

infile

Specifies the name of the input file written in the RPC language. *infile* should be a variable record format file (RECFM V).

enum clnt_stat Structure

The enumerated set `clnt_stat` structure is defined in the `CLNT.H` header file.

RPCs frequently return the enumerated set `clnt_stat` information. The following is the format and a description of the enumerated set `clnt_stat` structure:

```

enum clnt_stat {
    RPC_SUCCESS=0,          /* call succeeded */
    /*
     * local errors
     */
    RPC_CANTENCODEARGS=1,   /* can't encode arguments */
    RPC_CANTDECODERES=2,    /* can't decode results */
    RPC_CANTSEND=3,         /* failure in sending call */
    RPC_CANTRECV=4,         /* failure in receiving result */
    RPC_TIMEDOUT=5,         /* call timed out */
    /*
     * remote errors
     */
    RPC_VERSIONMISMATCH=6,  /* RPC versions not compatible */
    RPC_AUTHERROR=7,        /* authentication error */
    RPC_PROGUNAVAIL=8,      /* program not available */
    RPC_PROGVERSIONMISMATCH=9, /* program version mismatched */

```

```

RPC_PROCUUNAVAIL=10,      /* procedure unavailable */
RPC_CANTDECODEARGS=11,   /* decode arguments error */
RPC_SYSTEMERROR=12,      /* generic "other problem" */
/*
 * callrpc errors
 */
RPC_UNKNOWNHOST=13,       /* unknown host name */
/*
 * create errors
 */
RPC_PMAPFAILURE=14,       /* the pmapper failed in its call */
RPC_PROGNOTREGISTERED=15, /* remote program is not registered */
/*
 * unspecified error
 */
RPC_FAILED=16,            /* call failed */
RPC_UNKNOWNPROTO=17       /* unknown protocol */
};

```

Porting

This section contains information about porting RPC applications.

Accessing System Return Messages

To access system return values, you need only use the `ERRNO.H` include statement supplied with the compiler. To access network return values, you must add the following include statement:

```
#include <tcperrno.h>
```

Printing System Return Messages

To print only system errors, use `perror()`, a procedure available in the C compiler run-time library. To print both system and network errors, use `tcperror()`, a procedure included with TCP/IP.

Enumerations

To account for varying length enumerations, use the `xdr_enum()` and `xdr_union()` macros. `xdr_enum()` cannot be referenced by `callrpc()`, `svc_freeargs()`, `svc_getargs()`, or `svc_sendreply()`. An XDR routine for the specific enumeration must be created. The `xdr_union()` is not eligible for reference by these calls in any RPC environment. For more information, see [“xdr_enum\(\)”](#) on page 221.

Compiling, Linking, and Running an RPC Program

Note: If your program uses z/VM C sockets, follow the instructions in this section. If your program uses VM TCP/IP C sockets, see [“Recompiling with the TCP/IP C Sockets Library”](#) on page 28.

Before you compile and link an RPC program, read the information under [“Compiling and Linking a z/VM C Sockets Program”](#) on page 26 and [“Running a Sockets Program”](#) on page 29.

To compile, link and run an RPC program:

1. Access the TCP/IP Client-code disk (usually TCPMAINT 592), which contains the header files for RPC and the VMRPC TXTLIB, *after* the disk that contains the Language Environment header files (usually the Y-disk).
2. Specify the `_OE_SOCKETS` and VM preprocessor symbols in your source code or on the `c89` command.
3. Compile the program using `c89`. The following are examples of how to compile the sample RPC programs shown at the end of this chapter (see [“Sample RPC Programs”](#) on page 234):

```

c89 //genesend.c -D_OE_SOCKETS -l//VMRPC
c89 //geneserv.c -D_OE_SOCKETS -l//VMRPC
c89 //rawex.c -D_OE_SOCKETS -l//VMRPC

```

Note the use of the `//` syntax before the name of the `c` part. This convention informs `c89` that the `c` source part will be found in the CMS search order. The previous three `c89` commands will produce the `GENESEND MODULE`, `GENESERV MODULE`, and `RAWEX MODULE`, respectively. Additionally, note that `-DVM` is not specified on these compiles because the define for `VM` is in the C source.

4. Make sure that the `SCEERUN LOADLIB` is `GLOBALed` by issuing the command: `GLOBAL LOADLIB SCEERUN`
5. Run your program from either the CMS command line or from a POSIX shell command line. For example, run the `GENESERV MODULE` from the CMS command line as follows:

```
openvm run GENESERV
or
GENESERV
```

RPC Global Variables

This section describes the RPC global variables, `rpc_createerr`, `svc_fds`, and `svc_fdset`.

rpc_createerr

Description: A global variable that is set when any RPC client creation routine fails. Use `clnt_pcreateerror()` to print the message.

```
#include <rpc.h>
struct rpc_createerr rpc_createerr;
```

See Also: `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

svc_fds

```
#include <rpc.h>
int svc_fds;
```

Description: A global variable that specifies the read descriptor bit set on the service machine. This is of interest only if the service programmer decides to write an asynchronous event processing routine; otherwise `svc_run()` should be used. Writing asynchronous routines in the VM 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 VM programs for coordinating concurrent activities.

Attention: Do not modify this variable.

See Also: `svc_getreq()`.

svc_fdset

```
#include <rpc.h>
fd_set svc_fdset;
```

Description: A global variable that specifies the read descriptor bit set on the service machine. This is of interest only if the service programmer decides to write an asynchronous event processing routine; otherwise `svc_run()` should be used. Writing asynchronous routines in the VM environment is not simple,

auth_destroy()

because there is no direct relationship between the descriptors used by the socket routines and the Event Control Blocks commonly used by VM programs for coordinating concurrent activities.

Attention: Do not modify this variable.

See Also: svc_getreqset().

Remote Procedure Calls and External Data Representation

This section provides the syntax, operands, and other appropriate information for each remote procedure and external data representation call supported by TCP/IP.

auth_destroy()

```
#include <rpc.h>

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

Operand	Description
---------	-------------

<i>auth</i>	
--------------------	--

Points to authentication information.

Description: The auth_destroy() call deletes the authentication information for *auth*. Once this procedure is called, *auth* is undefined.

See Also: authnone_create(), authunix_create(), authunix_create_default().

authnone_create()

```
#include <rpc.h>
AUTH *
authnone_create()
```

The authnone_create() call has no operands.

Description: The authnone_create() call creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

See Also: auth_destroy(), authunix_create(), authunix_create_default().

authunix_create()

```
#include <rpc.h>

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

Operand	Description
---------	-------------

host

Specifies a pointer to the symbolic name of the host where the desired server is located.

uid

Identifies the user's user ID.

gid

Identifies the user's group ID.

len

Specifies the length of the information pointed to by *aup_gids*.

aup_gids

Specifies a pointer to an array of groups to which the user belongs.

Description: The `authunix_create()` call creates and returns an authentication handle that contains UNIX-based authentication information.

See Also: `auth_destroy()`, `authnone_create()`, `authunix_create_default()`.

authunix_create_default()

```
#include <rpc.h>

AUTH *
authunix_create_default()
```

The `authunix_create_default()` call has no operands.

Description: The `authunix_create_default()` call calls `authunix_create()` with default operands.

See Also: `auth_destroy()`, `authnone_create()`, `authunix_create()`.

callrpc()

```
#include <rpc.h>

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

Operand	Description
host	Specifies a pointer to the symbolic name of the host where the desired server is located.
prognum	Identifies the program number of the remote procedure.
versnum	Identifies the version number of the remote procedure.
procnum	Identifies the procedure number of the remote procedure.
inproc	Specifies the XDR procedure used to encode the arguments of the remote procedure.

clnt_broadcast()

in

Specifies a pointer to the arguments of the remote procedure.

outproc

Specifies the XDR procedure used to decode the results of the remote procedure.

out

Specifies a pointer to the results of the remote procedure.

Description: The `callrpc()` call calls the remote procedure described by *prognum*, *versnum*, and *procnum* running on the *host* system. `callrpc()` encodes and decodes the operands for transfer.

Note:

1. `clnt_perrno()` can be used to translate the return code into messages.
2. `callrpc()` cannot call the procedure `xdr_enum`. See [“xdr_enum\(\)” on page 221](#) for more information.
3. This procedure uses UDP as its transport layer, see [“clntudp_create\(\)” on page 204](#) for more information.

Return Values: `RPC_SUCCESS` indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

See Also: `clnt_broadcast()`, `clnt_call()`, `clnt_perrno()`, `clntudp_create()`, `clnt_sperrno()`, `xdr_enum()`.

clnt_broadcast()

```
#include <rpc.h>

enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
resultproc_t eachresult;
```

Operand

Description

prognum

Identifies the program number of the remote procedure.

versnum

Identifies the version number of the remote procedure.

procnum

Identifies the procedure number of the remote procedure.

inproc

Specifies the XDR procedure used to encode the arguments of the remote procedure.

in

Specifies a pointer to the arguments of the remote procedure.

outproc

Specifies the XDR procedure used to decode the results of the remote procedure.

out

Specifies a pointer to the results of the remote procedure.

eachresult

Specifies the procedure called after each response.

Note: `resultproc_t` is a type definition:

```
#include <rpc.h>

typedef bool_t (*resultproc_t) ();
```

Description: The `clnt_broadcast()` call broadcasts the remote procedure described by *prognum*, *versnum*, and *procnum* to all locally connected broadcast networks. Each time `clnt_broadcast()` receives a response it calls `eachresult()`. The format of `eachresult()` is:

```
#include <rpc.h>

bool_t eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

Operand Description

out

Has the same function as it does for `clnt_broadcast()`, except that the output of the remote procedure is decoded.

addr

Points to the address of the machine that sent the results.

Return Values: If `eachresult()` returns 0, `clnt_broadcast()` waits for more replies; otherwise, `eachresult()` returns the appropriate status.

Note: Broadcast sockets are limited in size to the maximum transfer unit of the data link.

See Also: `callrpc()`, `clnt_call()`.

clnt_call()

```
#include <rpc.h>

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

Operand Description

clnt

Points to a client handle that was previously obtained using `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()`.

procnum

Identifies the remote procedure number.

inproc

Identifies the XDR procedure used to encode *procnum*'s arguments.

in

Points to the remote procedure's arguments.

outproc

Specifies the XDR procedure used to decode the remote procedure's results.

out

Points to the remote procedure's results.

clnt_control()

tout

Specifies the time allowed for the server to respond.

Description: The `clnt_call()` call calls the remote procedure (*procnum*) associated with the client handle (*clnt*).

Return Values: `RPC_SUCCESS` indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_geterr()`, `clnt_perror()`, `clnt_sperror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_control()

```
#include <rpc.h>

bool_t
clnt_control(clnt, request, info)
CLIENT *clnt;
int request;
void *info;
```

Operand

Description

clnt

Specifies the pointer to a client handle that was previously obtained using `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()`.

request

Determines the operation (either `CLSET_TIMEOUT`, `CLGET_TIMEOUT`, `CLGET_SERVER_ADDR`, `CLSET_RETRY_TIMEOUT`, or `CLGET_RETRY_TIMEOUT`).

info

Points to information used by the request.

Description: The `clnt_control()` call performs one of the following control operations.

- Control operations that apply to both UDP and TCP transports:

CLSET_TIMEOUT

Sets time-out (*info* points to the `timeval` structure).

CLGET_TIMEOUT

Gets time-out (*info* points to the `timeval` structure).

CLGET_SERVER_ADDR

Gets server's address (*info* points to the `sockaddr_in` structure).

- UDP only control operations:

CLSET_RETRY_TIMEOUT

Sets retry time-out (information points to the `timeval` structure).

CLGET_RETRY_TIMEOUT

Gets retry time-out (*info* points to the `timeval` structure). If you set the timeout using `clnt_control()`, the timeout operand to `clnt_call()` will be ignored in all future calls.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `clnt_create()`, `clnt_destroy()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_create()

```
#include <rpc.h>

CLIENT *
clnt_create(host, prognum, versnum, protocol)
char *host;
u_long prognum;
u_long versnum;
char *protocol;
```

Operand Description

host

Points to the name of the host at which the remote program resides.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

protocol

Points to the protocol, which can be either tcp or udp.

Description: The `clnt_create()` call creates a generic RPC client transport handle for the remote program specified by (*prognum*, *versnum*). The client uses the specified protocol as the transport layer. Default timeouts are set, but can be modified using `clnt_control()`.

Return Values: NULL indicates failure.

See Also: `clnt_create()`, `clnt_destroy()`, `clnt_pcreateerror()`, `clnt_spccreateerror()`, `clnt_sperror()`, `clnttcp_create()`, `clntudp_create()`.

clnt_destroy()

```
#include <rpc.h>

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

Operand Description

clnt

Points to a client handle that was previously created using `clnt_create()`, `clntudp_create()`, `clnttcp_create()`, or `clntraw_create()`.

Description: The `clnt_destroy()` call deletes a client RPC transport handle. This procedure involves the deallocation of private data resources, including *clnt*. Once this procedure is used, *clnt* is undefined. If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

See Also: `clnt_control()`, `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_freeres()

```
#include <rpc.h>

bool_t
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

clnt_geterr()

Operand	Description
---------	-------------

clnt	Points to a client handle that was previously obtained using <code>clnt_create()</code> , <code>clntraw_create()</code> , <code>clnttcp_create()</code> , or <code>clntudp_create()</code> .
-------------	--

outproc	Specifies the XDR procedure used to decode the remote procedure's results.
----------------	--

out	Points to the results of the remote procedure.
------------	--

Description: The `clnt_freeres()` call deallocates any resources that were assigned by the system to decode the results of an RPC.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_geterr()

```
#include <rpc.h>

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

Operand	Description
---------	-------------

clnt	Points to a client handle that was previously obtained using <code>clnt_create()</code> , <code>clntraw_create()</code> , <code>clnttcp_create()</code> , or <code>clntudp_create()</code> .
-------------	--

errp	Points to the address into which the error structure is copied.
-------------	---

Description: The `clnt_geterr()` call copies the error structure from the client handle to the structure at address `errp`.

See Also: `clnt_call()`, `clnt_pcreateerror()`, `clnt_perrno()`, `clnt_perror()`, `clnt_spccreateerror()`, `clnt_sperrno()`, `clnt_sperror()`, `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_pcreateerror()

```
#include <rpc.h>

void
clnt_pcreateerror(s)
char *s;
```

Operand	Description
---------	-------------

s	Specifies a NULL or NULL-terminated character string. If <code>s</code> is non-NULL, <code>clnt_pcreateerror()</code> prints the string <code>s</code> followed by a colon, followed by a space, followed by the error message, and terminated with a newline character. If <code>s</code> is NULL or points to a NULL string, just the error message and the newline character are output.
----------	---

Description: The `clnt_pcreateerror()` call writes a message to the standard error device, indicating why a client handle cannot be created. This procedure is used after the `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()` calls fail.

See Also: `clnt_create()`, `clnt_geterr()`, `clnt_perrno()`, `clnt_perror()`, `clnt_spcreateerror()`, `clnt_sperrno()`, `clnt_sperror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_perrno()

```
#include <rpc.h>

void
clnt_perrno(stat)
enum clnt_stat stat;
```

Operand Description

stat

Specifies the client status.

Description: The `clnt_perrno()` call 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.

See Also: `callrpc()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_perror()`, `clnt_spcreateerror()`, `clnt_sperrno()`, `clnt_sperror()`.

clnt_perror()

```
#include <rpc.h>

void
clnt_perror(clnt, s)
CLIENT *clnt;
char *s;
```

Operand Description

clnt

Points to a client handle that was previously obtained using `clnt_create()`, `clntudp_create()`, `clnttcp_create()`, or `clntraw_create()`.

s

Specifies a NULL or NULL-terminated character string. If *s* is non-NULL, `clnt_perror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a new-line character. If *s* is NULL or points to a NULL string, just the error message and the new-line character are output.

Description: The `clnt_perror()` call writes a message to the standard error device, indicating why an RPC failed. This procedure should be used after `clnt_call()` if there is an error.

See Also: `clnt_call()`, `clnt_create()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_perrno()`, `clnt_spcreateerror()`, `clnt_sperrno()`, `clnt_sperror()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_spcreateerror()

clnt_sperno()

```
#include <rpc.h>

char *
clnt_spcreateerror(s)
char *s;
```

Operand Description

s

Specifies a NULL or NULL-terminated character string. If *s* is non-NULL, `clnt_spcreateerror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a new-line character. If *s* is NULL or points to a NULL string, just the error message and the new-line character are output.

Description: The `clnt_spcreateerror()` call returns the address of a message indicating why a client handle cannot be created. This procedure is used after the `clnt_create()`, `clntraw_create()`, `clnttcp_create()`, or `clntudp_create()` calls fail.

Return Values: Returns a pointer to a character string in a static data area. This data area is overwritten with each subsequent call. This function is not thread-safe.

See Also: `clnt_create()`, `clnt_geterr()`, `clnt_perrno()`, `clnt_perror()`, `clnt_pcreateerror()`, `clnt_sperno()`, `clnt_sperro()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clnt_sperro()

```
#include <rpc.h>

char *
clnt_sperro(stat)
enum clnt_stat stat;
```

Operand Description

stat

Specifies the client status.

Description: The `clnt_sperro()` call returns the address of a message corresponding to the condition indicated by *stat*. This procedure should be used after `callrpc()` if there is an error.

Return Values: Returns a pointer to a character string ending with a newline. This data area is overwritten with each subsequent call. This function is not thread-safe.

See Also: `callrpc()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_spcreateerror()`, `clnt_sperro()`, `clnt_perrno()`, `clnt_perror()`.

clnt_sperro()

```
#include <rpc.h>

char *
clnt_sperro(clnt, s)
CLIENT *clnt;
char *s;
```

Operand Description

clnt

Points to a client handle that was previously obtained using `clnt_create()`, `clntudp_create()`, `clnttcp_create()`, or `clntraw_create()`.

s

Specifies a NULL or a NULL-terminated character string. If *s* is non-NULL, `clnt_sperror()` prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a newline character. If *s* is NULL or points to a NULL string, just the error message and the newline character are output.

Description: The `clnt_sperror()` call returns the address of a message indicating why an RPC failed. This procedure should be used after `clnt_call()` if there is an error.

Return Values: Returns a pointer to a character string in a static data area. This data area is overwritten with each subsequent call. This function is not thread-safe.

See Also: `clnt_call()`, `clnt_create()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_perrno()`, `clnt_perror()`, `clnt_spcreateerror()`, `clnt_sperrno()`, `clntraw_create()`, `clnttcp_create()`, `clntudp_create()`.

clntraw_create()

```
#include <rpc.h>

CLIENT *
clntraw_create(prognum, versnum)
    u_long prognum;
    u_long versnum;
```

Operand**Description****prognum**

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

Description: The `clntraw_create()` call creates a dummy client for the remote double (*prognum*, *versnum*). Because messages are passed using a buffer within the virtual machine of the local process, the server should also use the same virtual machine, which simulates RPC programs within one virtual machine. For more information, see “[svcrw_create\(\)](#)” on page 216.

Return Values: NULL indicates failure.

See Also: `clnt_call()`, `clnt_destroy()`, `clnt_freeres()`, `clnt_geterr()`, `clnt_pcreateerror()`, `clnt_perror()`, `clnt_spcreateerror()`, `clnt_sperror()`, `clntudp_create()`, `clnttcp_create()`, `svcrw_create()`.

clnttcp_create()

```
#include <rpc.h>

CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
    struct sockaddr_in *addr;
    u_long prognum;
    u_long versnum;
    int *sockp;
    u_int sendsz;
    u_int recvsz;
```

Operand**Description**

clntudp_create()

addr

Points to the internet address of the remote program. If the *addr* port number is zero (*addr* -> *sin_port*), *addr* is set to the port on which the remote program is receiving.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

sockp

Points to the socket. If **sockp* is *RPC_ANYSOCK*, then this routine opens a new socket and sets **sockp*.

sendsz

Specifies the size of the send buffer. Specify 0 to choose the default.

rcvsvz

Specifies the size of the receive buffer. Specify 0 to choose the default.

Description: The *clnttcp_create()* call creates an RPC client transport handle for the remote program specified by (*prognum*, *versnum*). The client uses TCP as the transport layer.

Return Values: NULL indicates failure.

See Also: *clnt_call()*, *clnt_control()*, *clnt_create()*, *clnt_destroy()*, *clnt_freeres()*, *clnt_geterr()*, *clnt_pcreateerror()*, *clnt_perror()*, *clnt_spccreateerror()*, *clnt_sperror()*, *clntraw_create()*, *clntudp_create()*.

clntudp_create()

```
#include <rpc.h>

CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
    struct sockaddr_in *addr;
    u_long prognum;
    u_long versnum;
    struct timeval wait;
    int *sockp;
```

Operand

Description

addr

Points to the internet address of the remote program. If the *addr* port number is zero (*addr* -> *sin_port*), *addr* is set to the port on which the remote program is receiving. The remote portmap service is used for this.

prognum

Specifies the remote program number.

versnum

Specifies the version number of the remote program.

wait

Indicates that 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. If **sockp* is *RPC_ANYSOCK*, this routine opens a new socket and sets **sockp*.

Description: The *clntudp_create()* call 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. While UDP packet size is configurable to a maximum of 64–1 kilobytes, the default UDP packet size is only eight kilobytes.

Return Values: NULL indicates failure.

See Also: call_rpc(), clnt_call(), clnt_control(), clnt_create(), clnt_destroy(), clnt_freeres(), clnt_geterr(), clnt_pcreateerror(), clnt_perror(), clnt_spcreateerror(), clnt_sperror(), clntraw_create(), clnttcp_create().

get_myaddress()

```
#include <rpc.h>

void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Operand Description

addr

Points to the location where the local internet address is placed.

Description: The get_myaddress() call puts the local host's internet address into *addr*. The port number (*addr*→*sin_port*) is set to htons (PMAPPORT), which is 111.

See Also: getrpcport(), pmap_getmaps(), pmap_getport(), pmap_rmtcall(), pmap_set(), pmap_unset().

getrpcport()

```
#include <rpc.h>

u_short
getrpcport(host, prognum, versnum, protocol)
char *host;
u_long prognum;
u_long versnum;
int protocol;
```

Operand Description

host

Points to the name of the foreign host.

prognum

Specifies the program number to be mapped.

versnum

Specifies the version number of the program to be mapped.

protocol

Specifies the transport protocol used by the program (IPPROTO_TCP or IPPROTO_UDP).

Description: The getrpcport() call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

Return Values: The value 0 indicates that the mapping does not exist or that the remote portmap could not be contacted. If Portmapper cannot be contacted, rpc_createerr contains the RPC status.

See Also: get_myaddress(), pmap_getmaps(), pmap_getport(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_getmaps()

pmap_getport()

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

Operand Description

addr

Points to the internet address of the foreign host.

Description: The pmap_getmaps() call returns a list of current program-to-port mappings on the foreign host specified by *addr*.

Return Values: Returns a pointer to a pmaplist structure or NULL.

See Also: getrpcport(), pmap_getport(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_getport()

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
int protocol;
```

Operand Description

addr

Points to the internet address of the foreign host.

prognum

Identifies the program number to be mapped.

versnum

Identifies the version number of the program to be mapped.

protocol

Specifies the transport protocol used by the program (IPPROTO_TCP or IPPROTO_UDP).

Description: The pmap_getport() call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

Return Values: The value 0 indicates that the mapping does not exist or that the remote portmap could not be contacted. If Portmapper cannot be contacted, *rpc_createerr* contains the RPC status.

See Also: getrpcport() pmap_getmaps(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_rmtcall()

```

#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
u_long *portp;

```

Operand Description

addr

Points to the internet address of the foreign host.

prognum

Identifies the remote program number.

versnum

Identifies the version number of the remote program.

procnum

Identifies the procedure to be called.

inproc

Identifies the XDR procedure used to encode the arguments of the remote procedure.

in

Points to the arguments of the remote procedure.

outproc

Identifies the XDR procedure used to decode the results of the remote procedure.

out

Points to the results of the remote procedure.

tout

Identifies 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*).

Description: The `pmap_rmtcall()` call instructs Portmapper on the host at *addr* to make an RPC call to a procedure on that host, on your behalf. This procedure should be used only for ping type functions.

Return Values: Returns a `clnt_stat` enumerated type.

See Also: `getrpcport()`, `pmap_getmaps()`, `pmap_getport()`, `pmap_set()`, `pmap_unset()`.

pmap_set()

pmap_unset()

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
pmap_set(prognum, versnum, protocol, port)
u_long prognum;
u_long versnum;
int protocol;
u_short port;
```

Operand

Description

prognum

Identifies the local program number.

versnum

Identifies the version number of the local program.

protocol

Specifies the transport protocol used by the local program.

port

Identifies the port to which the local program is mapped.

Description: The `pmap_set()` call 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.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `getrpcport()`, `pmap_getmaps()`, `pmap_getport()`, `pmap_rmtcall()`, `pmap_unset()`.

pmap_unset()

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
pmap_unset(prognum, versnum)
u_long prognum;
u_long versnum;
```

Operand

Description

prognum

Identifies the local program number.

versnum

Identifies the version number of the local program.

Description: The `pmap_unset()` call 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.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `getrpcport()`, `pmap_getmaps()`, `pmap_getport()`, `pmap_rmtcall()`, `pmap_set()`.

registerrpc()

```
#include <rpc.h>

int
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum;
u_long versnum;
u_long procnum;
char *(*procname) ();
xdrproc_t inproc;
xdrproc_t outproc;
```

Operand Description

prognum

The program number to register.

versnum

Identifies the version number to register.

procnum

Specifies the procedure number to register.

procname

Specifies 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

Specifies the XDR routine used to decode the arguments.

outproc

Specifies the XDR routine that encodes the results.

Description: The `registerrpc()` call registers a procedure (*prognum*, *versnum*, *procnum*) with the local Portmapper, 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.

Note: `xdr_enum()` cannot be used as an argument to `registerrpc()`. See [“xdr_enum\(\)” on page 221](#) for more information.

Return Values: The value 0 indicates success; the value -1 indicates an error.

See Also: `svc_register()`, `svc_run()`.

svc_destroy()

```
#include <rpc.h>

void
svc_destroy(xprt)
SVCXPRT *xprt;
```

Operand Description

xprt

Points to the service transport handle.

Description: The `svc_destroy()` call deletes the RPC service transport handle *xprt*, which becomes undefined after this routine is called.

See Also: `svccraw_create()`, `svctcp_create()`, `svcudp_create()`.

svc_freeargs()

svc_freeargs()

```
#include <rpc.h>

bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Operand
Description

xprt

Points to the service transport handle.

inproc

Specifies the XDR routine used to decode the arguments.

in

Points to the input arguments.

Description: The `svc_freeargs()` call frees storage allocated to decode the arguments to a service procedure using `svc_getargs()`.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `svc_getargs()`.

svc_getargs()

```
#include <rpc.h>

bool_t
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Operand
Description

xprt

Points to the service transport handle.

inproc

Specifies the XDR routine used to decode the arguments.

in

Points to the decoded arguments.

Description: The `svc_getargs()` call 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*.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `svc_freeargs()`.

svc_getcaller()

```
#include <rpc.h>

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

Operand Description

xprt

Points to the service transport handle.

Description: This macro obtains the network address of the client associated with the service transport handle *xprt*.

Return Values: Returns a pointer to a `sockaddr_in` structure.

See Also: `get_myaddress()`.

svc_getreq()

```
#include <rpc.h>

void
svc_getreq(rdfds)
int rdfds;
```

Operand Description

rdfds

Specifies the read descriptor bit mask.

Description: The `svc_getreq()` call is used rather than `svc_run()` to implement asynchronous event processing. The routine returns control to the program when all sockets have been serviced.

Note: `svc_getreq()` limits you to 32 socket descriptors, of which 3 are reserved. Use `svc_getreqset()` if you have more than 29 socket descriptors.

See Also: `svc_run()`.

svc_getreqset()

```
#include <rpc.h>
void
svc_getreqset(rdfds)
fd_set rdfds;
```

Operand Description

rdfds

Specifies the read descriptor bit set.

Description: The `svc_getreqset()` call is used rather than `svc_run()` to implement asynchronous event processing. The routine returns control to the program when all sockets have been serviced.

A server would use a `select()` call to determine if there are any outstanding RPC requests at any of the sockets created when the programs were registered. The read bit descriptor set returned by `select()` is then used on the call to `svc_getreqset()`.

Note that you should not pass the global bit descriptor set `svc_fdset` on the call to `select()`, because `select()` changes the values. Instead, you should make a copy of `svc_fdset` before you call `select()`.

svc_register()

See Also: `svc_run()`.

svc_register()

```
#include <rpc.h>

bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum;
u_long versnum;
void (*dispatch) ();
int protocol;
```

Operand

Description

xprt

Points to the service transport handle.

prognum

Specifies the program number to be registered.

versnum

Specifies the version number of the program to be registered.

dispatch

Specifies the dispatch routine associated with *prognum* and *versnum*.

Specifies the structure of the dispatch routine is:

```
#include <rpc.h>

dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;
```

protocol

Specifies the protocol used. The value is generally one of the following:

- 0 (zero)
- IPPROTO_UDP
- IPPROTO_TCP

When a value of 0 is used, the service is not registered with Portmapper.

Note: When using a dummy RPC service transport created with `svccraw_create()`, a call to `xprt_register()` must be made immediately after a call to `svc_register()`.

Description: The `svc_register()` call associates the program described by (*prognum*, *versnum*) with the service dispatch routine *dispatch*.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `registerrpc()`, `svc_unregister()`, `xprt_register()`.

svc_run()

```
#include <rpc.h>

void
svc_run()
```

The `svc_run()` call has no operands.

Description: The `svc_run()` call does not return control. It accepts RPC requests and calls the appropriate service using `svc_getreqset()`.

See Also: `svc_getreqset()`.

svc_sendreply()

```
#include <rpc.h>

bool_t
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Operand Description

xprt

Points to the caller's transport handle.

outproc

Specifies the XDR procedure used to encode the results.

out

Points to the results.

Description: The `svc_sendreply()` call is called by the service dispatch routine to send the results of the call to the caller.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_call()`.

svc_unregister()

```
#include <rpc.h>

void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

Operand Description

prognum

Specifies the program number that is removed.

versnum

Specifies the version number of the program that is removed.

Description: The `svc_unregister()` call removes all local mappings of *(prognum, versnum)* to dispatch routines and *(prognum, versnum, *)* to port numbers.

See Also: `svc_register()`.

svcerr_auth()

svcerr_decode()

```
#include <rpc.h>

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

Operand Description

xprt

Points to the service transport handle.

why

Specifies the reason the call is refused.

Description: The `svcerr_auth()` call is called by a service dispatch routine that refuses to execute an RPC request because of authentication errors.

See Also: `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_decode()

```
#include <rpc.h>

void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

Operand Description

xprt

Points to the service transport handle.

Description: The `svcerr_decode()` call is called by a service dispatch routine that cannot decode its operands.

See Also: `svcerr_auth()`, `svcerr_noproc()`, `svcerr_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_noproc()

```
#include <rpc.h>

void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Operand Description

xprt

Points to the service transport handle.

Description: The `svcerr_noproc()` call is called by a service dispatch routine that does not implement the requested procedure.

See Also: `svcerr_auth()`, `svcerr_decode()`, `svcerr_noprogram()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_noprog()

```
#include <rpc.h>

void
svcerr_noprog(xprt)
SVCXPRT *xprt;
```

Operand Description

xprt

Points to the service transport handle.

Description: The `svcerr_noprog()` call is used when the desired program is not registered.

See Also: `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_progvers()

```
#include <rpc.h>

void
svcerr_progvers(xprt, low_vers, high_vers)
SVCXPRT *xprt;
u_long low_vers;
u_long high_vers;
```

Operand Description

xprt

Points to the service transport handle.

low_vers

Specifies the low version number that does not match.

high_vers

Specifies the high version number that does not match.

Description: The `svcerr_progvers()` call is called when the version numbers of two RPC programs do not match. The low version number corresponds to the lowest registered version, and the high version corresponds to the highest version registered on the Portmapper.

See Also: `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_systemerr()`, `svcerr_weakauth()`.

svcerr_systemerr()

```
#include <rpc.h>

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Operand Description

xprt

Points to the service transport handle.

svcerr_weakauth()

Description: The `svcerr_systemerr()` call is called by a service dispatch routine when it detects a system error that is not handled by the protocol.

See Also: `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_weakauth()`.

svcerr_weakauth()

```
#include <rpc.h>

void
svcerr_weakauth(xprt)
SVCXPRT *xprt;
```

Operand	Description
---------	-------------

<i>xprt</i>	Points to the service transport handle.
-------------	---

Note: This is the equivalent of: `svcerr_auth(xprt, AUTH_T00WEAK)`.

Description: The `svcerr_weakauth()` call is called by a service dispatch routine that cannot execute an RPC because of correct but weak authentication operands.

See Also: `svcerr_auth()`, `svcerr_decode()`, `svcerr_noproc()`, `svcerr_noprog()`, `svcerr_progvers()`, `svcerr_systemerr()`.

svcraw_create()

```
#include <rpc.h>

SVCXPRT *
svcraw_create()
```

The `svcraw_create()` call has no operands.

Description: The `svcraw_create()` call creates a local RPC service transport used for timings, to which it returns a pointer. Messages are passed using a buffer within the virtual machine of the local process; so, the client process must also use the same virtual machine. This allows the simulation of RPC programs within one computer. See [“clntraw_create\(\)” on page 203](#) for more information.

Return Values: NULL indicates failure.

See Also: `clntraw_create()`, `svc_destroy()`, `svctcp_create()`, `svcudp_create()`.

svctcp_create()

```
#include <rpc.h>

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size;
u_int recv_buf_size;
```

Operand	Description
---------	-------------

sock

Specifies the socket descriptor. If *sock* 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

Specifies the size of the send buffer. Specify 0 to choose the default.

recv_buf_size

Specifies the size of the receive buffer. Specify 0 to choose the default.

Description: The `svctcp_create()` call creates a TCP-based service transport to which it returns a pointer. `xprt->xp_sock` contains the transport's socket descriptor. `xprt->xp_port` contains the transport's port number.

Return Values: NULL indicates failure.

See Also: `svc_destroy()`, `svccraw_create()`, `svculdp_create()`.

svculdp_create()

```
#include <rpc.h>

SVCXPRT *
svculdp_create(sock, sendsz, recvsz)
int sock;
u_int sendsz;
u_int recvsz;
```

Operand**Description****sock**

Specifies the socket descriptor. If *sock* is `RPC_ANYSOCK`, a new socket is created. If the socket is not bound to a local UDP port, it is bound to an arbitrary port.

endsz

Specifies the size of the send buffer.

recvsz

Specifies the size of the receive buffer.

Description: The `svculdp_create()` call creates a UDP-based service transport to which it returns a pointer. `xprt->xp_sock` contains the transport's socket descriptor. `xprt->xp_port` contains the transport's port number.

Return Values: NULL indicates failure.

See Also: `svc_destroy()`, `svccraw_create()`, `svctcp_create()`.

xdr_accepted_reply()

```
#include <rpc.h>

bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Operand**Description****xdrs**

Points to an XDR stream.

xdr_array()

ar

Points to the reply to be represented.

Description: The xdr_accepted_reply() call translates RPC reply messages.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_array()

```
#include <rpc.h>

bool_t
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep;
u_int maxsize;
u_int elsize;
xdrproc_t elproc;
```

Operand

Description

xdrs

Points to an XDR stream.

arrp

Specifies the address of the pointer to the array.

sizep

Points to the element count of the array.

maxsize

Specifies the maximum number of elements accepted.

elsize

Specifies the size of each of the array's elements, found using sizeof().

elproc

Specifies the XDR routine that translates an individual array element.

Description: The xdr_array() call translates between an array and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_authunix_parms()

```
#include <rpc.h>

bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Operand

Description

xdrs

Points to an XDR stream.

aupp

Points to the authentication information.

Description: The xdr_authunix_parms() call translates UNIX-based authentication information.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_bool()

```
#include <rpc.h>

bool_t
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

Operand

Description

xdrs

Points to an XDR stream.

bp

Points to the Boolean.

Description: The xdr_bool() call translates between Booleans and their external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_bytes()

```
#include <rpc.h>

bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;
```

Operand

Description

xdrs

Points to an XDR stream.

sp

Points to a pointer to the byte string.

sizep

Points to the byte string size.

maxsize

Specifies the maximum size of the byte string.

Description: The xdr_bytes() call translates between byte strings and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_callhdr()

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_callhdr()

```
#include <rpc.h>

bool_t
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

Operand
Description

xdrs

Points to an XDR stream.

chdr

Points to the call header.

Description: The xdr_callhdr() call translates an RPC message header into XDR format.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_callmsg()

```
#include <rpc.h>

bool_t
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Operand
Description

xdrs

Points to an XDR stream.

cmsg

Points to the call message.

Description: The xdr_callmsg() call translates RPC messages (header and authentication, not argument data) to and from the xdr format.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_double()

```
#include <rpc.h>

bool_t
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

Operand Description

xdrs

Points to an XDR stream.

dp

Points to a double-precision number.

Description: The `xdr_double()` call translates between C double-precision numbers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_enum()

```
#include <rpc.h>

bool_t
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

Operand Description

xdrs

Points to an XDR stream.

ep

Points to the enumerated number. *enum_t* can be any enumeration type such as `enum colors`, with `colors` declared as `enum colors (black, brown, red)`.

Description: The `xdr_enum()` call translates between C-enumerated groups and their external representation. When calling the procedures `callrpc()` and `registerrpc()`, a stub procedure must be created for both the server and the client before the procedure of the application program using `xdr_enum()`. The following is the format of the stub procedure.

```
#include <rpc.h>

enum colors (black, brown, red)
void
static xdr_enum_t(xdrs, ep)
XDR *xdrs;
enum colors *ep;
{
    xdr_enum(xdrs, ep)
}
```

The `xdr_enum_t` procedure is used as the *inproc* and *outproc* in both the client and server RPCs.

For example, an RPC client would contain the following lines:

xdr_float()

```
        :
error = callrpc(argv[1],ENUMRCVPROC,VERSION,ENUMRCVPROC,
xdr_enum_t,&innumber,xdr_enum_t,
        &outnumber);
        :
```

An RPC server would contain the following line:

```
        :
registerrpc(ENUMRCVPROC,VERSION,ENUMRCVPROC,
xdr_enum_t,xdr_enum_t);
        :
```

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_float()

```
#include <rpc.h>

bool_t
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

Operand

Description

xdrs

Points to an XDR stream.

fp

Points to the floating-point number.

Description: The xdr_float() call translates between C floating-point numbers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_inline()

```
#include <rpc.h>

long *
xdr_inline(xdrs, len)
XDR *xdrs;
u_int len;
```

Operand

Description

xdrs

Points to an XDR stream.

len

Specifies the byte length of the desired buffer.

Description: The `xdr_inline()` call 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.

Note: `xdr_inline()` can return NULL if there is not sufficient space in the stream buffer to satisfy the request.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_int()

```
#include <rpc.h>

bool_t
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

Operand Description

xdrs
Points to an XDR stream.

ip
Points to the integer.

Description: The `xdr_int()` call translates between C integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_long()

```
#include <rpc.h>

bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

Operand Description

xdrs
Points to an XDR stream.

lp
Points to the long integer.

Description: The `xdr_long()` call translates between C long integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_opaque()

```
#include <rpc.h>

bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>cp</i>	Points to the opaque object.
------------------	------------------------------

<i>cnt</i>	Specifies the size of the opaque object.
-------------------	--

Description: The `xdr_opaque()` call translates between fixed-size opaque data and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_opaque_auth()

```
#include <rpc.h>

bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>ap</i>	Points to the opaque authentication information.
------------------	--

Description: The `xdr_opaque_auth()` call translates RPC message authentications.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_pmap()

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Operand Description

xdrs

Points to an XDR stream.

regs

Points to the portmap operands.

Description: The `xdr_pmap()` call translates an RPC procedure identification, such as is used in calls to `Portmapper`.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_pmaplist()

```
#include <rpc.h>
#include <pmap_pro.h>
#include <pmap_cln.h>

bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Operand Description

xdrs

Points to an XDR stream.

rp

Points to a pointer to the portmap data array.

Description: The `xdr_pmaplist()` call translates a variable number of RPC procedure identifications, such as `Portmapper` creates.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_pointer()

```
#include <rpc.h>

bool_t
xdr_pointer(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

xdr_reference()

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>pp</i>	Points to a pointer.
------------------	----------------------

<i>size</i>	Specifies the size of the target.
--------------------	-----------------------------------

<i>proc</i>	Specifies the XDR procedure that translates an individual element of the type addressed by the pointer.
--------------------	---

Description: The xdr_pointer() call provides pointer-chasing within structures. This differs from the xdr_reference() call in that it can serialize or deserialize trees correctly.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_reference()

```
#include <rpc.h>

bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
u_int size;
xdrproc_t proc;
```

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>pp</i>	Points to a pointer.
------------------	----------------------

<i>size</i>	Specifies the size of the target.
--------------------	-----------------------------------

<i>proc</i>	Specifies the XDR procedure that translates an individual element of the type addressed by the pointer.
--------------------	---

Description: The xdr_reference() call provides pointer-chasing within structures.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_rejected_reply()

```
#include <rpc.h>

bool_t
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Operand
Description

xdrs

Points to an XDR stream.

rr

Points to the rejected reply.

Description: The `xdr_rejected_reply()` call translates rejected RPC reply messages.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_replymsg()

```
#include <rpc.h>

bool_t
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Operand
Description

xdrs

Points to an XDR stream.

rmsg

Points to the reply message.

Description: The `xdr_replymsg()` call translates RPC reply messages.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_short()

```
#include <rpc.h>

bool_t
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

Operand
Description

xdrs

Points to an XDR stream.

sp

Points to the short integer.

Description: The `xdr_short()` call translates between C short integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_string()

```
#include <rpc.h>

bool_t
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>sp</i>	Points to a pointer to the string.
------------------	------------------------------------

<i>maxsize</i>	Specifies the maximum size of the string.
-----------------------	---

Description: The xdr_string() call translates between C strings and their external representations. The xdr_string() call is the only xdr routine to convert ASCII to EBCDIC.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_u_int()

```
#include <rpc.h>

bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>up</i>	Points to the unsigned integer.
------------------	---------------------------------

Description: The xdr_u_int() call translates between C unsigned integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: callrpc(), clnt_broadcast(), clnt_call(), clnt_freeres(), pmap_rmtcall(), registerrpc(), svc_freeargs(), svc_getargs(), svc_sendreply().

xdr_u_long()

```
#include <rpc.h>

bool_t
xdr_u_long(xdrs, ulp)
XDR *xdrs;
u_long *ulp;
```

Operand Description

xdrs

Points to an XDR stream.

ulp

Points to the unsigned long integer.

Description: The `xdr_u_long()` call translates between C unsigned long integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_u_short()

```
#include <rpc.h>

bool_t
xdr_u_short(xdrs, usp)
XDR *xdrs;
u_short *usp;
```

Operand Description

xdrs

Points to an XDR stream.

usp

Points to the unsigned short integer.

Description: The `xdr_u_short()` call translates between C unsigned short integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_union()

```
#include <rpc.h>

bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
enum_t *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Operand Description

xdrs

Points to an XDR stream.

xdr_vector()

dscmp

Points to the union's discriminant. *enum_t* can be any enumeration type.

unp

Points to the union.

choices

Points to an array detailing the XDR procedure to use on each arm of the union.

dfault

Specifies the default XDR procedure to use.

Description: The `xdr_union()` call translates between a discriminated C union and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

The following is an example of this call:

```
#include <rpc.h>

enum colors (black, brown, red);

bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
enum colors *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_vector()

```
#include <rpc.h>

bool_t
xdr_vector(xdrs, basep, nelem, elemsize, xdr_elem)
XDR *xdrs;
char *basep;
u_int nelem;
u_int elemsize;
xdrproc_t xdr_elem;
```

Operand

Description

xdrs

Points to an XDR stream.

basep

Specifies the base of the array.

nelem

Specifies the element count of the array.

elemsize

Specifies the size of each of the array's elements, found using `sizeof()`.

xdr_elem

Specifies the XDR routine that translates an individual array element.

Description: The `xdr_vector()` call translates between a fixed length array and its external representation. Unlike variable-length arrays, the storage of fixed length arrays is static and unfreeable.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_void()

```
#include <rpc.h>

bool_t
xdr_void()
```

The `xdr_void()` call has no operands.

Description: The `xdr_void()` call is used like a command that does not require any other xdr functions. This call can be placed in the *inproc* or *outproc* operand of the `clnt_call` function when the user does not need to move data.

Return Values: Always a value of 1.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdr_wrapstring()

```
#include <rpc.h>

bool_t
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

Operand Description

xdrs

Points to an XDR stream.

sp

Points to a pointer to the string.

Description: The `xdr_wrapstring()` call is the same as calling `xdr_string()` with a maximum size of `MAXUNSIGNED`. It is useful because many RPC procedures implicitly invoke two-operand XDR routines, and `xdr_string()` is a three-operand routine.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: `callrpc()`, `clnt_broadcast()`, `clnt_call()`, `clnt_freeres()`, `pmap_rmtcall()`, `registerrpc()`, `svc_freeargs()`, `svc_getargs()`, `svc_sendreply()`.

xdrmem_create()

```
#include <rpc.h>

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

Operand Description

xdrs

Points to an XDR stream.

xdrrec_create()

addr

Points to the memory location.

size

Specifies the maximum size of *addr*.

op

Determines the direction of the XDR stream (XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Description: The `xdrmem_create()` call initializes the XDR stream pointed to by *xdrs*. Data is written to, or read from, *addr*.

xdrrec_create()

```
#include <rpc.h>

void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
u_int sendsize;
u_int recvsize;
char *handle;
int (*readit) ();
int (*writeit) ();
```

Operand

Description

xdrs

Points to an XDR stream.

sendsize

Indicates the size of the send buffer. Specify 0 to choose the default.

recvsize

Indicates the size of the receive buffer. Specify 0 to choose the default.

handle

Specifies the first operand passed to *readit()* and *writeit()*.

readit()

Called when a stream's input buffer is empty.

writeit()

Called when a stream's output buffer is full.

Description: The `xdrrec_create()` call creates a record-oriented stream and initializes the XDR stream pointed to by *xdrs*.

Note:

1. The *x_op* field must be set by the caller.
2. This XDR procedure implements an intermediate record string.
3. Additional bytes in the XDR stream provide record boundary information.

xdrrec_endofrecord()

```
#include <rpc.h>
bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

Operand

Description

xdrs

Points to an XDR stream.

sendnow

Specifies nonzero to write out data in the output buffer.

Description: The `xdrrec_endofrecord()` call can be invoked only on streams created by `xdrrec_create()`. Data in the output buffer is marked as a complete record.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdrrec_eof()

```
#include <rpc.h>

bool_t
xdrrec_eof(xdrs)
XDR *xdrs;
```

Operand**Description*****xdrs***

Points to an XDR stream.

Description: The `xdrrec_eof()` call can be invoked only on streams created by `xdrrec_create()`.

Return Values: The value 1 indicates the current record has been consumed; the value 0 indicates continued input on the stream.

xdrrec_skiprecord()

```
#include <rpc.h>

bool_t
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

Operand**Description*****xdrs***

Points to an XDR stream.

Description: The `xdrrec_skiprecord()` call can be invoked only on streams created by `xdrrec_create()`. The XDR implementation is instructed to discard the remaining data in the input buffer.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdrstdio_create()

```
#include <rpc.h>
#include <stdio.h>

void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

xprt_register()

Operand	Description
---------	-------------

<i>xdrs</i>	Points to an XDR stream.
--------------------	--------------------------

<i>file</i>	Specifies the file name for the I/O stream.
--------------------	---

<i>op</i>	Determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).
------------------	--

Description: The xdrstdio_create() call initializes the XDR stream pointed to by *xdrs*. Data is written to, or read from, *file*.

Note: fflush() is the destroy routine associated with this procedure. fclose() is not called.

xprt_register()

```
#include <rpc.h>

void
xprt_register(xprt)
SVCXPRT *xprt;
```

Operand	Description
---------	-------------

<i>xprt</i>	Points to the service transport handle.
--------------------	---

Description: The xprt_register() call registers service transport handles with the RPC service package. This routine also modifies the global variable svc_fds

See Also: svc_register(), svc_fds.

xprt_unregister()

```
#include <rpc.h>

void
xprt_unregister(xprt)
SVCXPRT *xprt;
```

Operand	Description
---------	-------------

<i>xprt</i>	Points to the service transport handle.
--------------------	---

Description: The xprt_unregister() call 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 and svc_fdset.

See also: svc_fds, svc_fdset.

Sample RPC Programs

This appendix provides examples of the following programs:

- RPC Genesend client (see “RPC Genesend Client” on page 236)
- RPC Geneserv server (see “RPC Geneserv Server” on page 236)

- RPC Rawex raw data stream (see “RPC Rawex Raw Data Stream” on page 238)

Refer back to “Compiling, Linking, and Running an RPC Program” on page 192 for examples of how to compile, link, and run RPC programs.

Running the Geneserv server and Genesend client

The Geneserv server and Genesend client are a pair of client-server programs. Typically, the Geneserv server is run in one virtual machine and the Genesend client in another. If a POSIX shell command line is available, both can be run in the same virtual machine by starting the Geneserv server in the background and running the Genesend client in the foreground. The steps for running these programs are as follows:

1. Make sure the TCPIP Client-code disk is accessed (usually TCPMAINT 592).
2. Before the Geneserv server can be started, the Portmapper must be running. To determine if the Portmapper is running, contact the Portmapper with the command `RPCINFO -p`
3. Start the Geneserv server. From the CMS command line issue:

```
openvm run GENESERV
```

To start in the foreground from a POSIX shell command line issue:

```
geneserv
```

To start in the background from a POSIX shell command line issue:

```
geneserv &
```

After starting the Geneserv server you should see output similar to the following:

```
openvm run GENESERV
Intrcv Registration with Port Mapper completed
Floatrcv Registration with Port Mapper completed
integer received: 10
integer being returned: 10
```

4. Start the Genesend client. From the CMS command line issue:

```
openvm run GENESEND hostname some_number
```

To start in the foreground from a POSIX shell command line issue:

```
genesend hostname some_number
```

To start in the background from a POSIX shell command line issue:

```
genesend hostname some_number &
```

The *hostname* argument is the host running the Geneserv server. The *some_number* argument is an integer value that will be sent to the Geneserv server and then returned.

The following is a sample run of the Genesend client:

```
openvm run GENESEND myvmhost 10
value sent: 10 value received: 10
Ready;
```

Running the Rawex program

The rawex program uses the raw RPC interfaces and is a client and server program in the same program. To start Rawex from a CMS command line issue:

```
openvm run RAWEX some_number
```

RPC Client

To start in the foreground from a POSIX shell command line issue:

```
rawex some_number
```

To start in the background from a POSIX shell command line issue:

```
rawex some_number &
```

The following is a sample run of Rawex:

```
openvm run RAWEX 5678
Argument: 5678
Received: 5678
Sent: 5678
Result: 5678
Ready;
```

RPC Genesend Client

The following is an example of an RPC client program.

```
/* GENESEND.C */
/* Send an integer to the remote host and receive the integer back */
/* PORTMAPPER AND REMOTE SERVER MUST BE RUNNING */

#define VM
#include <stdio.h>
#include <rpc.h>
#include <socket.h>

#define intrcvprog ((u_long)150000)
#define version ((u_long)1)
#define intrcvproc ((u_long)1)

main(argc, argv)
    int argc;
    char *argv[];
{
    int innumber;
    int outnumber;
    int error;

    if (argc != 3) {
        fprintf(stderr, "usage: %s hostname integer\n", argv[0]);
        exit (-1);
    } /* endif */
    innumber = atoi(argv[2]);
    /*
     * Send the integer to the server. The server should
     * return the same integer.
     */
    error = callrpc(argv[1], intrcvprog, version, intrcvproc, xdr_int,
        (char *)&innumber, xdr_int, (char *)&outnumber);

    if (error != 0) {
        fprintf(stderr, "error: callrpc failed: %d \n", error);
        fprintf(stderr, "intrcvprog: %d version: %d intrcvproc: %d",
            intrcvprog, version, intrcvproc);
        exit(1);
    } /* endif */

    printf("value sent: %d    value received: %d\n", innumber, outnumber);
    exit(0);
}
```

RPC Geneserv Server

The following is an example of an RPC server program.

```

/* GENERIC SERVER      */
/* RECEIVE AN INTEGER OR FLOAT AND RETURN THEM RESPECTIVELY */
/* PORTMAPPER MUST BE RUNNING */

#define VM

#include <rpc.h>
#include <stdio.h>

#define intrcvprog ((u_long)150000)
#define fltrcvprog ((u_long)150102)
#define intvers    ((u_long)1)
#define intrcvproc ((u_long)1)
#define fltrcvproc ((u_long)1)
#define fltvers    ((u_long)1)

main()
{
    int *intrcv();
    float *floatrcv();

    /*REGISTER PROG, VERS AND PROC WITH THE PORTMAPPER*/

    /*FIRST PROGRAM*/
    registerrpc(intrcvprog,intvers,intrcvproc,intrcv,xdr_int,xdr_int);
    printf("Intrcv Registration with Port Mapper completed\n");

    /*OR MULTIPLE PROGRAMS*/
    registerrpc(fltrcvprog,fltvers,fltrcvproc,floatrcv,xdr_float,xdr_float);
    printf("Floatrcv Registration with Port Mapper completed\n");

    /*
     * svc_run will handle all requests for programs registered.
     */
    svc_run();
    printf("Error:svc_run returned!\n");
    exit(1);
}

/*
 * Procedure called by the server to receive and return an integer.
 */
int *
intrcv(in)
    int *in;
{
    int *out;

    printf("integer received: %d\n",*in);
    out = in;
    printf("integer being returned: %d\n",*out);
    return (out);
}

/*
 * Procedure called by the server to receive and return a float.
 */
float *
floatrcv(in)
    float *in;
{
    float *out;

    printf("float received: %e\n",*in);
    out=in;
    printf("float being returned: %e\n",*out);
    return(out);
}

```

RPC Rawex Raw Data Stream

The following is an example of an RPC raw data stream program.

```

/*RAWEX
/* AN EXAMPLE OF THE RAW CLIENT/SERVER USAGE */
/* PORTMAPPER MUST BE RUNNING */
/*
 * This program does not access an external interface. It provides
 * a test of the raw RPC interface allowing a client and server
 * program to be in the same process.
 */
#define VM
#include <rpc.h>
#include <stdio.h>
#define rawprog ((u_long)150104)
#define rawvers ((u_long)1)
#define rawproc ((u_long)1)

extern enum clnt_stat clntraw_call();
extern void raw2();

main(argc,argv)
int argc;
char *argv[];
{
    SVCXPRT *transp;
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int bout,in;
    register CLIENT *clnt;
    enum clnt_stat cs;
    int addrlen;

    /*
     * The only argument passed to the program is an integer to
     * be transferred from the client to the server and back.
     */
    if(argc!=2) {
        printf("usage:  %s    integer\n", argv[0]);
        exit(-1);
    }
    in = atoi(argv[1]);

    /*
     * Create the raw transport handle for the server.
     */
    transp = svcraw_create();
    if (transp == NULL) {
        fprintf(stderr, "can't create an RPC server transport\n");
        exit(-1);
    }

    /* In case the program is already registered, deregister it */
    pmap_unset(rawprog, rawvers);

    /* Register the server program with PORTMAPPER */
    if (!svc_register(transp,rawprog,rawvers,raw2, 0)) {
        fprintf(stderr, "can't register service\n");
        exit(-1);
    }
    /*
     * The following registers the transport handle with internal
     * data structures.
     */
    xprt_register(transp);

```

```

/*
 * Create the client transport handle.
 */
if ((clnt = clnt_raw_create(rawprog, rawvers)) == NULL ) {
    clnt_pcreateerror("clntudp_create");
    exit(-1);
}
total_timeout.tv_sec = 60;
total_timeout.tv_usec = 0;
printf("Argument:  %d\n",in);

/*
 * Make the call from the client to the server.
 */
cs=clnt_call(clnt,rawproc,xdr_int,
             (char *)&in,xdr_int,(char *)&bout,total_timeout);

printf("Result:  %d",bout);
if(cs!=0) {
    clnt_perror(clnt,"Client call failed");
    exit(1);
}
exit(0);
}

/*
 * Service procedure called by the server when it receives the client
 * request.
 */
void raw2(rqstp,transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    int in,out;
    if (rqstp->rq_proc==rawproc) {
        /*
         * Unpack the integer passed by the client.
         */
        svc_getargs(transp,xdr_int,&in);
        printf("Received:  %d\n",in);
        /*
         * Send the integer back to the client.
         */
        out=in;
        printf("Sent:  %d\n",out);
        if (!svc_sendreply(transp, xdr_int,&out)) {
            printf("Can't reply to RPC call.\n");
            exit(1);
        }
    }
}
}

```

Chapter 6. SNMP Agent Distributed Programming Interface

The Simple Network Management Protocol (SNMP) agent distributed programming interface (DPI) permits end users to dynamically add, delete, or replace management variables in the local Management Information Base (MIB) without requiring you to recompile the SNMP agent.

SNMP Agents and Subagents

SNMP defines an architecture that consists of network management stations (SNMP clients), network elements (hosts and gateways), and network management agents and subagents. The network management agents perform information management functions, such as gathering and maintaining network performance information and formatting and passing this data to clients when requested. This information is collectively called the Management Information Base (MIB). For more information about clients, agents, and the MIB, see *z/VM: TCP/IP User's Guide*.

A subagent provides an extension to the functionality provided by the SNMP agent. The subagent allows you to define your own MIB variables, which are useful in your environment, and register them with the SNMP agent. When requests for these variables are received by the SNMP agent, the agent passes the request to the subagent. The subagent then returns a response to the agent. The SNMP agent creates an SNMP response packet and sends the response to the remote network management station that initiated the request. The existence of the subagent is transparent to the network management station.

To allow the subagents to perform these functions, the SNMP agent binds to an arbitrarily chosen TCP port and listens for connection requests. A well-known port is not used. Every invocation of the SNMP agent potentially results in a different TCP port being used.

A subagent of the SNMP agent determines the port number by sending a GET request for the MIB variable, which represents the value of the TCP port. The subagent is not required to create and parse SNMP packets, because the DPI C language application program interface (API) has a library routine `query_DPI_port()`. This routine handles the GET request and response called Protocol Data Units (PDUs) necessary to obtain the port number of the TCP port used by the agent for DPI requests. After the subagent obtains the value of the DPI TCP port, it should make a TCP connection to the appropriate port. After a successful `connect()`, the subagent registers the set of variables it supports with the SNMP agent. When all variable classes are registered, the subagent waits for requests from the SNMP agent.

Processing DPI Requests

The SNMP agent can initiate three DPI requests: GET, SET, and GET-NEXT. These requests correspond to the three SNMP requests that a network management station can make. The subagent responds to a request with a response packet. The response packet can be created using the `mkDPIresponse()` library routine, which is part of the DPI API library.

The SNMP subagent can initiate only two requests: REGISTER and TRAP. For an overview of the SNMP DPI, see [Figure 37 on page 242](#).

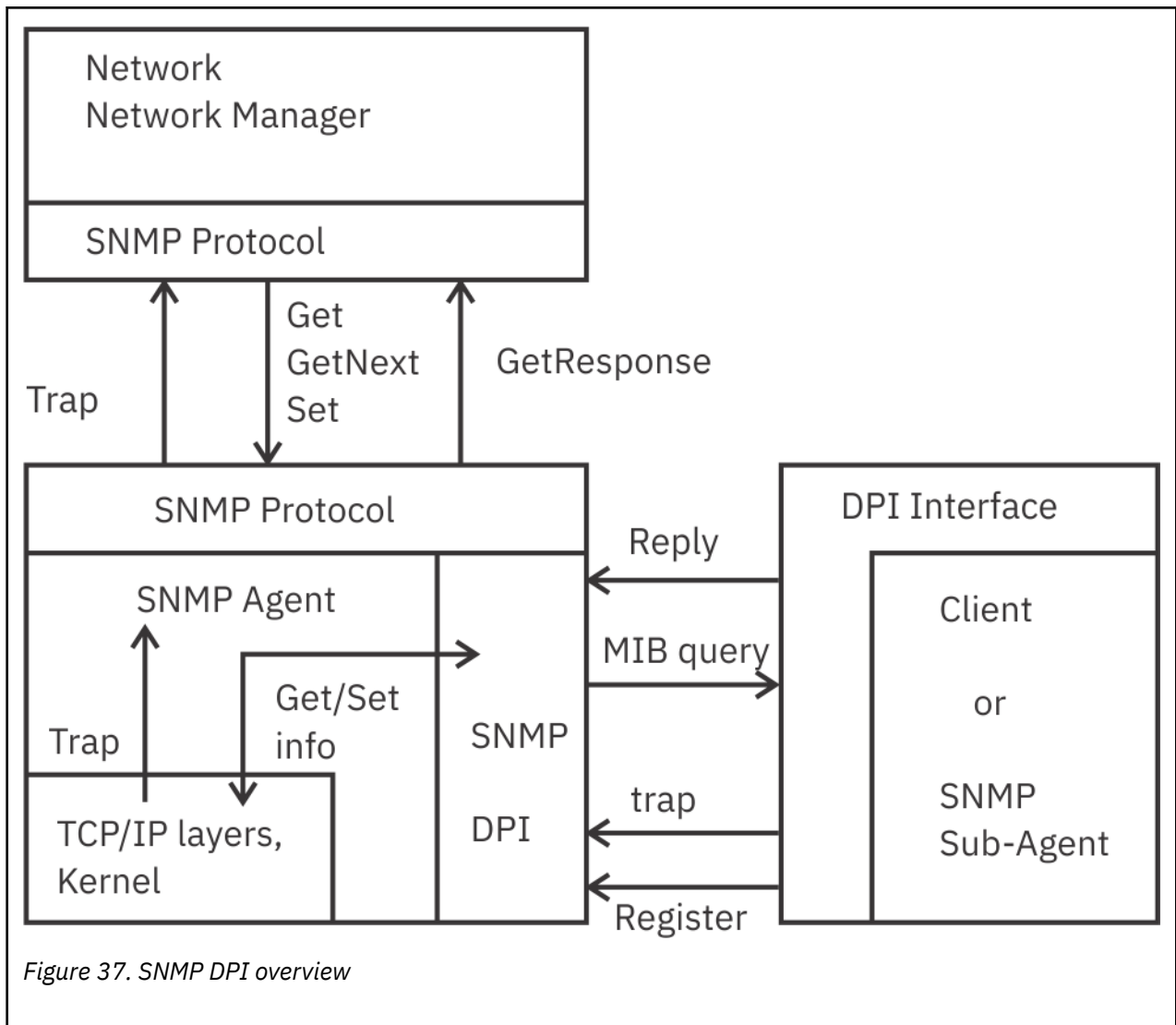


Figure 37. SNMP DPI overview

Note:

1. The SNMP agent communicates with the SNMP manager by the standard SNMP protocol.
2. The SNMP agent communicates with the TCP/IP layers and kernel (operating system) in an implementation-dependent manner. It implements the standard MIB II view.
3. An SNMP Subagent, running as a separate process (potentially even on another machine), can register objects with the SNMP agent.
4. The SNMP agent decodes SNMP Packets. If such a packet contains a Get, GetNext or Set request for an object registered by a subagent, it sends the request to the subagent by a query packet.
5. The SNMP subagent sends responses back by a reply packet.
6. The SNMP agent then encodes the reply into an SNMP packet and sends it back to the requesting SNMP manager.
7. If the subagent wants to report an important state change, it sends a trap packet to the SNMP agent, which encodes it into an SNMP trap packet and sends it to the manager(s).

Processing a GET Request

The DPI packet is parsed, using the `pDIPacket()` routine, to get the object ID of the requested variable. If the specified object ID of the requested variable is not supported by the subagent, the subagent returns an error indication of `SNMP_NO_SUCH_NAME`. Name, type, or value information is not returned. For example:

```
unsigned char *cp;

cp = mkDPIresponse(SNMP_NO_SUCH_NAME,0);
```

If the object ID of the variable is supported, an error is not returned and the name, type, and value of the object ID are returned using the `mkDPIset()` and `mkDPIresponse()` routines. The following is an example of an object ID, whose type is string, being returned.

```
char *obj_id;

unsigned char *cp;
struct dpi_set_packet *ret_value;
char *data;

/* obj_id = object ID of variable, like 1.3.6.1.2.1.1.1 */
/* should be identical to object ID sent in GET request */
data = a string to be returned;
ret_value = mkDPIset(obj_id,SNMP_TYPE_STRING,
                    strlen(data)+1,data);
cp = mkDPIresponse(0,ret_value);
```

Processing a SET Request

Processing a SET request is similar to processing a GET request, but you must pass additional information to the subagent. This additional information consists of the type, length, and value to be set.

If the object ID of the variable is not supported, the subagent returns an error indication of `SNMP_NO_SUCH_NAME`. If the object ID of the variable is supported, but cannot be set, an error indication of `SNMP_READ_ONLY` is returned. If the object ID of the variable is supported, and is successfully set, the message `SNMP_NO_ERROR` is returned.

Processing a GET_NEXT Request

Parsing a GET_NEXT request yields two operands: the object ID of the requested variable and the reason for this request. This allows the subagent to return the name, type, and value of the next supported variable, whose name lexicographically follows that of the passed object ID.

Subagents can support several different groups of the MIB tree. However, the subagent cannot jump from one group to another. You must first determine the reason for the request to then determine the path to traverse in the MIB tree. The second operand contains this reason and is the group prefix of the MIB tree that is supported by the subagent.

If the object ID of the next variable supported by the subagent does not match this group prefix, the subagent must return `SNMP_NO_SUCH_NAME`. If required, the SNMP agent will call on the subagent again and pass a different group prefix.

For example, if you have two subagents, the first subagent registers two group prefixes, A and C, and supports variables A.1, A.2, and C.1. The second subagent registers the group prefix B, and supports variable B.1.

When a remote management station begins dumping the MIB, starting from A, the following sequence of queries is performed.

Subagent 1 is called:

```
get_next(A,A) == A.1
get_next(A.1,A) == A.2
get_next(A.2,A) == error(no such name)
```

Subagent 2 is then called:

```
get_next(A.2,B) == B.1
get_next(B.1,B) == error(no such name)
```

Subagent 1 is then called:

```
get_next(B.1,C) == C.1
get_next(C.1,C) == error(no such name)
```

Processing a REGISTER Request

A subagent must register the variables that it supports with the SNMP agent. Packets can be created using the `mkDPIregister()` routine.

For example:

```
unsigned char *cp;
cp = mkDPIregister('1.3.6.1.2.1.1.2.');
```

Note: Object IDs are registered with a trailing dot ("."). Although DPI 1.0 level did accept an Object ID without a trailing dot, the new level (DPI 1.1) does not.

Processing a TRAP Request

A subagent can request that the SNMP agent generate a TRAP for it. The subagent must provide the desired values for the generic and specific operands of the TRAP. The subagent can optionally provide a name, type, and value operand. The DPI API library routine `mkDPItrap()` can be used to generate the TRAP packet.

Compiling and Linking

To compile your program, you must include the `SNMP_DPI.H` header file.

To compile and link your applications, use the following procedures:

1. To set up the C environment, enter the following commands:

```
SET LDRTBLS nn
GLOBAL LOADLIB SCEERUN
GLOBAL TXTLIB SCEELKED
```

2. To compile your program, enter one of the following commands:

- Place compile options on the CC command:

```
CC filename (def(VM)
```

- Place `#define VM` in the first line of all user's C source files:

```
CC filename
```

3. To generate an executable module, enter the following command:

```
TCPLOAD load_list control_file c (TXTLIB DPILIB
```

Note:

1. Make sure you have access to the IBM C for VM/ESA Compiler and to the TCPMAINT 592 minidisk.
2. For the syntax of the TCPLOAD EXEC, see [Appendix A, "TCPLOAD EXEC,"](#) on page 335 and for the syntax of the SET LDRTBLS command, see [z/VM: CMS Commands and Utilities Reference](#).

SNMP DPI Reference

The following table provides a reference for SNMP DPI. [Table 24 on page 245](#) describes each SNMP DPI routine supported by TCP/IP, and identifies the page in the book where you can find more information.

Table 24. SNMP DPI Reference

SNMP DPI Routine	Description	Location
DPIdebug()	Used to turn some DPI internal tracing on or off.	“DPIdebug()” on page 245
fDPIparse()	Frees a parse tree previously created by a call to pDPIpacket().	“fDPIparse()” on page 245
mkDPIlist()	Creates the portion of the parse tree that represents a list of name and value pairs.	“mkDPIlist()” on page 246
mkDPIregister()	Creates a register request packet and returns a pointer to a static buffer.	“mkDPIregister()” on page 246
mkDPIresponse()	Creates a response packet.	“mkDPIresponse()” on page 247
mkDPIset()	Creates a representation of a parse tree name and value pair.	“mkDPIset()” on page 248
mkDPItrap()	Creates a trap request packet.	“mkDPItrap()” on page 249
mkDPItrape()	Creates an extended trap. Basically the same as the mkDPItrap() routine but allows you to pass a list of variables and an enterprise object ID.	“mkDPItrape()” on page 249
pDPIpacket()	Parses a DPI packet and returns a parse tree representation.	“pDPIpacket()” on page 251
query_DPI_port()	Determines what TCP port is associated with DPI.	“query_DPI_port()” on page 252

DPI Library Routines

This section provides the syntax, operands, and other appropriate information for each DPI routine supported by TCP/IP for z/VM.

DPIdebug()

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

void DPIdebug(onoff)
int *onoff;
```

Operand

Description

onoff

Specifies an integer. A value of 0 turns tracing off and a value of 1 (or nonzero) turns tracing on.

Description: The DPIdebug() routine can be used to turn DPI internal tracing on or off.

fDPIparse()

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

void fDPIparse(hdr)
struct snmp_dpi_hdr *hdr;
```

Operand

Description

mkDPIList()

hdr

Specifies a parse tree.

Description: The `fDPiParse()` routine frees a parse tree that was previously created by a call to `pDPiPacket()`. After calling `fDPiParse()`, no further references to the parse tree can be made.

mkDPIList()

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

struct dpi_set_packet *mkDPIList(packet, oid_name, type, len, value)
struct dpi_set_packet *packet;
char *oid_name;
int type;
int len;
char *value;
```

Operand

Description

packet

Specifies a pointer to a structure `dpi_set_packet`.

oid_name

Specifies the object identifier of the variable.

type

Specifies the type of the value.

len

Specifies the length of the value.

value

Specifies a pointer to the value.

Description: The `mkDPIList()` routine can be used to create the portion of the parse tree that represents a list of name and value pairs. Each entry in the list represents a name and value pair (as would normally be returned in a response packet). If the pointer *packet* is NULL, then a new `dpi_set_packet` structure is dynamically allocated and the pointer to that structure is returned. The structure contains the new name and value pair. If the pointer *packet* is not NULL, then a new `dpi_set_packet` structure is dynamically allocated and chained to the list. The new structure contains the new name and value pair. The pointer *packet* is returned to the caller. If an error is detected, a NULL pointer is returned.

The value of *type* can be the same as for `mkDPISet()`. These values are defined in the *snmp_dpi.h* header file.

As a result, the structure `dpi_set_packet` has changed and now has a next pointer (zero in case of a `mkDPISet()` call and also zero upon the first `mkDPIList()` call). The following is the format of `dpi_set_packet`:

```
struct dpi_set_packet {
    char *object_id;
    unsigned char type;
    unsigned short value_len;
    char *value;
    struct dpi_set_packet *next;
};
```

A subagent writer would normally look only at the `dpi_set_packet` structure when receiving a `SNMP_DPI_SET` request and after having issued a `pDPiPacket()` call.

mkDPISet()

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

unsigned char *mkDPIregister(oid_name)
char *oid_name;
```

Operand Description

oid_name

Specifies the object identifier of the variable to be registered. Object identifiers are registered with a trailing dot (“.”).

Description: The mkDPIregister() routine creates a register request packet and returns a pointer to a static buffer, which holds the packet contents. The length of the remaining packet is stored in the first two bytes of the packet.

Return Values: If successful, returns a pointer to a static buffer containing the packet contents. A NULL pointer is returned if an error is detected during the creation of the packet.

Example: The following is an example of the mkDPIregister() routine.

```
unsigned char *packet;
int len;

/* register sysDescr variable */
packet = mkDPIregister("1.3.6.1.2.1.1.1.");

len = *packet * 256 + *(packet + 1);
len += 2; /* include length bytes */
```

mkDPIresponse()

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

unsigned char *mkDPIresponse(ret_code, value_list)
int ret_code;
struct dpi_set_packet *value_list;
```

Operand Description

ret_code

Determines the error code to be returned.

value_list

Points to a parse tree containing the name, type, and value information to be returned.

Description: The mkDPIresponse() routine creates a response packet. The first operand, *ret_code*, is the error code to be returned. Zero indicates no error. Possible errors include the following:

- SNMP_NO_ERROR
- SNMP_TOO_BIG
- SNMP_NO_SUCH_NAME
- SNMP_BAD_VALUE
- SNMP_READ_ONLY
- SNMP_GEN_ERR

See the SNMP_DPI.H header file for a description of these messages.

mkDPISet()

If *ret_code* does not indicate an error, then the second operand is a pointer to a parse tree created by `mkDPISet()`, which represents the name, type, and value information being returned. If an error is indicated, the second operand is passed as a NULL pointer.

The length of the remaining packet is stored in the first two bytes of the packet.

Note: `mkDPISet()` always frees the passed parse tree.

Return Values: If successful, `mkDPISet()` returns a pointer to a static buffer containing the packet contents. This is the same buffer used by `mkDPISet()`. A NULL pointer is returned if an error is detected during the creation of the packet.

Example: The following is an example of the `mkDPISet()` routine.

```
unsigned char *packet;

int error_code;
struct dpi_set_packet *ret_value;

packet = mkDPISet(error_code, ret_value);

len = *packet * 256 + *(packet + 1);
len += 2; /* include length bytes */
```

mkDPISet()

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

struct dpi_set_packet *mkDPISet(oid_name, type, len, value)
char *oid_name;
int type;
int len;
char *value;
```

Operand	Description
---------	-------------

<i>oid_name</i>	
------------------------	--

	Specifies the object identifier of the variable.
--	--

<i>type</i>	
--------------------	--

	Specifies the type of the object identifier.
--	--

<i>len</i>	
-------------------	--

	Indicates the length of the value.
--	------------------------------------

<i>value</i>	
---------------------	--

	Points to the first byte of the value of the object identifier.
--	---

Description: The `mkDPISet()` routine can be used to create the portion of a parse tree that represents a name and value pair (as would normally be returned in a response packet). It returns a pointer to a dynamically allocated parse tree representing the name, type, and value information. If there is an error detected while creating the parse tree, a NULL pointer is returned.

The value of *type* can be one of the following (which are defined in the `SNMP_DPI.H` header file):

- `SNMP_TYPE_NUMBER`
- `SNMP_TYPE_STRING`
- `SNMP_TYPE_OBJECT`
- `SNMP_TYPE_INTERNET`
- `SNMP_TYPE_COUNTER`
- `SNMP_TYPE_GAUGE`
- `SNMP_TYPE_TICKS`

The *value* operand is always a pointer to the first byte of the object ID's value.

Note: The parse tree is dynamically allocated, and copies are made of the passed operands. After a successful call to `mkDPISet()`, the application can dispose of the passed operands without affecting the contents of the parse tree.

Return Values: Returns a pointer to a parse tree containing the name, type, and value information.

mkDPITrap()

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

unsigned char *mkDPITrap(generic, specific, value_list)
int generic;
int specific;
struct dpi_set_packet *value_list;
```

Operand

Description

generic

Specifies the generic field in the SNMP TRAP packet.

specific

Identifies the specific field in the SNMP TRAP packet.

value_list

Passes the name and value pair to be placed into the SNMP packet.

Description: The `mkDPITrap()` routine creates a TRAP request packet. The information contained in *value_list* is passed as the `set_packet` portion of the parse tree.

The length of the remaining packet is stored in the first two bytes of the packet.

Note: `mkDPITrap()` always frees the passed parse tree.

Return Values: If the packet can be created, a pointer to a static buffer containing the packet contents is returned. This is the same buffer that is used by `mkDPISet()`. If an error is encountered while creating the packet, a NULL pointer is returned.

Example: The following is an example of the `mkDPITrap()` routine.

```
struct dpi_set_packet *if_index_value;
unsigned long data;
unsigned char *packet;
int len;

data = 3; /* interface number = 3 */
if_index_value = mkDPISet("1.3.6.1.2.1.2.2.1.1", SNMP_TYPE_NUMBER,
    sizeof(unsigned long), &data);
packet = mkDPITrap(2, 0, if_index_value);
len = *packet * 256 + *(packet + 1);
len += 2; /* include length bytes */
write(fd, packet, len);
```

mkDPITrape()

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

unsigned char *mkDPITrape(generic, specific, value_list, enterprise_oid)
long int generic; /* 4 octet integer */
long int specific;
struct dpi_set_packet *value_list;
char *enterprise_oid;
```

Operand Description

generic

Specifies the generic field for the SNMP TRAP packet.

specific

Specifies the specific field for the SNMP TRAP packet.

value_list

Specifies a pointer to a structure `dpi_set_packet`, which contains one or more variables to be sent with the SNMP TRAP packet. Or NULL if no variables are to be send.

enterprise_oid

Specifies a pointer to a character string representing the enterprise object ID (in ASN.1 notation, for example, 1.3.6.1.4.1.2.2.1.4). Specifies NULL if you want the SNMP agent to use its own enterprise object ID.

Description: The `mkDPITrape()` routine can be used to create an *extended* trap. An extended trap resembles the `mkDPITrap()` routine, but it allows you to pass a list of variables and an enterprise-object ID.

The structure for `dpi_trap_packet` has changed, but this structure is not exposed to subagent writers.

Example of an Extended Trap

The following is a piece of sample code to send an extended trap. No error checking is done.

```
struct dpi_set_packet *set;
int len;
long int num = 15; /* 4 octet integer */
unsigned long int ctr = 1234;
char str[] = "a string";
unsigned char *packet;

set = 0;
set = mkDPIList(set, "1.3.6.1.4.1.2.2.1.4.1", SNMP_TYPE_NUMBER, sizeof(num), &num);
set = mkDPIList(set, "1.3.6.1.4.1.2.2.1.4.2", SNMP_TYPE_STRING, strlen(str), str);
set = mkDPIList(set, "1.3.6.1.4.1.2.2.1.4.6", SNMP_TYPE_COUNTER, sizeof(ctr), &ctr);

packet = mkDPITrape(6L, 37L, set, "1.3.6.1.4.1.2.2.1.4");

len = *packet * 256 + *(packet+1);
len += 2;

write(fd, packet, len) /* use send on OS/2 */
```

You can use a `mkDPISet()` call to create an initial `dpi_set_packet` for the first name and value pair. So the following sample is equivalent to the one above.

```
struct dpi_set_packet *set;
int len;
long int num = 15; /* 4 octet integer */
unsigned long int ctr = 1234;
char str[] = "a string";
unsigned char *packet;

set = mkDPISet("1.3.6.1.4.1.2.2.1.4.1", SNMP_TYPE_NUMBER, sizeof(num), &num);
set = mkDPIList(set, "1.3.6.1.4.1.2.2.1.4.2", SNMP_TYPE_STRING, strlen(str), str);
set = mkDPIList(set, "1.3.6.1.4.1.2.2.1.4.6", SNMP_TYPE_COUNTER, sizeof(ctr), &ctr);

packet = mkDPITrape(6L, 37L, set, "1.3.6.1.4.1.2.2.1.4");

len = *packet * 256 + *(packet+1);
len += 2;

write(fd, packet, len) /* use send on OS/2 */
```

If the high order bit must be on for the specific trap type, then a negative integer must be passed.

pDPIpacket()

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

struct snmp_dpi_hdr *pDPIpacket(packet)
unsigned char *packet;
```

Operand Description

packet

Specifies the DPI packet to be parsed.

Description: The pDPIpacket() routine parses a DPI packet and returns a parse tree representing its contents. The parse tree is dynamically allocated and contains copies of the information within the DPI packet. After a successful call to pDPIpacket(), the packet can be disposed of in any manner the application chooses, without affecting the contents of the parse tree.

Return Values: If pDPIpacket() is successful, a parse tree is returned. If an error is encountered during the parse, a NULL pointer is returned.

Note: The parse tree structures are defined in the SNMP_DPI.H header file.

Example: The following is an example of the mkDPItrap() routine. The root of the parse tree is represented by an snmp_dpi_hdr structure.

```
struct snmp_dpi_hdr {
    unsigned char proto_major;
    unsigned char proto_minor;
    unsigned char proto_release;

    unsigned char packet_type;
    union {
        struct dpi_get_packet      *dpi_get;
        struct dpi_next_packet     *dpi_next;
        struct dpi_set_packet      *dpi_set;
        struct dpi_resp_packet     *dpi_response;
        struct dpi_trap_packet     *dpi_trap;
    } packet_body;
};
```

The *packet_type* field can have one of the following values, which are defined in the SNMP_DPI.H header file:

- SNMP_DPI_GET
- SNMP_DPI_GET_NEXT
- SNMP_DPI_SET

The *packet_type* field indicates the request that is made of the DPI client. For each of these requests, the remainder of the *packet_body* is different. If a GET request is indicated, the object ID of the desired variable is passed in a dpi_get_packet structure.

```
struct dpi_get_packet {
    char *object_id;
};
```

A GET-NEXT request is similar, but the dpi_next_packet structure also contains the object ID prefix of the group that is currently being traversed.

```
struct dpi_next_packet {
    char *object_id;
    char *group_id;
};
```

query_DPI_port()

If the next object, whose object ID lexicographically follows the object ID indicated by *object_id*, does not begin with the suffix indicated by the *group_id*, the DPI client must return an error indication of `SNMP_NO_SUCH_NAME`.

A SET request has the most data associated with it, and this is contained in a `dpi_set_packet` structure.

```
struct dpi_set_packet {
    char      *object_id;
    unsigned char  type;
    unsigned short value_len;
    char      *value;
};
```

The object ID of the variable to be modified is indicated by *object_id*. The type of the variable is provided in *type* and can have one of the following values:

- `SNMP_TYPE_NUMBER`
- `SNMP_TYPE_STRING`
- `SNMP_TYPE_OBJECT`
- `SNMP_TYPE_EMPTY`
- `SNMP_TYPE_INTERNET`
- `SNMP_TYPE_COUNTER`
- `SNMP_TYPE_GAUGE`
- `SNMP_TYPE_TICKS`

The length of the value to be set is stored in *value_len* and *value* contains a pointer to the value.

Note: The storage pointed to by *value* is reclaimed when the parse tree is freed. The DPI client must make provision for copying the value contents.

query_DPI_port()

```
#include <snmp_dpi.h>

int query_DPI_port (host_name, community_name)
char *host_name;
char *community_name;
```

Operand

Description

host_name

Points to the SNMP agent's host name or internet address.

community_name

Points to the community name to be used when making a request.

Description: The `query_DPI_port()` routine is used by a DPI client to determine the TCP port number that is associated with the DPI. This port number is needed to connect() to the SNMP agent. The port number is obtained through an SNMP GET request. *community_name* and *host_name* are the arguments that are passed to the `query_DPI_port()` routine.

Return Values: An integer representing the TCP port number is returned if successful; a -1 is returned if the port cannot be determined.

Sample SNMP DPI Client Program

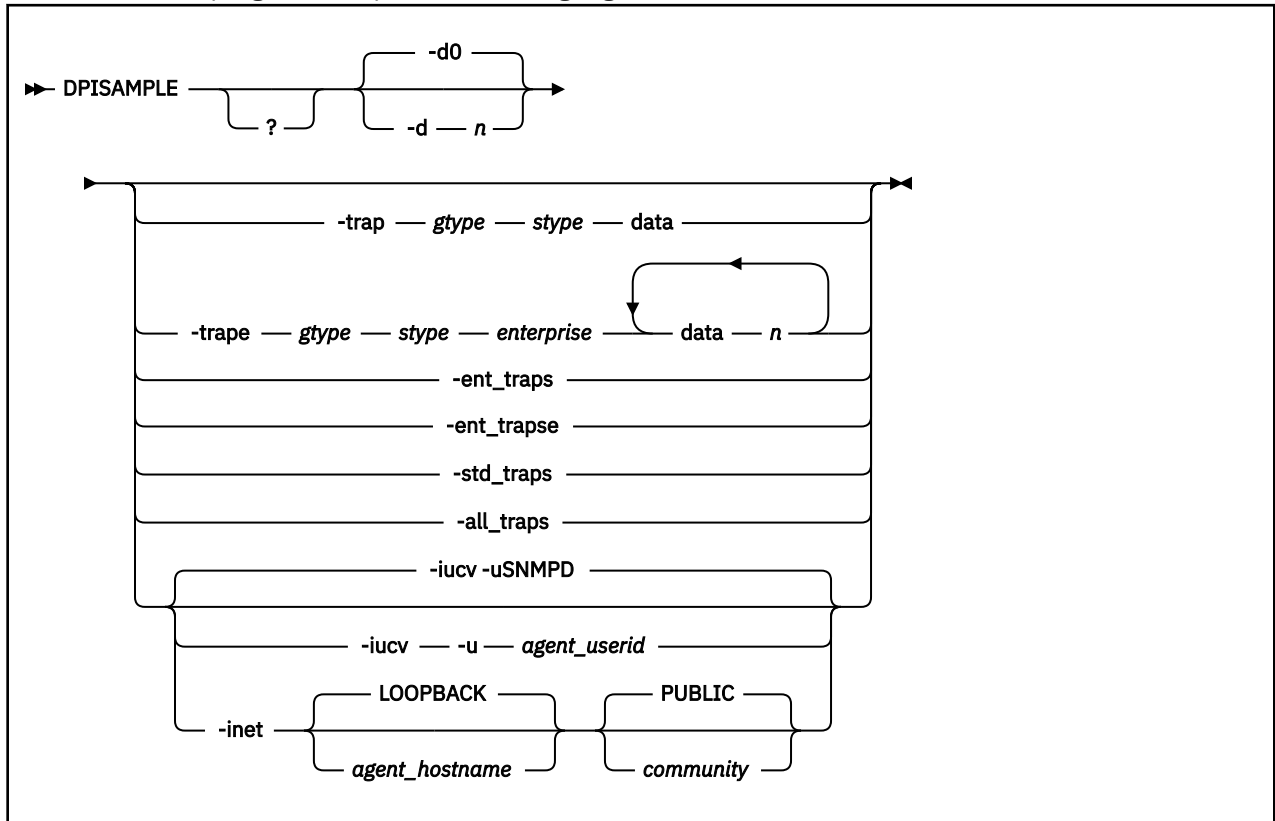
This section provides an example of an SNMP DPI agent program. You can run the `dpisample` program against the SNMP agents that support the SNMP-DPI interface, as described in RFC 1228.

The sample can be used to test agent DPI implementations because it provides variables of all types and also allows you to generate traps of all types.

The DPISAMPLE program implements a set of variables in the DPISAMPLE table which consists of a set of objects in the IBM Research tree (1.4.1.2.2.1.4). See [Figure 38 on page 255](#) for the object type and objectID.

The DPISAMPLE Program (Sample DPI Subagent)

The DPISAMPLE program accepts the following arguments:



Operand Description

?

Invokes output with an explanation about how the *dpisample* command is used. This option should be used in a C-shell environment.

-d *n*

Sets the debug level. The level, *n*, has a range from 0 - 4, 0 is silent and 4 is most verbose. The default level is 0.

-trap

Generates a trap with the following options:

gtype

Specifies the type as generic. The available ranges are 0 - 6.

stype

Specifies the type as specific.

data

Passes data as an additional value for the variable *dpisample.stype.0*. Data is interpreted depending on *stype*. The following list describes the available values for the *stype* operand and their data descriptions:

- 1**
number
- 2**
octet string
- 3**
object id
- 4**
empty (ignored)
- 5**
internet address
- 6**
counter
- 7**
gauge
- 8**
time ticks
- 9**
display string
- other**
octet string

-trape

Generates an extended trap (available with DPI 1.1 level) with the following defined options:

gtype

Specifies the trap as generic. The available ranges are 0 - 6.

stype

Specifies the type as specific.

enterprise

Provides the object ID for the extended trap.

data

Passes data values for additional variables. Data is passed as octet strings. Instances of data can be 1-n.

-ent_traps

Generates nine enterprise-specific traps with *stype* values of 1 - 9, using the internal dpiSample variables as data.

-ent_trapse

Generates nine enterprise-specific traps with *stype* values of 11 - 19, using the internal dpiSample variables as data.

-std_traps

Generates and simulates the standard five SNMP traps (generic types 1 - 5) including the link-down trap.

-all_traps

Generates both the standard traps (-std_traps) and the enterprise-specific traps with *stype* of 1 - 9 (-ent_traps).

-iucv

Specifies that an AF_IUCV socket is to be used to connect to the SNMP agent. The *-iucv* operand is the default.

-u agent_userid

Specifies the user ID where the SNMP agent (SNMPD) is running. The default is SNMPD.

-inet

Specifies that an AF_INET socket is to be used to connect to the SNMP agent.

agent_hostname

Specifies the host name of the system where an SNMP-DPI capable agent is running. The default, if *-inet* is specified, is LOOPBACK.

community_name

Specifies the community name to get the dpiPort. The default is PUBLIC.

DPISAMPLE TABLE

```
# DPISAMPLE.C supports these variables as an SNMP DPI sample sub-agent
# it also generates enterprise specific traps via DPI with these objects.
DPISample          1.3.6.1.4.1.2.2.1.4.    table      0
DPISampleNumber    1.3.6.1.4.1.2.2.1.4.1.    number     10
# next one is to be able to send a badValue with a SET request
DPISampleNumberString 1.3.6.1.4.1.2.2.1.4.1.1. string     10
DPISampleOctetString  1.3.6.1.4.1.2.2.1.4.2.    string     10
DPISampleObjectID    1.3.6.1.4.1.2.2.1.4.3.    object     10
# XGMON/SQESERV does not allow to specify empty (so use empty string)
DPISampleEmpty       1.3.6.1.4.1.2.2.1.4.4.    string     10
DPISampleInetAddress 1.3.6.1.4.1.2.2.1.4.5.    internet   10
DPISampleCounter     1.3.6.1.4.1.2.2.1.4.6.    counter    10
DPISampleGauge       1.3.6.1.4.1.2.2.1.4.7.    gauge      10
DPISampleTimeTicks   1.3.6.1.4.1.2.2.1.4.8.    ticks      10
DPISampleDisplayString 1.3.6.1.4.1.2.2.1.4.9.    display    10
DPISampleCommand     1.3.6.1.4.1.2.2.1.4.10.   display     1
```

Figure 38. DPISAMPLE Table MIB descriptions

Client Sample Program

The following is an example of a SNMP-DPI subagent program.

```
/*
 *
 * SNMP-DPI - SNMP Distributed Programming Interface
 *
 *
 * May 1991 - Version 1.0 - SNMP-DPI Version 1.0 (RFC1228)
 * Created by IBM Research.
 * Feb 1992 - Version 1.1 - Allow enterpriseID to be passed with
 * a (enterprise specific) trap
 * - allow multiple variables to be passed
 * - Use 4 octets (INTEGER from RFC1157)
 * for generic and specific type.
 * Jun 1992 - Make it run on OS/2 as well
 * Note: dpisample = dpisampl on OS/2
 *
 * Copyright None
 *
 * dpisample.c - a sample SNMP-DPI subagent
 * - can be used to test agent DPI implementations.
 *
 * For testing with XGMON and/or SQESERV (SNMP Query Engine)
 * it is best to keep the following define for OID in sync
 * with the dpiSample objectID in the MIB description file
 * (mib_desc for XGMON, MIB_DESC DATA for SQESERV on VM and
 * MIB@DESC.DATA for SQESERV on MVS, MIB2TBL on OS/2).
 */
#define OID "1.3.6.1.4.1.2.2.1.4."
#define ENTERPRISE_OID "1.3.6.1.4.1.2.2.1.4" /* dpiSample */
#define ifIndex "1.3.6.1.2.1.2.2.1.1.0"
#define egpNeighAddr "1.3.6.1.2.8.5.1.2.0"
#define PUBLIC_COMMUNITY_NAME "public"

#if defined(VM) || defined(MVS)

#define SNMPAGENTUSERID "SNMPD"
#define SNMPIUCVNAME "SNMP_DPI"
#pragma csect(CODE, "$DPISAMP")
#pragma csect(STATIC, "#DPISAMP")
#include <manifest.h> /* VM specific things */
#include "snmpnms.h" /* short external names for VM/MVS */
#include "snmp_vm.h" /* more of those short names */
#include <saiucv.h>
#include <bsdtime.h>
#include <bsdtypes.h>
```

```

#include <socket.h>
#include <in.h>
#include <netdb.h>
#include <inet.h>
extern char  ebcdicto[[], asciitoe][;
#pragma linkage(cmxlate,OS)
#define DO_ETOA(a) cmxlate((a),ebcdictoascii,strlen((a)))
#define DO_ATOE(a) cmxlate((a),asciitoebcdic,strlen((a)))
#define DO_ERROR(a) tcperror((a))
#define LOOPBACK "loopback"
#define IUCV TRUE
#define max(a,b) (((a) > (b)) ? (a) : (b))
#define min(a,b) (((a) < (b)) ? (a) : (b))

#else /* we are not on VM or MVS */

#ifdef OS2
#define INCL_DOSPROCESS
#include <stdlib.h>
#include <types.h>
#include <doscalls.h> /* GKS */
#include <os2.h> /* GKS */
#ifdef sleep
#define sleep(a) DosSleep(1000L * (a)) /*GKS*/
#endif
#define close soclose
/*char * malloc(); */
/*unsigned long strtoul(); */
#endif

#include <sys/time.h> /* GKS */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#define DO_ETOA(a) ; /* no need for this */
#define DO_ATOE(a) ; /* no need for this */
#define DO_ERROR(a) perror((a))
#define LOOPBACK "localhost"
#define IUCV FALSE
#ifdef AIX221
#define isdigit(c) (((c) >= '0') && ((c) <= '9'))
#else
#include <sys/select.h>
#endif /* AIX221 */

#endif /* defined(VM) || defined(MVS) */

#include <stdio.h>
#ifdef OS2
#include <dpi/snmp_dpi.h>
#else
#include "snmp_dpi.h"
#endif

#define WAIT_FOR_AGENT 3 /* time to wait before closing agent fd */

#ifdef TRUE
#define TRUE 1
#define FALSE 0
#endif

#ifdef _NO_PROTO /* for classic K&R C */
static void check_arguments();
static void send_packet();
static void print_val();
static void usage();
static void init_connection();
static void init_variables();
static void await_and_read_packet();
static void handle_packet();
static void do_get();
static void do_set();
static void issue_traps();
static void issue_one_trap();
static void issue_one_trap();
static void issue_std_traps();
static void issue_ent_traps();
static void issue_ent_trap();
static void do_register();
static void dump_bfr();

```

```

static struct dpi_set_packet *addtoset();
/extern unsigned long lookup_host();

#else /* _NO_PROTO */                                /* for ANSI-C compiler */

static void check_arguments(const int argc, char *argv[]);
static void send_packet(const char * packet);
static void print_val(const int index);
static void usage(const char *progname, const int exit_rc);
static void init_connection(void);
static void init_variables(void);
static void await_and_read_packet(void);
static void handle_packet(void);
static void do_get(void);
static void do_set(void);
static void issue_traps(void);
static void issue_one_trap(void);
static void issue_one_trapse(void);
static void issue_std_traps(void);
static void issue_ent_traps(void);
static void issue_ent_trapse(void);
static void do_register(void);
static void dump_bfr(const char *buf, const int len);
static struct dpi_set_packet *addtoset(struct dpi_set_packet *data,
                                       int stype);
static unsigned long lookup_host(const char *hostname);

#endif /* _NO_PROTO */

#define OSTRING          "hex01-04:"
#define DSTRING          "Initial Display String"
#define COMMAND          "None"
#define BUFSIZE          4096
#define TIMEOUT          3
#define PACKET_LEN(packet) (((unsigned char)*(packet)) * 256 + \
                             ((unsigned char)*((packet) + 1)) + 2)

/* We have the following instances for OID.x variables */
/* 0 - table */
static long number = 0; /* 1 - a number */
static unsigned char *ostring = 0; /* 2 - octet string */
static int ostring_len = 0; /* and its length */
static unsigned char *objectID = 0; /* 3 - objectID */
static int objectID_len = 0; /* and its length */
/* 4 - some empty variable */
static unsigned long ipaddr = 0; /* 5 - ipaddress */
static unsigned long counter = 1; /* 6 - a counter */
static unsigned long gauge = 1; /* 7 - a gauge */
static unsigned long ticks = 1; /* 8 - time ticks */
static unsigned char *dstring = 0; /* 9 - display string */
static unsigned char *command = 0; /* 10 - command */

static char *DPI_var[] = {
    "dpiSample",
    "dpiSampleNumber",
    "dpiSampleOctetString",
    "dpiSampleObjectID",
    "dpiSampleEmpty",
    "dpiSampleInetAddress",
    "dpiSampleCounter",
    "dpiSampleGauge",
    "dpiSampleTimeTicks",
    "dpiSampleDisplayString",
    "dpiSampleCommand"
};

static short int valid_types[] = { /* SNMP_TYPES accepted on SET */
    -1, /* 0 do not check type */
    SNMP_TYPE_NUMBER, /* 1 number */
    SNMP_TYPE_STRING, /* 2 octet string */
    SNMP_TYPE_OBJECT, /* 3 object identifier */
    -1, /* 4 do not check type */
    SNMP_TYPE_INTERNET, /* 5 internet address */
    SNMP_TYPE_COUNTER, /* 6 counter */
    SNMP_TYPE_GAUGE, /* 7 gauge */
    SNMP_TYPE_TICKS, /* 8 time ticks */
    SNMP_TYPE_STRING, /* 9 display string */
    SNMP_TYPE_STRING, /* 10 command (display string) */
};

#define OID_COUNT_FOR_TRAPS 9
#define OID_COUNT 10
};

```

```

static char      *packet = NULL; /* ptr to send packet. */
static char      inbuf[BUFSIZE]; /* buffer for receive packets */
static int       dpi_fd;         /* fd for socket to DPI agent */
static short int  dpi_port;      /* DPI_port at agent */
static unsigned long dpi_ipaddress; /* IP address of DPI agent */
static char      *dpi_hostname; /* hostname of DPI agent */
static char      *dpi_userid;   /* userid of DPI agent VM/MVS */
static char      *var_gid;      /* groupID received */
static char      *var_oid;      /* objectID received */
static int       var_index;      /* OID variable index */
static unsigned char var_type;   /* SET value type */
static char      *var_value;     /* SET value */
static short int  var_value_len; /* SET value length */
static int       debug_lvl = 0; /* current debug level */
static int       use_iucv = IUCV; /* optional use of AF_IUCV */
static int       do_quit = FALSE; /* Quit in await loop */
static int       trap_gtype = 0; /* trap generic type */
static int       trap_stype = 0; /* trap specific type */
static char      *trap_data = NULL; /* trap data */
static int       do_trap = 0; /* switch for traps */
#define ONE_TRAP 1
#define ONE_TRAPE 2
#define STD_TRAPS 3
#define ENT_TRAPS 4
#define ENT_TRAPSE 5
#define ALL_TRAPS 6
#define MAX_TRAPE_DATA 10 /* data for extended trap */
static long      trape_gtype = 6; /* trap generic type */
static long      trape_stype = 11; /* trap specific type */
static char      *trape_eprise = NULL; /* enterprise id */
static char      *trape_data[MAX_TRAPE_DATA]; /* pointers to data values */
static int       trape_datacnt; /* actual number of values */

#ifdef _NO_PROTO /* for classic K&R C */
main(argc, argv) /* main line */
int  argc;
char *argv[];
#else /* _NO_PROTO */ /* for ANSI-C compiler */
main(const int argc, char *argv[]) /* main line */
#endif /* _NO_PROTO */
{
    check_arguments(argc, argv); /* check callers arguments */
    dpi_ipaddress = lookup_host(dpi_hostname); /* get ip address */
    init_connection(); /* connect to specified agent */
    init_variables(); /* initialize our variables */
    if (do_trap) { /* we just need to do traps */
        issue_traps(); /* issue the trap(s) */
        sleep(WAIT_FOR_AGENT); /* sleep a bit, so agent can */
        close(dpi_fd); /* read data before we close */
        exit(0); /* and that's it */
    } /* end if (do_trap) */
    do_register(); /* register our objectIDs */
    printf("%s ready and awaiting queries from agent\n", argv[0]);
    while (do_quit == FALSE) { /* forever until quit or error */
        await_and_read_packet(); /* wait for next packet */
        handle_packet(); /* handle it */
        if (do_trap) issue_traps(); /* request to issue traps */
    } /* while loop */
    sleep(WAIT_FOR_AGENT); /* allow agent to read response */
    printf("Quitting, %s set to: quit\n", DPI_var[10]);
    exit(2); /* sampleDisplayString == quit */
}

#ifdef _NO_PROTO /* for classic K&R C */
static void issue_traps()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_traps(void)
#endif /* _NO_PROTO */
{
    switch (do_trap) { /* let's see which one(s) */
        case ONE_TRAP: /* only need to issue one trap */
            issue_one_trap(); /* go issue the one trap */
            break;
        case ONE_TRAPE: /* only need to issue one trape */
            issue_one_trape(); /* go issue the one trape */
            break;
        case STD_TRAPS: /* only need to issue std traps */
            issue_std_traps(); /* standard traps gtypes 0-5 */
            break;
        case ENT_TRAPS: /* only need to issue ent traps */
            issue_ent_traps(); /* enterprise specific traps */
            break;
    }
}

```

```

    case ENT_TRAPSE:                /* only need to issue ent trapse */
        issue_ent_trapse();        /* enterprise specific trapse */
        break;
    case ALL_TRAPS:                 /* only need to issue std traps */
        issue_std_trapse();        /* standard traps gtypes 0-5 */
        issue_ent_trapse();        /* enterprise specific traps */
        break;
    default:
        break;
}                                  /* end switch (do_trap) */
do_trap = 0;                      /* reset do_trap switch */
}

#ifdef _NO_PROTO                  /* for classic K&R C */
static void await_and_read_packet() /* await packet from DPI agent */
#else /* _NO_PROTO */             /* for ANSI-C compiler */
static void await_and_read_packet(void) /* await packet from DPI agent */
#endif /* _NO_PROTO */
{
    int len, rc, bytes_to_read, bytes_read = 0;
#ifdef OS2
    int socks[5];
#else
    fd_set read_mask;
#endif
    struct timeval timeout;

#ifdef OS2
    socks[0] = dpi_fd;
    rc = select(socks, 1, 0, 0, -1L);
#else
    FD_ZERO(&read_mask);
    FD_SET(dpi_fd, &read_mask);          /* wait for data */
    rc = select(dpi_fd+1, &read_mask, NULL, NULL, NULL);
#endif
    if (rc != 1) {                    /* exit on error */
        DO_ERROR("await_and_read_packet: select");
        close(dpi_fd);
        exit(1);
    }
#ifdef OS2
    len = recv(dpi_fd, inbuf, 2, 0);    /* read 2 bytes first */
#else
    len = read(dpi_fd, inbuf, 2);        /* read 2 bytes first */
#endif
    if (len <= 0) {                  /* exit on error or EOF */
        if (len < 0) DO_ERROR("await_and_read_packet: read");
        else printf("Quitting, EOF received from DPI-agent\n");
        close(dpi_fd);
        exit(1);
    }
    bytes_to_read = (inbuf[0] << 8) + inbuf[1]; /* bytes to follow */
    if (BUFSIZE < (bytes_to_read + 2)) { /* exit if too much */
        printf("Quitting, packet larger than %d byte buffer\n", BUFSIZE);
        close(dpi_fd);
        exit(1);
    }
    while (bytes_to_read > 0) {        /* while bytes to read */
#ifdef OS2
        socks[0] = dpi_fd;
        len = select(socks, 1, 0, 0, 3000L);
#else
        timeout.tv_sec = 3;           /* wait max 3 seconds */
        timeout.tv_usec = 0;
        FD_SET(dpi_fd, &read_mask);  /* check for data */
        len = select(dpi_fd+1, &read_mask, NULL, NULL, &timeout);
#endif
        if (len == 1) {               /* select returned OK */
#ifdef OS2
            len = recv(dpi_fd, &inbuf[2] + bytes_read, bytes_to_read, 0);
#else
            len = read(dpi_fd, &inbuf[2] + bytes_read, bytes_to_read);
#endif
        } /* end if (len == 1) */
        if (len <= 0) {               /* exit on error or EOF */
            if (len < 0) DO_ERROR("await_and_read_packet: read");
            printf("Can't read remainder of packet\n");
            close(dpi_fd);
            exit(1);
        } else {                     /* count bytes_read */
            bytes_read += len;
        }
    }
}

```

```

        bytes_to_read -= len;
    }
} /* while (bytes_to_read > 0) */
}

#ifdef _NO_PROTO                                /* for classic K&R C */
static void handle_packet() /* handle DPI packet from agent */
#else /* _NO_PROTO */      /* for ANSI-C compiler */
static void handle_packet(void) /* handle DPI packet from agent */
#endif /* _NO_PROTO */
{
    struct snmp_dpi_hdr *hdr;

    if (debug_lvl > 2) {
        printf("Received following SNMP-DPI packet:\n");
        dump_bfr(inbuf, PACKET_LEN(inbuf));
    }
    hdr = pDPIpacket(inbuf); /* parse received packet */
    if (hdr == 0) { /* ignore if can't parse */
        printf("Ignore received packet, could not parse it!\n");
        return;
    }
    packet = NULL;
    var_type = 0;
    var_oid = "";
    var_gid = "";
    switch (hdr->packet_type) {
        /* extract pointers and/or data from specific packet types, */
        /* such that we can use them independent of packet type. */
        case SNMP_DPI_GET:
            if (debug_lvl > 0) printf("SNMP_DPI_GET for ");
            var_oid = hdr->packet_body.dpi_get->object_id;
            break;
        case SNMP_DPI_GET_NEXT:
            if (debug_lvl > 0) printf("SNMP_DPI_GET_NEXT for ");
            var_oid = hdr->packet_body.dpi_next->object_id;
            var_gid = hdr->packet_body.dpi_next->group_id;
            break;
        case SNMP_DPI_SET:
            if (debug_lvl > 0) printf("SNMP_DPI_SET for ");
            var_value_len = hdr->packet_body.dpi_set->value_len;
            var_value = hdr->packet_body.dpi_set->value;
            var_oid = hdr->packet_body.dpi_set->object_id;
            var_type = hdr->packet_body.dpi_set->type;
            break;
        default: /* Return a GEN_ERROR */
            if (debug_lvl > 0) printf("Unexpected packet_type %d, genErr\n",
                                     hdr->packet_type);
            packet = mkDPIresponse(SNMP_GEN_ERR, NULL);
            fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
            send_packet(packet);
            return;
            break;
    } /* end switch(hdr->packet_type) */
    if (debug_lvl > 0) printf("objectID: %s \n", var_oid);

    if (strlen(var_oid) <= strlen(OID)) { /* not in our tree */
        if (hdr->packet_type == SNMP_DPI_GET_NEXT) var_index = 0; /* OK */
        else { /* cannot handle */
            if (debug_lvl > 0) printf("...Ignored %s, noSuchName\n", var_oid);
            packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
            fDPIparse(hdr); /* return storage allocated by pDPIpacket() */
            send_packet(packet);
            return;
        }
    }
    else { /* Extract our variable index (from OID.index.instance) */
        /* We handle any instance the same (we only have one instance) */
        var_index = atoi(&var_oid[strlen(OID)]);
    }
    if (debug_lvl > 1) {
        printf("...The groupID=%s\n", var_gid);
        printf("...Handle as if objectID=%s%d\n", OID, var_index);
    }
    switch (hdr->packet_type) {
        case SNMP_DPI_GET:
            do_get(); /* do a get to return response */
            break;
        case SNMP_DPI_GET_NEXT:
            { char toid[256]; /* space for temporary objectID */
              var_index++; /* do a get for the next variable */
              sprintf(toid, "%s%d", OID, var_index); /* construct objectID */
              var_oid = toid; /* point to it */
            }
    }
}

```

```

    do_get();                /* do a get to return response */
} break;
case SNMP_DPI_SET:
    if (debug_lvl > 1) printf("...value_type=%d\n",var_type);
    do_set();                /* set new value first */
    if (packet) break;       /* some error response was generated */
    do_get();                /* do a get to return response */
    break;
}
fDPiParse(hdr);             /* return storage allocated by pDPiPacket() */
}

#ifdef _NO_PROTO
static void do_get()         /* for classic K&R C */
/* handle SNMP_GET request */
#else /* _NO_PROTO */
static void do_get(void)     /* for ANSI-C compiler */
/* handle SNMP_GET request */
#endif /* _NO_PROTO */
{
    struct dpi_set_packet *data = NULL;

    switch (var_index) {
    case 0: /* table, cannot be queried by itself */
        printf("...Should not issue GET for table %s.0\n", OID);
        break;
    case 1: /* a number */
        data = mkDPiSet(var_oid,SNMP_TYPE_NUMBER,sizeof(number),&number);
        break;
    case 2: /* an octet_string (can have binary data) */
        data = mkDPiSet(var_oid,SNMP_TYPE_STRING,ostring_len,ostring);
        break;
    case 3: /* object id */
        data = mkDPiSet(var_oid,SNMP_TYPE_OBJECT,objectID_len,objectID);
        break;
    case 4: /* some empty variable */
        data = mkDPiSet(var_oid,SNMP_TYPE_EMPTY,0,NULL);
        break;
    case 5: /* internet address */
        data = mkDPiSet(var_oid,SNMP_TYPE_INTERNET,sizeof(ipaddr),&ipaddr);
        break;
    case 6: /* counter (unsigned) */
        data =mkDPiSet(var_oid,SNMP_TYPE_COUNTER,sizeof(counter),&counter);
        break;
    case 7: /* gauge (unsigned) */
        data = mkDPiSet(var_oid,SNMP_TYPE_GAUGE,sizeof(gauge),&gauge);
        break;
    case 8: /* time ticks (unsigned) */
        data = mkDPiSet(var_oid,SNMP_TYPE_TICKS,sizeof(ticks),&ticks);
        break;
    case 9: /* a display_string (printable ascii only) */
        DO_ETOA(dstring);
        data = mkDPiSet(var_oid,SNMP_TYPE_STRING,strlen(dstring),dstring);
        DO_ATOE(dstring);
        break;
    case 10: /* a command request (command is a display string) */
        DO_ETOA(command);
        data = mkDPiSet(var_oid,SNMP_TYPE_STRING,strlen(command),command);
        DO_ATOE(command);
        break;
    default: /* Return a NoSuchName */
        if (debug_lvl > 1)
            printf("...GET]NEXT[ for %s, not found\n", var_oid);
        break;
    } /* end switch (var_index) */

    if (data) {
        if (debug_lvl > 0) {
            printf("...Sending response oid: %s type: %d\n",
                var_oid, data->type);
            printf(".....Current value: ");
            print_val(var_index); /* prints \n at end */
        }
        packet = mkDPiResponse(SNMP_NO_ERROR,data);
    } else { /* Could have been an error in mkDPiSet though */
        if (debug_lvl > 0) printf("...Sending response noSuchName\n");
        packet = mkDPiResponse(SNMP_NO_SUCH_NAME,NULL);
    } /* end if (data) */
    if (packet) send_packet(packet);
}

#ifdef _NO_PROTO
static void do_set()         /* for classic K&R C */
/* handle SNMP_SET request */
#else /* _NO_PROTO */
static void do_set(void)     /* for ANSI-C compiler */

```

```

static void do_set(void) /* handle SNMP_SET request */
#endif /* _NO_PROTO */
{
    unsigned long *ulp;
    long *lp;

    if (valid_types[var_index] != var_type &&
        valid_types[var_index] != -1) {
        printf("...Ignored set request with type %d, expect type %d,",
            var_type, valid_types[var_index]);
        printf(" Returning badValue\n");
        packet = mkDPIresponse(SNMP_BAD_VALUE, NULL);
        if (packet) send_packet(packet);
        return;
    }
    switch (var_index) {
        case 0: /* table, cannot set table. */
            if (debug_lvl > 0) printf("...Ignored set TABLE, noSuchName\n");
            packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
            break;
        case 1: /* a number */
            lp = (long *)var_value;
            number = *lp;
            break;
        case 2: /* an octet_string (can have binary data) */
            free(ostring);
            ostring = (char *)malloc(var_value_len + 1);
            bcopy(var_value, ostring, var_value_len);
            ostring_len = var_value_len;
            ostring[var_value_len] = '\0'; /* so we can use it as a string */
            break;
        case 3: /* object id */
            free(objectID);
            objectID = (char *)malloc(var_value_len + 1);
            bcopy(var_value, objectID, var_value_len);
            objectID_len = var_value_len;
            if (objectID[objectID_len - 1]) {
                objectID[objectID_len++] = '\0'; /* a valid one needs a null */
                if (debug_lvl > 0)
                    printf("...added a terminating null to objectID\n");
            }
            break;
        case 4: /* an empty variable, cannot set */
            if (debug_lvl > 0) printf("...Ignored set EMPTY, readOnly\n");
            packet = mkDPIresponse(SNMP_READ_ONLY, NULL);
            break;
        case 5: /* Internet address */
            ulp = (unsigned long *)var_value;
            ipaddr = *ulp;
            break;
        case 6: /* counter (unsigned) */
            ulp = (unsigned long *)var_value;
            counter = *ulp;
            break;
        case 7: /* gauge (unsigned) */
            ulp = (unsigned long *)var_value;
            gauge = *ulp;
            break;
        case 8: /* time ticks (unsigned) */
            ulp = (unsigned long *)var_value;
            ticks = *ulp;
            break;
        case 9: /* a display_string (printable ascii only) */
            free(dstring);
            dstring = (char *)malloc(var_value_len + 1);
            bcopy(var_value, dstring, var_value_len);
            dstring[var_value_len] = '\0'; /* so we can use it as a string */
            DO_ATOE(dstring);
            break;
        case 10: /* a request to execute a command */
            free(command);
            command = (char *)malloc(var_value_len + 1);
            bcopy(var_value, command, var_value_len);
            command[var_value_len] = '\0'; /* so we can use it as a string */
            DO_ATOE(command);
            if (strcmp("all_traps", command) == 0) do_trap = ALL_TRAPS;
            else if (strcmp("std_traps", command) == 0) do_trap = STD_TRAPS;
            else if (strcmp("ent_traps", command) == 0) do_trap = ENT_TRAPS;
            else if (strcmp("ent_trapse", command) == 0) do_trap = ENT_TRAPSE;
            else if (strcmp("all_traps", command) == 0) do_trap = ALL_TRAPS;
            else if (strcmp("quit", command) == 0) do_quit = TRUE;
            else break;
    }
}

```

```

        if (debug_lvl > 0)
            printf("...Action requested: %s set to: %s\n",
                DPI_var[10], command);
        break;
    default: /* NoSuchName */
        if (debug_lvl > 0)
            printf("...Ignored set for %s, noSuchName\n", var_oid);
        packet = mkDPIresponse(SNMP_NO_SUCH_NAME, NULL);
        break;
    } /* end switch (var_index) */
    if (packet) send_packet(packet);
}

#ifdef _NO_PROTO /* for classic K&R C */
static void issue_std_traps()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_std_traps(void)
#endif /* _NO_PROTO */
{
    trap_stype = 0;
    trap_data = dpi_hostname;
    for (trap_gtype=0; trap_gtype<6; trap_gtype++) {
        issue_one_trap();
        if (trap_gtype == 0) sleep(10); /* some managers purge cache */
    }
}

#ifdef _NO_PROTO /* for classic K&R C */
static void issue_ent_traps()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_ent_traps(void)
#endif /* _NO_PROTO */
{
    char temp_string[256];

    trap_gtype = 6;
    for (trap_stype = 1; trap_stype < 10; trap_stype++) {
        trap_data = temp_string;
        switch (trap_stype) {
            case 1 :
                sprintf(temp_string, "%ld", number);
                break;
            case 2 :
                sprintf(temp_string, "%s", ostring);
                break;
            case 3 :
                trap_data = objectID;
                break;
            case 4 :
                trap_data = "";
                break;
            case 5 :
                trap_data = dpi_hostname;
                break;
            case 6 :
                sleep(1); /* give manager a break */
                sprintf(temp_string, "%lu", counter);
                break;
            case 7 :
                sprintf(temp_string, "%lu", gauge);
                break;
            case 8 :
                sprintf(temp_string, "%lu", ticks);
                break;
            case 9 :
                trap_data = dstring;
                break;
        } /* end switch (trap_stype) */
        issue_one_trap();
    }
}

/* issue a set of extended traps, pass enterprise ID and multiple
 * variable (assume octet string) as passed by caller
 */
#ifdef _NO_PROTO /* for classic K&R C */
static void issue_ent_trapse()
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void issue_ent_trapse(void)
#endif /* _NO_PROTO */
{
    int i, n;

```

```

struct dpi_set_packet *data = NULL;
unsigned char *packet = NULL;
unsigned long ipaddr, ulnum;
char oid[256];
char *cp;

trape_gtype = 6;
trape_eprise = ENTERPRISE_OID;
for (n=11; n < (11+OID_COUNT_FOR_TRAPS); n++) {
    data = 0;
    trape_stype = n;
    for (i=1; i<=(n-10); i++)
        data = addtoset(data, i);
    if (data == 0) {
        printf("Could not make dpi_set_packet\n");
        return;
    }
    packet = mkDPITrape(trape_gtype, trape_stype, data, trape_eprise);
    if ((debug_lvl > 0) && (packet)) {
        printf("sending trape packet: %lu %lu enterprise=%s\n",
            trape_gtype, trape_stype, trape_eprise);
    }
    if (packet) send_packet(packet);
    else printf("Could not make trape packet\n");
}
}

/* issue one extended trap, pass enterprise ID and multiple
 * variable (assume octet string) as passed by caller
 */
#ifdef _NO_PROTO
static void issue_one_trap() /* for classic K&R C */
#else /* _NO_PROTO */
static void issue_one_trap(void) /* for ANSI-C compiler */
#endif /* _NO_PROTO */
{
    struct dpi_set_packet *data = NULL;
    unsigned char *packet = NULL;
    char oid[256];
    char *cp;
    int i;

    for (i=0; i<trape_datacnt; i++) {
        sprintf(oid, "%s2.%d", OID, i);
        /* assume an octet_string (could have hex data) */
        data = mkDPIList(data, oid, SNMP_TYPE_STRING,
            strlen(trape_data[i]), trape_data[i]);
        if (data == 0) {
            printf("Could not make dpiset_packet\n");
        } else if (debug_lvl > 0) {
            printf("Preparing: ]oid=%s[ value: ", oid);
            printf("");
            for (cp = trape_data[i]; *cp; cp++) /* loop through data */
                printf("%2.2x", *cp); /* hex print one byte */
            printf("H\n");
        }
    }
    packet = mkDPITrape(trape_gtype, trape_stype, data, trape_eprise);
    if ((debug_lvl > 0) && (packet)) {
        printf("sending trape packet: %lu %lu enterprise=%s\n",
            trape_gtype, trape_stype, trape_eprise);
    }
    if (packet) send_packet(packet);
    else printf("Could not make trape packet\n");
}

#ifdef _NO_PROTO
static void issue_one_trap() /* for classic K&R C */
#else /* _NO_PROTO */
static void issue_one_trap(void) /* for ANSI-C compiler */
#endif /* _NO_PROTO */
{
    long int num; /* must be 4 bytes */
    struct dpi_set_packet *data = NULL;
    unsigned char *packet = NULL;
    unsigned long ipaddr, ulnum;
    char oid[256];
    char *cp;

    switch (trap_gtype) {
        /* all traps are handled more or less the same so far. */
        /* could put specific handling here if needed/wanted. */
    }

```

```

case 0: /* simulate cold start */
case 1: /* simulate warm start */
case 4: /* simulate authentication failure */
    strcpy(oid,"none");
    break;
case 2: /* simulate link down */
case 3: /* simulate link up */
    strcpy(oid,ifIndex);
    num = 1;
    data = mkDPISet(oid, SNMP_TYPE_NUMBER, sizeof(num), &num);
    break;
case 5: /* simulate EGP neighbor loss */
    strcpy(oid,egpNeighAddr);
    ipaddr = lookup_host(trap_data);
    data = mkDPISet(oid, SNMP_TYPE_INTERNET, sizeof(ipaddr), &ipaddr);
    break;
case 6: /* simulate enterprise specific trap */
    sprintf(oid,"%s%d.0",OID, trap_stype);
    switch (trap_stype) {
    case 1: /* a number */
        num = strtol(trap_data,(char **)0,10);
        data = mkDPISet(oid, SNMP_TYPE_NUMBER, sizeof(num), &num);
        break;
    case 2: /* an octet_string (could have hex data) */
        data = mkDPISet(oid,SNMP_TYPE_STRING,strlen(trap_data),trap_data);
        break;
    case 3: /* object id */
        data = mkDPISet(oid,SNMP_TYPE_OBJECT,strlen(trap_data) + 1,
            trap_data);
        break;
    case 4: /* an empty variable value */
        data = mkDPISet(oid, SNMP_TYPE_EMPTY, 0, 0);
        break;
    case 5: /* internet address */
        ipaddr = lookup_host(trap_data);
        data = mkDPISet(oid, SNMP_TYPE_INTERNET, sizeof(ipaddr), &ipaddr);
        break;
    case 6: /* counter (unsigned) */
        ulnum = strtoul(trap_data,(char **)0,10);
        data = mkDPISet(oid, SNMP_TYPE_COUNTER, sizeof(ulnum), &ulnum);
        break;
    case 7: /* gauge (unsigned) */
        ulnum = strtoul(trap_data,(char **)0,10);
        data = mkDPISet(oid, SNMP_TYPE_GAUGE, sizeof(ulnum), &ulnum);
        break;
    case 8: /* time ticks (unsigned) */
        ulnum = strtoul(trap_data,(char **)0,10);
        data = mkDPISet(oid, SNMP_TYPE_TICKS, sizeof(num), &ulnum);
        break;
    case 9: /* a display_string (ascii only) */
        DO_ETOA(trap_data);
        data = mkDPISet(oid,SNMP_TYPE_STRING,strlen(trap_data),trap_data);
        DO_ATOE(trap_data);
        break;
    default: /* handle as string */
        printf("Unknown specific trap type: %s, assume octet_string\n",
            trap_stype);
        data = mkDPISet(oid,SNMP_TYPE_STRING,strlen(trap_data),trap_data);
        break;
    } /* end switch (trap_stype) */
    break;
default: /* unknown trap */
    printf("Unknown general trap type: %s\n", trap_gtype);
    return;
    break;
} /* end switch (trap_gtype) */

packet = mkDPITrap(trap_gtype,trap_stype,data);
if ((debug_lvl > 0) && (packet)) {
    printf("sending trap packet: %u %u ]oid=%s[ value: ",
        trap_gtype, trap_stype, oid);
    if (trap_stype == 2) {
        printf("");
        for (cp = trap_data; *cp; cp++) /* loop through data */
            printf("%2.2x",*cp); /* hex print one byte */
        printf("H\n");
    } else printf("%s\n", trap_data);
}
if (packet) send_packet(packet);
else printf("Could not make trap packet\n");
}

```

```

#ifdef _NO_PROTO                                /* for classic K&R C */
static void send_packet(packet)                  /* DPI packet to agent */
char *packet;
#else /* _NO_PROTO */                            /* for ANSI-C compiler */
static void send_packet(const char *packet)      /* DPI packet to agent */
#endif /* _NO_PROTO */
{
    int rc;

    if (debug_lvl > 2) {
        printf("...Sending DPI packet:\n");
        dump_bfr(packet, PACKET_LEN(packet));
    }
#ifdef OS2
    rc = send(dpi_fd, packet, PACKET_LEN(packet), 0);
#else
    rc = write(dpi_fd, packet, PACKET_LEN(packet));
#endif
    if (rc != PACKET_LEN(packet)) DO_ERROR("send_packet: write");
    /* no need to free packet (static buffer in mkDPI.... routine) */
}

#ifdef _NO_PROTO                                /* for classic K&R C */
static void do_register() /* register our objectIDs with agent */
#else /* _NO_PROTO */                            /* for ANSI-C compiler */
static void do_register(void) /* register our objectIDs with agent */
#endif /* _NO_PROTO */
{
    int i, rc;
    char toid[256];

    if (debug_lvl > 0) printf("Registering variables:\n");
    for (i=1; i<=OID_COUNT; i++) {
        sprintf(toid, "%s%d.", OID, i);
        packet = mkDPIregister(toid);
#ifdef OS2
        rc = send(dpi_fd, packet, PACKET_LEN(packet), 0);
#else
        rc = write(dpi_fd, packet, PACKET_LEN(packet));
#endif
        if (rc <= 0) {
            DO_ERROR("do_register: write");
            printf("Quitting, unsuccessful register for %s\n", toid);
            close(dpi_fd);
            exit(1);
        }
        if (debug_lvl > 0) {
            printf("...Registered: %-25s oid: %s\n", DPI_var[i], toid);
            printf(".....Initial value: ");
            print_val(i); /* prints \n at end */
        }
    }
}

/* add specified variable to list of variable in the dpi_set_packet
*/
#ifdef _NO_PROTO                                /* for classic K&R C */
struct dpi_set_packet *addtoset(data, stype)
struct dpi_set_packet *data;
int stype;
#else /* _NO_PROTO */                            /* for ANSI-C compiler */
struct dpi_set_packet *addtoset(struct dpi_set_packet *data, int stype)
#endif /* _NO_PROTO */
{
    char var_oid[256];

    sprintf(var_oid, "%s%d.0", OID, stype);
    switch (stype) {
        case 1: /* a number */
            data = mkDPIlist(data, var_oid, SNMP_TYPE_NUMBER,
                             sizeof(number), &number);
            break;
        case 2: /* an octet_string (can have binary data) */
            data = mkDPIlist(data, var_oid, SNMP_TYPE_STRING,
                             ostring_len, ostring);
            break;
        case 3: /* object id */
            data = mkDPIlist(data, var_oid, SNMP_TYPE_OBJECT,
                             objectID_len, objectID);
            break;
        case 4: /* some empty variable */
            data = mkDPIlist(data, var_oid, SNMP_TYPE_EMPTY, 0, NULL);
    }
}

```

```

        break;
    case 5: /* internet address */
        data = mkDPList(data, var_oid, SNMP_TYPE_INETNET,
                        sizeof(ipaddr), &ipaddr);
        break;
    case 6: /* counter (unsigned) */
        data = mkDPList(data, var_oid, SNMP_TYPE_COUNTER,
                        sizeof(counter), &counter);
        break;
    case 7: /* gauge (unsigned) */
        data = mkDPList(data, var_oid, SNMP_TYPE_GAUGE,
                        sizeof(gauge), &gauge);
        break;
    case 8: /* time ticks (unsigned) */
        data = mkDPList(data, var_oid, SNMP_TYPE_TICKS,
                        sizeof(ticks), &ticks);
        break;
    case 9: /* a display_string (printable ascii only) */
        DO_ETOA(dstring);
        data = mkDPList(data, var_oid, SNMP_TYPE_STRING,
                        strlen(dstring), dstring);
        DO_ATOE(dstring);
        break;
    } /* end switch (stype) */
    return(data);
}

#ifdef _NO_PROTO /* for classic K&R C */
static void print_val(index)
int index;
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void print_val(const int index)
#endif /* _NO_PROTO */
{
    char *cp;
    struct in_addr display_ipaddr;

    switch (index) {
    case 1 :
        printf("%ld\n", number);
        break;
    case 2 :
        printf("");
        for (cp = ostring; cp < ostring + ostring_len; cp++)
            printf("%2.2x", *cp);
        printf("\n");
        break;
    case 3 :
        printf("%s\n", objectID_len, objectID);
        break;
    case 4 :
        printf("no value (EMPTY)\n");
        break;
    case 5 :
        display_ipaddr.s_addr = (u_long) ipaddr;
        printf("%s\n", inet_ntoa(display_ipaddr));
        /* This worked on VM, MVS and AIX, but not on OS/2
        * printf("%d.%d.%d.%d\n", (ipaddr >> 24), ((ipaddr << 8) >> 24),
        * ((ipaddr << 16) >> 24), ((ipaddr << 24) >> 24));
        */
        break;
    case 6 :
        printf("%lu\n", counter);
        break;
    case 7 :
        printf("%lu\n", gauge);
        break;
    case 8 :
        printf("%lu\n", ticks);
        break;
    case 9 :
        printf("%s\n", dstring);
        break;
    case 10 :
        printf("%s\n", command);
        break;
    } /* end switch(index) */
}

#ifdef _NO_PROTO /* for classic K&R C */
static void check_arguments(argc, argv) /* check arguments */
int argc;

```

```

char *argv[];
#else /* _NO_PROTO */
static void check_arguments(const int argc, char *argv[])
#endif /* _NO_PROTO */
{
    char *hname, *cname;
    int i, j;

    dpi_userid = hname = cname = NULL;
    for (i=1; argc > i; i++) {
        if (strcmp(argv[i], "-d") == 0) {
            i++;
            if (argc > i) {
                debug_lvl = atoi(argv[i]);
                if (debug_lvl >= 5) {
                    DPIDebug(1);
                }
            }
        }
        else if (strcmp(argv[i], "-trap") == 0) {
            if (argc > i+3) {
                trap_gtype = atoi(argv[i+1]);
                trap_stype = atoi(argv[i+2]);
                trap_data = argv[i+3];
                i = i + 3;
                do_trap = ONE_TRAP;
            }
            else usage(argv[0], 1);
        }
        else if (strcmp(argv[i], "-trape") == 0) {
            if (argc > i+4) {
                trape_gtype = strtoul(argv[i+1], (char**)0, 10);
                trape_stype = strtoul(argv[i+2], (char**)0, 10);
                trape_eprise = argv[i+3];
                for (i = i + 4, j = 0;
                     (argc > i) && (j < MAX_TRAPE_DATA);
                     i++, j++) {
                    trape_data[j] = argv[i];
                }
                trape_datacnt = j;
                do_trap = ONE_TRAPE;
                break; /* -trape must be last option */
            }
            else usage(argv[0], 1);
        }
        else if (strcmp(argv[i], "-all_traps") == 0) {
            do_trap = ALL_TRAPS;
        }
        else if (strcmp(argv[i], "-std_traps") == 0) {
            do_trap = STD_TRAPS;
        }
        else if (strcmp(argv[i], "-ent_traps") == 0) {
            do_trap = ENT_TRAPS;
        }
        else if (strcmp(argv[i], "-ent_trapse") == 0) {
            do_trap = ENT_TRAPSE;
        }
#ifdef VM || defined(MVS)
        else if (strcmp(argv[i], "-inet") == 0) {
            use_iucv = 0;
        }
        else if (strcmp(argv[i], "-iucv") == 0) {
            use_iucv = TRUE;
        }
        else if (strcmp(argv[i], "-u") == 0) {
            use_iucv = TRUE; /* -u implies -iucv */
            i++;
            if (argc > i) {
                dpi_userid = argv[i];
            }
        }
#endif
    }
    else if (strcmp(argv[i], "?") == 0) {
        usage(argv[0], 0);
    }
    else {
        if (hname == NULL) hname = argv[i];
        else if (cname == NULL) cname = argv[i];
        else usage(argv[0], 1);
    }
}

if (hname == NULL) hname = LOOPBACK; /* use default */
if (cname == NULL) cname = PUBLIC_COMMUNITY_NAME; /* use default */
#ifdef VM || defined(MVS)
if (dpi_userid == NULL) dpi_userid = SNMPAGENTUSERID;
if (debug_lvl > 2)
    printf("hname=%s, cname=%s, userid=%s\n", hname, cname, dpi_userid);
#else
if (debug_lvl > 2)
    printf("hname=%s, cname=%s\n", hname, cname);
#endif
if (use_iucv != TRUE) {
    DO_ETOA(cname); /* for VM or MVS */
    dpi_port = query_DPI_port(hname, cname);
    DO_ATOE(cname); /* for VM or MVS */
}

```

```

        if (dpi_port == -1) {
            printf("No response from agent at %s(%s)\n",hname,cname);
            exit(1);
        }
    } else dpi_port == -1;
    dpi_hostname = hname;
}

#ifdef _NO_PROTO /* for classic K&R C */
static void usage(pname, exit_rc)
char *pname;
int exit_rc;
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void usage(const char *pname, const int exit_rc)
#endif /* _NO_PROTO */
{
    printf("Usage: %s [-d debug_lvl] [-trap g_type s_type data[, pname)];
    printf("] -all_traps[\n");
    printf("%s]-trape g_type s_type enterprise data1 data2 .. datan[\n",
           strlen(pname)+8, "");
    printf("%s]-std_traps[ ]-ent_traps[ ]-ent_trapse[\n",
           strlen(pname)+8, "");
    #if defined(VM) || defined(MVS)
    printf("%s]-iucv[ ]-u agent_userid[\n",strlen(pname)+8, "");
    printf("%s", strlen(pname)+8, "");
    printf("]-inet[ ]agent_hostname ]community_name[[]\n");
    printf("default: -d 0 -iucv -u %s\n", SNMPAGENTUSERID);
    printf("        -inet %s %s\n", LOOPBACK, PUBLIC_COMMUNITY_NAME);
    #else
    printf("%s]agent_hostname ]community_name[[]\n",strlen(pname)+8, "");
    printf("default: -d 0 %s %s\n", LOOPBACK, PUBLIC_COMMUNITY_NAME);
    #endif
    exit(exit_rc);
}

#ifdef _NO_PROTO /* for classic K&R C */
static void init_variables() /* initialize our variables */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void init_variables(void) /* initialize our variables */
#endif /* _NO_PROTO */
{
    char ch, *cp;

    ostring = (char *)malloc(strlen(OSTRING) + 4 + 1 );
    bcopy(OSTRING,ostring,strlen(OSTRING));
    ostring_len = strlen(OSTRING);
    for (ch=1;ch<5;ch++) /* add hex data 0x01020304 */
        ostring[ostring_len++] = ch;
    ostring[ostring_len] = '\0'; /* so we can use it as a string */
    objectID = (char *)malloc(strlen(OID));
    objectID_len = strlen(OID);
    bcopy(OID,objectID,strlen(OID));
    if (objectID[objectID_len - 1] == '.') /* if trailing dot, */
        objectID[objectID_len - 1] = '\0'; /* remove it */
    else objectID_len++; /* length includes null */
    dstring = (char *)malloc(strlen(DSTRING)+1);
    bcopy(DSTRING,dstring,strlen(DSTRING)+1);
    command = (char *)malloc(strlen(COMMAND)+1);
    bcopy(COMMAND,command,strlen(COMMAND)+1);
    ipaddr = dpi_ipaddress;
}

#ifdef _NO_PROTO /* for classic K&R C */
static void init_connection() /* connect to the DPI agent */
#else /* _NO_PROTO */ /* for ANSI-C compiler */
static void init_connection(void) /* connect to the DPI agent */
#endif /* _NO_PROTO */
{
    int rc;
    int sasize; /* size of socket structure */
    struct sockaddr_in sin; /* socket address AF_INET */
    struct sockaddr *sa; /* socket address general */
    #if defined(VM) || defined(MVS)
    struct sockaddr_iucv siu; /* socket address AF_IUCV */

    if (use_iucv == TRUE) {
        printf("Connecting to %s DPI_port %d userid %s (TCP, AF_IUCV)\n",
               dpi_hostname,dpi_port,dpi_userid);
        bzero(&siu,sizeof(siu));
        siu.siucv_family = AF_IUCV;
        siu.siucv_addr = dpi_ipaddress;
    }

```

```

        siu.siucv_port = dpi_port;
        memset(siu.siucv_nodeid, ' ', sizeof(siu.siucv_nodeid));
        memset(siu.siucv_userid, ' ', sizeof(siu.siucv_userid));
        memset(siu.siucv_name, ' ', sizeof(siu.siucv_name));
        bcopy(dpi_userid, siu.siucv_userid, min(8, strlen(dpi_userid)));
        bcopy(SNMP_IUCVNAME, siu.siucv_name, min(8, strlen(SNMP_IUCVNAME)));
        dpi_fd = socket(AF_IUCV, SOCK_STREAM, 0);
        sa      = (struct sockaddr *) &siu;
        sasize = sizeof(struct sockaddr_iucv);
    } else {
#endif
        printf("Connecting to %s DPI_port %d (TCP, AF_INET)\n",
               dpi_hostname, dpi_port);
        bzero(&sin, sizeof(sin));
        sin.sin_family      = AF_INET;
        sin.sin_port        = htons(dpi_port);
        sin.sin_addr.s_addr = dpi_ipaddress;
        dpi_fd = socket(AF_INET, SOCK_STREAM, 0);
        sa      = (struct sockaddr *) &sin;
        sasize = sizeof(struct sockaddr_in);
#ifdef VM || defined (MVS)
    }
#endif
    if (dpi_fd < 0) {
        DO_ERROR("init_connection:  socket");
        exit(1);
    }
    rc = connect(dpi_fd, sa, sasize);
    if (rc != 0) {
        DO_ERROR("init_connection:  connect");
        close(dpi_fd);
        exit(1);
    }
}

#ifdef _NO_PROTO
static void dump_bfr(buf, len)
char *buf;
int len;
/* for classic K&R C */
/* hex dump buffer */
#else /* for ANSI-C compiler */
static void dump_bfr(const char *buf, const int len)
#endif /* _NO_PROTO */
{
    register int i;

    if (len == 0) printf("    empty buffer\n"); /* buffer is empty */
    for (i=0; i<len; i++) {
        if ((i&15) == 0) printf("    "); /* indent new line */
        printf("%2.2x", (unsigned char)buf[i]); /* hex print one byte */
        if ((i&15) == 15) printf("\n"); /* nl every 16 bytes */
        else if ((i&3) == 3) printf(" "); /* space every 4 bytes */
    }
    if (i&15) printf("\n"); /* always end with nl */
}

unsigned long lookup_host(const char *hostname)
{
    register unsigned long ret_addr;

    if ((*hostname >= '0') && (*hostname <= '9'))
        ret_addr = inet_addr(hostname);
    else {
        struct hostent *host;
        struct in_addr *addr;

        host = gethostbyname(hostname);
        if (host == NULL) return(0);
        addr = (struct in_addr *) (host->h_addr_list[0]);
        ret_addr = addr->s_addr;
    }
    return(ret_addr);
}

```

Compiling and Linking the DPISAMPLE.C Source Code

When compiling the Sample DPI Subagent program you may specify the following compile time flags:

NO_PROTO

The DPISAMPLE.C code assumes that it is compiled with an ANSI-C compliant compiler. It can be compiled without ANSI-C by defining this flag.

VM

Indicates that compilation is for VM and uses VM-specific includes. Some VM/MVS specific code is compiled.

Chapter 7. SMTP Virtual Machine Interfaces

Electronic mail (e-mail) is prepared using local mail preparation facilities (or, *user agents*) such as the CMS NOTE and SENDFILE commands; these facilities are not discussed here. This chapter describes the interfaces to the SMTP virtual machine itself, and may be of interest to users who implement electronic mail programs that communicate with the IBM z/VM implementation of SMTP.

The interfaces to the SMTP virtual machine are:

- 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 VM users (or users on an RSCS network attached to the local z/VM system) arrives over this interface. All commands and data received and transmitted through this interface must be composed of ASCII characters.

- The local z/VM system (and systems attached to the local z/VM system by an RSCS network)

SMTP commands can be written to a batch file and then spooled to the virtual reader of the SMTP virtual machine. SMTP processes each of the commands in this file, in order, as if they had been transmitted over a TCP connection. This is how mail is sent from local z/VM users (or users on an RSCS network attached to the local z/VM system) to recipients on the TCP network. Batch SMTP (or, BSMTP) files must contain commands and data composed of EBCDIC characters.

SMTP Transactions

Electronic mail is sent by a series of request/response transactions between a client, the *sender-SMTP*, and a server, the *receiver-SMTP*. These transactions pass (1) the message proper, which is composed of a *header* and a *body* (which by definition, are separated by the first blank line present in this information), and (2) SMTP commands, which are referred to as (and comprise) the mail *envelope*. These commands contain additional information about the mail, such as the host sending the mail and its source and destination addresses. Envelope addresses may be derived from information in the message header, supplied by the user interface, or derived from local configuration information.

The SMTP envelope is constructed at the *sender-SMTP* site. If this is the originating site, the information is typically provided by the user agent when the message is first queued for the *sender-SMTP* program. Each intermediate site receives the piece of mail and resends it on to the next site using an envelope that it creates. The content of the new envelope may be different from that of the one it received.

The envelope contains, at a minimum, the HELO or EHLO, MAIL FROM:, RCPT TO:, DATA, and QUIT commands. These, and other commands that can optionally appear in the envelope, are described in the next section. Some of these commands can appear more than once in an envelope. Also, more than one piece of mail can be sent using a given envelope.

SMTP Commands

This section describes SMTP commands that are recognized by the z/VM SMTP implementation. These commands are used to interface with user agent mail facilities (such as the CMS NOTE and SENDFILE commands) as well as with other SMTP servers.

For more complete information about SMTP and the commands that can be used with this protocol, it is suggested that you review the following RFCs:

- RFC 2821, *Simple Mail Transfer Protocol*
- RFC 822, *Standard for the Format of ARPA Internet Text Messages*
- RFC 1870, *SMTP Service Extension for Message Size Declaration*
- RFC 1652, *SMTP Service Extension for 8bit-MIME transport*

These RFCs are the basis for modern naming specifications associated with the SMTP protocol.

Note: The SMTP commands SEND, SOML, SAML, and TURN are not supported by the z/VM SMTP implementation, so are not described here.

HELO

The HELO command is used to identify the domain name of the sending host to SMTP. This command is used to initiate a mail transaction, and must be sent (once) before a MAIL FROM: command is used.

```
➡ HELO — domain_name ➡
```

Operand

Description

domain_name

Specifies the domain name of the sending host. The *domain_name* may be specified as either:

- a domain name
- an IP address in decimal integer form that is prefixed by the number or (US) pound sign ("#" or X'7B')
- an IP address in dotted-decimal form, enclosed in brackets.
- the string "IPv6:" followed by an IPv6 address in full or compressed form (IPv6 mapped IPv4 addresses are acceptable).

When HELO commands are received over a TCP connection, SMTP replies with the message 250 *SMTP_server_domain* is my domain name. The SMTP server client verification exit or built-in client verification function can be used to determine if the provided *domain_name* matches the client IP address and to include the result of that determination in the mail headers. See [z/VM: TCP/IP Planning and Customization](#) for detailed information about configuring SMTP to use this support.

When HELO commands are received over a batch SMTP connection, SMTP replies with the message 250 *SMTP_server_domain* is my domain name. Additional text is included with this message that indicates whether the provided *domain_name* does or does not match the host name of the spool file origination point. The 250 reply code indicates the HELO command is accepted and that SMTP commands can continue to be sent and received.

EHLO

The EHLO command operates and can be used in the same way as the HELO command. However, it additionally requests that the returned reply should identify specific SMTP service extensions that are supported by the SMTP server.

```
➡ EHLO — domain_name ➡
```

Operand

Description

domain_name

Specifies the domain name of the sending host. The *domain_name* may be specified as either:

- a domain name
- an IP address in decimal integer form that is prefixed by the number or (US) pound sign ("#" or X'7B')
- an IP address in dotted-decimal form, enclosed in brackets.
- the string "IPv6:" followed by an IPv6 address in full or compressed form (IPv6 mapped IPv4 addresses are acceptable).

If a server does not support SMTP service extensions, the client receives a negative reply to its EHLO command. When this occurs, the client should either supply a HELO command, if the mail being delivered can be processed without the use of SMTP service extensions, or it should end the current mail transaction.

If a client receives a positive response to an EHLO command, the server is then known to support one or more SMTP service extensions. This reply then can be further used by the client to determine whether certain kinds of mail can be effectively processed by that server.

For example, if the positive response includes the SIZE keyword, the server supports the SMTP service extension for Message Size Declaration. Whereas, if this response includes the 8BITMIME keyword, the server supports the SMTP service extension for 8-bit MIME transport.

SMTP supports the following service extensions:

EXPN HELP SIZE 8BITMIME

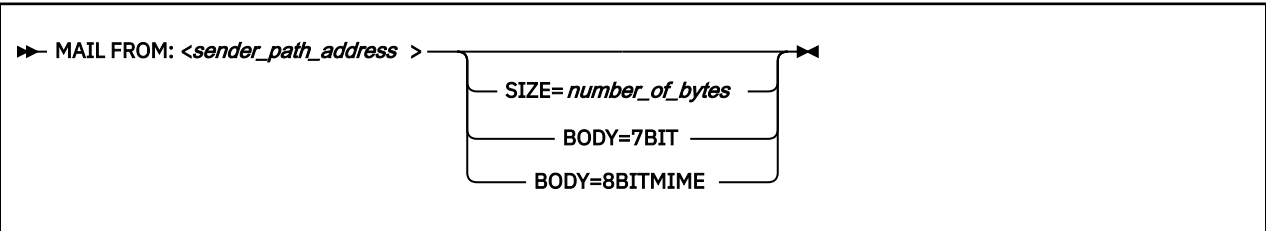
Following is an example of a positive reply to a client (c) EHLO command from an SMTP server (s) that supports these service extensions:

```
s: (wait for connection on TCP port 25)
c: (open connection to server)
s: 220 HOST1.HAL.COM running IBM VM SMTP Level 740 on Sat, 11 May 24 ...
c: EHLO HOST2.HAL.COM
s: 250-HOST1.HAL.COM is my domain name.
s: 250-EXPN
s: 250-HELP
s: 250-8BITMIME
s: 250 SIZE 524288
:
```

The hyphen (-), when present as the fourth character of a response, indicates the response is continued on the next line.

MAIL FROM

The MAIL FROM: command is used (once), after a HELO or EHLO command, to identify the sender of a piece of mail.



Operand

Description

sender_path_address

Specifies the full path address of the sender of the mail. Definitions for valid *sender_path_address* specifications can be obtained from the RFCs that define the naming conventions used throughout the Internet. For detailed information, consult the RFCs listed in the section “SMTP Commands” on page 273.

SIZE=number_of_bytes

Specifies the size of the mail, in bytes, including *carriage return/line feed* (CRLF, X'0D0A') pairs. The SIZE parameter has a range from 0 to 2,147,483,647.

BODY=7BIT

Specifies that the message is encoded using seven significant bits per 8-bit octet (byte). In practice, however, the body is typically encoded using all eight bits.

BODY=8BITMIME

Specifies that the message is encoded using all eight bits of each octet (byte) and may contain MIME headers.

Note: The SIZE, BODY=7BIT, and BODY=8BITMIME options of the MAIL FROM: command should be used only if an EHLO command was used to initiate a mail transaction. If an EHLO command was not used for this purpose, SMTP ignores these parameters if they are present.

If the SMTP server is known to support the SMTP service extension for Message Size Declaration, the client sending the mail can specify the optional SIZE= parameter with its MAIL FROM: commands. The client then can use the responses to these commands to determine whether the receiving SMTP server has sufficient resources available to process its mail before any data is transmitted to that server.

When a MAIL FROM: command is received that includes the optional SIZE= parameter, the SMTP server compares the supplied *number_of_bytes* value to its allowed maximum message size (defined by the MAXMAILBYTES statement in the SMTP CONFIG file) to determine if the mail should be accepted. If *number_of_bytes* exceeds the MAXMAILBYTES value, a reply code 552 is returned to the client.

The SIZE= parameter is evaluated only for MAIL FROM: commands received over a TCP connection; this parameter and its value are ignored when they are received over a batch connection.

RCPT TO

The RCPT TO: command specifies the recipient(s) of a piece of mail. This command can be repeated any number of times.

```
➡ RCPT TO: <recipient_path_address > ➡
```

Operand

Description

recipient_path_address

Specifies the full path address of a mail recipient. Definitions for valid *recipient_path_address* specifications can be obtained from the RFCs that define the naming conventions used throughout the Internet. For detailed information, consult the RFCs listed in the section [“SMTP Commands”](#) on page 273.

A RCPT TO: command must be used after a MAIL FROM: command. If the host system is not aware of the recipient's host, a negative reply is returned in response to the RCPT TO: command.

DATA

The DATA command indicates that the next information provided by the client should be construed as the text of the mail being delivered (that is, the *header* and *body* of the mail message).

```
➡ DATA ➡
```

The DATA command has no parameters.

The DATA command is used after a HELO or EHLO command, a MAIL FROM: command, and at least one RCPT TO: command have been accepted. When the DATA command has been accepted, the following response (reply code 354) is returned to indicate that the body of the mail can be transmitted:

```
354 Enter mail body. End new line with just a '.'.
```

The body of the mail is terminated by transmitting a single ASCII period (.) on a line by itself. When SMTP detects this "end of data" indicator, it returns the following reply:

```
250 Mail Delivered
```

When mail is received over a TCP connection, this ASCII period should be followed by the ASCII CR-LF sequence (*carriage return/line feed* sequence, X'0D0A'). 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 (..). Then, when the receiving SMTP encounters a record in the body of the mail that begins with two periods, it discards

the leading period. This convention permits the mail body to contain records that would otherwise be incorrectly interpreted as the "end of data" indicator. These rules must be followed over both TCP and batch SMTP connections. The CMS NOTE and SENDFILE execs perform this period doubling on all mail spooled to SMTP. If the body of the mail in a batch SMTP command file is not explicitly terminated by a record with a single period, SMTP supplies one.

After the "end of data" indicator has been received, the SMTP connection is reset to its initial state (that is, the state before any sender or recipients have been specified). Additional MAIL FROM:, RCPT TO:, DATA, and other commands can again be sent. If no further mail is to be delivered through this connection, the connection should then be terminated with a QUIT command. If the QUIT command is omitted from the end of a batch SMTP command file, the QUIT is implicit — SMTP will proceed as if it had been provided.

If SMTP runs out of local mail storage space, it returns a 451 reply code to the sender-SMTP client. Local mail storage space is constrained by the size of the SMTP server A-disk (191 minidisk). For a *large* batch SMTP file, disk storage equivalent to four times the size of that file may be required for it to be processed by SMTP.

If the body of the mail being delivered is found to exceed the MAXMAILBYTES value established in the SMTP CONFIG file, a reply code 552 is returned to the client. See [z/VM: TCP/IP Planning and Customization](#) for more information about the MAXMAILBYTES configuration statement.

When mail arrives over a batch SMTP connection from an RSCS network host, and the REWRITE822HEADER configuration option was specified in the SMTP configuration file, then header fields are modified to ensure that all addresses are fully qualified domain names. See [z/VM: TCP/IP Planning and Customization](#) for more information about the header rewriting.

RSET

The RSET command resets an SMTP connection to an initial state. That is, all information about the current mail transaction is discarded, and the connection is ready to process a new mail transaction.

```
➤➤ RSET ➤➤
```

The RSET command has no parameters.

QUIT

The QUIT command terminates an SMTP connection.

```
➤➤ QUIT ➤➤
```

The QUIT command has no parameters.

NOOP

The NOOP command has no intrinsic function. However, it will cause the receiver-SMTP to return an "OK" response (reply code 250).

```
➤➤ NOOP ➤➤
```

The NOOP command has no parameters.

HELP

The HELP command returns brief information about one or more SMTP commands.



Operand

Description

command_name

Identifies a specific SMTP command.

The HELP command returns a multiple-line reply with brief help information about the SMTP commands supported by a host. If an SMTP command is specified for *command_name*, information about that specific command is returned.

QUEU

The QUEU command returns a multiple-line reply with information about the content of the mail processing queues maintained within the SMTP server.



Operand

Description

DATE

Causes information about the age of any queued mail to be included in the QUEU command response. By default, age information is not included in such responses.

The z/VM SMTP server maintains various internal queues for handling mail, that can be generalized to two categories — the mail (delivery) queues, and the mail resolution (or, *resolver*) queues. The QUEU command returns a multiple-line reply with information about the content of these queues, which are described in more detail here.

Mail Delivery Queues

Queue Name

Description

Spool

Contains mail that is destined for recipients on the local z/VM system, or for recipients on an RSCS system attached to the local z/VM system. This queue is generally empty, because SMTP can deliver this mail quickly by spooling it directly to the local recipient, or to the RSCS virtual machine for delivery to an RSCS network recipient.

Active

Identifies mail that is currently being transmitted by SMTP to a TCP network destination. All mail queued for that same destination is shown to be *Active*.

Queued

Identifies mail that has arrived over either a TCP or batch SMTP connection that is to be forwarded to a TCP network destination (possibly because of source routing). When SMTP obtains sufficient resources from the TCPIP virtual machine to process this mail, it is transferred to the *Active* queue.

Retry

Identifies mail for which SMTP has made one or more previous delivery attempts that were not successful. Delivery attempts may fail for a variety of reasons; two common reasons are:

- The SMTP server could not open a connection to deliver the mail.
- Delivery of the mail was interrupted for some reason, such as a broken connection or a temporary error condition at the target host.

After the RETRYINT interval (defined in the SMTP CONFIG file) has passed, mail in the *Retry* queue is promoted to the *Queued* queue (or state) for another delivery attempt. For more information about the RETRYINT configuration parameter, see [z/VM: TCP/IP Planning and Customization](#).

Undeliverable

Identifies mail that SMTP cannot deliver to a local z/VM recipient, or to a recipient on the RSCS network attached to the local VM system, due to insufficient spooling resources on the local z/VM system. After spool space has been increased and SMTP has been reinitialized, delivery of this mail is again attempted.

Mail Resolution Queues

The mail resolution (or, *resolver*) queues are used to maintain the status of host resolution queries — performed through DNS services — for mail host domains, originators, and recipients, when such resolution is necessary. If the SMTP server is configured to *not* use a name server, but only local host tables, these queues are not used.

Several notes regarding the response information associated with mail resolution queues follow:

- If a queue is empty the word Empty appears in the response, to the right of the name of that queue.
- If a queue contains active queries, a line that identifies that queue will be present; information about the mail in that queue, and its associated query (or queries) is provided immediately after this identification line.
- Because of timing situations that can occur within the SMTP server, a queue identification line may at times show that a queue is active (that is, Empty is not indicated), but no mail entries will be present.

Queue Name Description

Process

This queue is generally empty, as it contains queries that have not yet been acted upon by the SMTP server. Once a query has been initially processed, it is placed in the *Resolver Send* queue.

Send

Identifies queries that are awaiting SMTP resolver processing. SMTP staggers the number of queries it submits to a name server, to prevent overloading the network and the name server.

Wait

Identifies queries for which the SMTP server is waiting a response from a name server. Queries remain in this queue for a specific amount of time, within which a reply should be received from the name server.

If a query is successful, that query is then placed in the *Resolver Completed* queue.

If a reply is not received for a query within the allotted time (that is, the resolver time-out has expired), that query is removed from this queue and placed in the *Resolver Retry* queue.

Note: The duration of the resolver time-out period can be controlled using the TCPIP DATA file RESOLVERTIMEOUT configuration statement. See [z/VM: TCP/IP Planning and Customization](#) for more information about this statement.

Retry

Identifies queries that have previously failed, possibly because:

- a name server response was not received for a query (within the designated time-out period), or
- the name server returned a temporary error that has forced the SMTP server to retry a query. A temporary error occurs if, for example, the name server truncates a packet, or if the name server detects a processing error.

Note: Mail for which queries are present in this queue can be significantly affected by the values defined for the RESOLVERRETRYINT and RETRYAGE configuration statements in the SMTP CONFIG file. See [z/VM: TCP/IP Planning and Customization](#) for more information about these statements.

Completed

Identifies queries that have been resolved and are waiting to be recorded by SMTP (and, possibly incorporated within a piece of mail). After the resolved information has been recorded, SMTP attempts to deliver the mail.

Error

Identifies queries for which a name server response was obtained, but for which no answer was obtained. Mail that corresponds to such queries is returned to the originator as undeliverable, with an unknown recipient error indicated.

VERFY

The VRFY ("verify") command determines if a given mailbox or user ID exists on the host where SMTP is running.

►► VRFY *verify_string* ◄◄

Operand**Description*****verify_string***

Specifies the name of a mailbox or user ID whose existence is to be verified.

The z/VM implementation of SMTP responds to the VRFY command and the EXPN command (see the EXPN command below) in the same manner. Thus, the VRFY command can be used with z/VM systems to expand a mailing list defined on such system; when this is done, a multiple-line reply may be returned in response to the VRFY command.

The VRFY command can also be used to verify the existence of the POSTMASTER mailbox or mailboxes defined for a system.

On z/VM systems, mailing lists are defined by the site administrator and are stored in the SMTP NAMES file; POSTMASTER mailboxes are defined by the POSTMASTER configuration statement in the SMTP CONFIG file. See the [z/VM: TCP/IP Planning and Customization](#) for more information about defining mailing lists and specifying POSTMASTER mailboxes.

Some example VRFY commands (issued against an SMTP server running on host TESTVM1 at "somewhere.com") and their corresponding responses follow:

```

vrfy tcpmaint 250 <tcpmaint@abcvml1.somewhere.com>

vrfy tcpadmin-list 250-<tcpadmin@abcvml1.somewhere.com>
250-<tcpadmin@adminpc.somewhere.com>
250 <maint@abcvml1.somewhere.com>

vrfy postmaster 250-<TCPMAINT@TESTVM1.somewhere.com>
250-<TCPADMIN@TESTVM1.SOMEWHERE.COM>
250 <TCPADMIN@ADMINPC.SOMEWHERE.COM>

```

The hyphen (-), when present as the fourth character of a response, indicates the response is continued on the next line.

EXPN

The EXPN ("expand") command expands a mailing list defined on the host where SMTP is running.

►► EXPN *expand_string* ◄◄

Operand**Description**

expand_string

Specifies the name of the mailing list to be expanded.

The EXPN command operates and can be used in the same way as the VRFY command.

VERB

The VERB command is used to enable or disable "verbose" mode for batch SMTP connections.



**Operand
Description**

ON

Specifies that verbose mode is to be enabled (turned on). When verbose mode is enabled for a batch SMTP connection, SMTP commands and their associated replies are recorded in a batch SMTP response file; this file is sent back to the origination point of the batch SMTP command file when the batch transaction is complete.

OFF

Specifies that verbose mode is to be disabled (turned off); this is the default. When verbose mode is disabled for a batch SMTP connection, only SMTP replies are recorded in the batch SMTP response file; this file is not returned to the origination point of the batch SMTP command file.

See “SMTP Command Responses” on page 282 for more information about the batch SMTP response file, how this file is handled, and how origination points are determined.

Note: The VERB command has no effect when issued over a TCP connection.

TICK

The TICK command can be used (in conjunction with the VERB ON command) to cause an identifier string to be inserted into a batch SMTP response file.



**Operand
Description**

identifier

Specifies an identification string to be included in a batch SMTP response file.

This command can be useful for some mail systems that keep track of batch SMTP response files and their content.

Note: The TICK command has no effect when it is issued over a TCP connection.

SMTP Command Example

The following is an example of an SMTP envelope and its contained piece of mail. The SMTP commands that comprise the *envelope* are in upper case boldface text. The information after the DATA command, and before the single ASCII period (the "end of data" indicator) is the message *header* and *body*. The body is distinguished from the header by the blank line that follows the "Subject: Update" line of text.

```
HELO yourhost.yourdomain.edu
MAIL FROM: <carol@yourhost.yourdomain.edu>
RCPT TO: <msgs@host1.somewhere.com>
RCPT TO: <alice@host2.somewhere.com>
DATA
Date: Sun, 30 Jun 24 nn:nn:nn EST
.
```

```

From: Carol <carol@yourhost.yourdomain.edu>
To: <msgs@host1.somewhere.com>
Cc: <alice@host2.somewhere.com>
Subject: Update

Mike: Cindy stubbed her toe. Bobby went to
      baseball camp. Marsha made the cheerleading team.
      Jan got glasses. Peter has an identity crisis.
      Greg made dates with 3 girls and couldn't
      remember their names.

.
QUIT

```

SMTP Command Responses

The z/VM SMTP server can accept SMTP commands that arrive over a TCP connection or over a batch SMTP (BSMTP) connection. With either type of connection, a response (or, reply) is generated for each command received by SMTP. Each reply is prefixed with a three-digit number, or code. The nature of each response can be determined by inspecting the first digit of this reply code; possible values for this digit are:

Digit

Description

0

Echo reply; used only in batch SMTP response files. Received commands are "echoed" in these files to provide contextual information for other reply codes.

1

Positive Preliminary reply. SMTP does not use a 1 as the first digit of a reply code, because there are no SMTP commands for which such a reply is applicable.

2

Positive Completion reply; command accepted.

3

Positive Intermediate reply; data associated with the command should now be provided.

4

Temporary Negative Completion reply; try the command again, but at a later time.

5

Permanent Negative Completion reply; the command has been rejected.

For SMTP commands that arrive over a TCP connection, all responses (positive or negative) are returned over that TCP connection.

Similarly, for SMTP commands that 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 (through use of the VERB ON command), SMTP returns this response file to the origination point of the spool file. The origination point is determined either from the ORIGINID field of the spool file (if the spool file was generated on the same z/VM system as the SMTP virtual machine) or from the spool file TAG field (if the spool file arrived from a remote system through the RSCS network). 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 a 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 valid MAIL FROM: command that was received by SMTP. 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.

Note: All SMTP commands and data that arrive over TCP or batch SMTP connections are subject to the restrictions imposed by both SMTP conventions and constants defined in either the SMTPGLOB COPY file, or within other SMTP source files. Any changes made to these files to overcome a restriction will require the affected source files to be recompiled, and the SMTP module to be rebuilt. Several significant restrictions, the relevant constants, and their default values are:

- Command lines must not exceed *MaxCommandLine* (552 characters).
- Data lines longer than *MaxDataLine* (32767 characters) are wrapped.
- Path addresses must not exceed *MaxPathLength* (256 characters).
- Domain names must not exceed *MaxDomainName* (256 characters).
- User names, the local part of a mailbox specification, must not exceed *MaxUserName* (256 characters).

Path Address Modifications

When SMTP processes MAIL FROM: and RCPT TO: commands, the *path addresses* specified with these commands may be modified by SMTP due to use of the SOURCEROUTES configuration statement, or based on the content of the path addresses themselves. See [z/VM: TCP/IP Planning and Customization](#) for more information about the SOURCEROUTES statement and its affect on path addresses. For content-based changes, certain path addresses will be rewritten by SMTP as follows:

1. If the local part of a mailbox name includes a percent sign (%) *and* the domain of the mailbox is that of the host system where SMTP is running, the given domain name is eliminated, and the portion of the "local part" to the right of the percent sign (%) is used as the destination domain. For example, the path address:

```
john%yourvm@ourvm.our.edu
```

is rewritten by SMTP running at "ourvm.our.edu" as:

```
john@yourvm
```

2. Path addresses with source routes are accepted and rewritten to remove the domain name of the host system where SMTP is running. For example, the path address:

```
@ourvm.our.edu,@next.host.edu:john@yourvm
```

is rewritten by SMTP running at "ourvm.our.edu" as:

```
@next.host.edu:john@yourvm
```

Definitions for "valid path format" specifications can be obtained from the RFCs that define the naming conventions used throughout the Internet. For detailed information, consult the RFCs listed in the section ["SMTP Commands"](#) on page 273.

Batch SMTP Command Files

Batch SMTP command files are files that contain an SMTP envelope (as described in ["SMTP Transactions"](#) on page 273) which are sent to the virtual reader of the SMTP virtual machine using the CMS PUNCH, DISK DUMP, SENDFILE, or NETDATA SEND commands. For a description of these commands, see the [z/VM: CMS Command Reference](#). These files are encoded using EBCDIC.

Batch SMTP command files can be sent by users on the same z/VM system, or any system connected through an RSCS network. For information about RSCS networks, see the [RSCS General Information](#).

Batch SMTP files may be modified when they are processed by the SMTP server, as follows:

- All trailing blanks are removed from each record of a file sent in PUNCH format. Trailing blanks are preserved for files sent in NETDATA or DISK DUMP format.
- A record that is entirely blank will be treated as a record with a single blank.

Batch SMTP Examples

The following sections contain examples that demonstrate batch SMTP capabilities.

Sending Mail to a TCP Network Recipient

The example that follows shows the content of a batch SMTP file used to send mail from a CMS user (CAROL at YOURHOST) to two TCP network recipients. The VERB ON command will cause a batch SMTP response file to be returned to the CMS user CAROL. The text included with the TICK command will appear in this file as well, so that the nature of the response file will be evident when it is returned.

```
VERB ON
TICK Carol's Batch Test File
HELO yourhost.yourdomain.edu
MAIL FROM: <carol@yourhost.yourdomain.edu>
RCPT TO: <msgs@host1.somewhere.com>
RCPT TO: <alice@host2.somewhere.com>
DATA
Date: Sun, 30 Jun 24 nn:nn:nn EST
From: Carol <carol@yourhost.yourdomain.edu>
To: <msgs@host1.somewhere.com>
Cc: <alice@host2.somewhere.com>
Subject: Update

Mike: Cindy stubbed her toe. Bobby went to
      baseball camp. Marsha made the cheerleading team.
      Jan got glasses. Peter has an identity crisis.
      Greg made dates with 3 girls and couldn't
      remember their names.

.
QUIT
```

With the exception of the VERB and TICK commands, this sample batch SMTP file contains commands that are identical to those shown in [“SMTP Command Example” on page 281](#).

Following is the batch SMTP response file (BSMTP REPLY) produced for the previous command file:

```
220-YOURHOST.YOURDOMAIN.EDU running IBM VM SMTP Level 740 on Sun, 30 Jun 24 nn
220 :nn:n EST
050 VERB ON
250 Verbose Mode On
050 TICK Carol's Batch Test File
250 OK
050 HELO yourhost.yourdomain.edu
250 YOURHOST.YOURDOMAIN.EDU is my domain name. Yours too, I see!
050 MAIL FROM: <carol@yourhost.yourdomain.edu>
250 OK
050 RCPT TO: <msgs@host1.somewhere.com>
250 OK
050 RCPT TO: <alice@host2.somewhere.com>
250 OK
050 DATA
354 Enter mail body. End by new line with just a '.'
250 Mail Delivered
050 QUIT
221
YOURHOST.YOURDOMAIN.EDU running IBM VM SMTP Level 740 closing connection
```

Querying SMTP Delivery Queues

The SMTP delivery queues can be queried by sending a file that contains VERB ON and QUEU commands to the SMTP virtual machine. A batch SMTP response file that contains the QUEU command results is then returned to the originating user ID.

The SMTPQUEU EXEC, which is supplied with TCP/IP for z/VM on the TCPMAINT 592 user (or client code) minidisk, generates such a file and sends it to the SMTP virtual machine.

Sample content for a BSMTP REPLY file returned in response to an SMTPQUEU command follows:

```
220-YOURHOST.YOURDOMAIN.EDU running IBM VM SMTP Level 740 on Sun, 30 Jun 24 nn
220 :nn:n EST
050-VERB ON
050
250 Verbose Mode On
050-QUEU
050
250-Queues on YOURHOST.YOURDOMAIN.EDU at nn:nn:nn EST on 06/30/24
```

```

250-Spool Queue: Empty
250-Queue for Site: 123.45.67.89 RETRY QUEUE Last Tried: nn:nn:nn
250-Note 000000005 to <MSG@HOST1.SOMEWHERE.COM>
250-Queue for Site: 98.76.54.32 RETRY QUEUE Last Tried: nn:nn:nn
250-Placeholder...no files queued for this site
250-Undeliverable Queue: Empty
250-Resolution Queues:
250-Resolver Process Queue: Empty
250-Resolver Send Queue: Empty
250-Resolver Wait Queue:
250-000000013 <userx@somehost.nowhereville.com>
250-Resolver Retry Queue: Empty
250-Resolver Completed Queue: Empty
250-Resolver Error Pending Queue: Empty
250 OK

```

SMTP Exit Routines

The SMTP user exits described in the next sections allow you greater control over each piece of mail that is processed by the SMTP server. To effectively use these exits and their parameters, it is necessary to understand SMTP transactions. Refer to the previous sections in this chapter for information about the commands, messages, and replies that are used to facilitate e-mail transactions between the sender and receiver of a piece of mail.

Prior to customizing the server exits described in this section, ensure that you have reviewed the exit limitations and customization recommendations presented in the "Customizing Server-specific Exits" section of the *z/VM: TCP/IP Planning and Customization*.

Client Verification Exit

When a client connects to SMTP, the originating mail domain must be provided. The client verification exit can be used to verify that the domain name provided by a client matches that client's IP address. Thus, this exit allows flexibility on actions you can take to deal with *spoofing* problems. In spoofing, the client provides a falsified domain in order to cause mail to appear to have come from someone (or somewhere) else. When client verification is performed, you might choose to include the verification results in mail headers, or possibly reject future communications on a connection.

With the client verification exit, you can perform any or all of the following:

- Reject mail from a particular host.
- Mark certain trusted sites as verified, but perform verification on all others.
- Control which users can use a particular SMTP server.

The exit can be further customized to perform additional actions that are unique or required for your environment.

Note: The client verification exit is called for each HELO or EHLO command processed for each mail item received from the network. Client verification is not performed for mail items received from the SMTP virtual reader.

Built-in Client Verification Function

In addition to the exit, SMTP can be configured to perform client verification through internal processing. When this support is enabled, this "built-in" client verification function will be called for each HELO or EHLO processed for each mail item. See *z/VM: TCP/IP Planning and Customization* for detailed information about configuring SMTP to use this support.

The built-in client verification function of the SMTP server can be used to determine if a client host name and client IP address match, and to include the result of that determination in the mail headers. This function will perform a DNS lookup against the HELO or EHLO command data provided by a client, and will then insert a message into the mail header that reflects the result of this lookup.

Client verification performed using the built-in function has three possible outcomes:

Success

The data the client provided in the HELO or EHLO command corresponds to the client address. The following line is inserted into the mail header:

```
X-Comment: localhost: Mail was sent by host
```

Failure

The data the client provided in the HELO or EHLO command is not associated with the client IP address. In this case, a reverse name lookup is done against the client IP address to determine the actual host name. The following line is inserted into the mail header:

```
X-Comment: localhost: Host host claimed to be helodata
```

Unknown

The validation could not be performed. This situation could occur if the name server is not responding, or the verification could not be performed in the allotted time (as controlled by the VERIFYCLIENTDELAY statement). The following line is inserted into the mail header:

```
X-Warning: localhost: Could not confirm that host [ipaddr] is helodata
```

The terms used in the previously listed mail header messages are described in more detail here:

localhost

the local VM host name

helodata

the data the client provided with the HELO or EHLO command

host

the host name determined by the reverse DNS lookup; if a host name is not found, "unknown host" will be used

ipaddr

the client IP address.

Client Verification Exit Parameter Lists

The parameter lists passed to the REXX and the assembler exit routines follow. When you customize either of these exits, keep in mind the following:

- Because an identical exit parameter list definition is used for *all* of the SMTP user exits, not all parameters may be meaningful for *this* exit. Parameters that are not used by this exit are indicated in the exit parameter lists; their values should be ignored.
- For the REXX exit, the value of an unused parameter will be such that any parsing will not be affected.

Parameter descriptions that pertain to both the REXX and assembler exits are provided in the [“Parameter Descriptions”](#) on page 288.

REXX Parameter List

Inputs

Table 25. Client Verification REXX Exit Parameter List

Argument	Description
ARG(1)	Parameter list defined as follows: <ul style="list-style-type: none"> • Exit type • Version number • Mail record ID • Port number of SMTP server • IPv4 address of SMTP server • Port number of client • IPv4 address of client • Filename of note on disk • Verify client status • Maximum length of Return String • IPv6 address of SMTP server • IPv6 address of client • Exit flags
ARG(2)	SMTP command string
ARG(3)	HELO/EHLO name
ARG(4-6)	Not used

Outputs

The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

RC	Return String
----	---------------

Argument Description

RC

The exit return code; this must be a 4-byte binary value.

Return String

An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.

Assembler Parameter List

Following is the parameter list that is passed to the assembler exit routine. General Register 1 points to the parameter list.

Table 26. Client Verification ASSEMBLER Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type

Table 26. Client Verification ASSEMBLER Exit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
+4	4	Input	Int	Version number
+8	4	Input	Int	Mail record ID
+12	4	Input	Int	Port number of SMTP server
+16	4	Input	Int	IPv4 address of SMTP server
+20	4	Input	Int	Port number of client
+24	4	Input	Int	IPv4 address of client
+28	4	Input	Ptr	Address of SMTP command string
+32	4	Input	Int	Length of SMTP command string
+36	4	Input	Ptr	Address of HELO/EHLO name
+40	4	Input	Int	Length of HELO/EHLO name
+44	24			Not used
+68	4	Input	Int	Verify Client status
+72	4	Output	Ptr	Address of Return String
+76	4	Output	Int	Length of Return String
+80	4	Input	Int	Maximum length of Return String
+84	16	Input	Char	IPv6 address of SMTP server
+100	16	Input	Char	IPv6 address of client
+116	1	Input	Char	Exit flags
+117	3			Not used
+120	8			Not used
+128	4	Input/ Output	Char	User Word 1
+132	4	Input/ Output	Char	User Word 2
+136	4	Output	Int	Return code from exit

Parameter Descriptions

Exit type

A four-character field that indicates the type of exit called. For the client verification exit, this is VERX.

Version number

The parameter list version number; if the parameter list format is changed, the version number will change. Your exit should verify it has received the expected version number. The current version number is 2.

Mail record ID

A number that uniquely identifies a piece of mail so that multiple exit calls can be correlated to the same piece of mail. A value of 0 indicates a mail record ID is not available.

Port number of SMTP server

The port number used by the SMTP server for this connection.

IP address of SMTP server

For the REXX exit, a dotted-decimal format IPv4 address is provided; for the assembler exit, this is an IPv4 address in decimal integer form. For multi-homed hosts, this address can be compared with the client IPv4 address to determine in which part of the network the client host resides.

Port number of client

If the connection no longer exists, -1 is supplied. Otherwise, this is the port number used by the foreign host for this connection.

IP address of client

For the REXX exit, a dotted-decimal format IPv4 address is provided; for the assembler exit, this is an IPv4 address in decimal integer form.

SMTP command string

Contains the HELO/EHLO command and the domain specified for this command. The string has been converted to uppercase (for example, "HELO DOMAIN1").

HELO/EHLO name

A string that contains the name specified on the HELO or EHLO command; this string may be either:

- a domain name
- an IP address in decimal integer form that is prefixed by the number or (US) pound sign ("#" or X'7B')
- an IP address in dotted-decimal form, enclosed in brackets.

For example, if the command HELO #123456 is provided by an SMTP client, this parameter would contain #123456.

The name has already been verified to have the correct syntax.

Verify Client status

A number that indicates client verification results. For this exit, client verification results are unknown when the exit receives control; thus, this field will contain a 3. Possible values and their meanings are:

- 0**
Client verification passed.
- 1**
Client verification failed.
- 2**
Client verification was not performed.
- 3**
Client verification results are unknown.

Return String

When the exit returns a return code of 3, this value is appended to the X-Comment that is inserted in the mail header. When the exit returns a return code of 5, the *Return String* value is appended to the 550 reply code.

Maximum length of Return String

The current maximum is 512 bytes; ensure the *Return String* length is less than this value. If the returned string is longer than the indicated maximum, the return string is truncated and the following message is displayed on the SMTP server console:

```
Return data from exit exitname exittype too long, data truncated
```

Normal processing continues.

IPv6 address of SMTP server

For the REXX exit, an IPv6 address in full or compressed form is provided; for the assembler exit, this is an IPv6 address in decimal integer form. For multi-homed hosts, this address can be compared with the client IPv6 address to determine in which part of the network the client host resides.

IPv6 address of client

For the REXX exit, an IPv6 address in full or compressed form is provided; for the assembler exit, this is an IPv6 address in decimal integer form.

Exit flags

Flags passed to the exit.

X'00'

IPv4 addresses passed

X'01'

IPv6 addresses passed

User Word 1

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

User Word 2

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

Return Codes from the Client Verification Exit Routine

Following are the return codes recognized by SMTP for this exit.

Table 27. Client Verification Exit Return Codes

Return Code	Explanation
0	Do not verify client. A comment will not be inserted in the mail header.
1	Perform verification using the built-in client verification function.
2	Mark as verified. The following comment will be inserted in the mail header: <div>X-Comment: localhost: Mail was sent by host</div>
3	The following comment will be inserted in the mail header: <div>X-Comment: Return String</div> where the value for <i>Return String</i> can be specified by the exit.
4	Disable the exit. The following message will be displayed on the SMTP console: <div>VERIFYCLIENT EXIT function disabled</div> The exit will no longer be called. The client will not be verified and no comment will be inserted in the mail header.
5	Reject this command with: 550 <i>Return String</i> If a return string is not provided by the exit, then the default message will be displayed: <div>550 Access denied</div> All future communications on this connection will be rejected with this 550 message.
Other	Any return code other than the above causes SMTP to issue this message: <div>Unexpected return from user exit <i>exitname</i> <i>exittype</i>, RC = <i>rc</i></div> SMTP treats this return code as if it were a return code of 0.

Client Verification Sample Exits

Sample Client Verification exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

SMTPVERX SAMPEXEC

REXX exit routine that contains a sample framework for performing client verification actions. Your customized exit should be stored on the TCPMAINT 198 disk as SMTPVERX EXEC.

SMTPVERX SAMPASM

Assembler exit routine that contains a sample framework for performing client verification actions. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPVERX ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM SMTPVERX DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment. The assembler exit will have better performance characteristics than the REXX exit. For best REXX performance, the SMTP server will EXECLOAD any REXX exit.

Using the Mail Forwarding Exit

When SMTP clients use the VM SMTP server to send mail to hosts that their workstations cannot reach directly, this is an instance of *mail forwarding*. The mail forwarding exit provides a mechanism to control this activity. When SMTP determines the addressee specified on a RCPT TO: command is not "*defined on*" the local system, it has detected mail forwarding, and it will call this exit routine.

The phrase "*defined on*" in the previous paragraph is meant to convey that SMTP considers a user to be a *local* user, in addition to any other criteria, if that user is defined in the SMTP NAMES file — regardless of whether mail delivery to that user is performed via spooling (RSCS services) or through a network TCP connection. Also, keep in mind that the determination of whether mail forwarding is occurring is made on a recipient-by-recipient basis, not on other aspects of a given piece of mail. A piece of mail with multiple recipients can contain occurrences of both mail forwarding and local delivery.

With the mail forwarding exit, you can perform any or all of the following:

- Allow mail forwarding and mail delivery to proceed without interruption.
- Disallow mail forwarding from a known sender of "junk" mail, and possibly reject future communications on a connection used for this purpose.
- Intercept mail from specific clients and forward that mail to a local VM user ID for further analysis.
- Restrict the ability to forward mail to a particular set of hosts.

The exit can be further customized to perform additional actions that are required for your environment.

Note: The mail forwarding exit is only called for mail items received from the network; it is not called for mail items generated on the VM system or received via RSCS.

This exit can also be used to control *spamming*. Spamming is the act of sending mail to a large number of e-mail addressees and is often compared to the term "junk mail", used to describe similar activities performed via postal services. *Spam* is a piece of mail that is perceived by the recipients to be unsolicited and unwanted. There are two aspects to consider when trying to control spamming problems:

- Is your system being used to relay spam messages to recipients throughout the internet?
- Are incoming spam messages to your local users seriously taxing or overloading your system?

The relaying of spam messages may be treated like any other type of mail forwarding. The exit can prevent delivery of all forwarded mail, prevent delivery of mail from particular sites known for spamming, or only allow delivery of mail from particular trusted sites. Handling spam messages directed to your local users will require the use of the SMTP command exit. When you address spamming problems, it's important to realize that one person may consider a piece of mail to be a spam, while the same piece of mail may be valuable to someone else. There are no explicit rules that determine what is and is not spam.

In addition to the exit, SMTP can be configured to enable or disable mail forwarding for **all** mail. If mail forwarding is disabled in this manner and SMTP determines the recipient specified on a RCPT TO: record is not defined on the local system, it has detected mail forwarding, and it will reject the delivery of the mail to that recipient. See *z/VM: TCP/IP Planning and Customization* for more information about configuring SMTP to accept or reject all forwarded mail.

Mail Forwarding Exit Parameter Lists

The parameter lists passed to the REXX and the assembler exit routines follow. When you customize either of these exits, keep in mind the following:

- Because an identical exit parameter list definition is used for *all* of the SMTP user exits, not all parameters may be meaningful for *this* exit. Parameters that are not used by this exit are indicated in the exit parameter lists; their values should be ignored.
- For the REXX exit, the value of an unused parameter will be such that any parsing will not be affected.

Parameter descriptions that pertain to both the REXX and assembler exits are provided in the [“Parameter Descriptions”](#) on page 294.

REXX Parameter List

Inputs

Table 28. Mail Forwarding REXX Exit Parameter List

Argument	Description
ARG(1)	Parameter list defined as follows: <ul style="list-style-type: none"> • Exit type • Version number • Mail record ID • Port number of SMTP server • IPv4 address of SMTP server • Port number of client • IPv4 address of client • Filename of note on disk • Verify client status • Maximum length of Return String • IPv6 address of SMTP server • IPv6 address of client • Exit flags
ARG(2)	SMTP command string
ARG(3)	HELO/EHLO name
ARG(4)	MAIL FROM: string
ARG(5)	Client domain name
ARG(6)	Not used

Outputs

The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

RC	Return String
----	---------------

Argument**Description****RC**

The exit return code; this must be a 4-byte binary value.

Return String

An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.

Assembler Parameter List

Following is the parameter list that is passed to the assembler exit routine. General Register 1 points to the parameter list.

Table 29. Mail Forwarding ASSEMBLER Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type
+4	4	Input	Int	Version number
+8	4	Input	Int	Mail record ID
+12	4	Input	Int	Port number of SMTP server
+16	4	Input	Int	IPv4 address of SMTP server
+20	4	Input	Int	Port number of client
+24	4	Input	Int	IPv4 address of client
+28	4	Input	Ptr	Address of SMTP command string
+32	4	Input	Int	Length of SMTP command string
+36	4	Input	Ptr	Address of HELO/EHLO name
+40	4	Input	Int	Length of HELO/EHLO name
+44	4	Input	Ptr	Address of client domain name
+48	4	Input	Int	Length of client domain name
+52	4	Input	Ptr	Address of MAIL FROM: string
+56	4	Input	Int	Length of MAIL FROM: string
+60	8	Input	Char	File name of note on disk
+68	4	Input	Int	Verify Client status
+72	4	Output	Ptr	Address of Return String
+76	4	Output	Int	Length of Return String
+80	4	Input	Int	Maximum length of Return String
+84	16	Input	Char	IPv6 address of SMTP server
+100	16	Input	Char	IPv6 address of client
+116	1	Input		Exit flags
+117	3			Not used
+120	8			Not used

Table 29. Mail Forwarding ASSEMBLER Exit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
+128	4	Input/ Output	Char	User Word 1
+132	4	Input/ Output	Char	User Word 2
+136	4	Output	Int	Return code from exit

Parameter Descriptions

Exit type

A four-character field that indicates the type of exit called. For the mail forwarding exit, this is FWDX.

Version number

The parameter list version number; if the parameter list format is changed, the version number will change. Your exit should verify it has received the expected version number. The current version number is 2.

Mail record ID

A number that uniquely identifies a piece of mail so that multiple exit calls can be correlated to the same piece of mail. A value of 0 indicates a mail record ID is not available.

Port number of SMTP server

The port number used by the SMTP server for this connection.

IP address of SMTP server

For the REXX exit, a dotted-decimal format IPv4 address is provided; for the assembler exit, this is an IPv4 address in decimal integer form. For multi-homed hosts, this address can be compared with the client IPv4 address to determine in which part of the network the client host resides.

Port number of client

If the connection no longer exists, -1 is supplied. Otherwise, this is the port number used by the foreign host for this connection.

IP address of client

For the REXX exit, a dotted-decimal format IPv4 address is provided; for the assembler exit, this is an IPv4 address in decimal integer form.

SMTP command string

Contains the name specified on the RCPT TO: command. The recipient path, enclosed in angle brackets (< and >), is included. The recipient path may be in any valid path format; it has already been verified to have the correct syntax. Because the recipient address has been resolved, this string may not exactly match the data provided with the RCPT TO: command.

For example, if the following has been specified by the SMTP client:

```
RCPT TO: <usera@host1>
```

the SMTP command string might contain: <usera@host1.com>

HELO/EHLO name

A string that contains the name specified on the HELO or EHLO command; this string may be either:

- a domain name
- an IP address in decimal integer form that is prefixed by the number or (US) pound sign ("#" or 'X'7B')
- an IP address in dotted-decimal form, enclosed in brackets.

For example, if the command HELO #123456 is provided by an SMTP client, this parameter would contain: #123456.

The name has already been verified to have the correct syntax.

Client domain name

The domain name that corresponds to the client IP address. The length of this field will be zero if:

- client verification was not performed
- the results of client verification are unknown
- a reverse lookup failed

In all other cases, this will be a domain name.

MAIL FROM: string

Contains the name specified on the MAIL FROM: command. The sender path, enclosed in angle brackets (< and >), is included. The sender path may be in any valid path format; it has already been verified to have the correct syntax. Because the sender address has been resolved, this string may not exactly match the data provided with the MAIL FROM: command.

For example, if the following has been specified by the SMTP client:

```
MAIL FROM: <userb@host2>
```

the SMTP command string might contain: <userb@host2.com>

File name of note on disk

Name of the file created after the "end of data" (EOD) condition, a period (.), is received. Prior to either of these conditions, the file name is not defined; in this case, an asterisk (*) will be supplied.

Verify Client status

A number that indicates client verification results. Possible values and their meanings are:

- 0**
Client verification passed.
- 1**
Client verification failed.
- 2**
Client verification was not performed.
- 3**
Client verification results are unknown.

Return String

When the exit returns a return code of 1 or 5, this value is appended to the 551 or 550 reply code.

When the exit returns a return code of 2, the *Return String* value should contain a VM user ID to which mail should be transferred.

Maximum length of Return String

The current maximum is 512 bytes; ensure the *Return String* length is less than this value. If the returned string is longer than the indicated maximum, the return string is truncated and the following message is displayed on the SMTP server console:

```
Return data from exit exitname exittype too long, data truncated
```

Normal processing continues.

IPv6 address of SMTP server

For the REXX exit, an IPv6 address in full or compressed form is provided; for the assembler exit, this is an IPv6 address in decimal integer form. For multi-homed hosts, this address can be compared with the client IPv6 address to determine in which part of the network the client host resides.

IPv6 address of client

For the REXX exit, an IPv6 address in full or compressed form is provided; for the assembler exit, this is an IPv6 address in decimal integer form.

Exit flags

Flags passed to the exit.

X'00'

IPv4 addresses passed

X'01'

IPv6 addresses passed

User Word 1

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

User Word 2

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

Return Codes from the Mail Forwarding Exit Routine

Following are the return codes recognized by SMTP for this exit.

Table 30. Mail Forwarding Exit Return Codes

Return Code	Explanation
0	Accept and attempt mail delivery.
1	Reject mail with: 551 <i>Return String</i> If a return string is not provided by the exit, the following default message will be used: <div>551 User not local; please try <i>user@otherhost</i></div> If the server has already responded to the command, this return code will result in error mail being sent back to the sender.
2	Accept and forward to the local VM user ID specified by <i>Return String</i> . If the VM user ID is null or is not valid, the mail will be delivered to the local postmaster; the mail will not be delivered to the addressee.
4	Disable the exit. The following message will be displayed on the SMTP console: <div>FORWARD MAIL EXIT function disabled</div> The exit will no longer be called. SMTP will attempt to deliver this mail.
5	Reject this command with: 550 <i>Return String</i> If a return string is not provided by the exit, then the default message will be displayed: <div>550 Access denied</div> All future communications on this connection will be rejected with this 550 message.
Other	Any return code other than the above causes SMTP to issue this message: <div>Unexpected return from user exit <i>exitname</i> <i>exittype</i>, RC = <i>rc</i></div> SMTP treats this return code as if it were a return code of 0.

Mail Forwarding Sample Exits

Sample Mail Forwarding exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

SMTPFWDX SAMPEXEC

REXX exit routine that contains a sample framework for handling forwarded mail items. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPFWD XEXEC.

SMTPFWDX SAMPASM

Assembler exit routine that contains a sample framework for handling forwarded mail items. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPFWDX ASSEMBLE.

The customized ASSEMBLE file must be assembled (by using the VMFHLASM SMTPFWDX DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment. The assembler exit will have better performance characteristics than the REXX exit. For best REXX performance, the SMTP server will EXECLOAD any REXX exit.

Using the SMTP Command Exit

The SMTP server can be configured to call an exit routine whenever certain SMTP commands are received, through use of the SMTP command exit. This exit can be defined such that is invoked for any or all the commands that follow:

HELO

The SMTP 'HELO' command.

EHLO

The SMTP 'EHLO' command.

MAIL

The SMTP 'MAIL FROM:' command.

RCPT

The SMTP 'RCPT TO:' command.

DATA

The SMTP 'DATA' command.

EOD

The "end of data" condition. This occurs when a period (.) is received by the server, usually after all data has been transmitted.

VRFY

The SMTP 'VRFY' command.

EXPN

The SMTP 'EXPN' command.

RSET

The SMTP 'RSET' command.

PUNCH

The point in time when the server is about to deliver mail to a local destination on the same node or RSCS network; this command is unique to the VM TCP/IP SMTP server.

Note:

1. The person responsible for creating or maintaining programs that exploit this capability should be knowledgeable of the protocol(s) related to the SMTP commands that are processed using this exit.
2. Only one SMTP command exit can be active at a time.

The SMTP command exit could be used for a wide variety of purposes; several possible uses are included here:

- Reject particular SMTP commands. For example, you may not want your server to support the VRFY and EXPN commands.

Note: The SMTP server answers to the EXPN and/or VRFY commands. The EXPN command can be used to find the delivery address of mail aliases, or even the full name of the recipients, and the VRFY command may be used to check the validity of an account. Your mailer should not allow remote users to use any of these commands, because it gives them too much information.

- Handle the delivery of local mail in a specific manner.

- Screen and reject mail that contains offensive language, or fails to meet other criteria defined by your installation.

Note: Scanning the content of a message will severely degrade server performance.

SMTP Command Exit Parameter Lists

The parameter lists passed to the REXX and the assembler exit routines follow. When you customize either of these exits, keep in mind the in mind the following:

- Because an identical exit parameter list definition is used for *all* of the SMTP user exits, not all parameters may be meaningful for *this* exit. Parameters that are not used by this exit are indicated in the exit parameter lists; their values should be ignored.
- For the REXX exit, the value of an unused parameter will be such that any parsing will not be affected.

Parameter descriptions that pertain to both the REXX and assembler exits are provided in the [“Parameter Descriptions”](#) on page 300.

REXX Parameter List

Inputs

Table 31. SMTP Commands REXX Exit Parameter List

Argument	Description
ARG(1)	Parameter list defined as follows: <ul style="list-style-type: none"> • Exit type • Version number • Mail record ID • Port number of SMTP server • IPv4 address of SMTP server • Port number of client • IPv4 address of client • Filename of note on disk • Verify client status • Maximum length of Return String • IPv6 address of SMTP server • IPv6 address of client • Exit flags
ARG(2)	SMTP command string
ARG(3)	HELO/EHLO name
ARG(4)	MAIL FROM: string
ARG(5)	Client domain name
ARG(6)	Batch VM user ID

Outputs

The following are returned to the caller in the RESULT variable via a REXX RETURN statement:

RC	Return String
----	---------------

Argument Description

RC

The exit return code; this must be a 4-byte numeric value.

Return String

An exit-specified string; the returned value must have a length less than or equal to the maximum length passed to the exit.

Assembler Parameter List

Following is the parameter list that is passed to the assembler exit routine. General Register 1 points to the parameter list.

Table 32. SMTP Commands ASSEMBLER Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type
+4	4	Input	Int	Version number
+8	4	Input	Int	Mail record ID
+12	4	Input	Int	Port number of SMTP server
+16	4	Input	Int	IPv4 address of SMTP server
+20	4	Input	Int	Port number of client
+24	4	Input	Int	IPv4 address of client
+28	4	Input	Ptr	Address of SMTP command string
+32	4	Input	Int	Length of SMTP command string
+36	4	Input	Ptr	Address of HELO/EHLO name
+40	4	Input	Int	Length of HELO/EHLO name
+44	4	Input	Ptr	Address of client domain name
+48	4	Input	Int	Length of client domain name
+52	4	Input	Ptr	Address of MAIL FROM: string
+56	4	Input	Int	Length of MAIL FROM: string
+60	8	Input	Char	File name of note on disk
+68	4	Input	Int	Verify client status
+72	4	Output	Ptr	Address of Return String
+76	4	Output	Int	Length of Return String
+80	4	Input	Int	Maximum length of Return String
+84	16	Input	Char	IPv6 address of SMTP server
+100	16	Input	Char	IPv6 address of SMTP server
+116	1	Input	Char	Exit flags
+117	3			Not used
+120	8	Input	Char	Batch VM User ID
+128	4	Input/ Output	Char	User Word 1
+132	4	Input/ Output	Char	User Word 2

Table 32. SMTP Commands ASSEMBLER Exit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
+136	4	Output	Int	Return code from exit

Parameter Descriptions

Exit type

A four-character field that indicates the type of exit called. For the SMTP command exit, this is CMDX.

Version number

The parameter list version number; if the parameter list format is changed, the version number will change. Your exit should verify it has received the expected version number. The current version number is 2.

Mail record ID

A number that uniquely identifies a piece of mail so that multiple exit calls can be correlated to the same piece of mail. A value of 0 indicates a mail record ID is not available.

Port number of SMTP server

The port number used by the SMTP server for this connection.

IP address of SMTP server

For the REXX exit, a dotted-decimal format IPv4 address is provided; for the assembler exit, this is an IPv4 address in decimal integer form. For multi-homed hosts, this address can be compared with the client IPv4 address to determine in which part of the network the client host resides.

Port number of client

If the connection no longer exists, or if the command was received over a batch (BSMTP) connection, -1 is supplied. Otherwise, this is the port number used by the foreign host for this connection.

IP address of client

For the REXX exit, a dotted-decimal format IPv4 address is provided; for the assembler exit, this is an IPv4 address in decimal integer form. If the relevant SMTP command was received over a batch SMTP (BSMTP) connection, this field is 0.

File name of note on disk

Name of the file created after the "end of data" (EOD) condition, a period (.), is received. Prior to either of these conditions, the file name is not defined; in this case, an asterisk (*) will be supplied.

Verify Client status

A number that indicates client verification results. Possible values and their meanings are:

- 0**
Client verification passed.
- 1**
Client verification failed.
- 2**
Client verification was not performed.
- 3**
Client verification results are unknown.

SMTP command string

Contains the current command and parameters; the string has been converted to uppercase. For example, if this exit was called for the MAIL FROM: command, this string might contain: MAIL FROM: <USERA@MYDOMAIN>.

HELO/EHLO name

A string that contains the name specified on the HELO or EHLO command; this string may be either:

- a domain name

- an IP address in decimal integer form that is prefixed by the number or (US) pound sign ("#" or X'7B')
- an IP address in dotted-decimal form, enclosed in brackets.

For example, if the command HELO #123456 is provided by an SMTP client, this parameter would contain: #123456.

The name has already been verified to have the correct syntax.

MAIL FROM: string

Contains the name specified on the MAIL FROM: command. The sender path, enclosed in angle brackets (< and >), is included. The sender path may be in any valid path format; it has already been verified to have the correct syntax. Because the sender address has been resolved, this string may not exactly match the data provided with the MAIL FROM: command.

For example, if the following has been specified by the SMTP client:

```
MAIL FROM: <userb@host2>
```

the SMTP command string might contain: <userb@host2.com>

Client domain name

The domain name that corresponds to the client IP address. This field will be a null string if:

- client verification was not performed
- the results of client verification are unknown
- a reverse lookup failed

In all other cases, this will be a domain name.

Batch VM user ID

This field is only used when SMTP commands arrive over a batch SMTP (BSMTP) connection. If this exit is called for batch SMTP connections, this field will contain the VM User ID that originated the mail. Otherwise, this field is not defined and will contain nulls.

User Word 1

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

User Word 2

Provided for use by the assembler exit only. The user word specified upon return from this exit will be passed back in this field for any future calls; 0 is the initial value. The SMTP server does not use this value in any way.

Return String

When the exit returns a return code of 1 or 5, this value is appended to the 550 reply code.

Maximum length of Return String

The current maximum is 512 bytes; ensure the *Return String* length is less than this value. If the returned string is longer than the indicated maximum, the return string is truncated and the following message is displayed on the SMTP server console:

```
Return data from exit exitname exittype too long, data truncated
```

Normal processing continues.

IPv6 address of SMTP server

For the REXX exit, an IPv6 address in full or compressed form is provided; for the assembler exit, this is an IPv6 address in decimal integer form. For multi-homed hosts, this address can be compared with the client IPv6 address to determine in which part of the network the client host resides.

IPv6 address of client

For the REXX exit, an IPv6 address in full or compressed form is provided; for the assembler exit, this is an IPv6 address in decimal integer form.

Exit flags

Flags passed to the exit.

X'00'

IPv4 addresses passed

X''

IPv6 addresses passed

Return Codes from the SMTP Command Exit Routine

Following are the return codes recognized by SMTP for this exit.

Table 33. SMTP Command Exit Return Codes

Return Code	Explanation
0	Accept command, and continue normal processing.
1	Reject mail with: 550 <i>Return String</i> If a return string is not provided by the exit, the following default message will be used: <div>550 Command Rejected</div> This return code is not valid for the PUNCH command; if 1 is returned for a PUNCH command exit call, it will be handled as an invalid exit return code.
2	The PUNCH command has been handled by the exit routine; therefore, bypass file delivery. This return code is valid for only the PUNCH command.
3	PUNCH the mail to an alternate user ID that is specified in the return string that is passed back by the exit routine. If a return string is not provided by the exit, or the return string is too long for a VM user ID, then this return code is treated in the same manner as a return code of 0 and normal file delivery will occur. If the return string does contain a valid user ID, then the SMTP server will deliver this mail to that user ID with the distribution code set to the user ID of the original recipient. This return code is only valid for the PUNCH command.
4	Disable the exit. The following message will be displayed on the SMTP console: <div>SMTPCMD5 EXIT function disabled</div> The exit will no longer be called. The command will be attempted, and processing will continue.
5	Reject this command with: 550 <i>Return String</i> If a return string is not provided by the exit, then the default message will be displayed: <div>550 Access denied</div> All future communications on this connection will be rejected with this 550 message.
6	Treat this RCPT TO command as a NOOP. The SMTP server will just reply to the RCPT TO with "250 OK", but will take no action on it. This return code can be used to ignore a null RCPT TO command or to ignore a RCPT TO command based on the recipient information it contains. This return code is only valid for the RCPT TO command.
Other	Any return code other than the above causes SMTP to issue this message: <div>Unexpected return from user exit <i>exitname</i> <i>exittype</i>, RC = <i>rc</i></div> SMTP treats this return code as if it were a return code of 0.

SMTP Command Sample Exits

Sample SMTP Command exit routines are supplied with TCP/IP on the TCPMAINT 591 disk. The supplied samples are:

SMTPCMDX SAMPEXEC

REXX exit routine that contains a sample framework for SMTP command processing. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPCMDX EXEC.

SMTPCMDX SAMPASM

Assembler exit routine that contains a sample framework for SMTP command processing. Your customized exit should be stored on the TCPMAINT 198 disk as file SMTPCMDX ASSEMBLE.

The customized ASSEMBLE file must be assembled (by using the VMFHLASM SMTPCMDX DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment. The assembler exit will have better performance characteristics than the REXX exit. For best REXX performance, the SMTP server will EXECLOAD any REXX exit.

Chapter 8. Telnet Exits

The Telnet server exits described in the sections that follow provide CP command simulation, TN3270E printer management, and system access control when Telnet connections are established with your host.

Prior to customizing the server exits described in this section, ensure that you have reviewed the exit limitations and customization recommendations presented in the "Customizing Server-specific Exits" section of the *z/VM: TCP/IP Planning and Customization*.

While the SCEXIT or PMEXIT is running, the TCP/IP service machine cannot service any other requests. Therefore, it is advised that processing performed within these exits should be minimized.

Also, in environments with a high rate of TN3270 and/or TN3270E session creation and termination, the use of a REXX exec could adversely affect performance. While calling such an exec may be useful for designing and testing a prototype, a production-level exit should be written in assembler. For such environments, the supplied sample Telnet session connection exit (SCEXIT SAMPASM) and printer management exit (PMEXIT SAMPASM) should be used as a basis for assemble files which directly perform any actions appropriate for your environment. It is recommended that execs be used only for designing and testing an exit prototype; for best performance, such execs should be EXECLOADED.

Telnet Session Connection Exit

When a Telnet client establishes a session with TCP/IP for VM and InternalClientParms ConnectExit has been specified, the exit routine receives control using standard OS linkage conventions. Register 1 points to a parameter list to be used by the exit.

Telnet Exit Parameter List

Table 34. Telnet Session Connection Exit Parameter List

Offset	Len	Name	In/Out	Description
+0	1	SCREASON	Input	Reason Exit was called X'01' - Client connect
+1	1	SCFLAGS1	Input	Flags 1... - Connection is IPv6xxx - Reserved
			Output	Flags 1... - Hide VM logo from client .1... - Hide command simulation ..xx xxxx - Reserved
+2	2	Reserved		
+4	4	SCIPADDR	Input	IPv4 address of client
+8	2	SCPORT	Input	Telnet server port number
+10	2	SCCMDL	Input Output	Length of command buffer Length of command placed in buffer
+12	4	SCPCMD	Input	Address of command buffer

Table 34. Telnet Session Connection Exit Parameter List (continued)

Offset	Len	Name	In/Out	Description
+16	4	SCRC	Output	Return code 0 = Give client VM logo 4 = Reject client, no message 8 = Perform command in SCCPCMD; SCCMDL must be non-zero 12 = Same as 0, and disable exit 16 = Same as 4, and disable exit 20 = Same as 8, and disable exit All others will reject the client, and a message is displayed on the TCP/IP virtual machine console.
+20	2	SCFPORT	Input	Client foreign port number
+22	2	Reserved		
+24	16	SCLUNAME	Input	Client-provided LU name
+40	4	SCLIPADD	Input	Local IPv4 address to which client connected
+44	32	SCCIPHER	Input	Encryption suite used by the connection. The string will be UNSECURED if it is not a secure connection.
+76	16	SCIP6ADD	Input	IPv6 address of client
+92	16	SCLIP6AD	Input	Local IPv6 address to which client connected

Sample Exit

Sample Telnet connection exit routines are supplied with TCP/IP on the TCPMAINT 591 minidisk. The supplied samples are:

SCEXIT SAMPEXEC

REXX exit routine that contains the logic for allowing or denying access by Telnet clients. Your customized exit should be stored on the TCPMAINT 198 disk as file SCEXIT EXEC.

SCEXIT SAMPASM

The assembler exit routine called by the Telnet server; the exit is used to call SCEXIT EXEC and to pass results back to the Telnet server. Your customized exit should be stored on the TCPMAINT 198 disk as file SCEXIT ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM SCEXIT DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

SCEXIT TEXTSAMP

TEXT file produced by assembly of the aforementioned SCEXIT SAMPASM assemble file. If the supplied SCEXIT assembler routine meets the needs of your installation in its supplied form, the SCEXIT TEXTSAMP file can be stored on the TCPMAINT 198 disk as file SCEXIT TEXT, and used as is, without the need to separately assemble the exit assembler source file.

The sample exit is enabled by including the following in PROFILE TCP/IP:

```
InternalClientParms
ConnectExit SCEXIT
EndInternalClientParms
```

Telnet Printer Management Exit

When a client establishes a TN3270E printer session with TCP/IP for VM and InternalClientParms TN3270EExit has been specified, the exit routine receives control when a printer session is established

or terminated. The exit is called using standard CMS linkage conventions. General Register 1 points to a parameter list that the exit may use.

Telnet Printer Management Exit Parameter List

Table 35. Telnet Exit Parameter List

Offset	Len	Name	In/Out	Description
+0	1	PMXVERSN	Input	Parameter list version number X'02'
+1	1	PMXREASN	Input	Reason exit called X'00' - Printer connected X'01' - Printer disconnected
+2	1	PMXFLAG	Input	Flags 1 - Connection is IPv6 .xxx xxxx - Reserved
+3	1	Reserved		
+4	4	PMXIPADD	Input	IPv4 address of client
+8	2	PMXFPORT	Input	Client port number
+A	2	PMXLPORT	Input	Telnet server port number
+C	4	PMXLDEV	Input	Logical device number
+10	8	PMXLUNAM	Input	Logical unit name specified by client
+18	8	PMXUSER	Input	Associated user identifier. If no matching TN3270E configuration statement exists, contains "?"
+20	4	PMXVDEV	Input	Associated virtual device address. If no matching TN3270E configuration statement exists, contains "?"
+24	4	PMXRC	Output	Return code 0 = Accept client 4 = Reject client 8 = Same as 0, and disable exit 12 = Same as 4, and disable exit All others will reject client, and a message is displayed on the TCPIP virtual machine console.
+28	16	PMXIP6AD	Input	IPv6 address of client

Sample Exit

Sample printer management exit routines are supplied with TCP/IP on the TCPMAINT 591 minidisk. The supplied samples are:

PMEXIT SAMPEXEC

REXX exit routine that contains the logic for allowing or denying access by TN3270 clients. Your customized exit should be stored on the TCPMAINT 198 disk as file PMEXIT EXEC.

PMEXIT SAMPASM

The assembler exit routine called by the Telnet server; the exit is used to call PMEXIT EXEC and to pass results back to the Telnet server. Your customized exit should be stored on the TCPMAINT 198

disk as file PMEXIT ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM PMEXIT DMSVM command, and the resulting text deck placed on the TCPMAINT 198 disk.

PMEXIT TEXTSAMP

TEXT file produced by assembly of the aforementioned PMEXIT SAMPASM assemble file. If the supplied PMEXIT assembler routine meets the needs of your installation in its supplied form, the PMEXIT TEXTSAMP file can be stored on the TCPMAINT 198 disk as file PMEXIT TEXT, and used as is, without the need to separately assemble the exit assembler source file.

Enable the sample exit by including the following in PROFILE TCPIP:

```
InternalClientParms
  TN3270EExit PMEXIT
EndInternalClientParms
```

Chapter 9. FTP Server Exit

The FTP server user exits described in the next sections allow you greater control over commands received by the FTP server and allows for auditing of FTP logins,logouts and file transfers.

Prior to customizing the server exit described in this section, ensure that you have reviewed the exit limitations and customization recommendations presented in the "Customizing Server-specific Exits" section of the *z/VM: TCP/IP Planning and Customization*.

The exit is enabled using the **FTAUDIT**, **FTCHKCMD**, and **FTCHKDIR** startup parameters on the SRVRFTP command or by using the FTP SMSG commands. The startup parameters and SMSG commands are documented in *z/VM: TCP/IP Planning and Customization*.

Since the use of the FTP exits adversely affects performance, it is advised that processing performed within the exit should be minimized. While calling a REXX exec is useful for designing an exit prototype, a production-level exit should be written entirely in assembler. For best performance, any REXX execs should be EXECLOADED.

The FTP Server Exit

The following are descriptions of sample FTP server exit routines.

Sample Exit

Sample FTP server exit routines are supplied with TCP/IP on the TCPMAINT 591 minidisk. The supplied samples are:

FTPEXIT SAMPEXEC

REXX exit routine that contains sample logic for login and directory control, and FTP command processing. Your customized exit should be stored on the TCPMAINT 198 disk as file FTPEXIT EXEC.

FTPEXIT SAMPASM

The assembler exit routine called by the FTP server; the exit is used to call FTPEXIT EXEC and to pass results back to the FTP server. Your customized exit should be stored on the TCPMAINT 198 disk as file FTPEXIT ASSEMBLE. The customized ASSEMBLE file must be assembled (by using the VMFHLASM FTPEXIT DMSVM command), and the resulting text deck placed on the TCPMAINT 198 disk.

FTPEXIT TEXTSAMP

TEXT file produced by assembly of the aforementioned FTPEXIT SAMPASM assemble file. If the supplied FTPEXIT assembler routine meets the needs of your installation in its supplied form, the FTPEXIT TEXTSAMP file can be stored on the TCPMAINT 198 disk as file FTPEXIT TEXT, and used as is, without the need to separately assemble the exit assembler source file.

These samples are for illustrative purposes only. They should be modified to meet the needs of your installation before placing them in a production environment.

Audit Processing

With the FTP server exit enabled for audit processing, the FTP Exit will be called for each of the following events:

- LOGIN

Auditing occurs following FTP user login validation

- LOGOUT

Logout occurs when a:

- user enters a QUIT command
- user enters a new USER command while already logged in

- connection is dropped by an SMSG DROP command
- client aborts the connection
- connection is closed because the server is shutting down
- connection times out
- DATA TRANSFER

Data transfers include the following commands:

- APPE (client append command)
- STOR, STOU (client put command)
- RETR (client get command)
- LIST, NLST (client dir, ls commands)

Note: Audit exit processing is enabled with the **FTAUDIT** startup parameter or with the SMSG command to enable exits.

Audit Processing Parameter List

Table 36. FTP Exit Audit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type (AUDX)
+4	4	Input	Int	Version number
+8	8	Input	Char	FTP server command
+16	4	Input	Ptr	Address of command argument string; the first halfword contains the length.
+20	8	Input	Char	Login user ID
+28	8	Input	Char	LOGONBY user ID
+36	4	Input	Int	IPv4 address of client
+40	4	Input	Ptr	Address of current working directory name; the first halfword contains the length.
+44	4	Input	Ptr	Address of target directory or file; the first halfword contains the length.
+48	8	Input	Unsigned double	Number of bytes transferred.
+56	4	Input	Int	Control connection secure setting
+60	4	Input	Int	Data connection secure setting
+64	2	Input	Int	Port number of FTP server
+66	2	Input	Int	Port number of FTP client
+68	8	Input	Char	Event time (yyyy:mm:dd)
+76	8	Input	Char	Event time (hh:mm:ss)
+84	8	-	-	Not used
+92	4	Output	Int	Return code from exit
+96	16	Input	Char	IPv6 address of client
+112	16	Input	Char	Local IPv6 address

Table 36. FTP Exit Audit Parameter List (continued)

Offset in Decimal	Len	In/Out	Type	Description
----------------------	-----	--------	------	-------------

Audit Processing Parameter Descriptions

Exit Type

A 4 character field that indicates the type of exit processing to be performed. For audit processing, this is **AUDX**.

Version number

If the parameter list format is changed, then the version number will change. Your exit should verify it has received the expected version number. The current version number is 4.

FTP server command

This field contains one of the following commands: **LOGIN, LOGOUT, XFER**.

FTP command argument string

- For data transfer (XFER) commands, this string indicates the transfer direction. **SENDING** indicates data is being transferred to a client; **RECEIVING** indicates data is being received from a client.
- For login commands, this string indicates the command that initiated login validation processing (**USER** for anonymous logins or **PASS**)
- For logout commands, this string indicates the command or function which initiated the logout (**QUIT, USER, DROPPED, TIMEOUT, SHUTDOWN, ABORTED**).

Login user ID

The VM user identifier associated with this FTP session. All FTP client authorization checks are made using the login user ID.

LOGONBY user ID

The alternate logon name whose password is used for login authorization checking. A user ID will be present in this field only when the client has issued a USER subcommand that includes the *userid/BY/byuserid* operands; otherwise, a hyphen (-) will be present.

IPv4 address of client

The IPv4 address in binary integer form.

Current working directory name

This field is not used for login or logout processing and will contain a hyphen (-). For data transfers, this field contains the type of directory in use, followed by the working directory name. For example:

Directory Type	Working Directory passed to FTPEXIT
Minidisk	DSK TERI.191
Shared File System	SFS SERVK1:TERI.
Byte File System	BFS /../VMBFS:BFS:TERI/
Virtual Reader	RDR TERI.RDR

Target file

Target file for data transfer. Minidisk, SFS, or RDR files are identified in upper case using the *filename.filetype* format. BFS files are identified using a mixed case *filename*, that can be up to 255 characters long. This field is not used for login and logout processing and will contain a hyphen (-).

Number of bytes transferred

- For data transfer (XFER) commands, this field contains the number of bytes transferred on the data connection.
- For login commands, this field contains a zero.

FTP Server Exit

- For logout commands, this field contains a zero.

Control connection secure setting

The secure setting of the control connection. 0 indicates not applicable. 1 indicates clear (non-secure). 2 indicates secure using TLS.

Data connection secure setting

The secure setting of the data connection. 0 indicates not applicable. 1 indicates clear (non-secure). 2 indicates secure using TLS.

Port number of FTP server

The port number used by the FTP server for this control connection.

Port number of client

The port number used by the foreign host for this control connection.

Event date

The date format for this parameter is *yyyymmdd*.

Event time

The time format for this parameter is *hh:mm:ss*.

Return code from exit

An integer return code. For a list of return codes recognized by the FTP server see [“Return Codes from Audit Processing”](#) on page 312.

IPv6 address of client

The remote IPv6 address of this client, in octet binary format.

Local IPv6 address

The local IPv6 address to which this client connected, in octet binary format.

Return Codes from Audit Processing

Return Code	Use / Description
0	Continue processing.
8	Continue processing, but disable audit exit. The following message is displayed on the FTP server console: "FTP AUDX exit has been disabled".
Other	Any return code other than the above causes FTP to issue the message: "Unexpected return from user exit FTPEXIT AUDX, RC = rc". The server will treat this return code as if it were a return code of 0.

General Command Processing

With the FTP server exit enabled for general command exit processing, the FTP exit will be called to perform command validation for every received FTP command.

Commands that may be passed to this exit follow:

ABOR	ACCT	ALLO	APPE	CDUP	CWD
DELE	EPRT	EPSV	HELP	LIST	MKD
MODE	NLST	NOOP	PASS	PASV	PORT
PWD	QUIT	REIN	REST	RETR	RMD
RNFR	RNT0	SITE	SYST	STAT	STOR
STOU	STRU	TYPE	USER	UNKNOWN	XCWD
XMKD	XPWD				

Note: See RFC 959 for details about the above commands.

Note: See RFC 2428 for more details on EPSV and EPRT.

Also, the AUTH, PBSZ, and PROT commands described in RFC 4217 may be passed to this exit. See RFC 4217 for more details.

The general command exit can be used to perform additional security checking and then take an appropriate action, such as the following:

- Reject commands from a particular IP address, user ID or LOGONBY user ID
- Reject a subset of commands for anonymous users
- Reject transfer requests for specific files
- Reject all users from issuing store (APPE, STOR, STOU) commands

Note: General command exit processing is enabled with the **FTCHKCMD** startup parameter or with the **SMSG** command to enable exits.

General Command Processing Parameter List

Table 37. FTP Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type (CMDX)
+4	4	Input	Int	Version number
+8	8	Input	Char	FTP server command
+16	4	Input	Ptr	Address of command argument string; the first halfword contains the length.
+20	8	Input	Char	Login user ID
+28	8	Input	Char	LOGONBY user ID
+36	4	Input	Int	IPv4 address of client
+40	4	Input	Ptr	Address of current working directory name; The first halfword contains the length.
+44	12	-	-	Not used
+56	4	Input	Int	Control connection secure setting
+60	4	Input	Int	Data connection secure setting
+64	2	Input	Int	Port number of FTP server
+66	2	Input	Int	Port number of FTP client
+68	16	-	-	Not used
+84	4	Input	Int	Maximum length of return string
+88	4	Output	Int	Address of return string (message text)
+92	4	Output	Int	Return code from exit
+96	16	Input	Char	IPv6 address of client
+112	16	Input	Char	Local IPv6 address

General Command Processing Parameter Descriptions

Exit Type

A 4 character field that indicates the type of exit processing to be performed. For command exit processing, this is **CMDX**.

Version number

If the parameter list format is changed, then the version number will change. Your exit should verify it has received the expected version number. The current version number is 4.

FTP server command

Commands received by the server such as USER, STOR, and DELE.

FTP command argument string

The argument string provided by the client. For ACCT and PASS commands, the argument string will contain all asterisks (*****).

Login user ID

The VM user identifier associated with this FTP session. All FTP client authorization checks are made using the login user ID.

LOGONBY user ID

The alternate logon name (*userid*) whose password is used for login authorization checking. A user ID will be present in this field only when the client has issued a USER subcommand that includes the *userid/BY/byuserid* operands; otherwise, a hyphen (-) will be present.

IPv4 address of client

The IPv4 address in decimal integer form.

Current working directory name

This field contains the type of directory, followed by the working directory name. For example:

Directory Type	Working Directory passed to FTPEXIT
Minidisk	DSK TERI.191
Shared File System	SFS SERV1:TERI.
Byte File System	BFS ../VMBFS:BFS:TERI/
Virtual reader	RDR TERI.RDR
No directory defined	-

Control connection secure setting

The secure setting of the control connection. 0 indicates not applicable. 1 indicates clear (non-secure). 2 indicates secure using TLS.

Data connection secure setting

The secure setting of data connections. 0 indicates not applicable. 1 indicates clear (non-secure). 2 indicates secure using TLS.

Port number of FTP server

The port number used by the FTP server for this control connection.

Port number of client

The port number used by the foreign host for this control connection.

Maximum length of return string

The current maximum is 1000 bytes. If the returned string is longer than the maximum, the return string is truncated.

Return string

A return string is to be included as part of the server reply to an FTP client. This string is used only when a return code of 4 or 12 is returned by the exit.

Return code from exit

An integer return code. For a list of return codes recognized by the FTP server see [“Return Codes from General Command Processing”](#) on page 315.

IPv6 address of client

The remote IPv6 address of this client, in octet binary format.

Local IPv6 address

The local IPv6 address to which this client connected, in octet binary format.

Example

If the FTP client provides the command "PUT PROFILE.EXEC", the parameter values provided to the FTPEXIT might be:

FTPEXIT Parameter Values

Exit Type	AUDX
FTP command	XFER
Client IP Address	9.111.32.29
UserID	TERI
ByUserID	-
Bytes transferred	2141
Control connection secure setting	1
Data connection secure setting	1
Server Port	1021
Client Port	21400
Event Date	19990309
Event Time	14:44:34
Working Directory	SFS SERV1:TERI.
Command args	RECEIVING
Target File	PROFILE.EXEC
Remote IPv6 Address	0000:0000:0000:0000:0000:0000:0000:0000
Local IPv6 Address	0000:0000:0000:0000:0000:0000:0000:0000

Return Codes from General Command Processing

Return Code	Use / Description
0	Accept command and continue processing
4	Reject client command with "502 <i>return_string</i> ". If <i>return_string</i> is not provided, return the default message "502 command rejected".
8	Accept command, continue normal processing, but disable command exit processing. The following message is displayed on the FTP server console: "FTP CMDX exit has been disabled".
12	Same as 4 and disable command exit processing. The following message is displayed on the FTP server console: "FTP CMDX exit has been disabled".
Other	Any return code other than the above causes FTP to issue the message: "Unexpected return from user exit FTPEXIT CMDX, RC = rc". The server will treat this return code as if it were a return code of 0.

Change Directory Processing

With the FTP server the exit will be called to validate FTP directory changes and provide greater control over access to system resources by selectively honoring or refusing a client change directory request. The exit is called when an FTP client provides one of the following commands:

- **CWD** or **CD** to change the working directory
- **CDUP** to change the working directory to the parent directory

- **PASS** with a default directory defined in CHKIPADR EXEC
- **USER** for an anonymous login with a default directory defined in CHKIPADR EXEC
- **APPE, DELE, LIST, NLST, RETR, SIZE, STOR, and STOU** commands that involve an explicit change in directory.

Note: See RFC 959 for details about the above commands.

Also, the AUTH, PBSZ, and PROT commands described in RFC 4217 may be passed to this exit. See RFC 4217 for more details.

Change Directory Processing Parameter List

Table 38. FTP Exit Parameter List

Offset in Decimal	Len	In/Out	Type	Description
+0	4	Input	Char	Exit type (DIRX)
+4	4	Input	Int	Version number
+8	8	Input	Char	FTP server command
+16	4	Input	Ptr	Address of command argument string; the first halfword contains the length.
+20	8	Input	Char	Login user ID
+28	8	Input	Char	LOGONBY user ID
+36	4	Input	Int	IPv4 address of client
+40	4	Input	Ptr	Address of current working directory name; the first halfword contains the length.
+44	4	Input	Ptr	Address of target directory or file; the first halfword contains the length.
+48	16	-	-	Not used
+64	2	Input	Int	Port number of FTP server
+66	2	Input	Int	Port number of FTP client
+68	16	-	-	Not used
+84	4	input	Int	Maximum length of return string
+88	4	Output	Int	Address of return string (message text)
+92	4	Output	Int	Return code from exit
+96	16	Input	Char	IPv6 address of client
+112	16	Input	Char	Local IPv6 address

Change Directory Processing Parameter Descriptions

Exit Type

A 4 character field that indicates the type of exit processing to be performed. For CD command exit processing, this is **DIRX**.

Version number

If the parameter list format is changed, then the version number will change. Your exit should verify it has received the expected version number. The current version number is 4.

FTP server command

This field will contain **APPE, CWD, CDUP, DELE, LIST, NLST, PASS, RETR, SIZE, STOR, STOU, or USER**.

FTP command argument string

The argument string entered by the client. For PASS commands, the argument string will contain asterisks (*****).

Login user ID

The VM user identifier associated with this FTP session. All FTP client authorization checks are made using the login user ID.

LOGONBY user ID

The alternate logon name (*userid*) whose password is used for login authorization checking. A user ID will be present in this field only when the client has issued a USER subcommand that includes the *userid/BY/byuserid* operands; otherwise, a hyphen (-) will be present.

IPv4 address of client

The IPv4 address in decimal integer form.

Current working directory name

This field contains the type of directory, followed by the working directory name. For example:

Directory Type	Working Directory Passed to FTPEXIT
Minidisk	DSK TERI.191
Shared File System	SFS SERV1:TERI.
Byte File System	BFS ../VMBFS:BFS:TERI/
Virtual reader	RDR TERI.RDR
No directory defined	-

Target directory

This field contains the fully-qualified target directory for the command. Format of the target directory is similar to the current working directory name format. The following examples show representative values that would be passed to the exit for certain actions or requests made by a client user.

For user login:

```
FTP command      : PASS
Working Directory : -
Command args     : *****
Target Directory  : DSK TERI.191
```

For a CD to a BFS directory request:

```
FTP command      : CWD
Working Directory : DSK TERI.191
Command args     : ../VMBFS:BFS:SCOTT/
Target Directory  : BFS ../VMBFS:BFS:SCOTT/
```

For a CD to a BFS subdirectory request:

```
FTP command      : CWD
Working Directory : BFS ../VMBFS:BFS:SCOTT/
Command args     : SUBDIR
Target Directory  : BFS ../VMBFS:BFS:SCOTT/SUBDIR/
```

Port number of FTP server

The port number used by the FTP server for this control connection.

Port number of client

The port number used by the foreign host for this control connection.

Maximum length of return string

The current maximum is 1000 bytes. If the returned string is longer than the maximum, the return string is truncated.

Return string

A return string is to be included as part of the server reply to an FTP client. This string is used only when a return code of 4 or 12 is returned by the exit.

Return code from exit

An integer return code. For a list of return codes recognized by the FTP server see [“Return Codes from the FTPEXIT Routine for CD Command Processing”](#) on page 318.

IPv6 address of client

The remote IPv6 address of this client, in octet binary format.

Local IPv6 address

The local IPv6 address to which this client connected, in octet binary format.

Return Codes from the FTPEXIT Routine for CD Command Processing

Return Code	Use / Description
0	Accept command and continue normal processing
4	Reject client command with "502 <i>return_string</i> ". If <i>return_string</i> is not provided, display the default message "502 command rejected".
8	Accept command, continue normal processing, but disable CD command exit processing. The following message is displayed on the FTP server console: "FTP DIRX exit has been disabled".
12	Same as 4 and disable CD command exit processing. The following message is displayed on the FTP server console: "FTP DIRX exit has been disabled".
Other	Any return code other than the above causes FTP to issue the message: "Unexpected return from user exit FTPEXIT DIRX, RC = <i>rc</i> ". The server will treat this return code as if it were a return code of 0.

Chapter 10. Remote authorization and auditing through LDAP

Remote authorization allows resource managers that do not reside on z/VM or resource managers that reside on z/VM as a guest to centralize authorization decisions using RACF® through the z/VM LDAP server. Two services are provided by the z/VM LDAP server that enable audit and authorization requests to be resolved using RACF Security Server for z/VM. These services are provided through LDAP extended operations. This same capability exists on both z/OS and z/VM. For remote authorization calls to z/VM, the request comes in the form of a DER-encoding of the ASN.1 syntax. The following sections provide details on these types of remote authorization and auditing requests.

Note that this information describes remote authorization and audit specifically. For full details on auditing controls in z/VM, see [z/VM: RACF Security Server Auditor's Guide](#).

Using remote authorization and auditing

An application or resource manager that uses the **Remote authorization** or **Remote auditing** extended operations must be able to generate requests, send it through the network to the appropriate z/VM LDAP server and interpret the response from the server. The following steps represent the typical sequence of events that are specific to the **Remote authorization** or **Remote auditing** extended operations:

1. The authenticated user must resolve to a SAF or RACF identity that is allowed to perform the authorization check or remote auditing request. The following binds in the z/VM LDAP server can resolve to a SAF or RACF identity:
 - Simple bind to the ICTX plug-in with using an authorized `racfid=userid,cn=ictx` bind distinguished name. If the RACF user ID begins with a number sign (#), it must be preceded by a backslash (\) escape character. Number sign characters in other positions of the user ID do not need to be escaped. For example:


```
racfid=\#user#id,cn=ictx
```
 - Simple bind to the SDBM backend. See SDBM authorization in [z/VM: TCP/IP Planning and Customization](#) for more information.
 - LDBM native authentication bind. Native authentication allows the use of an LDBM entry but the password or password phrase is stored in SAF. See [Native authentication](#) in [z/VM: TCP/IP Planning and Customization](#) for more information.
 - SASL EXTERNAL certificate bind where the certificate is mapped to a SAF or RACF user. See [Setting up for SSL/TLS](#) in [z/VM: TCP/IP Planning and Customization](#) for more information about mapping certificates to users.
2. The application must build a DER-encoded extended operation request having the defined ASN.1 syntax that is specific to the **Remote authorization** or **Remote auditing** extended operation request. That request can then be included with the LDAP handle and the specific request OID on the LDAP client application call, such as `ldap_extended_operation_s()`, to build the LDAP message and send it to the server.
3. The z/VM LDAP server receives the request and routes it to the ICTX plug-in, where it is decoded and processed. The ICTX plug-in verifies the correct syntax and the authority of the requester before invoking the SAF authorization check or audit service to satisfy the request. The result of the SAF service is a DER-encoded response that the LDAP server returns.
4. The application must decode the response to interpret the results. A nonzero **LdapResult** code indicates that the request was not processed by the ICTX plug-in. A nonzero **LdapResult** is accompanied by a reason code message in the response that might provide additional diagnostic information.

Note: A zero **LdapResult** code does not necessarily imply the request was processed successfully (or for authorization, that a user has the specified access). It does, however, indicate that an extended operation `responseValue` was returned. The application should verify that the ICTX `responseCode` within the `responseValue` indicates success (0). A nonzero `responseCode` indicates one or more request items resulted in errors (or unauthorized users). The application should check the `MajorCode` within each response item to determine which returned failures. The application should be aware that ICTX may not return a response item corresponding to each request item in the event of a severe error, such as an error encountered in the extended operation encoding.

The application can send as many requests as needed throughout a single bound session, and should unbind from the z/VM LDAP server when it has finished processing ICTX plug-in requests.

Setting up authorization for working with remote services

After the user successfully authenticated and issued the appropriate extended operation, the bound user must then have the appropriate authority to use the underlying SAF callable services.

For the **Remote authorization** extended operation, the bound user must have at least READ access to the FACILITY class profile IRR.LDAP.REMOTE.AUTH to check the user's own access to a resource. To check access of another user, the user must have at least UPDATE access to FACILITY class profile IRR.LDAP.REMOTE.AUTH.

For example:

```
RDEFINE FACILITY IRR.LDAP.REMOTE.AUTH UACC(NONE)
PERMIT IRR.LDAP.REMOTE.AUTH CLASS(FACILITY) ID(BINDUSER) ACCESS(UPDATE))
SETOPTS RACLIST(FACILITY) REFRESH
```

For the **Remote audit** extended operation, the bound user must have at least READ access to the FACILITY class profile IRR.LDAP.REMOTE.AUDIT.

For example:

```
RDEFINE FACILITY IRR.LDAP.REMOTE.AUDIT UACC(NONE)
PERMIT IRR.LDAP.REMOTE.AUDIT CLASS(FACILITY) ID(BINDUSER) ACCESS(READ)
SETOPTS RACLIST(FACILITY) REFRESH
```

Remote authorization extended operation

The **Remote authorization** extended operation request results in calls to the RACROUTE REQUEST=AUTH SAF service. The results of the RACROUTE REQUEST=AUTH service are returned to the caller. For more information about RACROUTE REQUEST=AUTH, see [z/VM: Security Server RACROUTE Macro Reference](#).

The **Remote authorization** extended operation request must contain the DER-encoding of the ASN.1 syntax. The request OID is 1.3.18.0.2.12.66. The following is the **Remote authorization** extended operation request syntax:

```
requestValue ::= SEQUENCE {
  requestVersion      INTEGER,
  itemList            SEQUENCE of
    item              SEQUENCE {
      itemVersion     INTEGER,
      itemTag          INTEGER,
      userOrGroup      OCTET STRING,
      resource         OCTET STRING,
      class            OCTET STRING,
      access           INTEGER,
      logString        OCTET STRING
    }
  }
}
```

Where:

requestValue

The name for the entire sequence of authorization request data.

requestVersion

The format of the request value. Version 1 indicates a user authorization request; each individual item in the `itemList` is an authorization request for a RACF user ID. Version 2 indicates a user authorization or a group authorization request; each individual item in the `itemList` is an authorization request for either a RACF user ID or a RACF group ID.

itemList

A sequence of one or more items, which allows for multiple authorization checks within a single ICTX request. The size of the entire encoded `requestValue` is limited to 16 million bytes unless your encoding routine or LDAP client imposes a stricter limit. If `requestVersion` is 2, the `itemList` can be a mixture of user authorization and group authorization items.

item

A sequence of data that represents a single authorization check.

itemVersion

The format of the individual item. Version 1 indicates an authorization request for a RACF user ID. Version 2 indicates an authorization request for a RACF group ID.

itemTag

An integer that is set by the client for each request item and echoed in each response item. Its purpose is to assist the client in correlating multiple request responses, and has no influence on the authorization logic or logging.

userOrGroup

If `itemVersion` is 1, a RACF user ID whose authority is being checked. Its length cannot exceed 8 characters. If the length is zero, the user value defaults to the user ID associated with the bind user.

If `itemVersion` is 2, a RACF group ID whose authority is being checked. Its length must be from 1 and 8 characters. Optimizations that are used when performing a user ID authorization check are not available when performing a group ID authorization check. For this reason, it is likely that group authorization check executes more slowly than user ID authorization checks.

This field must be specified in uppercase, because RACF user and group names are uppercase, and the remote authorization service does not convert lowercase characters to uppercase.

resource

A name to be matched against a RACF profile for authorization checking. The string may not include blank characters. Its length may be from 1 to the maximum RACF profile length defined for the specified class.

class

A defined RACF general resource class. It cannot be DATASET, USER, or GROUP. Its length must be from 1 to 8 characters.

If you are checking authorization to resources protected by profiles in a grouping/member class, specify the member class name in the remote authorization request. To obtain accurate results in this case, ensure that the administrator issued SETROPTS RACLIST for the member class.

access

The level of authority requested. It must be one of the following integer values:

```
X'01' READ
X'02' UPDATE
X'03' CONTROL
X'04' ALTER
```

logString

Any character data from 0 to 200 characters in length. It is appended to an ICTX-defined string in the SMF log record.

The following is the ASN.1 syntax for the **Remote authorization** extended operation. The response OID is 1.3.18.0.2.12.67.

```
responseValue ::= SEQUENCE {
    responseVersion      INTEGER,
    responseCode         INTEGER,
    itemList             SEQUENCE of
        item             SEQUENCE {
            itemVersion   INTEGER,
            itemTag       INTEGER,
            majorCode     INTEGER,
            minorCode1    INTEGER,
            minorCode2    INTEGER,
            minorCode3    INTEGER
        }
}
```

Where:

responseValue
The name for the entire sequence of authorization response data.

responseVersion
The format of the response value. Version 1 is the only supported format.

responseCode
The greatest error encountered while processing the request. See [Table 39 on page 322](#) for more details on supported responseCodes.

itemList
A sequence of one or more items, which allows for multiple authorization results within a single ICTX response.

item
A sequence of data that represents the results from a single authorization check.

itemVersion
The format of the individual item. Version 1 is the only supported format.

itemTag
An integer echoed from the corresponding request itemTag. The purpose of the itemTag is to assist the client in correlating multiple request responses. itemTag has no influence on the authorization logic or logging.

majorCode
An integer value representing the result of the authorization check. See [Table 40 on page 323](#) for more details on error major codes.

minorCode1
Additional details about the error. See [Table 41 on page 324](#) for more details on error minor codes.

minorCode2
Additional details about the error.

minorCode3
Additional details about the error.

Remote authorization extended operation response codes

Use the following table to understand the response codes that are generated from the remote authorization processing. The responseCode represents the greatest error encountered. You might experience situations where a request item generates an error that is not reflected in the responseCode because that value is overridden by a higher-severity error.

Table 39. Remote authorization responseCodes	
ResponseCode (decimal)	Meaning
0	All request items were processed successfully

Table 39. Remote authorization responseCodes (continued)

ResponseCode (decimal)	Meaning
28	Empty item list. No items are found within the <code>itemList</code> sequence of the extended operation request, so no response items are returned.
61-70	The specified <code>requestVersion</code> is not supported. Subtract 60 from the value to determine the highest <code>requestVersion</code> that the server supports. <code>responseCode 61</code> indicates the server supports version 1 requests only. <code>responseCode 62</code> indicates that the highest supported request level is 2.
other	Errors or warnings encountered while processing one or more request items. The value represents the highest <code>majorCode</code> in the set of all response items. Verify the major and minor codes returned for each item.

Table 40. Remote authorization majorCodes

MajorCode (decimal)	Meaning	Comment
0	Authorized	The user has the requested access to the resource.
2	Warning mode	The user has the requested access because warning mode is enabled for the resource. Warning mode is a feature of RACF that allows installations to try out security policies. Installations can define a profile with the WARNING attribute. When RACF performs an authorization check by using the profile, it logs the event (if there are audit settings) and allows the authorization check to pass successfully. The log records can be monitored to ensure that the new policy is operating as expected before putting the policy into production by turning off the WARNING attribute.
4	Undetermined	No decision is made. The specified resource is not protected by RACF, or RACF is not installed.
8	Unauthorized	The user does not have the requested access to the resource.
12	RACROUTE error	The RACROUTE REQUEST=AUTH service returned an unexpected error. Compare the returned minor codes with the SAF and RACF codes documented in <i>z/VM: Security Server RACROUTE Macro Reference</i> .
16	Request value error	A value specified in the extended operation request is incorrect or unsupported. Check the returned minor codes to narrow the reason.
20	Request encoding error	A decoding error was encountered indicating the extended operation request contains non-compliant DER encoding, or does not match the documented ASN.1 syntax.

Table 40. Remote authorization majorCodes (continued)

MajorCode (decimal)	Meaning	Comment
24	Insufficient authority	The requestor does not have sufficient authority for the requested function. The user ID associated with the LDAP bound user must have the appropriate access to the FACILITY class profile IRR.LDAP.REMOTE.AUTH.
100	Internal error	An internal error was encountered within the ICTX plug-in.

Table 41. Remote authorization minorCodes

MinorCode (decimal)	MinorCode Meaning
0-14	minorCode1- the SAF return code minorCode2 - the RACF return code minorCode3 - the RACF reason code
16-20	minorCode1 is the extended operation request parameter number within the item. 0 - item sequence 1 - itemVersion 2 - itemTag 3 - user 4 - resource 5 - class 6 - access 7 - logString
	minorCode2 value indicates one of the following: 32 - incorrect length 36 - incorrect value 40 - encoding error
	minorCode3 has no defined meaning.
24-100	minorCodes1, minorCode2, and minorCode3 do not have a defined meaning.

Remote authorization audit controls

The auditor can specify whether to log access attempts that are based on user, class, resource, or any criteria as described in *z/VM: RACF Security Server Auditor's Guide*. The SMF TYPE 80 records that are generated can be unloaded by using the IRRADU00 utility.

Remote auditing extended operation

The **Remote auditing** extended operation request must contain the DER-encoding of the ASN.1 syntax. The request OID is 1.3.18.0.2.12.68. The following is the **Remote auditing** extended operation request syntax:

```
requestValue ::= SEQUENCE {
    requestVersion          INTEGER,
```

```

itemList          SEQUENCE of
  item            SEQUENCE {
    itemVersion   INTEGER,
    itemTag       INTEGER,
    linkValue     OCTET STRING SIZE(8),
    violation     BOOLEAN,
    event         INTEGER,
    qualifier     INTEGER,
    class         OCTET STRING,
    resource      OCTET STRING,
    logString     OCTET STRING,
    dataFieldList SEQUENCE of
      dataField   SEQUENCE {
        type      INTEGER,
        value     OCTET STRING
      }
    }
  }
}

```

Where:

requestValue

The name for the entire sequence of audit request data.

requestVersion

The format of the request value. Version 1 is the only currently supported format.

itemList

A sequence of one or more items, allowing multiple audit records to be written with a single ICTX request. You should limit the size of the entire encoded RequestValue to 16 million bytes; however, your encoding routine or LDAP client might impose a stricter limit.

item

A sequence of data that represents a single audit record.

itemVersion

The format of the individual item. Version 1 is the only currently supported format.

itemTag

An integer that is set by the client for each request item and echoed in each response item. Its purpose is to assist the client in correlating multiple request responses. The itemTag value does not influence the audit processing, and does not appear in the audit record.

linkValue

8 bytes of data that is used to mark related audit records. Specify 8 bytes of zero (X'00') if no such marking is needed.

violation

A boolean value that indicates whether the event represents a violation (nonzero ~ TRUE) or not (zero ~ FALSE). The value is used in the audit logging decision.

event

An integer 1 - 7 that identifies the security event type. The possible values are:

- 1** Authentication
- 2** Authorization
- 3** Authorization Mapping
- 4** Key Management
- 5** Policy Management
- 6** Administrator Configuration

7

Administrator Action

qualifier

An integer 0 - 3 that describe the event result. The possible values are:

0

Success

1

Information

2

Warning

3

Failure

class

A defined RACF general resource class that might be used for audit logging determination. It cannot be DATASET, USER, or GROUP. Its length must be from 0 to 8 characters.

resource

A name that might be matched against a RACF profile in the specified class for audit logging determination. Its length may be from 0 to 246 characters.

logString

Any character data from 0 to 200 characters in length. It is appended to an ICTX-defined string in the SMF log record.

dataFieldList

A sequence of type and value pairs that will be logged as SMF relocates. Any number of relocates might be included, but the audit service limits the total amount of this relocate data to 20 kilobytes per record.

dataField

A sequence of data that represents a single relocate section in an audit record.

type

An integer 100 to 114 corresponding to a defined relocate number. The possible values are:

100

SAF identifier for bind user

101

Requester's bind user identifier

102

Originating security domain

103

Originating registry / realm

104

Originating user name

105

Mapped security domain

106

Mapped registry / realm

107

Mapped user name

108

Operation performed

109

Mechanism / object name

110

Method / function used

111

Key / certificate name

112

Caller subject initiating security event

113

Date and time security event occurred

114

Application specific data

115

Identifier for the client submitting the remote audit request

116

Version of the client submitting the remote audit request

value

Character data of the associated type that is included in the audit record.

The following is the ASN.1 syntax for the **Remote auditing** extended operation response. The response OID is 1.3.18.0.2.12.69.

```
responseValue ::= SEQUENCE {
    responseVersion      INTEGER,
    responseCode         INTEGER,
    itemList             SEQUENCE of
        item             SEQUENCE{
            itemVersion  INTEGER,
            itemTag       INTEGER,
            majorCode     INTEGER,
            minorCode1    INTEGER,
            minorCode2    INTEGER,
            minorCode3    INTEGER
        }
}
```

Where:

responseValue

The name for the entire sequence of audit response data.

responseVersion

The format of the response value. Version 1 is the only supported format.

responseCode

The greatest error encountered while processing the request. See [Table 42 on page 328](#) for more details about supported responseCodes.

itemList

A sequence of one or more items, which allows for multiple audit results within a single ICTX response.

item

A sequence of data that represents the results from a single audit request.

itemVersion

The format of the individual item. Version 1 is the only supported format.

itemTag

An integer that is echoed from the corresponding request itemTag. The purpose of the itemTag is to assist the client in correlating multiple request responses. The itemTag does not influence the audit processing, and does not appear in the audit record.

majorCode

An integer value representing the result of the audit request. See [Table 43 on page 328](#) for more details about error major codes.

minorCode1

Additional details about the error. See [Table 44 on page 330](#) for more details about error minor codes.

minorCode2

Additional details about the error.

minorCode3

Additional details about the error.

Remote auditing extended operation response codes

Use the following table to understand the response codes that are generated from the remote auditing response. The `responseCode` represents the greatest error encountered. You might experience situations in which a request item generates an error that is not reflected in the `responseCode`, because that value is overridden by a higher-severity error.

<i>Table 42. Remote auditing responseCodes</i>	
ResponseCode (decimal)	Meaning
0	All request items were processed successfully.
28	Empty item list. No items are found within the <code>itemList</code> sequence of the extended operation request, so no response items are returned.
61-70	The specified <code>requestVersion</code> is not supported. Subtract 60 from the value to determine the highest <code>requestVersion</code> that the server supports. <code>responseCode 61</code> indicates that the server supports version 1 requests only.
other	Errors or warnings that are encountered while processing one or more request items. The value represents the highest <code>majorCode</code> in the set of all response items. Verify the major and minor codes returned for each item.

<i>Table 43. Remote auditing majorCodes</i>		
MajorCode (decimal)	Meaning	Comment
0	Success	The event is logged successfully.

Table 43. Remote auditing majorCodes (continued)

MajorCode (decimal)	Meaning	Comment
2	Warning mode	<p>The event is logged, and warning mode is set for the specified resource. Warning mode is a feature of RACF that allows installations to try out security policies. Installations can define a profile with the WARNING attribute. When RACF performs an authorization check by using the profile, it logs the event (if there are audit settings) and allows the authorization check to pass successfully. The log records can be monitored to ensure that the new policy is operating as expected before putting the policy into production by turning off the WARNING attribute.</p> <p>A remote client resource manager using the remote audit service might simulate RACF warning mode logic after submitting an audit request for a failing authorization event. If the majorCode in the response item indicates the matching resource profile has the warning mode set, the remote client resource manager might allow the check to pass successfully.</p>
3	Logging not required	The event is not logged because no audit controls are set to require it.
4	Undetermined	<p>The event is not logged. The conditions suggested by the following minorCode combinations might be intentional administrator settings:</p> <p>4,0,0 - RACF is not installed or not active</p> <p>8,8,8 - UAUDIT is not set, and class is not active or not RACLISTed</p> <p>8,8,12 - UAUDIT is not set, class is active and RACLISTed, and a covering resource profile is not found</p>
8	Unauthorized	The user does not have authority for the audit service. The userid associated with the LDAP server must have at least READ access to the FACILITY class profile IRR.RAUDITX.
12	Audit error	The audit service returned an unexpected error. Record the returned minor codes and contact your service representative.
16	Request value error	A value specified in the extended operation request is incorrect or unsupported. Check the returned minor codes to narrow the reason.
20	Request encoding error	A decoding error was encountered indicating the extended operation request contains non-compliant DER encoding, or does not match the documented ASN.1 syntax.

Table 43. Remote auditing majorCodes (continued)

MajorCode (decimal)	Meaning	Comment
24	Insufficient authority	The requestor does not have sufficient authority for the requested function. The user ID associated with the LDAP bound user must have at least READ access to the FACILITY class profile IRR.LDAP.REMOTE.AUDIT.
100	Internal error	An internal error was encountered within the ICTX plug-in.

Table 44. Remote auditing minorCodes

MinorCode (decimal)	MinorCode Meaning
0-12	minorCode1- the SAF return code minorCode2 - the RACF return code minorCode3 - the RACF reason code
16-20	minorCode1 is the extended operation request parameter number within the item. 0 - item sequence 1 - itemVersion 2 - itemTag 3 - linkValue 4 - violation 5 - event 6 - qualifier 7 - class 8 - resource 9 - logstring 10 - dataFieldList sequence 11 - dataField sequence 12 - type 13 - value minorCode2 value indicates one of the following: 32 - incorrect length 36 - incorrect value 40 - encoding error minorCode3 has no defined meaning.
24-100	minorCodes1, minorCodes2, and minorCodes3 do not have defined meaning.

Remote audit controls

The auditor can enable logging for all remote audit events by setting UAUDIT for the RACF user ID associated with the z/VM LDAP server. To narrow the set of events logged, the auditor may set SETROPTS LOGOPTIONS for the matching class or set AUDIT/GLOBALAUDIT for the matching resource profile. To do this, the auditor must know the class and resource specified by the application submitting the remote

audit requests. The SMF Unload utility can then be used to unload the generated SMF type 83 subtype 4 records.

SMF Record Type 83 subtype 4 records

The remote audit service logs events as SMF Type 83 subtype 4 records that can be unloaded by using the IRRADU00 utility. Each logged event has a unique event code with a corresponding event code qualifier, or value that indicates whether the event succeeded, resulted in warning or failure, or was logging event information. The event codes are described in the following table:

<i>Table 45. Remote audit event codes</i>	
Event	Command / Service
1	Authentication
2	Authorization
3	Authorization mapping
4	Key management
5	Policy management
6	Administrator configuration
7	Administrator action

The following table describes the event code qualifiers:

<i>Table 46. Remote audit event code qualifiers</i>		
(Common) Event Code Qualifier Dec (Hex)	Description	(Common) Relocate type sections
0	Successful request or authorization.	Common relocates, 100-114
1	Event information.	Common relocates, 100-114
2	Not a failure, but might warrant investigation. For authorization event, grace period might be in effect.	Common relocates, 100-114
3	Unsuccessful request; unauthorized.	Common relocates, 100-114

The following are the remote audit specific extended relocates:

Table 47. Event-specific fields for remote audit events							
Relocate	XML Tag	SQL field name	Type	Length	Position		Comments
					Start	End	
100	localUser	SAF_LOCAL_USER	Char	8	3000	3007	SAF identifier for bind user
101	bindUser	SAF_BIND_USER	Char	256	3010	3265	Requesters bind user identifier
102	domain	SAF_DOMAIN	Char	512	3268	3779	Originating security domain

<i>Table 47. Event-specific fields for remote audit events (continued)</i>							
Relocate	XML Tag	SQL field name	Type	Length	Position		Comments
					Start	End	
103	regName	SAF_REG_NAME	Char	256	3782	4037	Originating registry / real m
104	regUser	SAF_REG_USER	Char	256	4040	4295	Originating user name
105	mapDomain	SAF_MAP_DOMAIN	Char	512	4298	4809	Mapped security domain
106	mapRegName	SAF_MAP_REG_NAME	Char	256	4812	5067	Mapped registry / realm
107	mapRegUser	SAF_MAP_REG_USER	Char	256	5070	5325	Mapped user name
108	action	SAF_ACTION	Char	64	5328	5391	Operation performed
109	object	SAF_OBJECT	Char	64	5394	5457	Mechanism / object name
110	method	SAF_METHOD	Char	64	5460	5523	Method / function used
111	key	SAF_KEY	Char	256	5526	5781	Key / certificate name
112	subjectName	SAF_SUBJECT_NAME	Char	256	5784	6039	Caller subject initiating security event
113	dateTime	SAF_DATE_TIME	Char	32	6042	6073	Date and time security event occurred
114	otherData	SAF_OTHER_DATA	Char	2048	6076	8123	Application specific data
115	clientID	SAF_CLIENT_ID	Char	16	8126	8141	Identifier for client submitting remote audit request.
116	clientVer	SAF_CLIENT_VER	Char	8	8144	8151	Version of client submitting remote audit request.

Chapter 11. Building an LDAP Server Plug-in

This topic explains how to build an LDAP server plug-in on z/VM and explains differences between the z/VM implementation and the IBM Tivoli® Directory Server for z/OS implementation. For details on the plug-in application service routines, see [z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS \(https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/glpa300_v2r5.pdf\)](https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/glpa300_v2r5.pdf).

An LDAP server plug-in is a software module that extends the capabilities of your directory server.

Plug-ins are dynamically loaded into the LDAP server's virtual machine when the server is started. Once the plug-ins are loaded, the server calls the functions in a shared library by using function pointers.

A server frontend listens to the wire, receives and parses requests from clients, and then processes the requests by calling an appropriate database backend function. A server backend reads and writes data to the database containing the directory entries.

If the frontend fails to process a request, it returns an error message to the client; otherwise, the backends are searched for the appropriate backend to process the request. If a backend is found to process the request, it is passed to the backend. If a backend is not found to process the request, the plug-ins are searched for an appropriate plug-in to process the request. If a backend is called, it must return a message to the client. If a plug-in is called, it must return a message to the client. The frontend, backend, or the plug-in can return a message to the client, but only one can return the message.

The following types of plug-ins are supported by LDAP:

pre-operation

A plug-in that is executed before a client request is processed; for example, a plug-in that checks for a new entry before the new entry is added to a directory.

post-operation

A plug-in that is executed after a client request is processed; for example, a plug-in that audits clients after they bind to the server.

client-operation

A plug-in that is called to process a client request.

Each plug-in is a separate dynamic link library (DLL) that is loaded by the LDAP server. The `SLAPI_PL H` include file defines the various structures and service routine prototypes that are available to the plug-in, and the service functions are provided in a DLL load module. The `GLDSLP31 TEXT` side file defines the plug-in import definitions for this DLL. The `SLAPI_PL H` and `GLDSLP31 TEXT` files are on the TCPMAINT 591 disk.

The LDAP server plugin configuration option is used to define a plug-in to the LDAP server. See [plugin in z/VM: TCP/IP Planning and Customization](#). The option has three required parameters and one optional parameter:

1. The plug-in type: `preOperation`, `clientOperation`, or `postOperation`
2. The plug-in DLL name.
3. The name of the plug-in initialization routine, which is called during LDAP server initialization
4. Optional parameters that the plug-in can retrieve.

For example:

```
plugin postOperation PLUGSAMP plugin_init "auditFile"
```

Steps for writing an LDAP plug-in

To build an LDAP plug-in:

1. Design and write the plug-in initialization routine and SLAPI service functions.

The plug-in initialization routine must register the following that are supported by the plug-in:

- Service functions
- Message types
- Distinguished name suffixes
- Extended operation object identifiers.

Return code 0 must be returned when successful and non-zero when not successful. The plug-in initialization routine receives as input, the plug-in parameter block (**Slapi_PBlock**) and returns an integer as the return value. An example of an initialization routine prototype:

```
int plugin_init ( Slapi_PBlock * pb );
```

Note: For this example, the name `plugin_init` would be the initialization routine name used with the **plugin** configuration option.

2. When writing the SLAPI service functions that implement the plug-in design, see *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS* (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/glpa300_v2r5.pdf) for application service routines to use and for defined prototypes. You can also see `SLAPI_PL H` for defined prototypes.
3. Decide on any input parameters for the plug-in.

Plug-in input parameters can be retrieved using the `SLAPI_PLUGIN_ARGC` or `SLAPI_PLUGIN_ARGV` parameters with the **slapi_pblock_get()** service routine.
4. Include `SLAPI_PL H`, which contains defined SLAPI data structures and prototypes.
5. Export the plug-in initialization routine.
6. Compile the plug-in code into object files.
7. Bind the plug-in object files with the LDAP server `GLDSLP31 TEXT` side file.
8. Ensure the plug-in DLL module is on a CMS minidisk or directory accessible by the LDAP server.
9. Edit and add the **plugin** configuration option to the LDAP server configuration file. See *plugin* in *z/VM: TCP/IP Planning and Customization*.
10. Restart the LDAP server.

You may want to program trace statements to follow processing flow in the plug-in. The trace macro, **SLAPI_TRACE()**, is provided in `SLAPI_PL H` to assist in tracing. This macro uses the **slapi_trace()** service routine, described in *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS* (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/glpa300_v2r5.pdf). For example:

```
SLAPI_TRACE((LDAP_DEBUG_PLUGIN, "PLUGSAMP", "Entered."));
```

A sample plug-in showing several examples of using SLAPI service routines is in file `PLUGSAMP CSAMPLE` on the `TCPMAINT 591` disk.

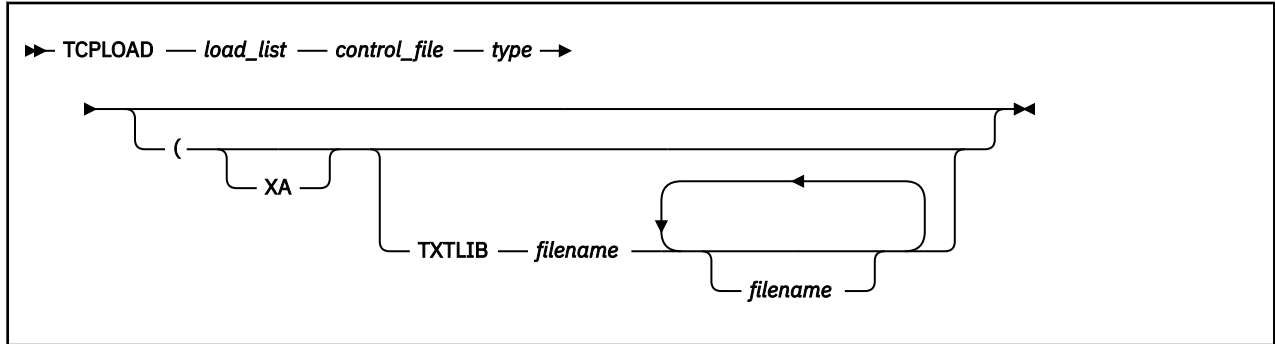
Note about LDAP support on z/VM

When referring to *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS* (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/glpa300_v2r5.pdf), be aware of the following:

- z/VM does not support the DB2® back-end.
- z/VM does not support the 64-bit DLL.
- The z/OS term "address space" is equivalent to "virtual machine" on z/VM.

Appendix A. TCPLOAD EXEC

The TCPLOAD EXEC is provided to generate an executable module from your compiled program. When running TCPLOAD, all disks containing object files must be accessed as extensions of the A-disk. The TCPLOAD EXEC generates a module when given a list of text file names and a control file.



Parameter Description

load_list

Specifies the file name of a file with the file type LOADLIST that contains file names to be included in the load module. The first line in the *load_list* specifies the name of the main object module. Subsequent lines specify additional object modules to be included in the load module.

control_file

Specifies the *control_file*, which determines the file types of text files according to the standard update identifier procedure.

type

Specifies the *type* parameter as one of the following:

C

Includes SCEELKED, CMSLIB, RPCLIB, TCPASCAL, TCPLANG, COMMTXT, and CLIB txtlibs.

C-ONLY

Includes SCEELKED, RPCLIB, COMMTXT, CMSLIB, and CLIB txtlib.

PASCAL

Includes TCPASCAL, TCPLANG, and COMMTXT txtlibs. PASCAL is the default for the *type* parameter.

XA

Specify this option if the application requires storage above the 16Mb line. The application will be generated RMODE ANY and AMODE 31. Pascal applications will require the GLOBAL LOADLIB TCPRTLIB command be issued before being run.

TXTLIB

Specifies the TXTLIB option, which allows you to specify up to 50 *filenames* that will be added to the GLOBAL TXTLIB command.

See [Table 3 on page 29](#) for a list of the files necessary for each application.

If TCPLOAD is not used, you must global the appropriate TXTLIB files.

Using TCPLOAD

The following example describes how to use TCPLOAD to generate an executable module from object files.

1. Create a file with file type LOADLIST, which contains all the object (TEXT) files to be linked. For example, `l1istfn loadlist`.

2. Create a control file with file type CNTRL, which contains the list of TEXT file types. For example, `ctrlfn cntrl`.
3. Invoke the TCPLOAD command, as shown in the following example.

```
TCPLOAD llistfn ctrlfn C (TXTLIB mylib1 mylib2
```

Where:

- `llistfn` is the LOADLIST file name
- `ctrlfn` is the control file name
- `C` is the language of the main program
- `TXTLIB` is the keyword that specifies the libraries to link
- `mylib1` and `mylib2` are the libraries to link.

The following is an example of how to create an executable module from a list of object files. In the example, `OBJ1`, `OBJ2`, `OBJ3`, `OBJ4`, and `OBJ5` are TEXT files created by compiling C programs, and `MYLIB1` and `MYLIB2` are libraries.

1. Create the file SAMPLE LOADLIST that lists the object files:

```
OBJ1  
OBJ2  
OBJ3  
OBJ4  
OBJ5
```

2. Create the file TEST CNTRL with the following:

```
TEXT
```

3. Invoke TCPLOAD with the following command:

```
TCPLOAD SAMPLE TEST C (TXTLIB MYLIB1 MYLIB2
```

This creates the executable file SAMPLE MODULE.

Appendix B. Pascal Return Codes

When using Pascal procedure calls, check to determine whether the call has been completed successfully. Use the SayCalRe function (see [“SayCalRe” on page 75](#)) 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 buffer length specified. For a description of Pascal return codes and their equivalent message text from SayCalRe, see [Table 48 on page 337](#).

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. For more information, see the Explanatory Notes at the end of [Table 48 on page 337](#).

Table 48. 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=xxxxxxx)
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 file; details are in StackID.TCPERROR.
FATALerror ³	-9	Fatal inter-address communications error. (VMCF. RC=nn User=xxxxxxx)
HASnoPASSWORD	-10	No password in RACF directory.
INCORRECTpassword	-11	TCPIP not authorized to access file.
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	File is in use by someone else and cannot be accessed.
MINIDISKnotAVAILABLE	-18	File 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.

Table 48. Pascal Language Return Codes (continued)

Return Code	Numeric Value	Message Text
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 time-out 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 file.
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 time-out
TOOmanyOPENS	-41	Too many open connections already exist.
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 virtual machine has too little storage.
WRONGsecORprc	-50	Foreign host disagreed on security or precedence.
YOURend	-55	Client has ended TCP/IP service.
Oresources	-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.

Table 48. Pascal Language Return Codes (continued)

Return Code	Numeric Value	Message Text
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 file.
FILEformatINVALID	-81	File format invalid.
CANNOTreadFILE	-85	A file used by the operation cannot be read. The file may not exist, or it may be a file mode 0 file.
CANNOTwriteFILE	-86	A file used by the operation cannot be written to. The file may not exist, or it may be a file mode 0 file.
TLSnotAVAILABLE	-87	The SSL server is currently not available.
LABELnotRECOGNIZED	-88	The security server is available but the label is not recognized.
LABELnotPERMITTED	-89	The security server is available but the user is not authorized to use the label specified.
KEYRINGnotRECOGNIZED	-90	The security server is available but the keyring is not recognized.
KEYRINGnotPERMITTED	-91	The security server is available but the user is not authorized to use the keyring specified.
ALREADYsecured	-92	The request to secure the connection failed. The connection is already secure.
STATICALLYsecured	-93	The request to dynamically secure the connection failed. The connection is already statically secured.
CONNECTIONnotSECURE	-94	The request to close a secure connection failed. The connection is not secure.
SSLhandshakeINprogress	-95	The request cannot be made at this time. There is an SSL handshake currently in progress on this connection.
MIXEdaddresses	-96	A mixture of IPv4 and IPv6 addresses have been specified in the Status6InfoType record, this is not allowed.
IPv6connection	-97	An IPv4 function has been issued against an IPv6 connection, the data returned is not valid.
SSLnotRESPONDING	-98	A QueryTLS request was made to the SSL server but the server is not responding.
BACKlevelSSL	-99	The current SSL server is backlevel and does not support TLS negotiated security.
TCPipALREADYstarted	-101	TCP/IP services have already been started.
PERSTISTclose	-102	The request cannot be made at this time. The PersistConnectionLimit has been reached.
SSLserverNOTfound	-103	The specified SSL server does not exist.
SSLserverISactive	-104	The specified SSL server is already active.

Table 48. Pascal Language Return Codes (continued)

Return Code	Numeric Value	Message Text
SSLcloseINprogress	-105	The request cannot be made at this time. There is an SSL close currently in progress on this connection.
UNKNOWNinterface	-106	The interface specified on the QUERY OSA command is not a valid interface.
TIMEOUTosaREQUEST	-107	The request was sent to the OSA but no response was received.
TRYagainLater	-108	An OSA request is already in progress. Try the request again at a later time.
QUERYoatNOTsupported	-109	The query request is not supported by the OSA hardware.
CERTdataNOTavail	-110	The requested certificate data is not available. Check the CDRetCode field in the CertDataCompleteDetailType structure that is returned on the call.

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. Space may still be available for other connections.

Appendix C. C API System Return Codes

This appendix provides the system return codes for IUCV socket calls. These return codes are also contained in the compiler file TCPERRNO H. C socket return codes can be found in compiler file ERRNO H.

Table 49. System Return Codes

Message	Code	Description
EPERM	1	Permission denied.
ENOENT	2	No such file or directory.
ESRCH	3	No such process.
EINTR	4	Interrupted system call.
EIO	5	I/O error.
ENXIO	6	No such device or address.
E2BIG	7	Argument list too long.
ENOEXEC	8	Exec format error.
EBADF	9	Bad file number.
ECHILD	10	No children.
EAGAIN	11	No more processes.
ENOMEM	12	Not enough memory.
EACCES	13	Permission denied.
EFAULT	14	Bad address.
ENOTBLK	15	Block device required.
EBUSY	16	Device busy.
EEXIST	17	File exists.
EXDEV	18	Cross device link.
ENODEV	19	No such device.
ENOTDIR	20	Not a directory.
EISDIR	21	Is a directory.
EINVAL	22	Invalid argument.
ENFILE	23	File table overflow.
EMFILE	24	Too many open files.
ENOTTY	25	Inappropriate device call.
ETXTBSY	26	Text file busy.
EFBIG	27	File too large.
ENOSPC	28	No space left on device.
ESPIPE	29	Illegal seek.
EROFS	30	Read only file system.
EMLINK	31	Too many links.
EPIPE	32	Broken pipe.

Table 49. System Return Codes (continued)

Message	Code	Description
EDOM	33	Argument too large.
ERANGE	34	Result too large.
EWOULDBLOCK	35	Operation would block.
EINPROGRESS	36	Operation now in progress.
EALREADY	37	Operation already in progress.
ENOTSOCK	38	Socket operation on non-socket.
EDESTADDRREQ	39	Destination address required.
EMSGSIZE	40	Message too long.
EPROTOTYPE	41	Protocol wrong type for socket.
ENOPROTOOPT	42	Protocol not available.
EPROTONOSUPPORT	43	Protocol not supported.
ESOCKTNOSUPPORT	44	Socket type not supported.
EOPNOTSUPP	45	Operation not supported on socket.
EPFNOSUPPORT	46	Protocol family not supported.
EAFNOSUPPORT	47	Address family not supported by protocol family.
EADDRINUSE	48	Address already in use.
EADDRNOTAVAIL	49	Cannot assign requested address.
ENETDOWN	50	Network is down.
ENETUNREACH	51	Network is unreachable.
ENETRESET	52	Network dropped connection on reset.
ECONNABORTED	53	Software caused connection abort.
ECONNRESET	54	Connection reset by peer.
ENOBUFS	55	No buffer space available.
EISCONN	56	Socket is already connected.
ENOTCONN	57	Socket is not connected.
ESHUTDOWN	58	Cannot send after socket shutdown.
ETOOMANYREFS	59	Too many references: cannot splice.
ETIMEDOUT	60	Connection timed out.
ECONNREFUSED	61	Connection refused.
ELOOP	62	Too many levels of symbolic loops.
ENAMETOOLONG	63	File name too long.
EHOSTDOWN	64	Host is down.
EHOSTUNREACH	65	No route to host.
ENOTEMPTY	66	Directory not empty.
EPROCLIM	67	Too many processes.
EUSERS	68	Too many users.

Table 49. System Return Codes (continued)

Message	Code	Description
EDQUOT	69	Disc quota exceeded.
ESTALE	70	Stale NFS file handle.
EREMOTE	71	Too many levels of remote in path.
ENOSTR	72	Device is not a stream.
ETIME	73	Timer expired.
ENOSR	74	Out of streams resources.
ENOMSG	75	No message of desired type.
EBADMSG	76	Trying to read unreadable message.
EIDRM	77	Identifier removed.
EDEADLK	78	Deadlock condition.
ENOLCK	79	No record locks available.
ENONET	80	Machine is not on the network.
ERREMOTE	81	Object is remote.
ENOLINK	82	Link has been severed.
EADV	83	Advertise error.
ESRMNT	84	Srmount error.
ECOMM	85	Communication error on send.
EPROTO	86	Protocol error.
EMULTIHOP	87	Multihop attempted.
EDOTDOT	88	Cross mount point.
EREMCHG	89	Remote address changed.

Appendix D. Well-Known Port Assignments

This appendix lists the well-known port assignments for transport protocols TCP and UDP, and includes port number, keyword, and a description of the reserved port assignment. You can also find a list of these well-known port numbers in the ETC SERVICES file.

For further port assignment information, see the [Internet Assigned Numbers Authority \(IANA\)](https://www.iana.org/) website.

TCP Well-Known Port Assignments

Table 50 on page 345 lists the well-known port assignments for TCP.

Table 50. TCP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	rje	remote job entry	remote job entry
7	echo	echo	echo
9	discard	discard	sink null
11	systat	active users	active users
13	daytime	daytime	daytime
15	netstat	Netstat	who is up or Netstat
19	chargen	ttytst source	character generator
21	ftp	FTP	File Transfer Protocol
23	telnet	Telnet	Telnet
25	smtp	mail	Simple Mail Transfer Protocol
37	time	timeserver	timeserver
39	rlp	resource	Resource Location Protocol
42	nameserver	name	host name server
43	nicname	who is	who is
53	domain	name server	domain name server
57	mtp	private terminal access	private terminal access
77	rje	netrjs	any private RJE service
79	finger	finger	finger
87	link	ttylink	any private terminal link
95	supdup	supdup	SUPDUP Protocol
101	hostname	hostname	nic hostname server, usually from SRI-NIC
109	pop	postoffice	Post Office Protocol
111	sunrpc	sunrpc	Sun remote procedure call
113	auth	authentication	authentication service
115	sftp	sftp	Simple File Transfer Protocol
117	uucp-path	UUCP path service	UUCP path service

Table 50. TCP Well-Known Port Assignments (continued)

Port Number	Keyword	Reserved for	Services Description
119	untp	readnews untp	USENET News Transfer Protocol
123	ntp	NTP	Network Time Protocol
160–223		reserved	
512	REXEC	REXEC	Remote Execution Protocol
514	RSH	RSHELL	Remote Shell Service
992	telnet	Telnet	Telnet over TLS/SSL

UDP Well-Known Port Assignments

Table 51 on page 346 lists the well-known port assignments for UDP.

Table 51. UDP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	rje	remote job entry	remote job entry
7	echo	echo	echo
9	discard	discard	sink null
11	users	active users	active users
13	daytime	daytime	daytime
15	netstat	Netstat	Netstat
19	chargen	ttytst source	character generator
37	time	timeserver	timeserver
39	rlp	resource	Resource Location Protocol
42	nameserver	name	host name server
43	nickname	who is	who is
53	domain	nameserver	domain name server
67	bootpd	BOOTP	BOOTP Daemon
75			any private dial out service
77	rje	netrjs	any private RJE service
79	finger	finger	finger
111	sunrpc	sunrpc	Sun remote procedure call
123	ntp	NTP	Network Time Protocol
160–223		reserved	
520			RouteD and MPROUTE using Rip
531	rxd-control		rxd control port
2001	rauth2		Andrew File System service, for the Venus process

Table 51. UDP Well-Known Port Assignments (continued)

Port Number	Keyword	Reserved for	Services Description
2002	rfilebulk		Andrew File System service, for the Venus process
2003	rfilesrv		Andrew File System service, for the Venus process
2018	console		Andrew File System service
2115	ropcons		Andrew File System service, for the Venus process
2131	rupdsrv		assigned in pairs; bulk must be <code>srv +1</code>
2132	rupdbulk;		assigned in pairs; bulk must be <code>srv +1</code>
2133	rupdsrv1		assigned in pairs; bulk must be <code>srv +1</code>
2134	rupdbulk1;		assigned in pairs; bulk must be <code>srv +1</code>

Appendix E. Related Protocol Specifications

Many features of TCP/IP for z/VM are based on the following RFCs:

RFC	Title	Author
768	<i>User Datagram Protocol</i>	J.B. Postel
791	<i>Internet Protocol</i>	J.B. Postel
792	<i>Internet Control Message Protocol</i>	J.B. Postel
793	<i>Transmission Control Protocol</i>	J.B. Postel
821	<i>Simple Mail Transfer Protocol</i>	J.B. Postel
822	<i>Standard for the Format of ARPA Internet Text Messages</i>	D. Crocker
823	<i>DARPA Internet Gateway</i>	R.M. Hinden, A. Sheltzer
826	<i>Ethernet Address Resolution Protocol: or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware</i>	D.C. Plummer
854	<i>Telnet Protocol Specification</i>	J.B. Postel, J.K. Reynolds
856	<i>Telnet Binary Transmission</i>	J.B. Postel, J.K. Reynolds
857	<i>Telnet Echo Option</i>	J.B. Postel, J.K. Reynolds
877	<i>Standard for the Transmission of IP Datagrams over Public Data Networks</i>	J.T. Korb
885	<i>Telnet End of Record Option</i>	J.B. Postel
903	<i>Reverse Address Resolution Protocol</i>	R. Finlayson, T. Mann, J.C. Mogul, M. Theimer
904	<i>Exterior Gateway Protocol Formal Specification</i>	D.L. Mills
919	<i>Broadcasting Internet Datagrams</i>	J.C. Mogul
922	<i>Broadcasting Internet Datagrams in the Presence of Subnets</i>	J.C. Mogul
950	<i>Internet Standard Subnetting Procedure</i>	J.C. Mogul, J.B. Postel
952	<i>DoD Internet Host Table Specification</i>	K. Harrenstien, M.K. Stahl, E.J. Feinler
959	<i>File Transfer Protocol</i>	J.B. Postel, J.K. Reynolds
974	<i>Mail Routing and the Domain Name System</i>	C. Partridge
1009	<i>Requirements for Internet Gateways</i>	R.T. Braden, J.B. Postel
1014	<i>XDR: External Data Representation Standard</i>	Sun Microsystems Incorporated
1027	<i>Using ARP to Implement Transparent Subnet Gateways</i>	S. Carl-Mitchell, J.S. Quarterman
1032	<i>Domain Administrators Guide</i>	M.K. Stahl
1033	<i>Domain Administrators Operations Guide</i>	M. Lottor
1034	<i>Domain Names—Concepts and Facilities</i>	P.V. Mockapetris

RFC	Title	Author
1035	<i>Domain Names—Implementation and Specification</i>	P.V. Mockapetris
1042	<i>Standard for the Transmission of IP Datagrams over IEEE 802 Networks</i>	J.B. Postel, J.K. Reynolds
1055	<i>Nonstandard for Transmission of IP Datagrams over Serial Lines: SLIP</i>	J.L. Romkey
1057	<i>RPC: Remote Procedure Call Protocol Version 2 Specification</i>	Sun Microsystems Incorporated
1058	<i>Routing Information Protocol</i>	C.L. Hedrick
1091	<i>Telnet Terminal-Type Option</i>	J. VanBokkelen
1094	<i>NFS: Network File System Protocol Specification</i>	Sun Microsystems Incorporated
1112	<i>Host Extensions for IP Multicasting</i>	S. Deering
1118	<i>Hitchhikers Guide to the Internet</i>	E. Krol
1122	<i>Requirements for Internet Hosts-Communication Layers</i>	R.T. Braden
1123	<i>Requirements for Internet Hosts-Application and Support</i>	R.T. Braden
1155	<i>Structure and Identification of Management Information for TCP/IP-Based Internets</i>	M.T. Rose, K. McCloghrie
1156	<i>Management Information Base for Network Management of TCP/IP-based Internets</i>	K. McCloghrie, M.T. Rose
1157	<i>Simple Network Management Protocol (SNMP),</i>	J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin
1179	<i>Line Printer Daemon Protocol</i>	The Wollongong Group, L. McLaughlin III
1180	<i>TCP/IP Tutorial,</i>	T. J. Socolofsky, C.J. Kale
1183	<i>New DNS RR Definitions (Updates RFC 1034, RFC 1035)</i>	C.F. Everhart, L.A. Mamakos, R. Ullmann, P.V. Mockapetris,
1187	<i>Bulk Table Retrieval with the SNMP</i>	M.T. Rose, K. McCloghrie, J.R. Davin
1207	<i>FYI on Questions and Answers: Answers to Commonly Asked Experienced Internet User Questions</i>	G.S. Malkin, A.N. Marine, J.K. Reynolds
1208	<i>Glossary of Networking Terms</i>	O.J. Jacobsen, D.C. Lynch
1213	<i>Management Information Base for Network Management of TCP/IP-Based Internets: MIB-II,</i>	K. McCloghrie, M.T. Rose
1215	<i>Convention for Defining Traps for Use with the SNMP</i>	M.T. Rose
1228	<i>SNMP-DPI Simple Network Management Protocol Distributed Program Interface</i>	G.C. Carpenter, B. Wijnen
1229	<i>Extensions to the Generic-Interface MIB</i>	K. McCloghrie
1267	<i>A Border Gateway Protocol 3 (BGP-3)</i>	K. Lougheed, Y. Rekhter
1268	<i>Application of the Border Gateway Protocol in the Internet</i>	Y. Rekhter, P. Gross

RFC	Title	Author
1269	<i>Definitions of Managed Objects for the Border Gateway Protocol (Version 3)</i>	S. Willis, J. Burruss
1293	<i>Inverse Address Resolution Protocol</i>	T. Bradley, C. Brown
1270	<i>SNMP Communications Services</i>	F. Kastenholz, ed.
1323	<i>TCP Extensions for High Performance</i>	V. Jacobson, R. Braden, D. Borman
1325	<i>FYI on Questions and Answers: Answers to Commonly Asked New Internet User Questions</i>	G.S. Malkin, A.N. Marine
1351	<i>SNMP Administrative Model</i>	J. Davin, J. Galvin, K. McCloghrie
1352	<i>SNMP Security Protocols</i>	J. Galvin, K. McCloghrie, J. Davin
1353	<i>Definitions of Managed Objects for Administration of SNMP Parties</i>	K. McCloghrie, J. Davin, J. Galvin
1354	<i>IP Forwarding Table MIB</i>	F. Baker
1387	<i>RIP Version 2 Protocol Analysis</i>	G. Malkin
1389	<i>RIP Version 2 MIB Extension</i>	G. Malkin
1393	<i>Traceroute Using an IP Option</i>	G. Malkin
1397	<i>Default Route Advertisement In BGP2 And BGP3 Versions of the Border Gateway Protocol</i>	D. Haskin
1398	<i>Definitions of Managed Objects for the Ethernet-like Interface Types</i>	F. Kastenholz
1440	<i>SIFT/UFT:Sender-Initiated/Unsolicited File Transfer</i>	R. Troth
1493	<i>Definition of Managed Objects for Bridges</i>	E. Decker, P. Langille, A. Rijssinghani, K. McCloghrie
1540	<i>IAB Official Protocol Standards</i>	J.B. Postel
1583	<i>OSPF Version 2</i>	J.Moy
1647	<i>TN3270 Enhancements</i>	B. Kelly
1700	<i>Assigned Numbers</i>	J.K. Reynolds, J.B. Postel
1723	<i>RIP Version 2 – Carrying Additional Information</i>	G. Malkin
1738	<i>Uniform Resource Locators (URL)</i>	T. Berners-Lee, L. Masinter, M. McCahill
1813	<i>NFS Version 3 Protocol Specification</i>	B. Callaghan, B. Pawlowski, P. Stauback, Sun Microsystems Incorporated
1823	<i>The LDAP Application Program Interface</i>	T. Howes, M. Smith
2460	<i>Internet Protocol, Version 6 (IPv6) Specification</i>	S. Deering, R. Hinden
2052	<i>A DNS RR for specifying the location of services (DNS SRV)</i>	A. Gulbrandsen, P. Vixie

RFC	Title	Author
2104	<i>HMAC: Keyed-Hashing for Message Authentication</i>	H. Krawczyk, M. Bellare, R. Canetti
2222	<i>Simple Authentication and Security Layer (SASL)</i>	J. Myers
2247	<i>Using Domains in LDAP/X.500 Distinguished Names</i>	S. Kille, M. Wahl, A. Grimstad, R. Huber, S. Sataluri
2251	<i>Lightweight Directory Access Protocol (v3)</i>	M. Wahl, T. Howes, S. Kille
2252	<i>Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions</i>	M. Wahl, A. Coulbeck, T. Howes, S. Kille
2253	<i>Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names</i>	M. Wahl, S. Kille, T. Howes
2254	<i>The String Representation of LDAP Search Filters</i>	T. Howes
2255	<i>The LDAP URL Format</i>	T. Howes, M. Smith
2256	<i>A Summary of the X.500 (96) User Schema for use with LDAPv3</i>	M. Wahl
2279	<i>UTF-8, a transformation format of ISO 10646</i>	F. Yergeau
2373	<i>IP Version 6 Addressing Architecture</i>	R. Hinden, S. Deering
2461	<i>Neighbor Discovery for IP Version 6 (IPv6)</i>	T. Narten, E. Nordmark, W. Simpson
2462	<i>IPv6 Stateless Address Autoconfiguration</i>	S. Thomson, T. Narten
2463	<i>Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification</i>	A. Conta, S. Deering
2710	<i>Multicast Listener Discovery (MLD) for IPv6</i>	S. Deering, W. Fenner, B. Haberman
2713	<i>Schema for Representing Java Objects in an LDAP Directory</i>	V. Ryan, S. Seligman, R. Lee
2714	<i>Schema for Representing CORBA Object References in an LDAP Directory</i>	V. Ryan, R. Lee, S. Seligman
2732	<i>Format for Literal IPv6 Addresses in URLs</i>	R. Hinden, B. Carpenter, L. Masinter
2743	<i>Generic Security Service Application Program Interface Version 2, Update 1</i>	J. Linn
2744	<i>Generic Security Service API Version 2 : C-bindings</i>	J. Wray
2820	<i>Access Control Requirements for LDAP</i>	E. Stokes, D. Byrne, B. Blakley, P. Behera
2829	<i>Authentication Methods for LDAP</i>	M. Wahl, H. Alvestrand, J. Hodges, R. Morgan
2830	<i>Lightweight Directory Access Protocol (v3): Extension for Transport Layer Security</i>	J. Hodges, R. Morgan, M. Wahl
2831	<i>Using Digest Authentication as a SASL Mechanism</i>	P. Leach, C. Newman
2849	<i>The LDAP Data Interchange Format (LDIF)</i>	G. Good

RFC	Title	Author
2873	<i>TCP Processing of the IPv4 Precedence Field</i>	X. Xiao, A. Hannan, V. Paxson, E. Crabble
3377	<i>Lightweight Directory Access Protocol (v3): Technical Specification</i>	J. Hodges, R. Morgan
3484	<i>Default Address Selection for Internet Protocol version 6 (IPv6)</i>	R. Draves
3513	<i>Internet Protocol Version 6 (IPv6) Addressing Architecture</i>	R. Hinden, S. Deering
4191	<i>Default Router Preferences and More-Specific Routes</i>	R. Draves, D. Thaler
4517	<i>LDAP Syntaxes and Matching Rules</i>	S. Legg
4523	<i>LDAP Schema Definitions for X.509 Certificates</i>	K. Zeilenga
5095	<i>Deprecation of Type 0 Routing Headers in IPv6</i>	J. Abley, P. Savola, G. Neville-Nei
5175	<i>IPv6 Router Advertisement Flags Option</i>	B. Haberman, R. Hinden
5722	<i>Handling of Overlapping IPv6 Fragments</i>	S. Krishnan
6946	<i>Processing of IPv6 "Atomic" Fragments</i>	F. Gont
6980	<i>Security Implications of IPv6 Fragmentation with IPv6</i>	F. Gont

These documents can be obtained from:

Government Systems, Inc.
Attn: Network Information Center
14200 Park Meadow Drive
Suite 200
Chantilly, VA 22021

Many RFCs are available online. Hard copies of all RFCs are available from the NIC, either individually or on a subscription basis. Online copies are available using FTP from the NIC at `nic.ddn.mil`. Use FTP to download the files, using the following format:

```
RFC:RFC-INDEX.TXT
RFC:RFCnnnn.TXT
RFC:RFCnnnn.PS
```

Where:

nnnn

Is the RFC number.

TXT

Is the text format.

PS

Is the PostScript format.

You can also request RFCs through electronic mail, from the automated NIC mail server, by sending a message to `service@nic.ddn.mil` with a subject line of RFC *nnnn* for text versions or a subject line of RFC *nnnn*.PS for PostScript versions. To request a copy of the RFC index, send a message with a subject line of RFC INDEX.

For more information, contact `nic@nic.ddn.mil`. Information is also available at [Internet Engineering Task Force](http://www.ietf.org) (www.ietf.org).

Appendix F. Abbreviations and acronyms

The following abbreviations and acronyms are used throughout this book.

Acronym	What it stands for
AIX®	Advanced Interactive Executive
ANSI	American National Standards Institute
API	Application Program Interface
APPC	Advanced Program-to-Program Communications
APPN	Advanced Peer-to-Peer Networking
ARP	Address Resolution Protocol
ASCII	American National Standard Code for Information Interchange
ASN.1	Abstract Syntax Notation One
AUI	Attachment Unit Interface
BFS	Byte File System
BIOS	Basic Input/Output System
BNC	Bayonet Neill-Concelman
CCITT	Comite Consultatif International Telegraphique et Telephonique (The International Telegraph and Telephone Consultative Committee)
CLIST	Command List
CMS	Conversational Monitor System
CP	Control Program
CPI	Common Programming Interface
CREN	Corporation for Research and Education Networking
CSD	Corrective Service Diskette
CTC	Channel-to-Channel
CU	Control Unit
CUA	Common User Access
DASD	Direct Access Storage Device
DBCS	Double Byte Character Set
DLL	Dynamic Link Library
DNS	Domain Name System
DOS	Disk Operating System
DPI	Distributed Program Interface
EBCDIC	Extended Binary-Coded Decimal Interchange Code
EISA	Enhanced Industry Standard Adapter
ESCON	Enterprise Systems Connection Architecture
FAT	File Allocation Table

Acronym	What it stands for
FTAM	File Transfer Access Management
FTP	File Transfer Protocol
FTP API	File Transfer Protocol Applications Programming Interface
GCS	Group Control System
GDF	Graphics Data File
HPFS	High Performance File System
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
IGMP	Internet Group Management Protocol
IP	Internet Protocol
IPL	Initial Program Load
ISA	Industry Standard Adapter
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
IUCV	Inter-User Communication Vehicle
JES	Job Entry Subsystem
JIS	Japanese Institute of Standards
JCL	Job Control Language
LAN	Local Area Network
LAPS	LAN Adapter Protocol Support
LDAP	Lightweight Directory Access Protocol
LPQ	Line Printer Query
LPR	Line Printer Client
LPRM	Line Printer Remove
LPRMON	Line Printer Monitor
LU	Logical Unit
MAC	Media Access Control
Mbps	Megabits per second
MBps	Megabytes per second
MCA	Micro Channel Adapter
MIB	Management Information Base
MIH	Missing Interrupt Handler
MILNET	Military Network
MHS	Message Handling System
MTU	Maximum Transmission Unit
MVS™	Multiple Virtual Storage

Acronym	What it stands for
MX	Mail Exchange
NCP	Network Control Program
NDIS	Network Driver Interface Specification
NFS	Network File System
NIC	Network Information Center
NLS	National Language Support
NSFNET	National Science Foundation Network
OS/2	Operating System/2®
OSA	Open Systems Adapter
OSF	Open Software Foundation, Inc.
OSI	Open Systems Interconnection
OSIMF/6000	Open Systems Interconnection Messaging and Filing/6000
OV/MVS	OfficeVision/MVS
OV/VM	OfficeVision/VM
PAD	Packet Assembly/Disassembly
PC	Personal Computer
PCA	Parallel Channel Adapter
PDN	Public Data Network
PDU	Protocol Data Units
PING	Packet Internet Groper
PIOAM	Parallel I/O Access Method
POP	Post Office Protocol
PROFS	Professional Office Systems
PSCA	Personal System Channel Attach
PSDN	Packet Switching Data Network
PU	Physical Unit
PVM	Passthrough Virtual Machine
RACF	Resource Access Control Facility
RARP	Reverse Address Resolution Protocol
REXEC	Remote Execution
REXX	Restructured Extended Executor Language
RFC	Request For Comments
RIP	Routing Information Protocol
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RSCS	Remote Spooling Communications Subsystem
SAA	Systems Application Architecture®

Abbreviations and Acronyms

Acronym	What it stands for
SBCS	Single Byte Character Set
SFS	Shared File System
SLIP	Serial Line Internet Protocol
SMIL	Structure for Management Information
SMTP	Simple Mail Transfer Protocol
SNA	Systems Network Architecture
SNMP	Simple Network Management Protocol
SOA	Start of Authority
SPOOL	Simultaneous Peripheral Operations Online
SQL	IBM Structured Query Language
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TSO	Time Sharing Option
TTL	Time-to-Live
UDP	User Datagram Protocol
VGA	Video Graphic Array
VM	Virtual Machine
VMCF	Virtual Machine Communication Facility
VM/ESA	Virtual Machine/Enterprise System Architecture
VMSES/E	Virtual Machine Serviceability Enhancements Staged/Extended
VTAM®	Virtual Telecommunications Access Method
WAN	Wide Area Network
XDR	eXternal Data Representation

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/VM.

Trademarks

IBM, the IBM logo, and [ibm.com](https://www.ibm.com/legal/copytrade)® are trademarks or registered trademarks of International Business Machines Corp., in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on [IBM Copyright and trademark information](https://www.ibm.com/legal/copytrade) (<https://www.ibm.com/legal/copytrade>).

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java™ and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see:

- The section entitled **IBM Websites** at [IBM Privacy Statement](https://www.ibm.com/privacy) (<https://www.ibm.com/privacy>)
- [Cookies and Similar Technologies](https://www.ibm.com/privacy#Cookies_and_Similar_Technologies) (https://www.ibm.com/privacy#Cookies_and_Similar_Technologies)

Bibliography

This topic lists the publications in the z/VM library. For abstracts of the z/VM publications, see [z/VM: General Information](#).

Where to Get z/VM Information

The current z/VM product documentation is available in [IBM Documentation - z/VM \(https://www.ibm.com/docs/en/zvm\)](https://www.ibm.com/docs/en/zvm).

z/VM Base Library

Overview

- [z/VM: License Information](#), GI13-4377
- [z/VM: General Information](#), GC24-6286

Installation, Migration, and Service

- [z/VM: Installation Guide](#), GC24-6292
- [z/VM: Migration Guide](#), GC24-6294
- [z/VM: Service Guide](#), GC24-6325
- [z/VM: VMSES/E Introduction and Reference](#), GC24-6336

Planning and Administration

- [z/VM: CMS File Pool Planning, Administration, and Operation](#), SC24-6261
- [z/VM: CMS Planning and Administration](#), SC24-6264
- [z/VM: Connectivity](#), SC24-6267
- [z/VM: CP Planning and Administration](#), SC24-6271
- [z/VM: Getting Started with Linux on IBM Z](#), SC24-6287
- [z/VM: Group Control System](#), SC24-6289
- [z/VM: I/O Configuration](#), SC24-6291
- [z/VM: Running Guest Operating Systems](#), SC24-6321
- [z/VM: Saved Segments Planning and Administration](#), SC24-6322
- [z/VM: Secure Configuration Guide](#), SC24-6323

Customization and Tuning

- [z/VM: CP Exit Customization](#), SC24-6269
- [z/VM: Performance](#), SC24-6301

Operation and Use

- [z/VM: CMS Commands and Utilities Reference](#), SC24-6260
- [z/VM: CMS Primer](#), SC24-6265
- [z/VM: CMS User's Guide](#), SC24-6266
- [z/VM: CP Commands and Utilities Reference](#), SC24-6268

- [z/VM: System Operation](#), SC24-6326
- [z/VM: Virtual Machine Operation](#), SC24-6334
- [z/VM: XEDIT Commands and Macros Reference](#), SC24-6337
- [z/VM: XEDIT User's Guide](#), SC24-6338

Application Programming

- [z/VM: CMS Application Development Guide](#), SC24-6256
- [z/VM: CMS Application Development Guide for Assembler](#), SC24-6257
- [z/VM: CMS Application Multitasking](#), SC24-6258
- [z/VM: CMS Callable Services Reference](#), SC24-6259
- [z/VM: CMS Macros and Functions Reference](#), SC24-6262
- [z/VM: CMS Pipelines User's Guide and Reference](#), SC24-6252
- [z/VM: CP Programming Services](#), SC24-6272
- [z/VM: CPI Communications User's Guide](#), SC24-6273
- [z/VM: ESA/XC Principles of Operation](#), SC24-6285
- [z/VM: Language Environment User's Guide](#), SC24-6293
- [z/VM: OpenExtensions Advanced Application Programming Tools](#), SC24-6295
- [z/VM: OpenExtensions Callable Services Reference](#), SC24-6296
- [z/VM: OpenExtensions Commands Reference](#), SC24-6297
- [z/VM: OpenExtensions POSIX Conformance Document](#), GC24-6298
- [z/VM: OpenExtensions User's Guide](#), SC24-6299
- [z/VM: Program Management Binder for CMS](#), SC24-6304
- [z/VM: Reusable Server Kernel Programmer's Guide and Reference](#), SC24-6313
- [z/VM: REXX/VM Reference](#), SC24-6314
- [z/VM: REXX/VM User's Guide](#), SC24-6315
- [z/VM: Systems Management Application Programming](#), SC24-6327
- [z/VM: z/Architecture Extended Configuration \(z/XC\) Principles of Operation](#), SC27-4940

Diagnosis

- [z/VM: CMS and REXX/VM Messages and Codes](#), GC24-6255
- [z/VM: CP Messages and Codes](#), GC24-6270
- [z/VM: Diagnosis Guide](#), GC24-6280
- [z/VM: Dump Viewing Facility](#), GC24-6284
- [z/VM: Other Components Messages and Codes](#), GC24-6300
- [z/VM: VM Dump Tool](#), GC24-6335

z/VM Facilities and Features

Data Facility Storage Management Subsystem for z/VM

- [z/VM: DFSMS/VM Customization](#), SC24-6274
- [z/VM: DFSMS/VM Diagnosis Guide](#), GC24-6275
- [z/VM: DFSMS/VM Messages and Codes](#), GC24-6276
- [z/VM: DFSMS/VM Planning Guide](#), SC24-6277

- *z/VM: DFSMS/VM Removable Media Services*, SC24-6278
- *z/VM: DFSMS/VM Storage Administration*, SC24-6279

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6281
- *z/VM: Directory Maintenance Facility Messages*, GC24-6282
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6283

Open Systems Adapter

- Open Systems Adapter/Support Facility on the Hardware Management Console (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC14-7580-02.pdf), SC14-7580
- Open Systems Adapter-Express ICC 3215 Support (<https://www.ibm.com/docs/en/zos/2.3.0?topic=osa-icc-3215-support>), SA23-2247
- Open Systems Adapter Integrated Console Controller User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC27-9003-02.pdf), SC27-9003
- Open Systems Adapter-Express Customer's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/iaa2z1f0.pdf), SA22-7935

Performance Toolkit for z/VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

The following publications contain sections that provide information about z/VM Performance Data Pump, which is licensed with Performance Toolkit for z/VM.

- *z/VM: Performance*, SC24-6301. See *z/VM Performance Data Pump*.
- *z/VM: Other Components Messages and Codes*, GC24-6300. See *Data Pump Messages*.

RACF Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6305
- *z/VM: RACF Security Server Command Language Reference*, SC24-6306
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6307
- *z/VM: RACF Security Server General User's Guide*, SC24-6308
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6309
- *z/VM: RACF Security Server Messages and Codes*, GC24-6310
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6311
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6312
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6316
- *z/VM: RSCS Networking Exit Customization*, SC24-6317
- *z/VM: RSCS Networking Messages and Codes*, GC24-6318
- *z/VM: RSCS Networking Operation and Use*, SC24-6319
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6320

TCP/IP for z/VM

- *z/VM: TCP/IP Diagnosis Guide*, GC24-6328
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6329
- *z/VM: TCP/IP Messages and Codes*, GC24-6330
- *z/VM: TCP/IP Planning and Customization*, SC24-6331
- *z/VM: TCP/IP Programmer's Reference*, SC24-6332
- *z/VM: TCP/IP User's Guide*, SC24-6333

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ickug00_v2r5.pdf), GC35-0033

Related Products

XL C++ for z/VM

- *XL C/C++ for z/VM: Runtime Library Reference*, SC09-7624
- *XL C/C++ for z/VM: User's Guide*, SC09-7625

z/OS

IBM Documentation - z/OS (<https://www.ibm.com/docs/en/zos>)

Other TCP/IP Related Publications

This section lists other publications, outside the z/VM 7.4 library, that you may find helpful.

- *TCP/IP Tutorial and Technical Overview*, GG24-3376
- *TCP/IP Illustrated, Volume 1: The Protocols*, SR28-5586
- *Internetworking with TCP/IP Volume I: Principles, Protocols, and Architecture*, SC31-6144
- *Internetworking With TCP/IP Volume II: Implementation and Internals*, SC31-6145
- *Internetworking With TCP/IP Volume III: Client-Server Programming and Applications*, SC31-6146
- *DNS and BIND in a Nutshell*, SR28-4970
- "MIB II Extends SNMP Interoperability," C. Vanderberg, *Data Communications*, October 1990.
- "Network Management and the Design of SNMP," J.D. Case, J.R. Davin, M.S. Fedor, M.L. Schoffstall.
- "Network Management of TCP/IP Networks: Present and Future," A. Ben-Artzi, A. Chandna, V. Warriar.
- "Special Issue: Network Management and Network Security," *ConneXions-The Interoperability Report*, Volume 4, No. 8, August 1990.
- *The Art of Distributed Application: Programming Techniques for Remote Procedure Calls*, John R. Corbin, Springer-Verlog, 1991.
- *The Simple Book: An Introduction to Management of TCP/IP-based Internets*, Marshall T Rose, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Index

A

- abbreviations and acronyms [355](#)
- ACCEPT (IUCV) [159](#)
- address families, socket [3](#)
- address information file, specifying [31](#)
- address, socket [5](#)
- AddUserNote [60](#)
- AF_INET socket domain
 - definition [4](#)
- AF_INET6 socket domain
 - definition [4](#)
- AF_IUCV socket domain
 - definition [4](#)
- AF_UNIX socket domain
 - definition [4](#)
- aliases information file, specifying [30](#)
- APITYPE=3 (multiple request) [146](#)
- applications program interface (API)
 - IUCV sockets API [142](#)
- APPTYPE environment variable [30](#)
- ASCII to EBCDIC translation tables, specifying [31](#)
- asynchronous communication, sequence (Pascal API) [41](#)
- auth_destroy() [194](#)
- authnone_create() [194](#)
- authunix_create_default() [195](#)
- authunix_create() [194](#)

B

- BeginTcpIp (Pascal) [60](#)
- Berkeley socket implementation [25](#)
- big-endian byte ordering convention [6](#)
- BIND (IUCV) [160](#)
- BUFFERspaceAVAILABLE (VMCF) [134](#)
- byte order conventions [6](#)

C

- C socket application programming interface [1](#)
- C socket programs, examples
 - TCP client [35](#)
 - TCP server [36](#)
 - UDP client [38](#)
 - UDP server [38](#)
- C sockets quick reference [32](#)
- callrpc() [195](#)
- calls
 - IUCV socket [159](#)
- CLEARtextRESUMED (VMCF) [135](#)
- ClearTimer [60](#)
- client
 - remote procedure calls [187](#)
 - SNMP DPI programs [252](#)
- client verification exit, SMTP [285](#)
- clnt_broadcast() [196](#)
- clnt_call() [197](#)

- clnt_destroy() [199](#)
- clnt_freeres() [199](#)
- clnt_geterr() [200](#)
- clnt_pcreateerror() [200](#)
- clnt_perrno() [201](#)
- clnt_perror() [201](#)
- clnt_spcreateerror() [201](#)
- clnt_sperrno() [202](#)
- clnt_sperror() [202](#)
- clnttcp_create() [198](#)
- clntraw_create() [203](#)
- clnttcp_create() [203](#)
- clntudp_create() [204](#)
- command exit, SMTP [297](#)
- compiling and linking
 - SNMP DPI [244](#)
- CONNECT (IUCV) [162](#)
- connection information record (Pascal) [43](#)
- connection states (Pascal) [42](#)
- CONNECTIONclosing (Pascal) [42](#)
- CONNECTIONstateCHANGED (VMCF) [135](#)
- CreateTimer [61](#)

D

- DATA [276](#)
- data structures
 - Pascal [41](#)
 - VMCF [113](#)
- DATAdelivered (VMCF) [136](#)
- datagram sockets [4](#)
- DestroyTimer [61](#)
- DPI client program [252](#), [255](#)

E

- EBCDIC to ASCII translation tables, specifying [31](#)
- EHLO [274](#)
- encrypting data on an IUCV socket [147](#)
- EndTcpIp (Pascal) [61](#)
- envelope, SMTP
 - description [273](#)
 - example [281](#)
- environment variables
 - APPTYPE [30](#)
 - HOSTALIASES [30](#)
 - X_ADDR [31](#)
 - X_SITE [31](#)
 - X_XLATE [31](#)
- ETC SERVICES file [345](#)
- exit routines, SMTP [285](#), [303](#)
- Exits, Server
 - Telnet [305](#)
- EXPN [280](#)
- eXternal Data Representation protocol, general information [187](#)

F

FCNTL (IUCV) [163](#)
fDPIparse() [245](#)
file specification record (Pascal) [53](#)
files
 ETC SERVICES [345](#)
ForeignSocket [43](#), [44](#)

G

get_myaddress() [205](#)
GET-NEXT, SNMP DPI request [243](#)
GET, SNMP DPI request [242](#)
GETCLIENTID (IUCV) [164](#)
GETHOSTID (IUCV) [165](#)
Gethostname (REXX) [165](#)
GetHostNumber [62](#)
GetHostResol [62](#)
GetHostString [62](#)
GetIdentity [63](#)
GetNextNote [63](#)
GETPEERNAME (IUCV) [166](#)
GetSmsg [64](#)
GETSOCKNAME (IUCV) [167](#)
GETSOCKOPT (IUCV) [168](#)
GIVESOCKET (IUCV) [169](#)

H

Handle (Pascal) [64](#)
handling external interrupts [55](#)
HELO [274](#)
HELP [277](#)
host byte order [6](#)
host information file, specifying [31](#)
Host lookup routines [58](#)
HOSTALIASES environment variable [30](#)
HOSTS ADDRINFO file, replacing [31](#)
HOSTS SITEINFO file, replacing [31](#)

I

ICTX plug-in
 SMF Record Type 83 subtype 4 records [331](#)
initialization procedures, TCP/UDP (Pascal) [54](#)
inter-communication vehicle sockets [141](#)
IOCTL (IUCV) [170](#)
IPUSER variable, returned by socket call [145](#)
IsLocalAddress [65](#)
IsLocalHost [65](#)
IUCV socket API [142](#)
IUCV socket call, buffer formats
 ACCEPT [160](#)
 BIND [161](#)
 CANCEL [161](#)
 CLOSE [162](#)
 CONNECT [163](#)
 FCNTL [164](#)
 GETCLIENTID [165](#)
 GETHOSTID [165](#)
 GETHOSTNAME [166](#)
 GETPEERNAME [167](#)

IUCV socket call, buffer formats (*continued*)

 GETSOCKNAME [168](#)
 GETSOCKOPT [169](#)
 GIVESOCKET [170](#)
 IOCTL [171](#)
 LASTERRNO [186](#)
 LISTEN [174](#)
 READ [175](#)
 READV [175](#)
 RECV [176](#)
 RECVMSG [176](#)
 RRCVFROM [176](#)
 SELECT [177](#)
 SELECT and SELECTEX
 descriptor sets [176](#)
 DESCRIPTOR_SET macro [177](#)
 FD_CLR macro [177](#)
 FD_ISSET macro [177](#)
 SELECTEX [177](#)
 SEND [179](#)
 SENDTO [180](#)
 SHUTDOWN [183](#)
 SOCKET [183](#)
 TAKESOCKET [184](#)
 WRITE [185](#)
 WRITEV [185](#)

IUCV socket calls

 ACCEPT [159](#)
 BIND [160](#)
 CLOSE [162](#)
 CONNECT [162](#)
 FCNTL [163](#)
 GETCLIENTID [164](#)
 GETHOSTID [165](#)
 GETHOSTNAME [165](#)
 GETPEERNAME [166](#)
 GETSOCKNAME [167](#)
 GETSOCKOPT [168](#)
 GIVESOCKET [169](#)
 IOCTL [170](#)
 LISTEN [173](#)
 MAXDESC [174](#)
 READ [174](#)
 READV [174](#)
 RECV [175](#)
 RRCVFROM [175](#)
 RECVMSG [175](#)
 SELECT [176](#)
 SELECTEX [176](#)
 SEND [178](#)
 SENDMSG [179](#)
 SENDTO [180](#)
 SETSOCKOPT [181](#)
 SHUTDOWN [182](#)
 SOCKET [183](#)
 TAKESOCKET [184](#)
 WRITE [185](#)
 WRITEV [185](#)

IUCV sockets, general

 connect parameters [143](#)
 general information [141](#)
 issuing socket calls [145](#)
 lasterno special request [186](#)
 multiple-req socket program (apitype=3) [143](#), [146](#)

- IUCV sockets, general (*continued*)
 - path severance [144](#)
 - response from initial message [143](#)
 - response from TCPIP [147](#)
 - restrictions [141](#)
 - send parameters, initial message [143](#)
 - sever, application initiated [145](#)
 - sever, clean_up of stream sockets [145](#)
 - sever, TCPIP initiated [145](#)
 - socket API [142](#)
 - socket call syntax [148](#)
 - waiting for response from TCPIP [147](#)
- IUCV Sockets, prerequisite knowledge [141](#)
- IUCV, subsystem communication macros
 - IUCV CONNECT [143](#)
 - IUCV PURGE [147](#)
 - IUCV REJECT [147](#), [186](#)
 - IUCV REPLY [147](#)
 - IUCV SEND [143](#)
 - IUCVMCOM SEVER [146](#)

L

- libraries
 - remote procedure calls [26](#), [29](#)
 - SNMP DPI [245](#)
 - sockets [1](#)
- LISTEN (IUCV) [173](#)
- LISTENING (Pascal) [42](#)
- little-endian byte ordering convention [6](#)

M

- mail forwarding exit, SMTP [291](#)
- MAILFROM [275](#)
- Management Information Base (MIB) [241](#), [243](#)
- MAXDESC (IUCV) [174](#)
- message examples, notation used in [xxii](#)
- messages
 - Pascal [51](#)
- mkDPiRegister() [246](#)
- mkDPiResponse() [247](#)
- mkDPiSet() [248](#)
- mkDPiTrap() [249](#)
- MonCommand [66](#)
- Monitor procedures [57](#)
- monitor query [67](#)
- MonQuery [67](#)

N

- network byte order [6](#)
- NONEXISTENT (Pascal) [42](#)
- NOOP [277](#)
- notation used in message and response examples [xxii](#)
- notification record (Pascal) [45](#)
- notifications
 - notifications (Pascal) [54](#)
 - notifications (VMCF) [134](#)
 - notifications, specifying those to receive (VMCF) [119](#)
- NotifyIo [68](#)

O

- OPEN (Pascal) [42](#)
- OpenAttemptTimeout [43](#)

P

- parse [246](#)
- partner certificate
 - requesting details from [22](#)
- Pascal
 - API, description [41](#)
 - assembler calls
 - RTcpExtRupt [74](#)
 - RTcpVmcfRupt [74](#)
 - asynchronous communication, general sequence [41](#)
 - Compiler, IBM VS Pascal & Library [41](#)
 - connection state type
 - CONNECTIONclosing [42](#)
 - LISTENING [42](#)
 - NONEXISTENT [42](#)
 - OPEN [42](#)
 - RECEIVINGonly [42](#)
 - SENDINGonly [42](#)
 - TRYINGtoOPEN [42](#)
 - data structures [41](#)
 - description
 - connection information record [43](#)
 - connection states [42](#)
 - file specification record [53](#)
 - notification record [45](#)
 - return codes [337](#), [340](#)
 - sample program [108](#)
 - software requirements [41](#)
 - path addresses, SMTP [283](#)
 - pDPiPacket() [251](#)
 - PING interface [57](#)
 - PingRequest [69](#)
 - PINGresponse (VMCF) [136](#)
 - pmap_getmaps() [205](#)
 - pmap_getport() [206](#)
 - pmap_rmtcall() [206](#)
 - pmap_set() [207](#)
 - pmap_unset() [208](#)
 - port
 - port assignments [190](#)
 - unspecified ports [89](#)
 - port assignments [345](#)
 - porting
 - remote procedure calls [192](#)
 - socket applications [23](#), [25](#)
 - portmap [190](#)
 - Portmapper [190](#)
 - procedure calls, Pascal
 - descriptions [53](#)
 - handling external interrupts
 - RTcpExtRupt [74](#)
 - RTcpVmcfRupt [74](#)
 - TcpExtRupt [83](#)
 - TcpVmcfRupt [101](#)
 - Host lookup routines
 - GetHostNumber [62](#)
 - GetHostResol [62](#)
 - GetHostString [62](#)

procedure calls, Pascal (*continued*)

Host lookup routines (*continued*)

GetIdentity [63](#)
IsLocalAddress [65](#)
IsLocalHost [65](#)

Monitor procedures

MonCommand [66](#)
MonQuery [67](#)

notifications

description [54](#)
GetNextNote [63](#)
Handle [64](#)
Unhandle [107](#)

Other routines

AddUserNote [60](#)
GetSmsg [64](#)
ReadXlateTable [73](#)
SayCalRe [75](#)
SayConSt [75](#)
SayIntAd [76](#)
SayIntNum [76](#)
SayNotEn [76](#)
SayPorTy [77](#)
SayProTy [77](#)

Raw IP interface

RawIpClose [70](#)
RawIpOpen [71](#)
RawIpReceive [72](#)
RawIpSend [72](#)

TCP communication procedures

TcpAbort [81](#)
TcpClose [82](#)
TcpFReceive, TcpReceive, and TcpWaitReceive [83](#)
TcpFSend, TcpSend, and TcpWaitSend [86](#)
TcpOpen and TcpWaitOpen [88](#)
TcpOption [90](#)
TcpStatus [100](#)

TCP/UDP initialization procedures

BeginTcpIp [60](#)
StartTcpNotice [78](#)
TcpNameChange [88](#)

TCP/UDP termination procedure, EndTcpIp [55](#)

Timer routines

ClearTimer [60](#)
CreateTimer [61](#)
DestroyTimer [61](#)
SetTimer [77](#)

UDP communication procedures

Udp6Open [102](#)
Udp6Send [102](#)
UdpClose [103](#)
UdpNReceive [104](#)
UdpOpen [104](#)
UdpReceive [105](#)
UdpSend [106](#)

Q

query_DPI_port() [252](#)

QUEUE [278](#)

quick reference tables

SNMP DPI routines [244](#)
socket calls [32](#)

QUIT [277](#)

R

Raw IP interface [58](#)

raw sockets [4](#)

RawIpClose (Pascal) [70](#)

RawIpOpen (Pascal) [71](#)

RAWIPpacketsDELIVERED (VMCF) [137](#)

RawIpReceive (Pascal) [72](#)

RawIpSend (Pascal) [72](#)

RAWIPspaceAVAILABLE (VMCF) [137](#)

RCPT TO [276](#)

ReadXlateTable [73](#)

RECEIVINGonly (Pascal) [42](#)

REGISTER, SNMP DPI request [244](#)

registerrpc() [208](#)

related protocols [349](#)

remote auditing extended operation [324](#)

remote auditing extended operation response codes [328](#)

remote authorization extended operation [320](#)

remote authorization extended operation response codes [322](#)

remote procedure calls (RPCs)

accessing system return messages [192](#)

auth_destroy() [194](#)

authnone_create() [194](#)

authunix_create_default() [195](#)

authunix_create() [194](#)

callrpc() [195](#)

clnt_broadcast() [196](#)

clnt_call() [197](#)

clnt_control() [198](#)

clnt_create() [198](#)

clnt_destroy() [199](#)

clnt_freeres() [199](#)

clnt_geterr() [200](#)

clnt_pcreateerror() [200](#)

clnt_perrno() [201](#)

clnt_perror() [201](#)

clnt_screateerror() [201](#)

clnt_sperrno() [202](#)

clnt_sperror() [202](#)

clntraw_create() [203](#)

clnttcp_create() [203](#)

clntudp_create() [204](#)

enum clnt_stat structure [191](#)

enumerations [192](#)

general information [187](#)

get_myaddress() [205](#)

getrpcport() [205](#)

interface [187](#)

library [26, 29](#)

pmap_getmaps() [205](#)

pmap_getport() [206](#)

pmap_rmtcall() [206](#)

pmap_set() [207](#)

pmap_unset() [208](#)

porting [192](#)

Portmapper

contacting [190](#)

target assistance [190](#)

printing system return messages [192](#)

registerrpc() [208](#)

rpc_createerr [193](#)

RPCGEN command [190](#)

remote procedure calls (RPCs) (*continued*)

- [svc_destroy\(\)](#) [209](#)
- [svc_fds\(\)](#) [193](#)
- [svc_freeargs\(\)](#) [210](#)
- [svc_getargs\(\)](#) [210](#)
- [svc_getcaller\(\)](#) [210](#)
- [svc_getreq\(\)](#) [211](#)
- [svc_register\(\)](#) [212](#)
- [svc_run\(\)](#) [212](#)
- [svc_sendreply\(\)](#) [213](#)
- [svc_unregister\(\)](#) [213](#)
- [svcerr_auth\(\)](#) [213](#)
- [svcerr_decode\(\)](#) [214](#)
- [svcerr_noproc\(\)](#) [214](#)
- [svcerr_noprog\(\)](#) [215](#)
- [svcerr_progvers\(\)](#) [215](#)
- [svcerr_systemerr\(\)](#) [215](#)
- [svcerr_weakauth\(\)](#) [216](#)
- [svccraw_create\(\)](#) [216](#)
- [svctcp_create\(\)](#) [216](#)
- [svcdup_create\(\)](#) [217](#)
- [xdr_accepted_reply\(\)](#) [217](#)
- [xdr_array\(\)](#) [218](#)
- [xdr_authunix_parms\(\)](#) [218](#)
- [xdr_bool\(\)](#) [219](#)
- [xdr_bytes\(\)](#) [219](#)
- [xdr_callhdr\(\)](#) [220](#)
- [xdr_callmsg\(\)](#) [220](#)
- [xdr_double\(\)](#) [220](#)
- [xdr_enum\(\)](#) [221](#)
- [xdr_float\(\)](#) [222](#)
- [xdr_inline\(\)](#) [222](#)
- [xdr_int\(\)](#) [223](#)
- [xdr_long\(\)](#) [223](#)
- [xdr_opaque_auth\(\)](#) [224](#)
- [xdr_opaque\(\)](#) [224](#)
- [xdr_pmap\(\)](#) [224](#)
- [xdr_pmaplist\(\)](#) [225](#)
- [xdr_pointer\(\)](#) [225](#)
- [xdr_reference\(\)](#) [226](#)
- [xdr_rejected_reply\(\)](#) [226](#)
- [xdr_replymsg\(\)](#) [227](#)
- [xdr_short\(\)](#) [227](#)
- [xdr_string\(\)](#) [228](#)
- [xdr_u_int\(\)](#) [228](#)
- [xdr_u_long\(\)](#) [228](#)
- [xdr_u_short\(\)](#) [229](#)
- [xdr_union\(\)](#) [229](#)
- [xdr_vector\(\)](#) [230](#)
- [xdr_void\(\)](#) [231](#)
- [xdr_wrapstring\(\)](#) [231](#)
- [xdrmem_create\(\)](#) [231](#)
- [xdrrec_create\(\)](#) [232](#)
- [xdrrec_endofrecord\(\)](#) [232](#)
- [xdrrec_eof\(\)](#) [233](#)
- [xdrrec_skiprecord\(\)](#) [233](#)
- [xdrstdio_create\(\)](#) [233](#)
- [xpirt_register\(\)](#) [234](#)
- [xpirt_unregister\(\)](#) [234](#)
- requesting details for a secure connection [22](#)
- RESOURCESavailable (VMCF) [138](#)
- response examples, notation used in [xxii](#)
- return codes
 - Pascal [337](#), [340](#)

- RPC sample programs
 - client [236](#)
 - raw data stream [238](#)
 - server [236](#)
- rpc_createerr [193](#)
- RPCGEN command [190](#)
- RSET [277](#)

S

- SayCalRe [75](#)
- SayConSt [75](#)
- SayIntAd [43](#), [44](#), [76](#)
- SayIntNum [76](#)
- SayNotEn [76](#)
- SayPorTy [77](#)
- SayProTy [77](#)
- secure connection
 - starting [21](#)
- secure connection considerations [21](#)
- SELECT (IUCV) [176](#)
- SELECTEX (IUCV) [176](#)
- SEND (IUCV) [178](#)
- SENDINGonly [42](#)
- SENDMSG (IUCV) [179](#)
- SENDTO (IUCV) [180](#)
- server
 - remote procedure calls [189](#), [190](#), [236](#), [239](#)
 - sockets [36](#)
- SET, SNMP DPI request [243](#)
- SETSOCKOPT (IUCV) [181](#)
- SetTimer [77](#)
- setting up authorization for working with remote services [320](#)
- SHUTDOWN (IUCV) [182](#)
- SMSG command (VMCF) [64](#)
- SMTP exit routines [285](#), [303](#)
- SMTP interface
 - batch command files, format [283](#)
 - batch examples
 - converting to batch format [283](#)
 - querying delivery queues [284](#)
 - sending mail [284](#)
 - envelope, description of [281](#)
 - path addresses [283](#)
 - responses [282](#)
- SMTP commands
 - DATA [276](#)
 - EHLO [274](#)
 - EXPN [280](#)
 - HELO [274](#)
 - HELP [277](#)
 - MAILFROM [275](#)
 - NOOP [277](#)
 - QUEUE [278](#)
 - QUIT [277](#)
 - RCPT TO [276](#)
 - RSET [277](#)
 - TICK [281](#)
 - VERB [281](#)
 - VERFY [280](#)
- SMTP transactions [273](#)
- SMTPSEND EXEC [284](#)
- SNMP agent distributed program interface (DPI) [241](#), [252](#)

SNMP DPI

- agents [241](#)
- compiling and linking [244](#)
- requests
 - GET [242](#)
 - GET-NEXT [243](#)
 - REGISTER [244](#)
 - SET [243](#)
 - TRAP [244](#)

routines

- DPIdebug() [245](#)
- fDIParse() [245](#)
- mkDPIlist() [246](#)
- mkDPIregister() [246](#)
- mkDPIresponse() [247](#)
- mkDPIset() [248](#)
- mkDPItrap() [249](#)
- mkDPItrape() [249](#)
- pDIPacket() [251](#)
- query_DPI_port() [252](#)
- Quick Reference [244](#)

software requirements [244](#)

subagents [241](#)

SOCKET (IUCV) [183](#)

socket calls

IUCV [159](#)

socket record [44](#)

sockets, C

- address [5](#)
- address families [3](#)
- addressing
 - AF_INET domain [4, 5](#)
 - AF_IUCV domain [8](#)
 - AF_UNIX domain [7](#)
- AF_INET domain
 - addressing [5](#)
 - client perspective [11](#)
 - definition [4](#)
 - server perspective [9](#)
 - TCP client program example [35](#)
 - TCP server program example [36](#)
 - UDP client program example [38](#)
 - UDP server program example [38](#)

AF_INET6 domain

addressing [5](#)

AF_IUCV domain

addressing [8](#)

definition [4](#)

AF_UNIX domain

addressing [7](#)

definition [4](#)

API [1](#)

compiling and linking a sockets program

VM TCP/IP C sockets program [28](#)

z/VM C sockets program [26](#)

conversation, client/server

client perspective for AF_INET [11](#)

server perspective for AF_INET [9](#)

TCP socket session, typical [11](#)

UDP socket session, typical [12](#)

definition [2](#)

environment variables [30](#)

example programs

TCP client [35](#)

sockets, C (*continued*)

example programs (*continued*)

TCP server [36](#)

UDP client [38](#)

UDP server [38](#)

header files [18](#)

incompatibilities

with Berkeley socket implementation [25](#)

with OS/390 C sockets implementation [25](#)

with VM TCP/IP C sockets implementation [23](#)

internetworking overview [2](#)

multithreading [19](#)

network application example [14](#)

POSIX signals [20](#)

quick reference [32](#)

running a sockets program

BFS, residing in [31](#)

environment variables, using [30](#)

minidisk or SFS directory, residing on [32](#)

preparing for [29](#)

transport protocols [2](#)

types

datagram [4](#)

guidelines for using [5](#)

raw [4](#)

stream [4](#)

z/VM implementation, details of

header files [18](#)

incompatibilities with Berkeley sockets [25](#)

incompatibilities with OS/390 C sockets [25](#)

incompatibilities with VM TCP/IP C sockets [23](#)

miscellaneous implementation notes [22](#)

multithreading [19](#)

POSIX signals [20](#)

software requirements

Pascal [41](#)

sockets [1](#)

STANDARD TCPXLBIN file, replacing [31](#)

StartTcpNotice (Pascal) [78](#)

stopping a secure connection [21](#)

stream sockets [4](#)

stubs [191](#)

svc_destroy() [209](#)

svc_fds() [193](#)

svc_freeargs() [210](#)

svc_getargs() [210](#)

svc_getcaller() [210](#)

svc_getreq() [211](#)

svc_register() [212](#)

svc_run() [212](#)

svc_sendreply() [213](#)

svc_unregister() [213](#)

svcerr_auth() [213](#)

svcerr_decode() [214](#)

svcerr_noproc() [214](#)

svcerr_noprogram() [215](#)

svcerr_progvers() [215](#)

svcerr_systemerr() [215](#)

svcerr_weakauth() [216](#)

svccraw_create() [216](#)

svctcp_create() [216](#)

svcudp_create() [217](#)

syntax diagrams, how to read [xx](#)

sys/socket.h header file [5](#)

system return codes [341](#)

T

TAKESOCKET (IUCV) [184](#)

TCP communication procedures (Pascal) [55](#)

TCP port assignments [345](#)

TCP socket session, typical [11](#)

TCP/IP initialization and termination procedures (VMCF)

begin TCP/IP service [119](#)

close a TCP connection [120](#)

close a UDP port [127](#)

determine whether an address is local [131](#)

end TCP/IP service [119](#)

instruct TCPIP to obey a file of commands [132](#)

obtain current status of TCP connection [124](#)

obtain status information from TCPIP [132](#)

open a UDP port [128](#), [129](#)

open TCP connection [121](#)

receive raw IP packets of a given protocol [130](#)

receive TCP data with FRECEIVETcp function [120](#)

receive TCP data with RECEIVETcp function [123](#)

receive UDP data [128](#)

send an ICMP echo request [133](#)

send raw IP packets [131](#)

send TCP data [123](#)

send UDP data [128](#), [129](#)

specifying the notifications to receive [119](#)

tell TCPIP that your program will no longer use a particular IP protocol [130](#)

tell TCPIP that your program will use a particular IP protocol [130](#)

TCP/UDP initialization procedures (Pascal) [54](#)

TCP/UDP termination procedure (Pascal) [55](#)

TCP/UDP/IP API (Pascal)

connection information record [43](#)

connection state [42](#)

data structures [41](#)

file specification record [53](#)

handling external interrupts [55](#)

notification record [45](#)

notifications [54](#)

socket record [44](#)

software requirements [41](#)

using procedure calls [53](#)

TcpAbort (Pascal) [81](#)

TcpClose (Pascal) [82](#)

TcpExtRupt [83](#)

TcpFReceive (Pascal) [83](#)

TcpFSend (Pascal) [86](#)

TCPIP ATCPPSRC file (Pascal) [41](#)

TCPLOAD

EXEC [335](#)

using [335](#)

TcpNameChange [88](#)

TcpOpen (Pascal) [52](#), [88](#)

TcpOption (Pascal) [90](#)

TcpReceive (Pascal) [83](#)

TcpSend (Pascal) [86](#)

TcpStatus (Pascal) [100](#)

TcpVmcfRupt [101](#)

TcpWaitOpen (Pascal) [52](#), [88](#)

TcpWaitReceive [83](#)

TcpWaitSend [86](#)

Textlib (TXTLIB) Files

CLIB [335](#)

CMSLIB [335](#)

COMMTXT [335](#)

GLOBAL [335](#)

IBMLIB [335](#)

PASCAL [335](#)

RPCLIB [335](#)

SCEELKED [335](#)

TCPASCAL [335](#)

TCPLANG [335](#)

TICK [281](#)

Timer routines [58](#)

TLS/SSL server

determining availability [22](#)

trademarks [360](#)

transactions, SMTP [273](#)

translation information file, specifying [31](#)

TRAP, SNMP DPI request [244](#)

TRYINGtoOPEN (Pascal) [42](#)

U

UDP communication procedure [57](#), [102](#), [104](#)

UDP port assignments [345](#)

UDP socket session, typical [12](#)

UdpClose (Pascal) [103](#)

UDPdatagramDELIVERED (VMCF) [52](#), [53](#), [138](#)

UDPdatagramSPACEavailable (VMCF) [139](#)

UdpNReceive [104](#)

UdpReceive (Pascal) [53](#), [105](#)

UDPresourcesAVAILABLE (VMCF) [139](#)

UdpSend (Pascal) [102](#), [106](#)

Unhandle (Pascal) [107](#)

UnNotifyIo [107](#)

UnpackedBytes [43](#), [44](#)

URGENTpending (VMCF) [139](#)

user exit routines, SMTP [285](#), [303](#)

using remote authorization and auditing [319](#)

V

variables, environment [30](#)

VERB [281](#)

Virtual Machine Communication Facility (VMCF) Interface

CALLCODE notifications

ACTIVEprobe [134](#)

BUFFERspaceAVAILABLE [134](#)

CONNECTIONstate- CHANGED [135](#)

DATAdelivered [136](#)

DUMMYprobe [136](#)

PINGresponse [136](#)

RAWIPpacketsDELIVERED [137](#)

RAWIPspaceAVAILABLE [137](#)

RESOURCESavailable [138](#)

UDPdatagramDELIVERED [138](#)

UDPdatagramSPACEavailable [139](#)

UDPresourcesAVAILABLE [139](#)

URGENTpending [139](#)

CALLCODE system queries

IShostLOCAL [131](#)

MONITORcommand [132](#)

MONITORquery [132](#)

CALLCODE system queries (*continued*)

PINGreq [133](#)

functions [115](#)

general information

data structures [113](#), [122](#)

use of VMCF interrupt header fields [114](#)

use of VMCF parameter list fields [114](#)

IP CALLCODE requests

CLOSErawip [130](#)

OPENrawip [130](#)

RECEIVERawip [130](#)

SENDrawip [131](#)

TCP CALLCODE requests

CLOSEtcp [120](#)

FRECEIVtcp [120](#)

FSENDtcp [123](#)

OPENTcp [121](#)

OPTIONtcp [122](#)

RECEIVtcp [123](#)

SENDtcp [123](#)

STATUSTcp [124](#)

TCP/IP initialization and termination procedures

begin TCP/IP service [119](#)

close a TCP connection [120](#)

close a UDP port [127](#)

determine whether an address is local [131](#)

end TCP/IP service [119](#)

instruct TCPIP to obey a file of commands [132](#)

obtain current status of TCP connection [124](#)

obtain status information from TCPIP [132](#)

open a UDP port [128](#), [129](#)

open TCP connection [121](#)

receive raw IP packets of a given protocol [130](#)

receive TCP data with FRECEIVtcp function [120](#)

receive TCP data with RECEIVtcp function [123](#)

receive UDP data [128](#)

send an ICMP echo request [133](#)

send raw IP packets [131](#)

send TCP data [123](#)

send UDP data [128](#), [129](#)

specifying the notifications to receive [119](#)

tell TCPIP that your program will no longer use a particular IP protocol [130](#)

tell TCPIP that your program will use a particular IP protocol [130](#)

TCP/UDP/IP initialization and termination procedures

BEGINTcpIPservice [119](#)

ENDtcpIPservice [119](#)

HANDLEnotice [119](#)

TCPIP communication CALLCODE notifications [117](#)

TCPIP communication CALLCODE requests [115](#)

UDP CALLCODE requests

CLOSEudp [127](#)

NRECEIVEudp [128](#)

OPENudp [128](#)

SENDudp [128](#)

V6OPENudp [129](#)

V6SENDudp [129](#)

when to use [113](#)

VERFY [280](#)

well-known port assignments

TCP [345](#)

UDP [346](#)

WRITE (IUCV) [185](#)

WRITEV (IUCV) [185](#)

X

X_ADDR environment variable [31](#)

X_SITE environment variable [31](#)

X_XLATE environment variable [31](#)

xdr_accepted_reply() [217](#)

xdr_array() [218](#)

xdr_authunix_parms() [218](#)

xdr_bool() [219](#)

xdr_bytes() [219](#)

xdr_callhdr() [220](#)

xdr_callmsg() [220](#)

xdr_double() [220](#)

xdr_enum() [221](#)

xdr_float() [222](#)

xdr_inline() [222](#)

xdr_int() [223](#)

xdr_long() [223](#)

xdr_opaque_auth() [224](#)

xdr_opaque() [224](#)

xdr_pmap() [224](#)

xdr_pmaplist() [225](#)

xdr_pointer() [225](#)

xdr_reference() [226](#)

xdr_rejected_reply() [226](#)

xdr_replymsg() [227](#)

xdr_short() [227](#)

xdr_string() [228](#)

xdr_u_int() [228](#)

xdr_u_long() [228](#)

xdr_u_short() [229](#)

xdr_union() [229](#)

xdr_vector() [230](#)

xdr_void() [231](#)

xdr_wrapstring() [231](#)

xdrmem_create() [231](#)

xdrrec_create() [232](#)

xdrrec_endofrecord() [232](#)

xdrrec_eof() [233](#)

xdrrec_skiprecord() [233](#)

xdrstdio_create() [233](#)

xprt_register() [234](#)

xprt_unregister() [234](#)



Product Number: 5741-A09

Printed in USA

SC24-6332-74

